Computing project – Analysis section

Introduction:

My project is essentially a second-hand bookstore that will allow members to sell their unwanted second-hand books to other members, as well as purchase (or trade?) books. Members will have to state the condition of the book when posting a sale and, based on this (and a few other things e.g. hardback/paperback, demand of book), an appropriate price will be suggested. The webstore will also make book recommendations based on previously bought books (e.g. same genre of book will be recommended). Users will also have a book list where they can add books they want to purchase/read in the future.

This project is for the students at my school to allow for an easy way to share books. Prices will be allocated via a credit-based system, meaning that no money will be involved, and it will be the exchange of books rather than the purchase of books, making it more accessible to everyone. This also means that you have to sell books to earn credits and you need credits to buy books, which will help the functionality of my project since people will have an incentive to sell their books.

I have chosen this project because I love reading and have so many books I have already read just lying around in my room. Therefore, I thought this project would be a good way of repurposing these books whilst being able to read the books one wants to read for cheaper than buying them at a regular bookstore, and in an eco-friendlier way as well.

1: described and justified the features that make the problem solvable by computational methods, explaining why it is amenable to a computational approach.

Computational methods

Through identifying computational methods and applying them to my project, I will be able to break down the complexity of my project and better understand how to tackle it. Computational thinking is the ability to think logically about a problem and apply techniques for solving it. This process will allow me to identify different solutions to my project, and which one will be the most efficient and effective for my project. Essentially, the process of computational thinking and breaking down the different aspects of my project will help me understand how I can actually create it.

Thinking abstractly:
Abstraction is essential for designing any computer science project because it helps us simplify our project and focus only on what is essential to it rather than the irrelevant details. To apply abstraction means to omit all the details that don't contribute to the essential characteristics of the project.

Within my project, I will make use of abstraction by filtering out aspects of a typical bookstore that wouldn't be needed in mine. Firstly, for my project to calculate a reasonable price for a book based on what information the user has provided, I will need to use abstraction to decide what information will contribute to this. I would abstract out details such as the colour of the book or the picture on the front cover of the books that wouldn't need to be considered when coming to a decision on how to price the book. This would allow me to focus on the details that do need to be considered, such as what condition the book is in, which is important to consider for second-hand books, whether the book is a paperback or a hardback book, and how in demand the book is across the store. By this I mean that if there are a lot of the same books being sold on the website, the suggested price would be lower than if there are very few of the same books being sold. I will also need to abstract when thinking about how to program the recommendation aspect of my project. The recommendations would have to be tailored to each user and based on certain things, so I will have to abstract irrelevant details for this specific aspect of my project that may be relevant in other aspects of the project. For example, I would abstract information on book demand when coding the recommendation aspect, even though this information is useful in the price suggestion aspect of my project. Therefore, it is clear to me that I will have to break down my project into separate steps that focus on different things, and abstraction will need to be applied to each one based on its purpose. Overall, my project is quite abstract in the way that there are a lot of different aspects to consider and not much to abstract when looking at it as a whole, but by breaking it down, abstraction can be applied much more effectively and in a more useful way.

Thinking ahead:
Thinking ahead is the process of thorough planning to ensure efficiency as well as specifying inputs, outputs, and preconditions of a computational problem. Identifying inputs and outputs is advantageous because it leaves no ambiguity in what is inputted and returned. I will also have to consider that the algorithm must work for all inputs, even incorrect ones. Rather than crash, it should be able to respond and notify the user that the input is incorrect. For example, when a user is creating their account, they will need a username. One of the preconditions of this would be that the username has to be at least 8 characters long. If the username doesn't include this, my algorithm should respond that the input is invalid, and they must input something else. By including preconditions, my program should be easier to debug because the conditions will be clearer, and it will also be easier to test because I will know what checks need to be made.

As part of thinking ahead, I will be able to maximise efficiency by reusing components of my program. By clearly documenting functions and procedures, I can save time on writing and testing new code by reusing already debugged and tested code wherever I can. For example, whether or not the book is hardback or paperback can be considered when suggesting a price to sell as well as making book recommendations because someone may have a preference of one based on their purchase history. Therefore, I can reuse parts of my code and include it in different steps of my program. Code optimisation will reduce the overall project time.

Thinking procedurally and problem decomposition:
Thinking procedurally is all about identifying the components of a problem and breaking down the steps of the problem via decomposition. By breaking down my project into smaller steps, I am also partaking in a method known as divide-and-conquer. Through thinking procedurally, I will be able to better identify the necessary sub-procedures and the overall solution of each step. If I were to decompose my problem into a series of sub-problems, it would look somewhat like this:

1. Creating a membership for the user
2. The process of a user posting a book for sale
3. The process of a user purchasing a book for sale
4. Using past information to make book recommendations
5. Making the book list user-friendly and functional

By decomposing my project into just these 5 simple, still very abstract steps, I have already made it easier for myself as I am now able to focus one each one and identify the sub-routines and conditions needed for each one. We can also order the steps in order of importance and prioritise the aspects of each step. For example, the basic functionality of selling and purchasing books is probably the most important feature of the whole project, so these two steps would be prioritised. Additionally, I can make use of a top-down design model to further break down the identified sub-procedures of my problem. This process of thinking procedurally will allow my program to be easier to maintain and debug because it is much easier to find errors in a smaller, self-contained piece of code.

Thinking logically:
When thinking logically to solve a problem, it is necessary to understand and identify when decision making is needed and what decisions need to be made. Programming constructs are an important part of programming. IF and ELSE statements are an example of sequences and can be used in my program to allow for a variety of inputs. They are necessary in many aspects of my project, such as when suggesting the sell price of a book. Using IF statements will allow the price suggestion to be based on the different inputs, and the output will be different based on the combination of different inputs and the process of sequential branching will allow this. Iteration can also be used to enter and exit loops. For example, when a user is logging in, an iterative statement may be used for the password, and the program would exit the loop once the password has been entered correctly or until the user has used up the maximum number of tries. The use of sequences, iterations and branches are extremely important in ensuring that the program functions properly and doesn't crash.

Thinking concurrently:
Thinking concurrently is all about executing parts of a problem simultaneously, which saves time. Concurrent processing is used to reduce wait time by running instructions simultaneously on multiple processors on a single computing device. While this is useful, I am not sure yet if I will need to make use of this, although it is helpful in reducing loading time of everything. Also, concurrent processing is hard to implement as programs need to be specifically written to run on multiple processors which makes the code more complex, and it may be too ambitious. However, a lot of my

project requires for things to be done simultaneously. For example, when suggesting the price of a book, multiple things must be taken into account, so pipelining could be utilised to reduce the wait time for a price to be suggested.

Performance modelling:
Performance modelling is the processing of creating a model of a computer system or application using mathematical approximation rather than performance testing. It is useful because the model gives an indication of system operation and delays in processing. I can utilise performance modelling to measure the runtime of my algorithms and identify where there may be any issues. It can help me pinpoint what parts of my program need to be optimised to help make my store run more smoothly.

Heuristics:
Heuristic methods are designed to find the solution to a problem based on an educated guess when there may not be a simple, one-answer solution. This is relevant to my program because the aspect of suggesting prices relies heavily on educated guessing and does not have one simple, straight-forward answer for each book. Therefore, my program could make use of heuristics by taking the inputs given by the user when posting the book (condition of book, demand of book, etc.) and outputting a solution that may not be exact but is an educated guess and accurate enough output based on the inputs given. It is also faster than using the standard approach and applies very well to this scenario.

Data mining:
Data mining is a method used to discover patterns through collecting and then analysing huge amounts of data, trying to find connections within the data. This could potentially be very useful for my program, as a large part of my project is the book suggestion based on past purchases or based on what is on a user's booklist. Therefore, data mining could be used to collect a user's data and analyse it to figure out their most commonly bought or viewed genre, as well as whether they only lie their book in perfect condition or they don't mind, and many other things. However, data mining is extremely hard to implement, so I am not sure if I will be able to utilise it in my project, as it involves managing big data tasks, which often needs multiple machines or supercomputers to do so.

2. Identified suitable stakeholders for the project and described them explaining how they will make use of the proposed solution and why it is appropriate to their needs

Stakeholders:

Librarians:	**Students:**	**Parents:**
One of the main stakeholders is the school librarians because the transaction of the books would be managed by them in that the books would be dropped off and picked up in the library, and the transaction would be managed there through a credit system. I will therefore interview them to get an idea of what they think the requirements of the project may be, and also if there is anything my program isn't taking into account that it should be when using past information to make recommendations for users. My project could also potentially be appropriate to their needs because they could include the already existing library books on there and would also have to spend less money buying new books as students would hopefully be able to provide a lot of books too. This would also be helpful to the librarians because it would mean that if all the copies of a book had already been borrowed from the library, a student could potentially have their own copy that they are selling.	Another main stakeholder of the project and the people who will most likely be making use of my project the most is the students in my school. They will be able to have access to the books posted by their fellow students as well as sell their own books that they've already read. This is appropriate to their needs because it would give them a platform from which to browse and purchase books as well as get personalised recommendations, whilst simultaneously getting rid of the finished books that are just taking up space in their room now. Furthermore, because the cost is credit-based, it is suitable for students who may be unwilling to spend actual money on books but want to read, as my project allows for this. Because the students will be using my project the most, it is important that the end result is user-friendly and easy to use. Therefore, I need to consider making the user interface self-explanatory and easily navigable.	A perhaps less common but still a potential stakeholder could be the parents of the students. Although they may not be using it as much as the students, some parents may want access to the project so that they are able to purchase books for their children, as well as sell their kids old books. It is therefore important to again keep in mind how user-friendly my program is, as the parents may not be using my program as frequently as the students will be, and also may not be as technologically knowledgeable.

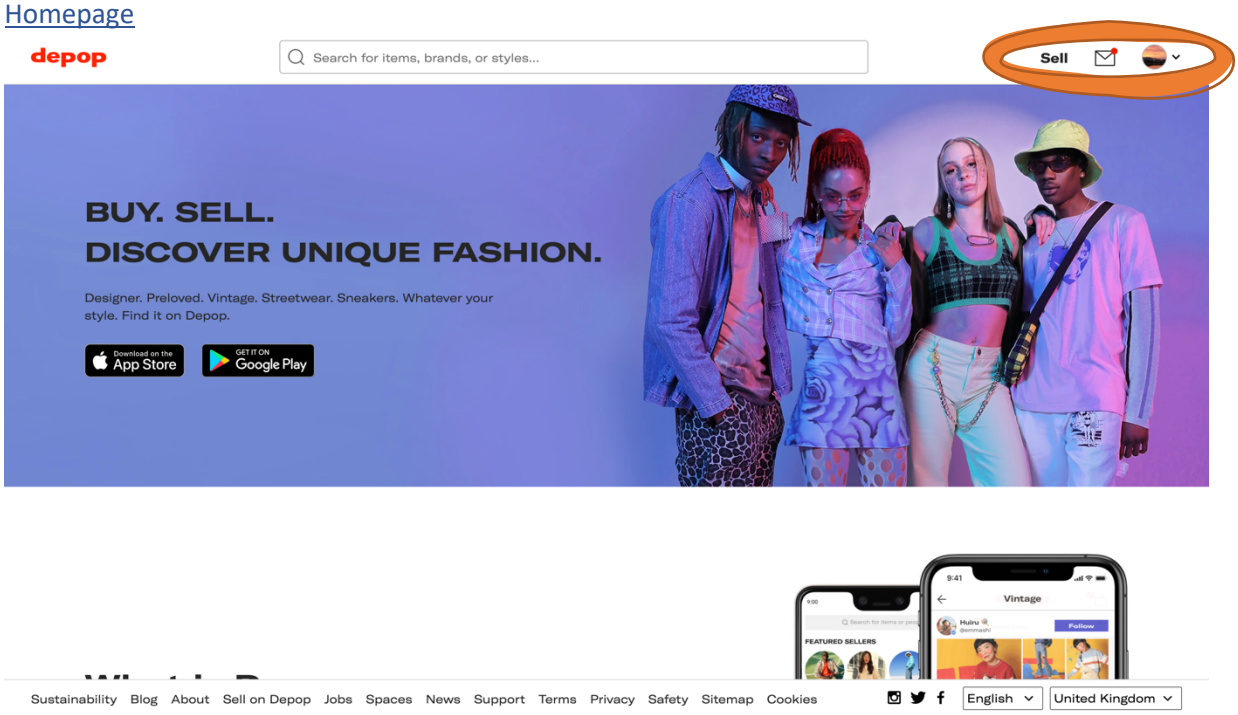
3. researched the problem in depth looking at existing solutions to similar problems, identifying and justifying suitable approaches based on this research

Research

In this section, I will be researching existing solutions and evaluating what I like or don't like about them, giving me a better idea of how I want my project to be implemented.

Depop:

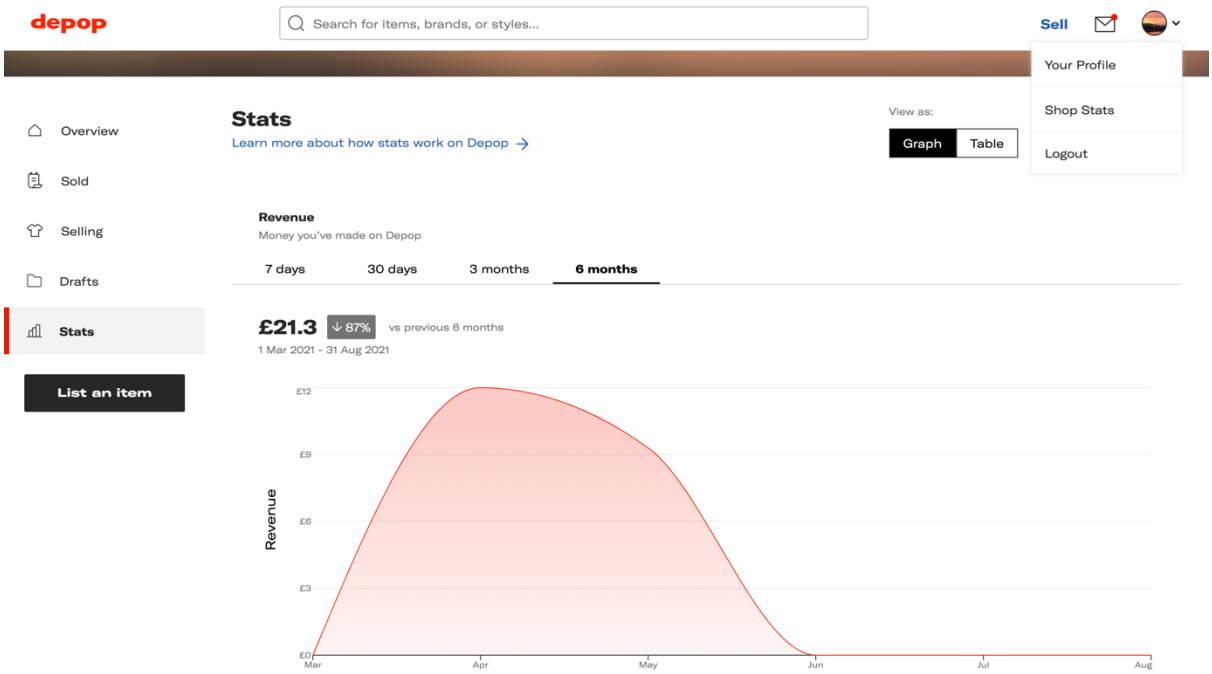
Homepage



I started with researching depop since it is a similar concept to my project but based around selling second-hand clothes rather than books. What I first noticed when logging in is that I liked the three buttons in the top right-hand corner, where one is for selling, another is for messaging buyers/sellers, and the third is your profile. This seems easy to navigate as they are clear and also aesthetically pleasing and user-friendly so I would definitely like to incorporate this kind of usability into my project. However, maybe instead of a messaging icon, I could have a book icon and have it as the book list.

What I didn't like as much about the homepage of depop is that there isn't much help with finding what you want or browsing. There is simply just a search bar for you to look up what it is exactly that you want, however, it means that you need to know what you want or what you're looking for before you go on, which isn't helpful for people who are just looking for something they want to buy. For my project, I would like to have some kind of filtering system where there can be more help searching for what you want, whether you're searching by genre, by condition or by price. Other than that, I like the simplicity of the homepage and how it is quite easy to navigate as a new user, but I think the homepage could be a bit more productive and helpful than just a photo and a search bar. Perhaps my project could have more on the homepage and maybe have two sections called buy and sell and you can click either to make it more easily navigable and even more obvious to the user. Also, the small sell button in the top right corner may not be very obvious, and if my project is equally about buying and selling, then maybe the sell button should be a bigger part of the homepage than it is on depop.

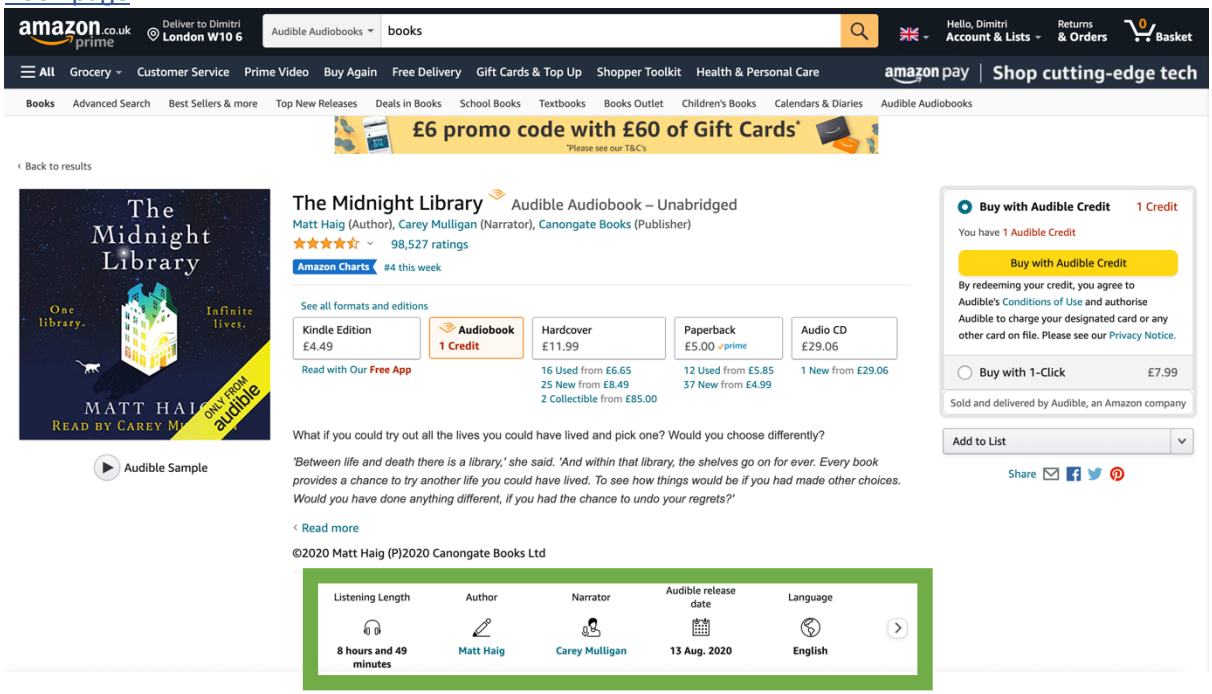
Stats page



Another thing I liked about depop is that when you clicked on your profile picture icon, you had the option of going to your profile and seeing your user details, going to shop stats, or logging out. I especially liked the page on stats, which I have shown in this picture, because I think it would be nice to incorporate into my project even though it's not necessary. Having my own stats page showing the user's past sales and past purchases, and maybe even a section on tailoring their recommendations by setting their favourite genre and so on could be a useful resource for the user. I also like the look of it because the graph on revenue is easy to read and the timeline for it can be changed, so hopefully I cold portray the user's selling revenue or something along the lines of that in a graph as simple and easy to read as the one on depop.

Amazon:

Book page

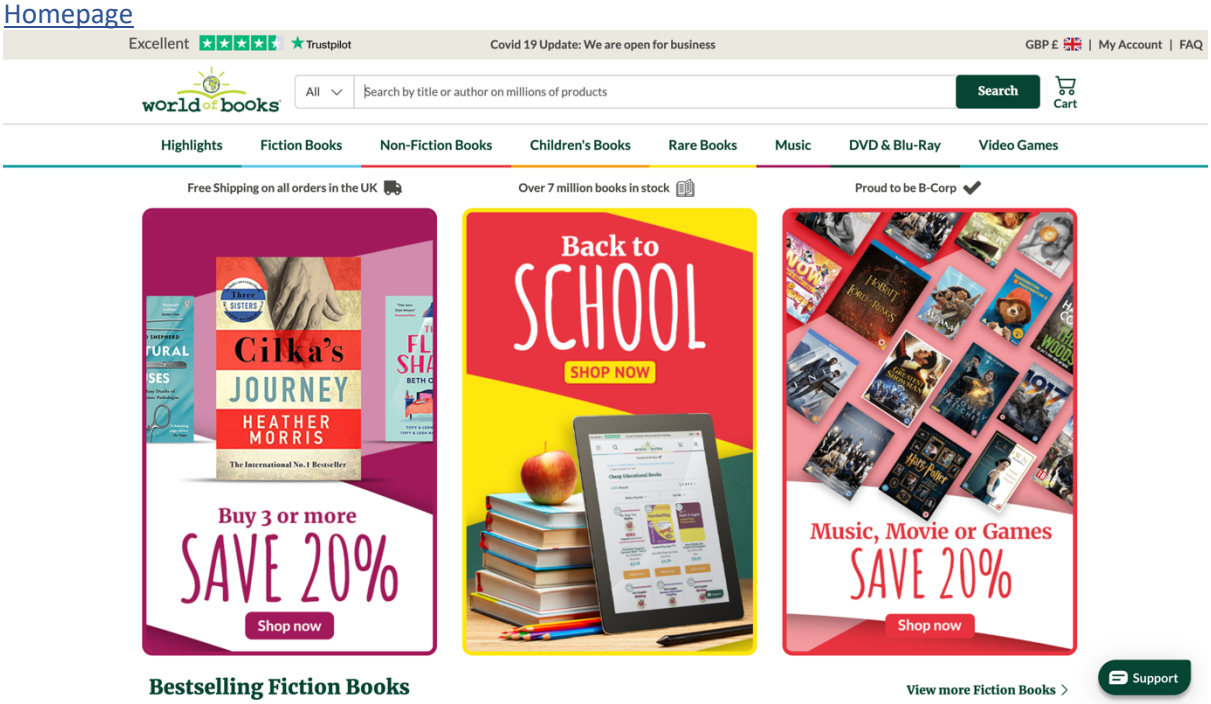


What I like about amazon is that when you click on a book, it displays the information nicely. The picture is on the left-hand side and all the information on the book next to the picture. I think it is very clear and very easy to use and understand, whoever the client is. What I especially like about the picture I have put is what I put the green box around. I like how amazon includes information about the book with little icons, making it easy to read and fun to look at. Perhaps I could do something similar to this when a user posts a book to sell, using the information they posted about the book and displaying it in this fun way. For example, I could have an icon showing whether or not the book is hardback or paperback, an icon for the author like amazon, an icon for the condition, and whatever else. This could be instead of having a listed description next to the book photo, as it could be a more fun and easy way to look at the information of the book.

I also really liked that when you scrolled down from the book you were looking at, there was a little list showcasing books that people viewed based on the book you are reading. I could incorporate something like this but have it be the recommendation feature, so it would just say 'your recommendations' instead and would be suggesting books that the user might like and can easily click on and look at.

World of books:

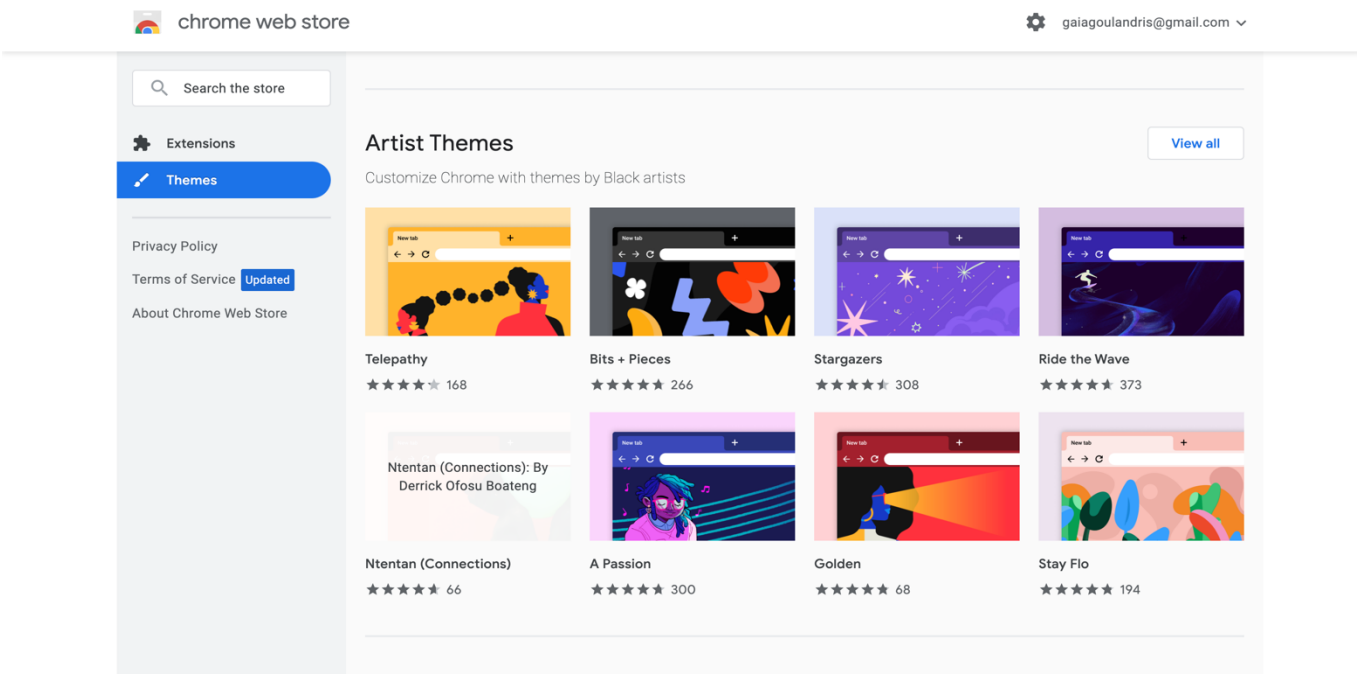
Homepage



What I like about the homepage of the world of books as opposed to the homepage of depop is that under the search bar it has a few genres of books and other things that you can click on to browse that genre. This is definitely a good feature to consider for my project because it means that it can be used for browsing books and using my program when you're not sure what you're looking for or if you're simply just looking for something new to read. I also like the logo in the top left-hand corner and think it would be nice to have a logo of my own for my project.

What I don't like as much about world of books is the colour scheme. I think the use of neutral colours as the background and bright colours as the foreground looks less professional and less well-done than some other webpages, and it gives the illusion that the webpage is not as user-friendly. This will help me better understand what I want for the aesthetics as my project, as it will save me time trying out a colour scheme like this that I now know I am not a fan of.

Chrome web store:



I like that in the chrome web store, you are able to customise your google homepage with themes. This could be a nice idea to incorporate into my project because I could then worry less about the aesthetics and what might be preferrable to each client, because I won't be able to do something that every student likes, so giving them the ability to customise what my project looks like could be fun for them and also a way to make it line up with their aesthetic and what they find to be the most user-friendly.
However, as cool as this would be, I think it is too ambitious for my project so I don't think I will be able to include it as a feature.

4. identified the essential features of the proposed computational solution explaining these choices

Features of my proposed computational solution:

Account system

This feature will allow students to create an account and have login details. They will have a profile where they will be able to input their email so that they can receive notifications of when an item of their own has been sold or when they have bought an item. In their profile, they can also tailor their suggestions feature by inputting the genres they like and don't like, and other preferences. Their data will need to be stored on a database so that their information, booklist, suggestions, purchases and more can be saved. I would also like to include some kind of stats page or dashboard within their profile where they can easily view how many books they have sold, how many books they have purchased, and some other statistics involving their interactivity, however, I'm not sure that this will be achievable in the time scale I have for this project.

Browsing system

A key feature of my system will be being able to browse books to buy. Again, I want this to be as easy and accommodating as possible for the user, so I'd like to include filters to allow for more specific browsing as well as categories to browse from. I'd like the categories to be presented horizontally near the top of the homepage, below a search bar. Having both the ability to browse and a search bar means that the user can both be specific with what they want or look for a new idea of something to read.

Booklist

The booklist is essentially a Wishlist for books. It will enable users to add books that they see while browsing to their booklist. It basically functions as a save for later feature, as users can go back to their booklist and later purchase the books they have read. This will also require a database as the data stored in the booklist will need to be saved somewhere so the user can revisit it.

Suggestion system

The suggestion system will be another way of browsing for books, whereby suggestions are made to the user based on their past purchases and what they may have set as their favourite genre/s and other things in their profile section.

Resell feature

Once a user has bought a book, the ability to resell the same book will become available for a quick and easy post of the book to the user's profile, because hopefully once the user has read the book they bought, they will resell it again since it is after all a second-hand bookstore. This feature doesn't mean that they have to post the book that they bought for sale, it's just a way to post the book quicker than manually putting in all the information of the book themselves.

Transaction system

The transaction – both monetary and books-wise – will be managed by the librarians whereby the books will be dropped off by the seller at the library for the buyer to pick up. In terms of what the users will get for selling a book, they will get credits that can then be used to buy more books from other people. When a book is purchased, the credit value would be deducted from the user's total credits and when a book is sold, the credit value will be added to the user's total credit value. So that each user can buy books at the start of my project, each user starts off with 5 credits each once they make an account. The reason there is a credit system is because making a monetary system whereby students can make money would be too challenging given the time constraints of this project.

Graphical user interface/look of my solution

I previously discussed having the feature of users being able to personalise their homepage and cater it to their likings to maximise user-friendliness and satisfy preferences but given that this may be challenging under the time constraints of this project, I must also think about the aesthetics of my solution if this is not possible. I would want my solution to be as easy to use and user-friendly as possible, with everything being self-explanatory and lots of uses of icons to limit the amount of text, thus making the look of it more fun. I also would use a colour scheme that isn't too bright and crazy, as I want my solution to look professional and clean rather than messy. However, I also don't want it to look too boring and monotone, so small pops of colour would be nice.

Why I chose these features

I chose these features because, after doing research, I was able to see what worked and what didn't work for the webstores and other projects that I looked at and tried to take the best parts of all of them and incorporate them into my solution, whilst simultaneously avoiding or providing better ways to do the things that I didn't like. I also found that with the project I am taking on, a key aspect is user-friendliness and usability, so I have tried to emphasise this throughout all my features and maximise it as much as possible.

5. identified and explained with justification any limitations of the proposed solution

Limitations:

Time scale

Seeing as this is my first time attempting a project of this size, it is likely that I have been too ambitious with all I want to do, and also haven't taken into account the time I will need to spend figuring out how to do something I haven't tried before such as creating and making use of a database. Therefore, it is important that during this project, I ensure that I prioritise the most important aspects of the proposed solution and leave the less important aspects to the end of the project. I may not have time to implement a stats page within the user's profile and I may not have time to include customisability.

Monetary aspect

Seeing as there is a time constraint on this project, after looking into how transactions work and how I would be able to implement this within my project, I realised that it would be too ambitious to try to implement this and seeing as this is my first project of this scale, I have no experience with creating some kind of store involving real money and the level of security that must come with that. Therefore, I have decided that my project would be more realistically achieved and would simultaneously be more beneficial to the users if I used a credit-based system instead, which is much simpler to program, and doesn't involve real money so doesn't require as much security.

Customisability

The problem with the idea of allowing the user to customise the webpage is that I'm unsure what they would actually customise it with. If I used a Canva colour scheme and gave them the ability to change the colours of the homepage and profile, that could maybe be achieved, but in terms of some kind of artwork or nice design that google allows you to customise your homepage with, I'm not sure how I could achieve this or where to find designs that I am allowed to use for my solution. I also worry that it may overcomplicate things and providing too many options for the user to change may be too demanding to program. Therefore, considering the time constraints and complexity of implementing a customisability feature, I will most likely not include this in the end result.

6. Specified and justified the requirements for the solution including (as appropriate) any hardware and software requirements

Hardware and software requirements:

Permanent storage

- Need permanent storage to be able to hold all the user's information as well as store the data from the booklist
- Need to be able to modify, add, and remove information in storage

Input device

- Need a mouse/touchpad to be able to navigate the webpage
- Need a keyboard to be able to navigate and input information

Output device

- Need a monitor/device to be able to display the webpage

Systems software

- Need a processor
- Need RAM to be able to use the webpage
- Don't need cache but the webpage will run quicker with cache

User interface

- Need an easily navigable, professional yet not boring, consistent user interface that is self-explanatory to use

Security

- Need sufficient security features to protect users' personal and private information, as well as sufficient security to prevent outsiders from hacking into profiles
- Data should only be able to be modified by administrators
- Double authentication system and a limit to the number of password guesses, as well as a minimum number of characters for the password and the use of different types of numbers and symbols in the password

7. Identified and justified measurable success criteria for the proposed solution

Success criteria:

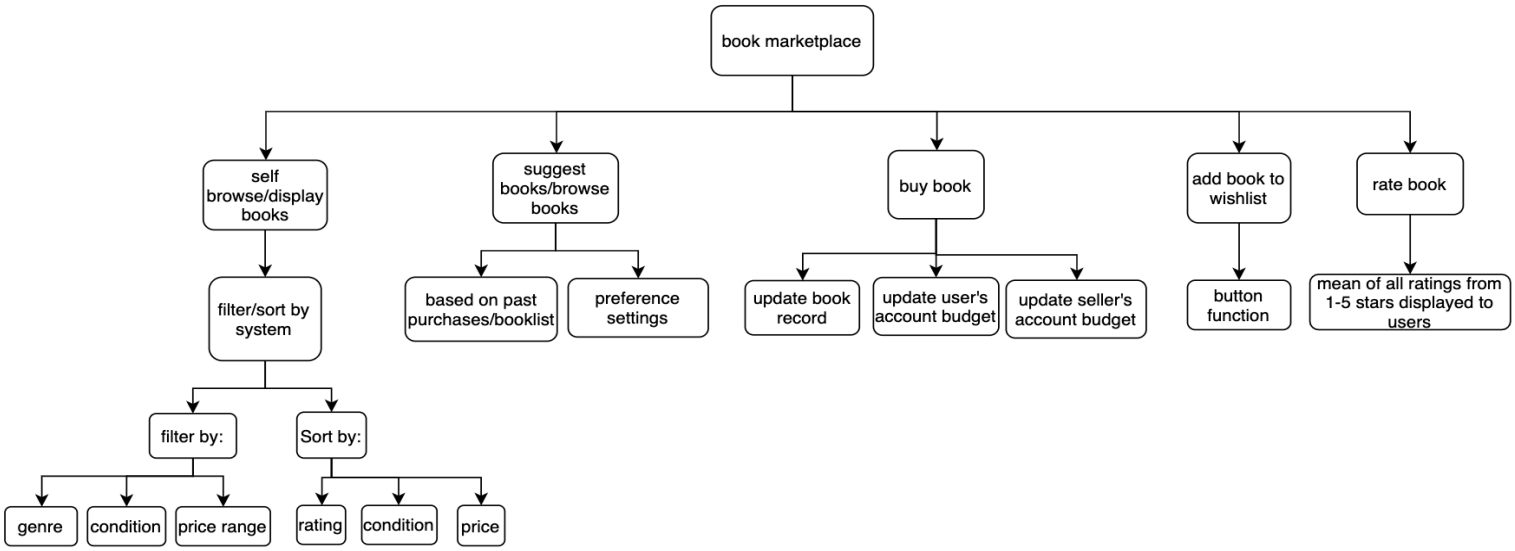
No.	Criteria:	How to evidence:
1	Creation of a user is functioning	Provide screenshot of a user profile
2	Login with password is functioning	Test by logging in to user profile
3	Database is queried to make sure user is valid	Test by using non-existent username and by using incorrect password. Should not log in
4	Books can be added to booklist	Provide screenshot of a book in user's booklist
5	Books can be removed from booklist	Provide screenshot of same book removed
6	Dashboard view displays user's current credit	Provide screenshot of current credit in dashboard view
7	Dashboard view accurately displays user's past sales	Test by buying book and checking in past sales that the book is correctly displayed there
8	Dashboard view accurately displays user's past purchases	Test by creating new user to buy previous test user's posted book, then logging back into previous test user's account and checking that their sold book is correctly displayed in dashboard view
9	Suggestion system is working based on user's preference settings	Test by checking that the genre of the suggestions are mainly based on the user's inputted preferred genres
10	A book can be posted to sell	Provide screenshot of book posted on test user's profile
11	Picture of book is displayed for book posted for sale	Provide screenshot of picture of book being displayed by book sale
12	Books search bar still displays results despite spelling errors or capitalisation	Provide screenshot of search results with slight spelling error
13	Details of book already existing in database can be loaded into book being posted to sell	Test by posting a book already on the database and seeing if the option to load info is available and accurate
14	Credit is added to user's total credit once a book is sold	Test by checking user's total credit before sale and after sale, checking that the correct amount has been added to total credit
15	Book can be bought by user	Provide screenshot of a book in past sale
16	Credit is deducted from user's total credit once a book is purchased	Test by checking user's total credit before purchase and after purchase, checking that the correct amount has been deducted from total credit
17	Book can be sold by user	Test by posting a book for sale, creating new user tp buy it, logging back in to first user and checking past sales
18	New activity is stored after logging out	Test by buying book, logging out, logging in, and checking if book is in past sales
19	Books page has a functional search bar	Provide screenshot of something inputted into the search bar and the results shown
20	Filtering system during browsing is functioning	Provide screenshot of filtering system showing only one genre of book
21	Pages include readable text and usability features such as buttons and a navbar	Provide screenshot of page showing readable text and button
22	Navbar is functioning	Test by clicking on a link to a page on the navbar and ensuring it brings the user to the correct page

Computing project – Design phase

Top-down level diagram:

I decomposed my project down into a series of smaller problems by creating three top-down level diagrams. I did this because it makes my project easier to manage and helps me understand where I should start and what parts of my project I should be prioritising in case I have been too ambitious given my time limitation.

The main top-down level diagram is the 'book marketplace' one, and I decomposed it this way because it is essentially an easier way for me to view all the different main functionality aspects of my project, and what steps are needed to successfully implement each aspect.



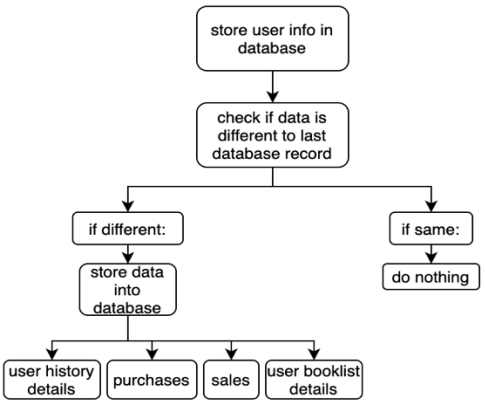
Self-browse/display books – demonstrates the ways in which browsing can be filtered or sorted by

Suggest books/browse books – demonstrates that suggestions can happen based on past purchases but also based on the user setting their own preferences. If the user does not set preferences, suggestions will be based solely on past purchases.

Buy book – demonstrates that a user is able to buy a book and how that would work transaction-wise

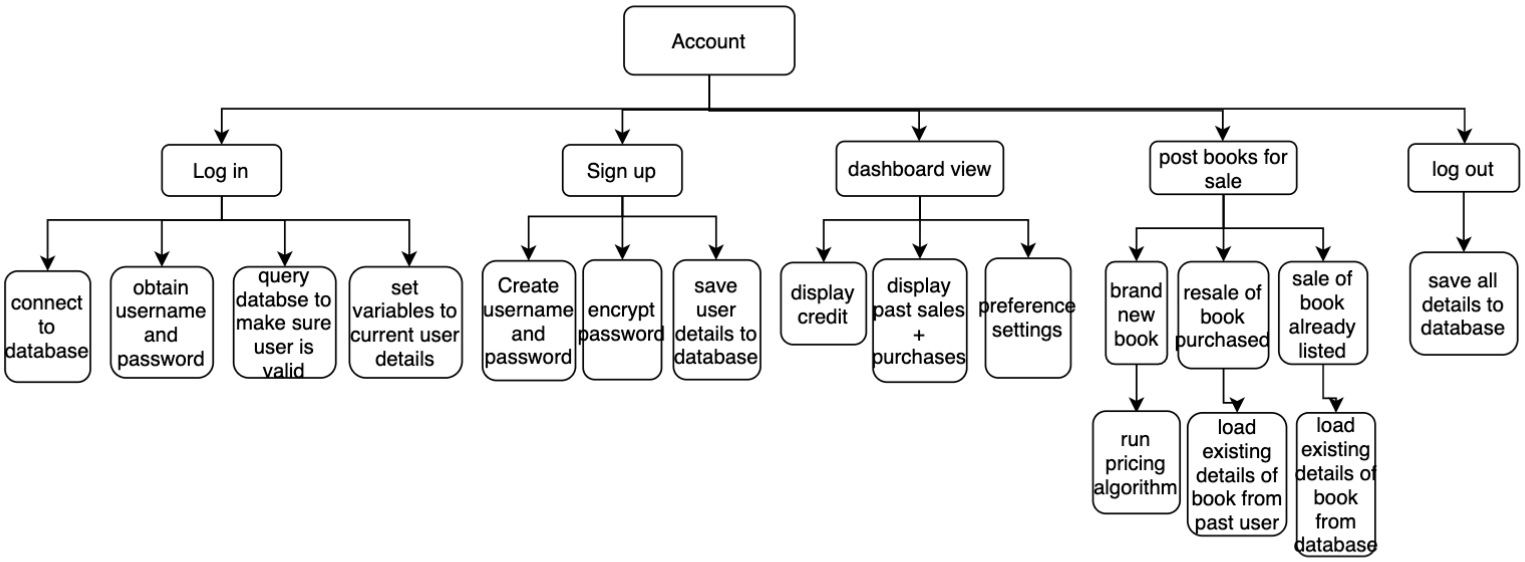
Booklist – this demonstrates how the booklist functions, and it is important to mention that the book is saved to a database because if the books aren't saved to a database, then every time the user logs in or refreshes the page there will be no books in their booklist

rate book – demonstrates that a user is able to rate a book and the mean of all the ratings will be displayed on all books posted with the same title



My database top-down level diagram is useful to me because I have never done a project using databases, so it helps me understand how I should be going about it, and helps me think about ways to implement it in my project.

My account top-down level diagram is useful for breaking down the different aspects of a user's accounts and its features, as well as differentiating between a user signing up and a user logging on.



Log in – demonstrates how logging in works and how it interacts with the databases

Sign up – breaks down the process of signing up , and mentions encryption and databases, which are two things that are important for me to remember to implement and good to break down because they are both new to me so it will be good for me to implement these features into my project separately

Dashboard view – demonstrates what the user should be able to access on their dashboard

Post books for sale – demonstrates the different ways a book can be sold, which is helpful in clearly viewing the three different ways and will ensure that I avoid mixing them up when implementing them into my project. The details for a brand new book will need to be entered in manually by the user. The details for a resale can be loaded into the post from the user who previously posted the book, and can be modified by the user if need be. The details for a book already listed by a different user can be loaded from a database by seeing if the title inputted by the user matches a title already in the database

Documentation of what I have done so far:

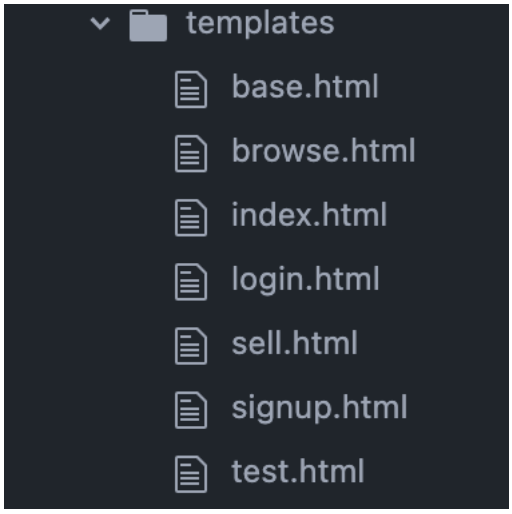
So far, I have done:

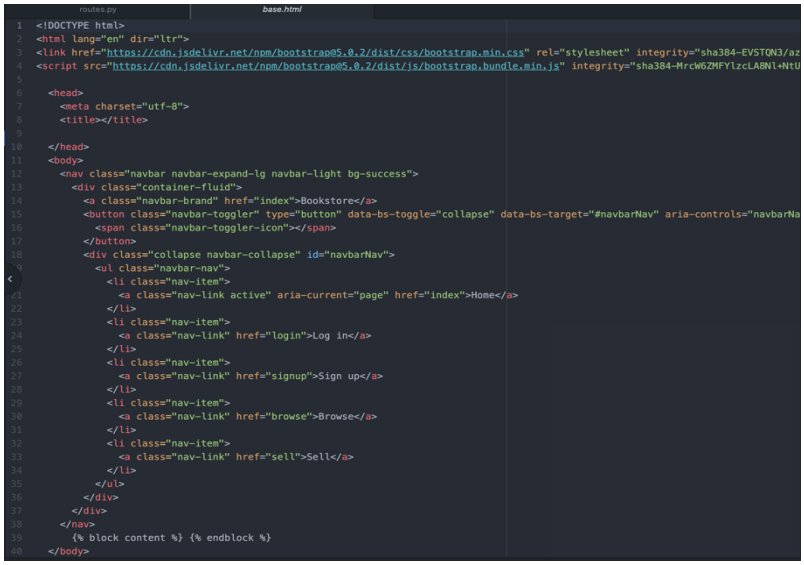
- Basic app setup

```python
from flask import render_template
from myapp import app

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'gaiag'}
    return render_template('index.html', title = 'title', user = user)

@app.route('/login')
def login():
    return render_template('login.html')

@app.route('/signup')
def signup():
    return render_template('signup.html')

@app.route('/browse')
def browse():
    return render_template('browse.html')

@app.route('/sell')
def sell():
    return render_template('sell.html')
```

I created a route for each of the different pages and created an html for each page

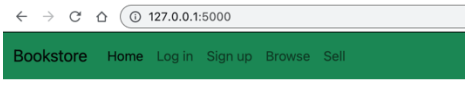
- Created views and associated templates
    - Have created these templates: base.html, browse.html, index.html, login.html, sell.html, signup.html, test.html



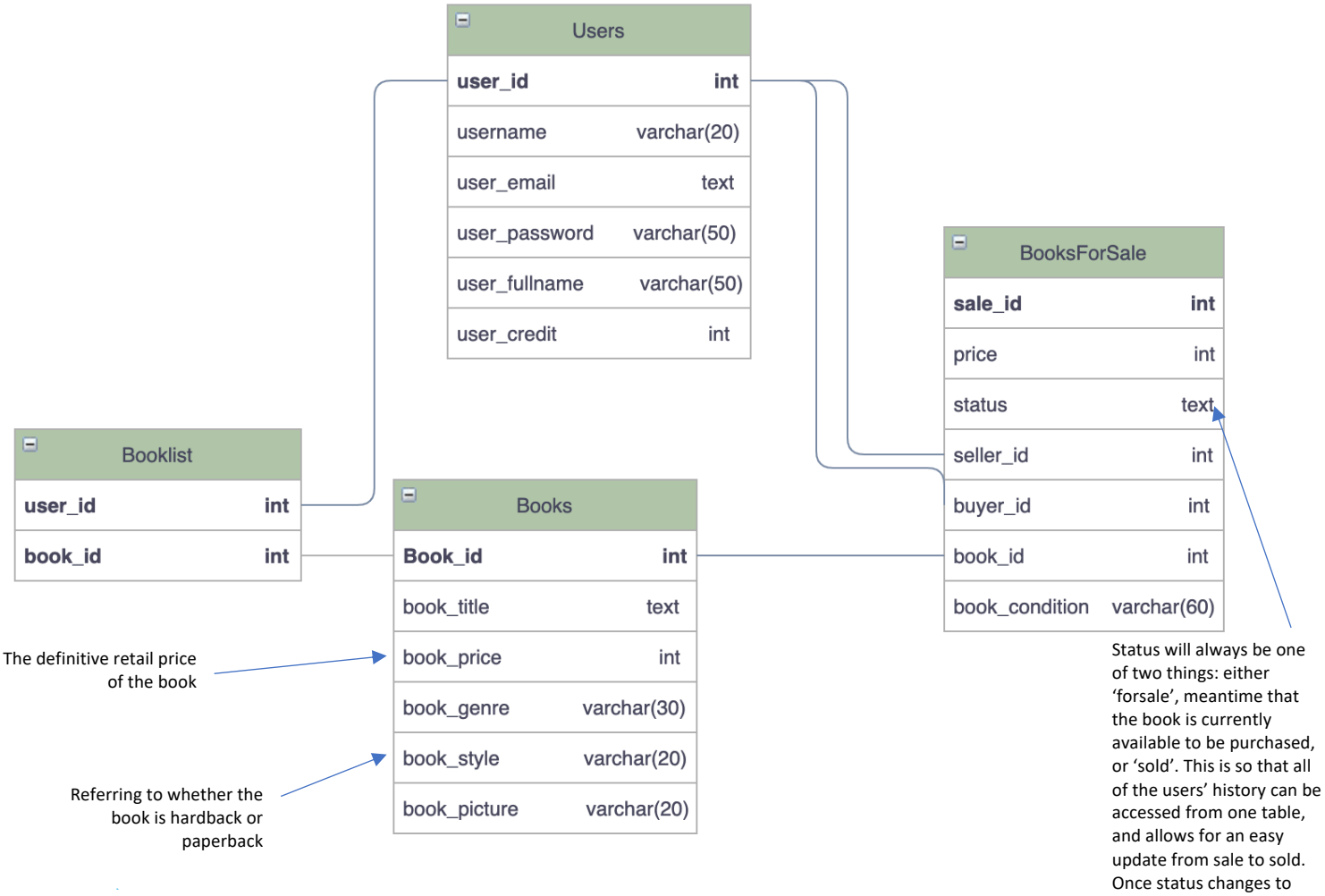


```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-EVSTQN3/az
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtU

  <head>
    <meta charset="utf-8">
    <title></title>

  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-light bg-success">
      <div class="container-fluid">
        <a class="navbar-brand" href="index">Bookstore</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNa
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
          <ul class="navbar-nav">
            <li class="nav-item">
              <a class="nav-link active" aria-current="page" href="index">Home</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="login">Log in</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="signup">Sign up</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="browse">Browse</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="sell">Sell</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
      {% block content %} {% endblock %}
  </body>
```

- Implemented navigation and look/feel with bootstrap
    - Implemented a navbar with bootstrap to make navigation smoother and



Database design:

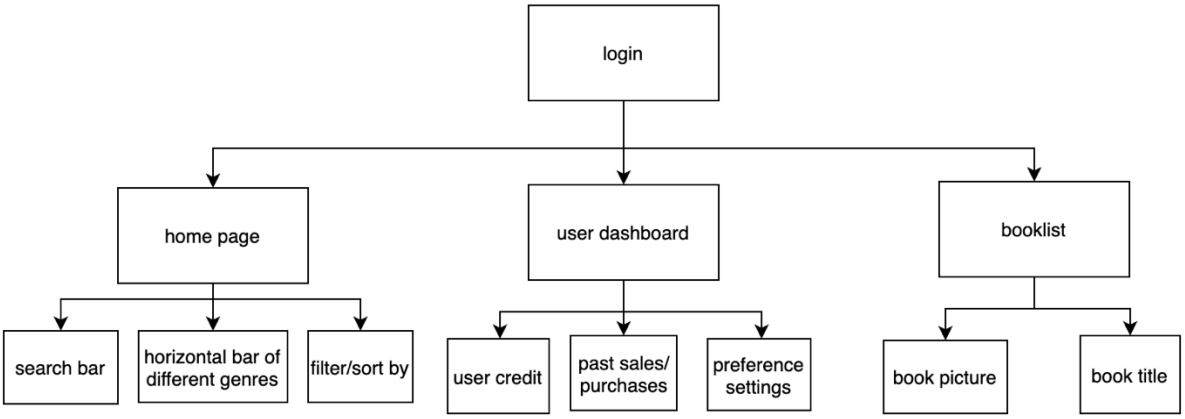


Buyer_id and seller_id are technically the same as user_id in that they will both match a user's user_id, but they will be different user_id's because the same user can't buy the book they are selling.

The difference between the Books table and the BooksForSale is that Books is the table of all books that have ever been entered into my flask app, and is used to autofill the sections when a user lists a previously entered book to sell. BooksForSale, however is used to display the books currently for sale, as well as to display user history.

In a later phase of my project, it came to my attention that status was not a necessary field within the BooksForSale table, so it ended up being removed from the final database. It has been included here to highlight the iterative development process of my project, and how it evolves.

Navigation decomposition:



Login page - each user needs an account to make use of the features of my project. This will consist of a username and password. The username must be unique to the user but the password does not have to be unique. It does, however, have to have a minimum amount of characters and must contain numbers and a special character to ensure that it is not guessed too easily

Home page – the home page will be the main page for browsing books. The majority of the home page will show suggested books based on past purchases or user suggestion settings. There will be a search bar to search for a specific book, and a horizontal bar displaying different genres that can be clicked on and will display the books under this genre. There will also be an option to filter books based on their genre, condition and price range, as well as sort books by rating, condition and price

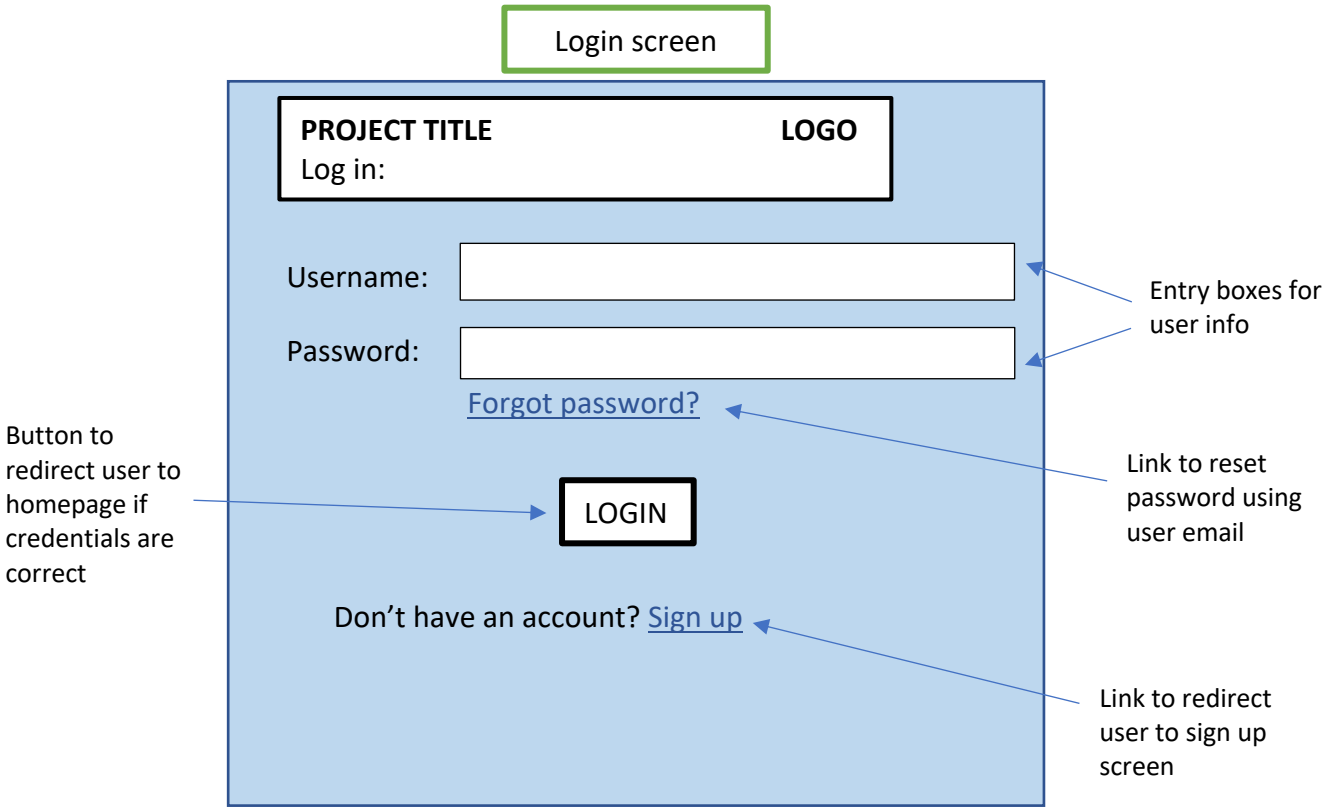
User dashboard – this will display the users credit, their past sales and purchases, and their preferences settings, as well as general account settings. It will also display the user's profile picture and username

Booklist – this will display the books the user has added to their booklist. The books will be displayed with their picture and title

Phase 1:

Design and development:

I am going to deliver a login page as per my description above.
Here is the **user interface design** of the page:



Login page in pseudocode

User enters login name
User enters password
On submit button:
    Query database users to find username=login input
    If not found:
        Inform user
        Refresh screen
    Else:
        Query database users to find password=password input
        If not found:
            Inform user
            Refresh screen
        Else:
            Load user info to profile page from database

Usability features to be included in the solution:

As seen above in the design of the login screen, there are a number of features included to for usability:

- 'forgot password' link
    - This is both a usability and functionality feature – if the user forgets their password, it is a helpful feature to allow them to reset their password so that they don't have to make a completely new account
    - It will be small so as not to distract from the rest of the page, but will be right under the password box so that it is not hard to find
- 'login' button
    - The button will be big and coloured so that it is obvious to the user
- Entry boxes
    - The entry boxes will be large and a sufficiently large text can be entered so that the user can easily spot a spelling error in their entry details

To conclude, usability of this page will focus on text that is appropriately sized so that it is easy to use, titles for the entry boxes so that the user knows where to input their details, clearly shown buttons, and links that can be easily seen but don't distract from the main functionality of the page.

Link to success criteria:

- 'Pages include readable text and usability features such as buttons and a navbar'

Validation:

For the login page, the user will input their username and password. The user's inputs must be validated to ensure the correct data has been inputted. The user's inputted username and password must be checked against data in the User table in the database. If the inputted information matches, then the user can log in successfully, but if not, the user should not be able to log in and an error message should be flashed to the user to make the process more user-friendly. The validation will run once the login button has been pressed.

Key variables:

User – a variable as part of validation that queries for a user that has the same email as the one inputted by the user

- This variable is vital because it firstly ensures that the email the user inputted exists in the User table, and can then be used to compare the user's inputted password with the password of the User instance in the database

Variables within user table in database that are relevant to log in:

id – integer, primary key, needed to identify each user
username – string, inputted by user during sign up, used to store user's chosen name
email – string, inputted by user during signup, used to store user's email
password – string, inputted by user during signup, used to store user's password and validate user during login

Evidence of development:

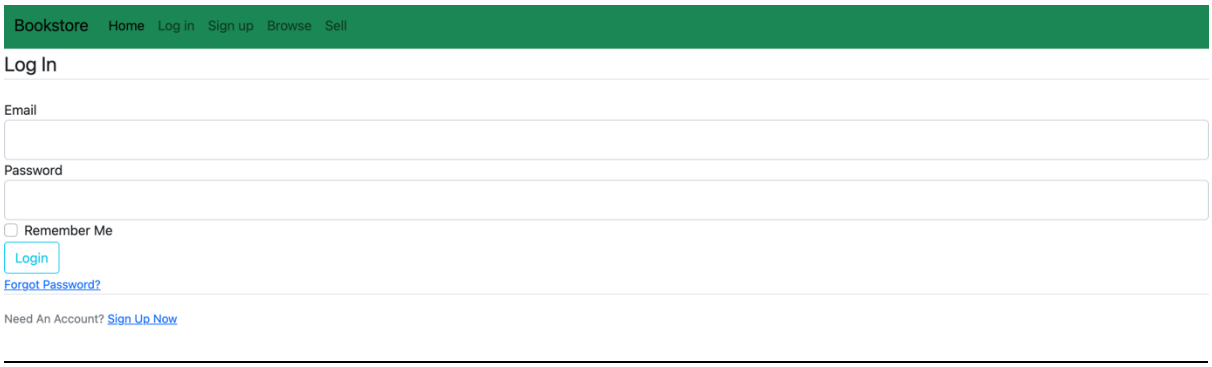
Whilst testing my login page, I ran into an error message saying that my redirect line in my login.html template was wrong.

```
  File "/Users/gaiagoulandris/Documents/GitHub/Gaia/phase2/myapp/templates/login
.html", line 50, in block 'content'
    Need An Account? <a class="ml-2" href="{{ url_for('register') }}">Sign Up No
w</a>
```

I realised that it was because my url_for was redirecting to 'register' – a page that doesn't exist – rather than 'signup' which is what I actually called it.

```
50          Need An Account? <a class="ml-2" href="{{ url_for('signup') }}">Sign Up Now</a>
```

So, to debug this, I changed 'register' to 'signup', and sure enough, my flask app was able to run again.



Now that I have a login page with a button and forms, I now need to create my database so that the login page can actually be functional.

After having created the tables of my database, I tested it and got this error:

```
  File "/Users/gaiagoulandris/Documents/GitHub/Gaia/phase2/myapp/routes.py", lin
e 45
    user_id = db.Column(db.Integer, nullable=False, db.ForeignKey('user.id'), nu
llable=False)
                ^
SyntaxError: positional argument follows keyword argument
(flasklessons) gag-MacBook-Pro:phase2 gaiagoulandris$
```

I realised after reading It that I had included "nullable=False" twice, and that nullable shouldn't come before the foreign key part either. So I changed my code accordingly:

```
class Booklist(db.model):
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
    book_id = db.Column(db.Integer, db.ForeignKey('book.id'), nullable=False)
```

Testing:

Testing what	test data	Expected result	Actual result	Success?
Database is queried to make sure user is valid (point 3 on success criteria)	use incorrect username and incorrect password	Should display an informative error message and not allow log in	Displayed informative error message and didn't allow login	success
Login with password is functioning (point 2 on success criteria)	Login using existing username and password	Should pass validation and login successfully	Logged in successfully	success
Navbar is functioning	User clicks on navbar links	Should bring user to correct corresponding page	Brought user to correct corresponding page for all linked pages in navbar	success

Phase 1 review:

What was done:
I implemented the login part of my project by creating the User table of my database and allowing the user to input their login details, and then validating them by using queries and comparing data in the database to the inputted data.

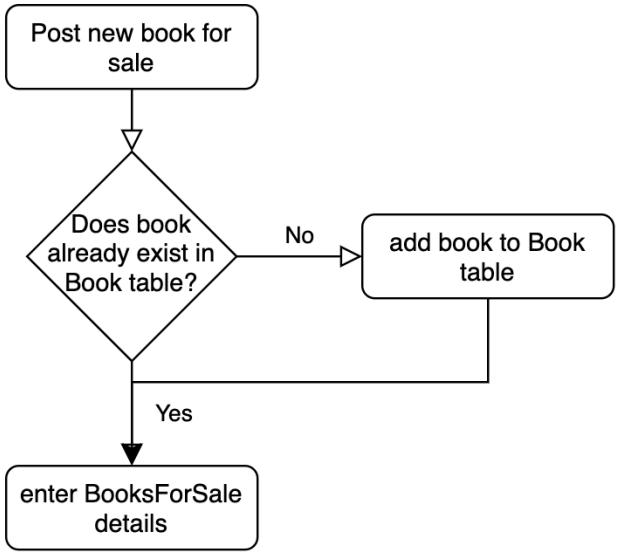
Success criteria I supported:
2 – 'login with password is functioning'
- I supported this success criteria during testing of phase 1

3 – 'database is queried to make sure user is valid'
- I supported this success criteria during testing of phase 1

21 – 'pages include readable text and usability features such as buttons and a navbar'
22 – 'navbar is functioning'

Phase 2:

In my next design phase, I will be working on the 'post books for sale' part of my top-down design, which means initialising my Books and BooksForSale database tables to make them functionable with the users, as well as implementing the different ways users can upload books and how those would be saved to the tables. By the end of this phase, I hope to allow the users to post books for sale. The actual purchasing of the book in terms of transaction will not be functioning at the end of this phase, but that will come at a later phase.



Development steps

Step 1:

- Amend home screen to include button 'post books for sale'
- If user clicks button it takes them to sale page, if not logged in it takes them to login page

Step 2:

- New form called Book form in forms.py
- New template called post_new_book.html
- New function in routes.py that uploads new books to database

step 3:

- new form called Bookforsale in forms.py
- new template
- new function in routes.py

Step 4:

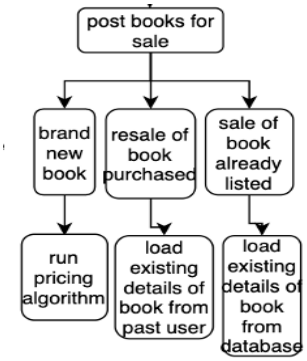
1. new form called checkbytitle in forms.py
2. New template called check.html
3. New function in routes.py
    a. function will query the database for the title entered by the user (make sure that caps/no caps doesn't matter) and if there is a title, it returns it to user and considers books page being done, automatically takes user to booksforsale stage

b. If there is no title, function will redirect to sell function and get user to manually enter book before moving on as usual

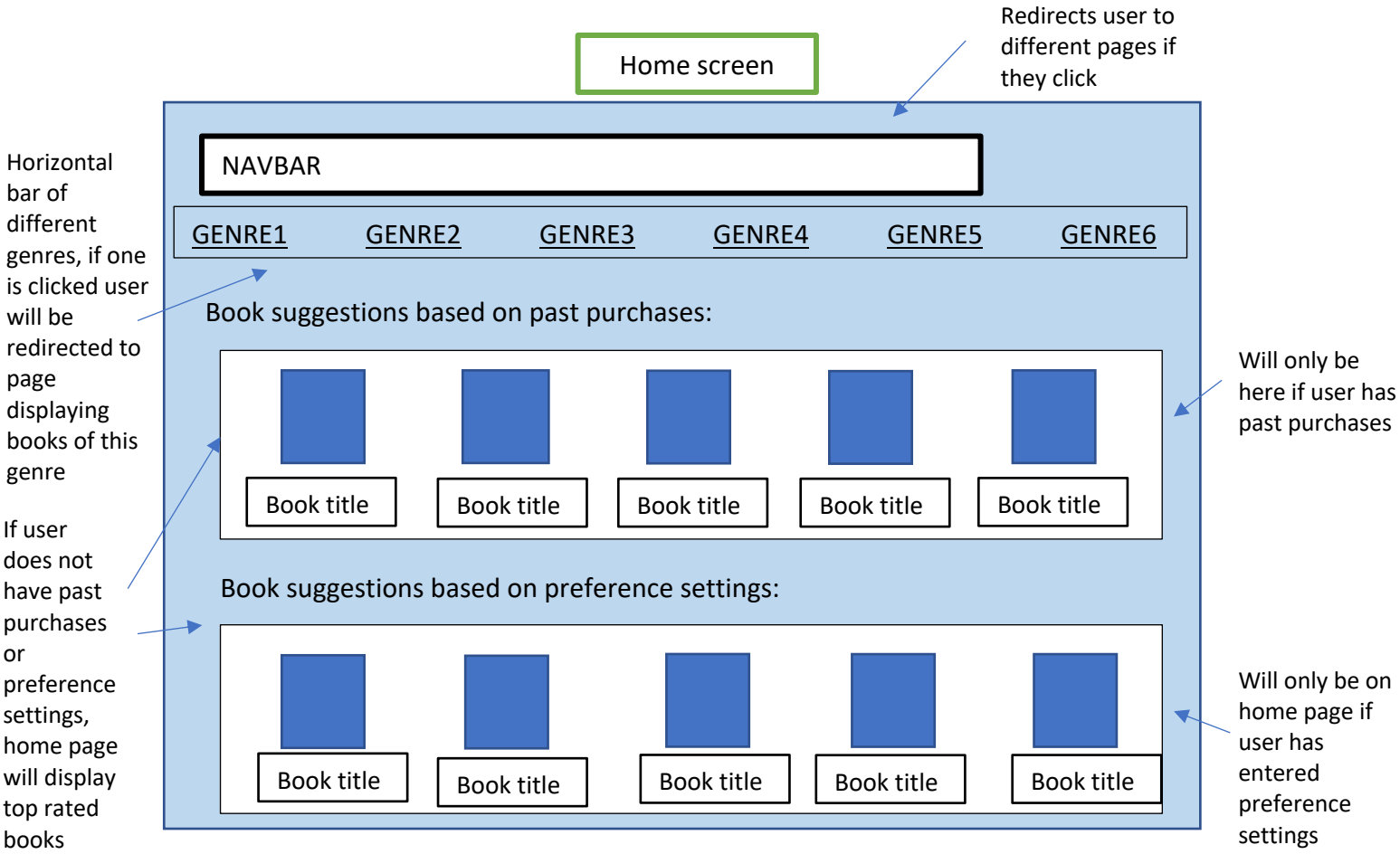
- Check via title if book exists in book table
- If yes, then enter bookforsale details to database and link with already existing book's information
- if no, then user enters info into books form and then enters bookforsale details

step 5:

- Amend home screen to display 6 most recently posted books



Here is the finished design of what I expect my home page to look like at the end of the project. For this phase, I would like for only the books to be displayed, not according to preference or past purchase, just any books that have been posted by any user, as well as a functioning button that can be pressed to post a book for sale.



Home screen page in pseudocode

User clicks ‘post book for sale’ button
User redirected to sale page
User enters title
Query database for title
If title in Books:
    Option for user to select book and automatically fill in book info
    User enters condition of book, book style
Else:
    User enters rest of book info
Endif
User presses ‘post’ button
Sale is saved to BooksForSale and displayed on home screen along with 5 other most recently posted books

Assumptions

- Assuming user will choose price – price set to min 0, max 100 in forms.py
- Assuming home screen will default to displaying most recently posted books (preferences will be done later)

Usability features to be included in the solution:

- ‘post for sale’ button on homepage
    - Button available for logged in users to click on in homepage to redirect to the sale page quickly and easily
- ‘post book for sale’ in navbar
    - Logged in users will also have the option to redirect to the sale page via the navbar – allowing the user multiple options to redirect to sale page increases usability by allowing them quick access to the sale page based on which page they might currently be on
- The way the book is displayed in books page
    - Books page shows all previously posted books, and they are displayed to show the book picture, title and other details of the book that will make it easy for the user to distinguish whether or not that is the book they are looking to sell
- Search bar in books page
    - The search bar is a usability feature that helps the user more easily find if the book they are attempting to sell is already in the Books table – this feature would be especially helpful since the stakeholders are the librarians and the target audience the students of my school, meaning that there would eventually be many books posted, and usability would be made much more difficult if the user had to scroll through all previously posted books to see if the book they are trying to sell is there

Validation:

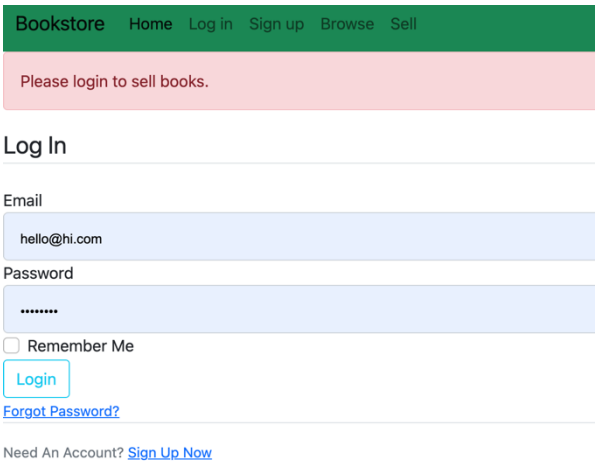
There isn't much need for validation in this phase because when the user inputs data about the book they are trying to sell, they are usually selecting from drop downs. For example, the books genre is selected from a drop down list and therefore requires no validation.

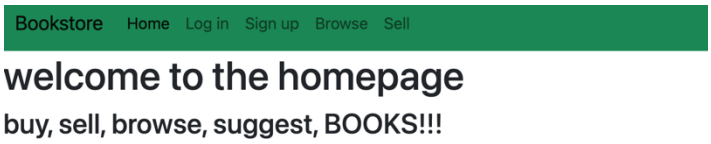
Something like title, however, is an entry text box, in which case validation might be important. However, unless I were to import some kind of library that holds all books in the world so that I can compare the user's inputted title with the library to see if it exists, validation is extremely difficult to implement in this case. Therefore, I must rely on the fact that the user will check over what they have typed, and hopefully not make some type of spelling error. The text will be easy to read to improve usability in this way.

For book price, the user gets to pick it, and the only requirement is that it must be an integer. This means that if the user were to input a negative number, this would count as correct, which clearly does not make sense. Also, there should be some limit as to how high the price can be set, seeing as it is unrealistic that anyone would buy a book priced at 1000 credits when they only start at 5 credits. The point of this project is to be an eco-friendly, accessible point for students to get books, and less of a business focused on the monetary aspect. The main point of credits is to encourage selling as well as buying, rather than actually making any money, which is why a credit limit of maximum 15 for example, makes sense.

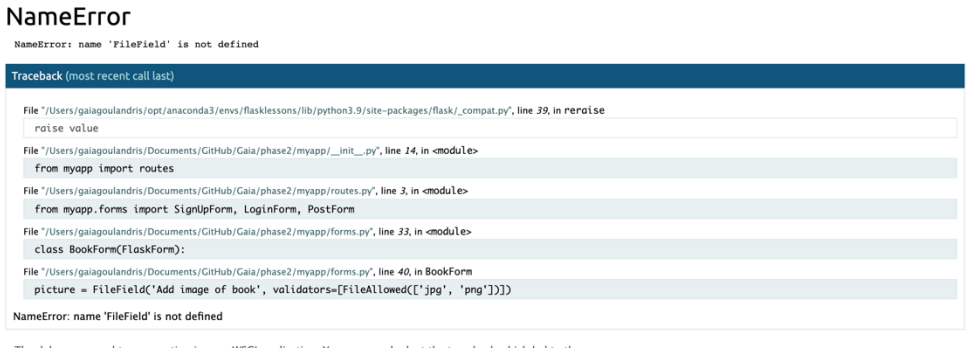
Evidence of development:

I have coded in my button for easy access to selling books from the homepage for a logged in user, and made sure that if someone is not logged on and tries to access the sell page through the navbar, they get redirected to the login page and an error message is shown asking them to log in. The button does not show up for logged out users





I came into the error below whilst trying to create my form for posting a new book for sale, but I realised that it was because I had forgotten to import 'FileField' at the start of my code, so I quickly fixed this error.




```
from wtforms import StringField, PasswordField, SubmitField, BooleanField, IntegerField, SelectField, FileField
```

Step 1: created a form to enter the book details

```
class BookForm(FlaskForm):
    title = StringField('Title',
                        validators =[DataRequired(), Length(min=2, max=100)])
    price = IntegerField('Price', validators=[DataRequired()])
    genre = SelectField('Genre', choices=['Fiction', 'Non-fiction', 'Fantasy', 'Sci-fi', 'Mystery', 'Thriller', '
    style = SelectField('Hardback or paperback?', choices=['Hardback', 'Paperback'], validators=[DataRequired()])
    picture = StringField('Picture')
    submit = SubmitField('Post for sale')
```

Step 2: created a new template called post_new_book.html where I displayed my form for the user

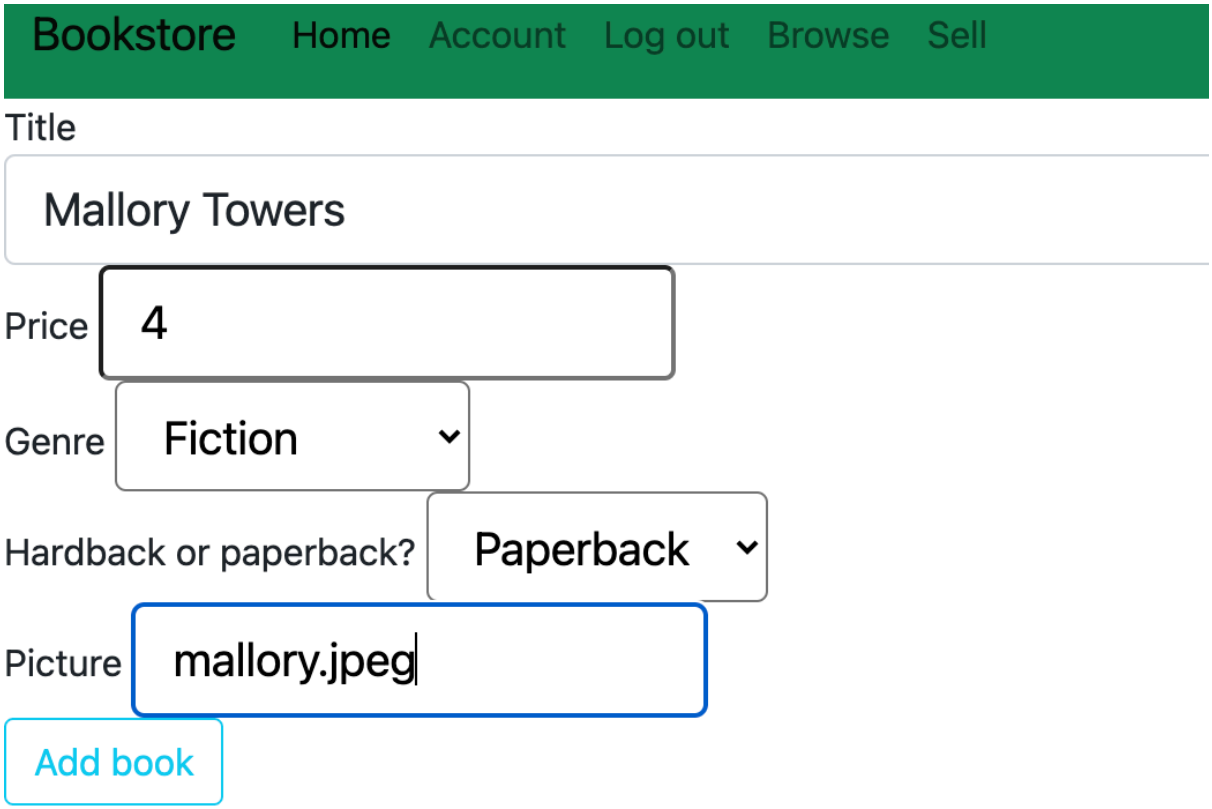
```
{% extends 'base.html' %}

{% block content %}
<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <div class="form-group">
                        {{ form.title.label(class="form-control-label") }}
                        {{ form.title(class="form-control form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.price.label(class="form-control-label") }}
                    {{ form.price(class="form-price form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.genre.label(class="form-control-label") }}
                    {{ form.genre(class="form-genre form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.style.label(class="form-control-label") }}
                    {{ form.style(class="form-style form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.picture.label(class="form-control-label") }}
                    {{ form.picture(class="form-picture form-control-lg") }}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
        </form>
    </div>
{% endblock %}
```

Step 3: created a new function in routes.py to render this new template and update to database

```
@app.route('/PostNewBook', methods=['GET', 'POST'])
@login_required
def new_book():
    form = BookForm()
    if form.validate_on_submit():
        flash('Your book has been created!', 'success')
        book = Book(title=form.title.data, price=form.price.data, genre=form.genre.data, style=form.style
        db.session.add(book)
        db.session.commit()
        return redirect(url_for('sell'))
    return render_template('post_new_book.html', title='Post Book', form=form)
```

Step 4: I then tested this

```
>>> from myapp.models import Book
>>> result = Book.query.all()
>>> result
[Book('Harry Potter and the Philosopher's Stone', '3', 'Fiction', 'Paperback', '
harry.jpeg')]
```



From testing, I identified the following issues:

1. Author is missing (need to drop table and recreate again)
2. Tidy up URLs 'sell' and 'PostNewBook' – they are different
3. User needs to be able to upload an actual picture rather than just type in a jpeg
4. Move hardback/paperback to booksforsale (makes more sense to avoid double versions of a book in books table)

As I moved on to coding the 'booksforsale' part of my code, whereby the user has to enter the other relevant details of the book that change with each book submission, I ran into a few problems, the main ones being the foreign key book_id. I struggled with how to pass it to my booksforsale table so that I could commit it and then refer to the books table through the book_id later on. I kept getting this error message:

TypeError

```
TypeError: booksforsale() missing 1 required positional argument: 'book_id'
```

However, after some logical thinking and problem-solving as well as backtracking, I figured out what needed to be added to make this finally work (as shown in red boxes below). I also checked my database in terminal to make sure my test book had indeed been committed.

```python
@app.route('/sell', methods=['GET', 'POST'])
@login_required
def sell():
    if current_user.is_authenticated:
        form = BookForm()
        if form.validate_on_submit():
            flash('Please enter next details', 'success')
            book = Book(title=form.title.data, author = form.author.data, genre=form.genre.data, picture=form.picture.data)
            db.session.add(book)
            db.session.commit()
            print(book.id)
            return redirect(url_for('booksforsale', selling_book_id = book.id))
        return render_template('sell.html', title='Post Book', form=form)
    else:
            flash('Please login to sell books.', 'danger')
            form = LoginForm()
            return render_template('sell.html', title='sell', form=form)

@app.route('/booksforsale/<int:selling_book_id>', methods=['GET', 'POST'])
@login_required
def booksforsale(selling_book_id):
        form = BooksForSaleForm()
        if form.validate_on_submit():
            flash('Your book has been created!', 'success')
            bookforsale = BooksForSale(price=form.price.data,
                status = True, style=form.style.data,
                    condition=form.condition.data, seller_id = current_user.id, book_id = selling_book_id)
            db.session.add(bookforsale)
            db.session.commit()
            return redirect(url_for('index'))
        return render_template('booksforsale.html', title='Post book', form=form)
```

```
>>> result = BooksForSale.query.all()
>>> result
[BooksForSale('2', '1', 'Brand new', '1', 'None', '1'), BooksForSale('3', '1', '
Brand new', '1', 'None', '8')]
```

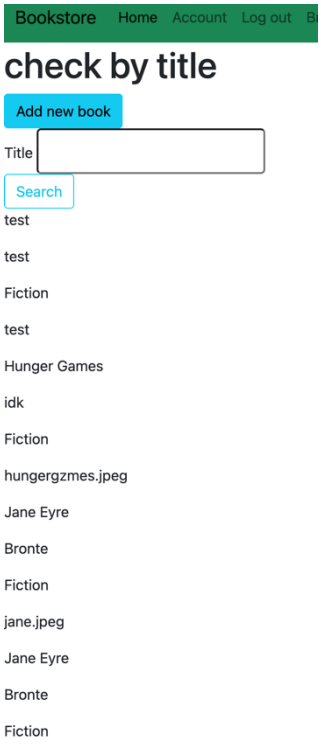
<u>Check by title documentation:</u>

Next steps:

- Form where they enter book title and submit button
- In routes.py, I render that form, and query the database looking for books that match that title
- Display those books to the user on the same template (user can select book or create the book themselves (by taking them to PostnewBook html) if no results come up
- New template booksforsale

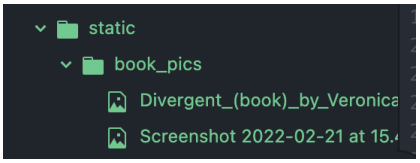
```python
class CheckByTitleForm(FlaskForm):
    title = StringField('Title')
    submit = SubmitField('Search')
```

Made new form with submit field



Shows my template for checking if the book the user is trying to sell exists. It is also displaying existing books underneath, and there should eventually be a select button under each book to select the book if that is the book the user is trying to sell. Currently, it looks like a list of writing rather than a book, so I need css and my picture functioning to make it look more like an actual book being displayed.

Managed to get my check by title page to display book with its picture. The picture for the book is stored in a folder in static.



Now my picture is working and my check for title page is not yet functioning but has all the parts it needs to work before I get the check for title search bar to work. Before I do that though, I need to change my website to make sure that when a user clicks the sell button, it takes them to the books page for checking for a title before bringing them to the sell page, and if they don't find the book they are looking for, they can click the button 'add new book' to take them to the sell page. If they do find the book they are looking for, they should be able to press a 'select' button under the book that will take them to the booksforsale page to enter price, condition and whether the book is hardback or paperback.

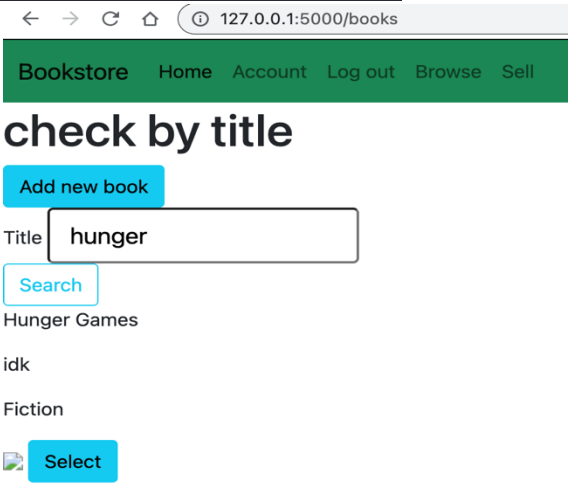
Steps:

1. Change base.html to make the sell tab on the navbar redirect to books page rather than sell page
2. Change button on index.html to redirect to books page rather than sell page

I have now fixed the search bar on my books page, and when the user searches for a book by its title, it displays the books in the database that have that title. It also allows for spelling errors or capital/non capital letters as I used filter like rather than filter by:

```
def books():
    form = CheckByTitleForm()
    books = Book.query.all()
    if form.validate_on_submit():
        search_title = form.title.data
        search = "%{}%".format(search_title)
        books = Book.query.filter(Book.title.like(search)).all()
```

This screenshot shows how the search bar works, and that it displays the books by title according to the search bar when the search button is pressed. Also, once the select button for the book is pressed, it takes the user straight to the books for sale page, and all the information for the book is linked to the booksforsale information through the book_id foreign key. I tested this by checking in terminal that they had the same book id to make sure it was working correctly.



Phase 2 review:

My flow diagram for selling a book has now been completed, and I can move on to phase 3, which will be allowing users to buy a book. However, before I do so I will review phase 2.

Testing:
I made sure to test as I was developing so examples of testing and fixing failed tests can be found in my 'evidence of development' section.

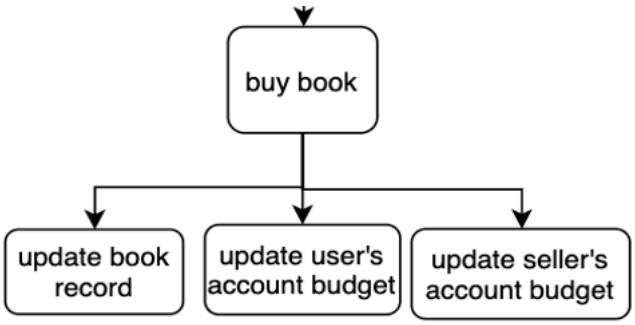
Success criteria I supported:
10 – 'a book can be posted to sell'
11 – 'picture of book is displayed for book posted for sale'
12 – 'previously posted books can be searched when posting a new book to see if it already exists'
13 – 'details of book already existing in database can be loaded into book being posted to sell'

What I did not complete:
This phase proved to be more difficult than I initially expected, and ended up being more about the functionality of the code rather than the usability of my project. I did not have the time to display the 6 most recently posted books in my homepage as I had initially hoped to do. Although this isn't a necessary part of my project and more so something that would make it more user-friendly, it is something that I would have definitely included if I had more time.

Also, I did not manage to include validation for the book price, which is quite an important feature I missed out. This is definitely one of the first things I would implement in my project if I had more time.

In conclusion:
This phase definitely didn't turn out as I expected it to. At the start it was very much focused on the homepage, how it would look, and what would be displayed on it. I clearly was slightly too ambitious in terms of the look of my project and how everything would be displayed and I overlooked how challenging the functionality of my project would prove to be. However, this phase accomplished a lot, even if it may not have been exactly what I had set out to do at the start of this phase.

Phase 3:



In this design phase, I will be working on the 'buy book' part of my top-down design. This should be quite simple, but one thing that I can't forget is that once a book is bought, it can no longer be displayed.

Development steps

Create buy page that displays all books in BooksForSale if their status is false:

- New form called buy in forms.py
- New template called buy.html
- New function in routes.py
    - Function will change buyer and seller credit and will change the book status to False so that it will no longer be displayed

phase in pseudocode

```
user goes to buy page
if booksforsale status = True:
        book is displayed
else:
        book not displayed
endif
buyer clicks buy button on book
query database for buyer credit and book price
if buyer credit is more than book price:
        add book price to seller credit
        deduct book price from buyer credit
        set buyer_id to current user
        change booksforsale status to false
else:
        display error message to buyer saying not enough credit
endif
```

Usability features to be included in the solution:

- Flash messages
    - Flash messages are important for this phase because this phase involves credit, and users may not have enough credit to buy a book. If they don't have enough credit, the user should not be able to buy the book provided the code is working correctly, in which case they should be given a message telling them they have insufficient funds because without this, it could be confusing to the user as to why the transaction didn't work. A flash message should also be given for a successful purchase
    - The flash messages are also coloured according to whether or not they are good or bad. For example, if the user doesn't have enough credit, their flash message will be red as opposed to green if they do have enough credit
- Cancel purchase button
    - This button is important in case the user changes their mind last minute
- Buy book needed to be pressed twice
    - The user will need to press 'buy' twice – once in buy.html where they select a book to buy and it redirects them to buybook.html, and again when they are confirming their purchase – this double authentication ensures that the user will not accidentally buy a book, and makes the buying process more user-friendly

Validation:

Validation is extremely important in this phase since it is all about a book being purchased and therefore, the buyer's and seller's credit being altered.

In the buybook route, there is an if statement saying that if the user's credit after buying the book will be less than 0, then the book cannot be bought and the user will be redirected to the home page. The code works out whether or not the user has enough credit using the variables mentioned below.

Key variables:

These are the main variables that the code will use to work out the buyer's credit and the seller's credit:

SellerUser – uses a query to set the variable as the seller of the book the current user wants to buy. This is done so that the seller can be referred to as sellerUser and their credit can be altered once a book has been sold

New_seller_credit – an integer variable that holds the new credit of the book seller if the book is to be sold

New_credit – an integer variable that holds the new credit of the current user if they were to buy the book. The variable is used to check if the user has enough credits to buy the bok, because if this new credit is less than 0, they can't buy the book.

Documentation:

- Upon coding my buy route, I realised that I can instead display only the books that haven't yet been bought by filtering by the books that don't have a buyer id, rather than having a status for my book at all. This means I can delete status from my booksforsale table and display the books that haven't been bought in a simpler way

Buy book documentation – changing user credit and setting buyer id

I managed to display the books on the buy page according to whether or not they had a buyer id. If they did not have a buyer id, they are displayed seeing as those boks have not yet been sold. This wasn't too difficult to do, as it was quite similar to how I displayed the books for the selling phase of my project, but there were a few more complications with displaying the books for buying, because it meant that the books table and the BooksForSale table in my database had to be functional via the foreign keys, and I needed to figure out how to properly link them. I did this as seen below in the screenshots. As you can see below, in my books query, it filters by the buyer_id=None, and the second books line is what allows me to join my Books and BooksForSale table, allowing me to display fields on a posted book from either table as seen in my code for buy.html, whereby I use book.Book or book.BooksForSale. I also added a button under each book to say buy, but the functionality has not yet been coded in in the screenshots below, which is why the button redirects us to the same page when pressed.

```python
@app.route('/buy', methods=['GET', 'POST'])
def buy():
    books = BooksForSale.query.filter_by(buyer_id=None).all()
    books = db.session.query(BooksForSale, Book).join(Book).all()
    return render_template('buy.html', title='Buy', books=books)
```

```
<h1> Buy a book </h1>

<div class="row">
    {% for book in books %}
          <div class="col-md-3">
              <p class="article-content">{{ book.Book.title }}</p>
              <p class="article-content">{{ book.Book.author }}</p>
              <p class="article-content">{{ book.Book.genre }}</p>
              <p class="article-content">{{ book.BooksForSale.price }} Credits</p>
              <p class="article-content">{{ book.BooksForSale.style }}</p>
              <p class="article-content">{{ book.BooksForSale.condition }}</p>
              <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">
              <a class="btn btn-info" href="{{url_for('buy', buying_book_id=book.BooksForSale.id)}}">Buy</a>
          </div>

        {%  if loop.index is divisibleby 3 and not loop.last %}
        </div> <br> <div class="row">
          {% endif %}

    {% endfor %}
</div>
```

Next step – pressing the 'buy' button on a book

For this step, I created a button (as seen in buy.html above) that will run a function called buybook in routes.py, as well as a form with a submit button called 'confirm purchase', making sure not to forget to import the form name at the top of routes.py.

```
class BuyBookForm(FlaskForm):
    submit = SubmitField('Confirm Purchase')
```

I also coded this route, that ensures the user has enough credit before allowing the purchase to be done, as well as updating the buyer id and the buyer's credit. The flash messages are important for usability and user friendliness.

```
@app.route('/buybook/<int:buying_book_id>', methods=['GET', 'POST'])
@login_required
def buybook(buying_book_id):
        form = BuyBookForm()
        if request.method=='POST':
            book = BooksForSale.query.filter_by(BooksForSale.id==buying_book_id).first()
            new_credit = current_user.credit - book.price
            if new_credit < 0:
                flash('Insufficient credit', 'danger')
                return redirect(url_for('index'))
            else:
                BooksForSale.buyer_id = current_user.id
                current_user.credit = new_credit
                db.session.commit()
                flash('Purchase successful', 'success')
                return redirect(url_for('index'))
        book = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.id==buying_book_id).first()
        return render_template('buybook.html', title='Buy book', book=book, credit=current_user.credit, form=form)
```

And coded this in my buybook.html page, which displays the book that the user selected to buy, and has a button for them to cancel their purchase and will redirect them to the home page (which is an important feature for usability), and also has a button to confirm purchase. Upon clicking the confirm purchase button, the buyer credit will be updated if they have enough, and the buyer id for the book will be set to the current user. Subsequently, this will also mean that the book will stop being displayed on the buy page seeing as there is now a value in the buyer id. Without this, there would be big issues in the code as people could end up buying an already sold book.

```
{% extends 'base.html' %}

{% block content %}

<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <h1> Buy this book </h1>


            <p class="article-content">{{ book.Book.title }}</p>
            <p class="article-content">{{ book.Book.author }}</p>
            <p class="article-content">{{ book.Book.genre }}</p>
            <p class="article-content">{{ book.BooksForSale.price }} Credits</p>
            <p class="article-content">{{ book.Book.picture }}</p>

            <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">

            <p class="article-content">You have {{ credit }} credits</p>


            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
            <br>
            <br>

            <a class="btn btn-info" href="{{url_for('index')}}">Cancel</a>

        </form>
    </div>
{% endblock %}
```

However, when testing to make sure everything functioned correctly, I got an error :

```
File "/Users/gaiagoulandris/Documents/GitHub/Gaia/phase3/myapp/routes.py", line 125, in buybook
  book = BooksForSale.query.filter_by(BooksForSale.id==buying_book_id).first()
TypeError: filter_by() takes 1 positional argument but 2 were given
```

This error is about the query in my buybook route where I am making sure that the book the user selected is the one actually being sold.

I fixed this by changing filter_by to filter, seeing as I didn't realise the distinction between them – filter is used for more powerful/complex queries, and filter_by is used for kwargs. Nonetheless it was a quick and easy fix:

```
book = BooksForSale.query.filter(BooksForSale.id==buying_book_id).first()
```

I also realised a major flaw in my code, in that I forgot to change the seller's credit when their book is bought, meaning they wouldn't gain any credits from selling a book.
I struggled with linking the user table and the book table, and how to access and change someone's credit that wasn't the current user, but I eventually figure out how to do it and update my code accordingly:

```
@app.route('/buybook/<int:buying_book_id>', methods=['GET', 'POST'])
@login_required
def buybook(buying_book_id):
        form = BuyBookForm()
        if request.method=='POST':
            book = BooksForSale.query.filter(BooksForSale.id==buying_book_id).first()
            sellerUser = User.query.filter(book.seller_id==User.id).first()
            new_seller_credit = sellerUser.credit + book.price
            new_credit = current_user.credit - book.price
            if new_credit < 0:
                flash('Insufficient credit', 'danger')
                return redirect(url_for('index'))
            else:
                BooksForSale.buyer_id = current_user.id
                current_user.credit = new_credit
                sellerUser.credit = new_seller_credit
                db.session.commit()
                flash('Purchase successful', 'success')
                return redirect(url_for('index'))
        book = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.id==buying_book_id).first()
        return render_template('buybook.html', title='Buy book', book=book, credit=current_user.credit, form=form)
```

I also used prints to test in terminal that it was functioning correctly. I checked that the seller's id and the later assigned seller id for the book were the same, and that the seller's credit before and after the sale changed appropriately:

```
print(sellerUser.id)
print(sellerUser.credit)
# seller_credit = sellerUser.credit
new_seller_credit = sellerUser.credit + book.price
new_credit = current_user.credit - book.price
if new_credit < 0:
    flash('Insufficient credit', 'danger')
    return redirect(url_for('index'))
else:
    BooksForSale.buyer_id = current_user.id
    current_user.credit = new_credit
    sellerUser.credit = new_seller_credit
    db.session.commit()
    flash('Purchase successful', 'success')
    print(sellerUser.credit)
    print(book.seller_id)
```

```
127.0.0.1 - - [09/Mar/2022 11:28:28] "GET /buybook/7 HTTP/
1
7
10
1
```

However, I did notice when I was testing my project's buy feature that the books could be bought multiple times and would not stop being displayed on buy.html after being bought. I thought I had solved this with the line in my buy route that filtered by buyer_id=None but this clearly isn't functioning correctly.

```
books = BooksForSale.query.filter_by(buyer_id=None).all()
```

I then realised that I had two books queries, and I had to combine them for them to work properly. Upon checking after implementing this new line of code, my books stop being displayed on buy.html after being bought, and the buy page is now fully functioning.

```
books = db.session.query(BooksForSale, Book).filter(buyer_id==None).join(Book.id==BooksforSale.id).all()
```

Phase 3 review:

I completed what the top-down diagram asked for this phase, and users are now able to purchase books. Their credits and the seller's credits change accordingly, and validation that checks the user has sufficient credit to purchase the book is also working correctly.

Testing:
I tested each part of my code as I went along, which can be seen in the evidence of development section.

Because this phase was heavily integer-based and therefore required flawless accuracy, the use of prints in my code whilst testing proved to be of great help because it allowed me to check that all the values being calculated and assigned to variables were of the correct value.

Although testing in this phase mainly concerned testing the functionality of my code, I also made sure to test the usability by checking that the font was readable and the function of the code was easily understood by the user.

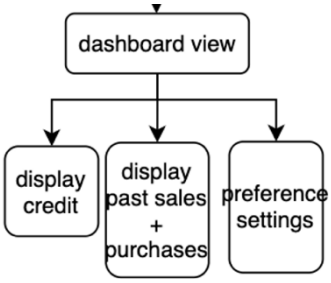
As part of usability, I also tested the flash messages by setting the user's credit to below what they needed. The result I expected from this was a flash message saying 'insufficient credit' and for the user to be redirected to the home page. This test was successful.

Success criteria I supported:
14 – 'credit is added to user's total credit once a book is sold'
15 – 'book can be bought by user'
16 – 'credit is deducted from user's total credit once a book is purchased'
21 – 'page includes readable text and usability features such as buttons and a navbar'

In conclusion:
This phase turned out to be efficient and successful, and I was able to complete everything I set out to complete at the start of it. I was even able to notice new things such as the fact that the status filed in my BooksForSale table was unnecessary.

Phase 4:

In this phase, I will be working on the dashboard/account part of my project. All of this will be displayed on the account page.



Development steps:

STEP 1

1. New function in routes.py that queries relevant tables on database
2. Display user credit on account.html
3. Display past sales on account.html
4. Display total credits spent on account.html under past purchases
5. Display past purchases on account.html
6. Display total credits made on account.html under past sales

I unfortunately did not have time to complete step 2, which is the preference settings part of my dashboard view, but I have outlined how I would have completed it if I had more time to do so.

STEP 2:

1. Button on account.html that leads to preference page
2. New form that asks user for:
    - Favourite genre
    - Preference of hardback/paperback
    - Favourite author
3. New table in models.py that commits the user's preferences to the database
4. New function in routes.py that queries books according to user's preferences
5. Button on buy.html that says 'display books by preference' or 'display all books'
    - User can choose whether they would like to see all books posted or just see the books that fall under what they have selected as their preference

Usability features to be included in the solution:

This phase is the most about usability in that all its features don't affect the functionality of the code but rather benefit the user.

- Credit displayed in account page
    - This helps the user by allowing them to keep track of how many credits they have so they know if they are running low and should sell more books or if they have enough credits to buy a new book
- Displaying past sales and past purchases

- This feature benefits the user because they can keep track of their past transactions should they like to – the fact that they are also able to see their total credits spent and total credits made is a fun user-friendly feature that allows the user to see how much they have actually done since signing up

Validation:

For this phase, validation isn't so important as there isn't anything being inputted by the user or any new information. Rather, this phase is focused on the accuracy and function of the code, and testing is definitely a more important feature to carry out than validation.

Key variables:

These are the main variables that the code will use to work out total credits made and total credits spent, as well as the variables that will be needed to display the books in past sales/purchases.

Total_credits_made – set at 0, and then an iteration is run to add up the credit of all books in sales so as to set the variable to the total number of credits the user has made

Total_credits_spent – set at 0, and then an iteration is run to add up the credit of all books in purchases so as to set the variable to the total number of credits the user has spent

Purchases – a query rather than a variable, but nevertheless an imperative aspect of the code as it filters through the BooksForSale table and finds the books whereby the buyer id is the same as the current user's id, and also joins the Book, BooksForSale, and User table together so that the information linking to the book found from this query can be taken from any of these tables and displayed on account.html under past purchases

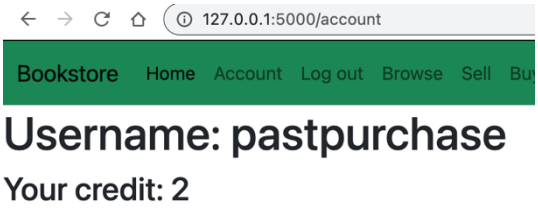
Sales - a query rather than a variable, but nevertheless an imperative aspect of the code as it filters through the BooksForSale table and finds the books whereby the seller id is the same as the current user's id, and also joins the Book, BooksForSale, and User table together so that the information linking to the book found from this query can be taken from any of these tables and displayed on account.html under past sales

Evidence of development:

I started off by displaying the user's username at the top of the account page and their credit:

account.html routes.py

```
{% extends 'base.html' %}

{% block content %}
<h1>Username: {{ current_user.username }}</h1>
<h3>Your credit: {{ current_user.credit }}</h3>

<hr>
```



This was extremely simple. I next moved on to attempting to display the current user's past purchases. I did this by adding a query to the account function in routes.py that linked the Book table and BooksForSale table, and also filtered by the books that had a buyer id equal to the current user's id, and did a very similar thing for sales:

```
@app.route('/account')
@login_required
def account():
    purchases = db.session.query(BooksForSale, Book).filter(BooksForSale.buyer_id==current_user.id).all()
    sales = db.session.query(BooksForSale, Book).filter(BooksForSale.seller_id==current_user.id).all()
    return render_template('account.html', title='Account', purchases=purchases)
```

This seemed to work at first, because it displayed all the books

But I realised that a book shows up on the user's past sales as soon as they post it, even when it hasn't been bought yet. To fix this, I added another filter to the sales query in the account function:

```
@app.route('/account')
@login_required
def account():
    purchases = db.session.query(BooksForSale, Book).filter(BooksForSale.buyer_id==current_user.id).all()
    print(purchases)
    sales = db.session.query(BooksForSale, Book).filter(BooksForSale.seller_id==current_user.id, BooksForSale.buyer_id!=None).all()
    return render_template('account.html', title='Account', purchases=purchases, sales=sales)
```

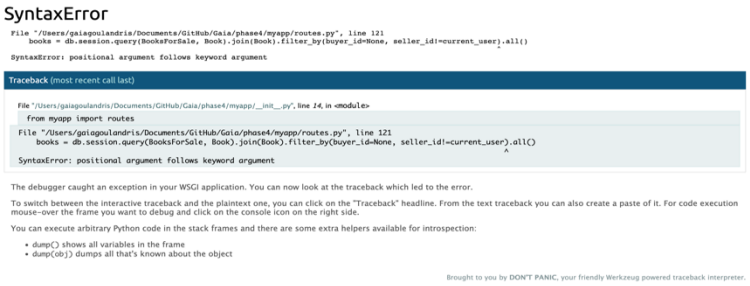
I also realised that when I created a new account and bought one book with this new account, suddenly all books showed up on the account page under past purchases. It seemed that this had to be a problem with the buy function rather than the account function, because all the purchase query was doing was showing books that's buyer id is the same as the current user's id. However, I then realised that the purchases and sales queries didn't actually join the Book and BooksForSale tables together because I forgot the join(Book) part in the query:

```
account():
purchases = db.session.query(BooksForSale, Book).join(Book).
print(purchases)
sales = db.session.query(BooksForSale, Book).join(Book).filt
```

I also noticed (from dropping and recreating my database so that the status field would officially be gone) when posting new books for sale, that a user is able to buy a book that they have posted for sale themselves. I thought this was an easy fix though, and all it required was another variable to filter by in the books query in the buy function:

```
@app.route('/buy', methods=['GET', 'POST'])
def buy():
    books = db.session.query(BooksForSale, Book).filter_by(buyer_id=None, seller_id!=current_user.id).join(Book).all()
    return render_template('buy.html', title='Buy', books=books)
```

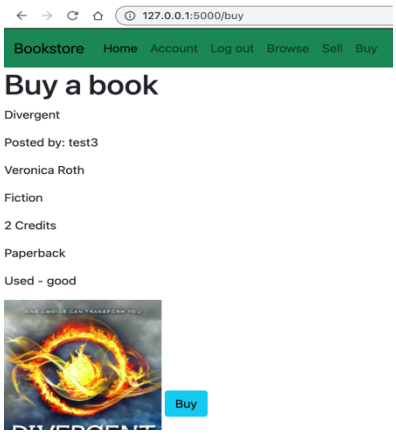
However, when I did this and tried to run my website, I got this error message:



It seemed there was a problem with the books query, where I was trying to display all the books where the buyer_id is null whilst excluding ones where the seller_id is the current user.

So I amended the query and also joined the User in the same query so that I could display the seller's username instead of just their id when a book has been posted:

```
@app.route('/buy', methods=['GET', 'POST'])
def buy():
    books = db.session.query(BooksForSale, Book, User).filter(BooksForSale.seller_id!=current_user.id, BooksForSale.buyer_id==None).join(Bo
    return render_template('buy.html', title='Buy', books=books)
```



Upon testing, my website came up with no error due to a flaw in the query, and books posted for sale now displayed the seller's username rather than their id.

Another thing I realised is that it would make sense for me to display the username of the seller on the books in past purchases, and the username of the buyer on the books in past sales. I tried to do this by adding a seller and buyer query in my account function that would link the user table and BooksForSale table so that I could reference the username of the buyer/seller and not just the id:

```
@app.route('/account')
@login_required
def account():
    purchases = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.buyer_id==current_user.id).all()
    print(purchases)
    seller = db.session.query(BooksForSale, User).join(User).filter(BooksForSale.buyer_id==User.id).first()
    sales = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.seller_id==current_user.id, BooksForSa
    buyer = db.session.query(BooksForSale, User).join(User).filter(BooksForSale.seller_id==User.id).first()
    return render_template('account.html', title='Account', purchases=purchases, sales=sales)
```

```
<p class="article-content">Sold by: {{ seller.User.username}}</p>
```

But, I ran into this error, that says I didn't specify which foreign key I want to link the BooksForSale table and the User table with for my seller query:

To fix this, I specified the foreign key I wanted to link the tables with (seller id for seller and buyer id for buyer:

```
seller = db.session.query(BooksForSale, User).join(User, BooksForSale.seller_id==User.id).filt
total_credits_spent = 0
for book in purchases:
    total_credits_spent = total_credits_spent + book.BooksForSale.price
pass
sales = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.seller_id==current
buyer = db.session.query(BooksForSale, User).join(User, BooksForSale.buyer_id==User.id).filter
```

However, on my account template, I was trying to display the username of the seller of the book (for past purchases) and the buyer of the book (for past sales), which is why I had another query called seller and buyer that tried to join the User table. However, to do this properly, I should actually join all 3 tables in the same query:

```
purchases = db.session.query(BooksForSale, Book, User).filter(BooksForSale.buyer_id==current_user.id).join(Book, BooksForSale.book_id==
# seller = db.session.query(BooksForSale, User).join(User, BooksForSale.seller_id==User.id).filter(BooksForSale.buyer_id==User.id).firs
```

This meant that I did not need to reference the seller in the account template as I had previously thought I had to, and I could get the field from the purchases database query instead:

```
{% for book in purchases %}
      <div class="col-md-3">
          <p class="article-content">{{ book.Book.title }}</p>
          <p class="article-content">Sold by: {{ book.User.username}}</p>
          <p class="article-content">{{ book.Book.author }}</p>
          <p class="article-content">{{ book.Book.genre }}</p>
          <p class="article-content">Bought for {{ book.BooksForSale.price }} Credits</p>
          <p class="article-content">{{ book.BooksForSale.style }}</p>
          <p class="article-content">{{ book.BooksForSale.condition }}</p>
          <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">

      </div>
```

I then did the same for the seller query, and for the buyer in the account template.

Adding in the total credits spent and total credits made feature for the account page proved to be simple, with no errors. All I did was create a variable for total credits made and total credits spent in the account route and added together the credits of all the books the user sold/purchased to the variable using a for loop:

```
@app.route('/account')
@login_required
def account():
    purchases = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.buyer_id==current_user.id).a
    print(purchases)
    seller = db.session.query(BooksForSale, User).join(User).filter(BooksForSale.buyer_id==User.id).first()
    total_credits_spent = 0
    for book in purchases:
        total_credits_spent = total_credits_spent + book.BooksForSale.price
    pass
    sales = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.seller_id==current_user.id, Book
    buyer = db.session.query(BooksForSale, User).join(User).filter(BooksForSale.seller_id==User.id).first()
    total_credits_made = 0
    for book in sales:
        total_credits_made = total_credits_spent + book.BooksForSale.price
    pass
    return render_template('account.html', title='Account', purchases=purchases, sales=sales, total_credits_spen
```

And to display it on account.html, I wrote this:

```
<br>
<p class="article-content">Total credits spent: {{ total_credits_spent }}</p>
<br>
<br>
```

Testing:

Some of my testing can be seen in my evidence of development, along with the errors I ran into and an explanation of how I overcame them, but I also tested according to the success criteria at the end of this phase:

Testing what	Test data	Expected result	Actual result	Success?
1 - User's username and current credit is displayed at top of account page (criteria 6 of success criteria)	Login to a test user and click on account.	Screenshot of correct username and credit	Screenshot as seen at top of p. 43	success
2 - Past purchases/sales queries working correctly (criteria 7 and 8 of success criteria)	Buy book and check past purchases	Book bought by user displayed on account.html under past purchases	All books posted for sale showed up under past purchases (p. 43)	Fail – remedial action displayed at end of p. 43-44
3 - Total credits spent is accurate	New user buys 2 test books of 2 credits each	Total credits spent should be 4 credits	Total credits spent was 4 credits	success
4 – total credits made is accurate	User posts 2 books worth 2 credits each for sale and new test user buys both books	When logging back into first test user, total credits made should be 4 credits	Total credits made was 4 credits	success

Success criteria I supported:

6 – ‘dashboard view displays user’s current credit’

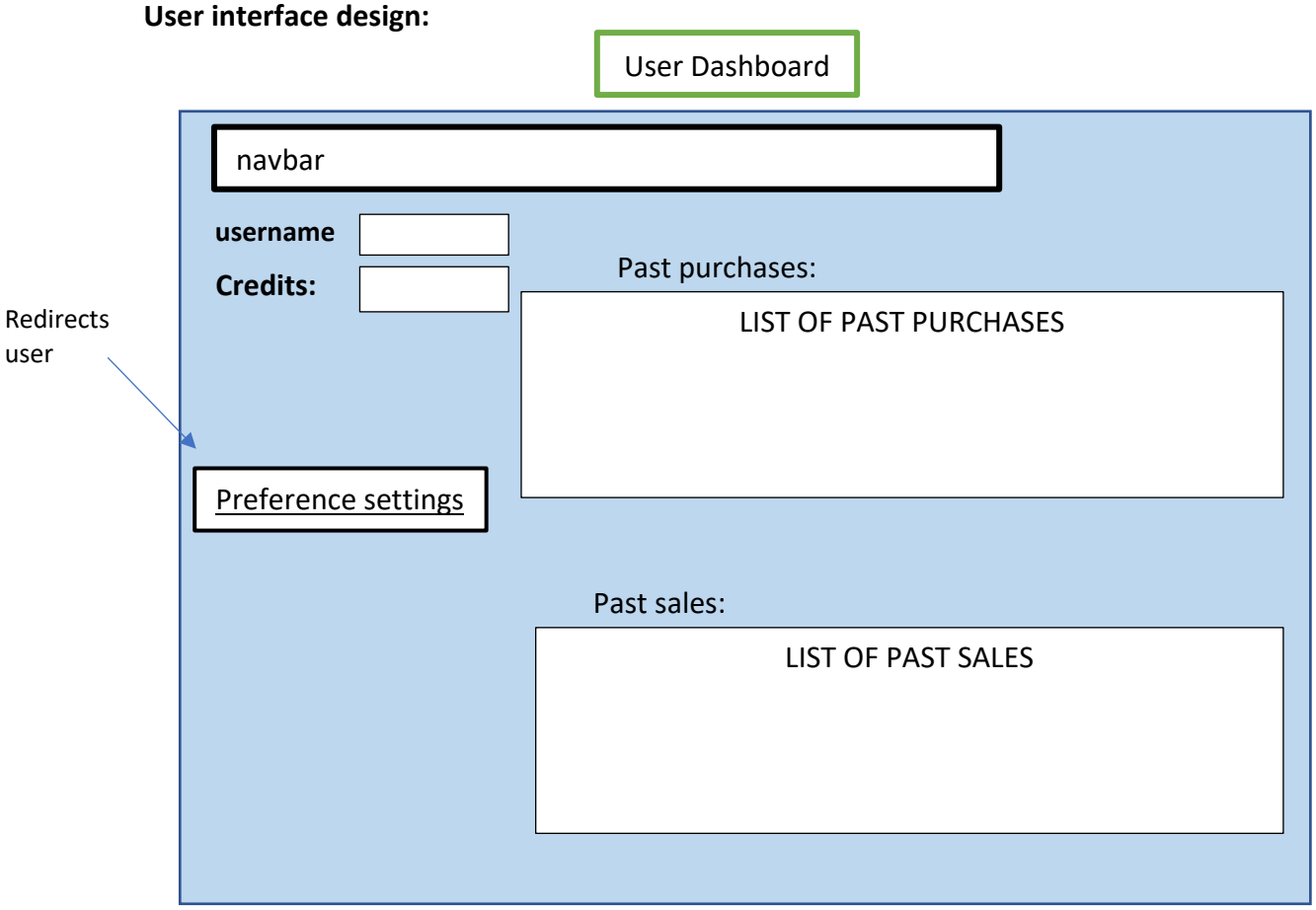
7 – ‘dashboard view accurately displays user’s past sales’

8 – ‘dashboard view accurately displays users past purchases’

9 – ‘suggestion system is working based on user’s preference settings’

- This was not explicitly tested in my testing table because it does not exist, but it therefore counts as a failed test. I did explain how I would implement it at the start of phase 4 though, which is essentially the solution to the ‘test’

User interface design:



Phase 4 review:

I completed most of what the top-down diagram asked for this phase (‘display credit’ and ‘display past sales + purchases’), and users are now able to see their current credit and their past activity on the account page. However, I did not manage to implement step 2 in my development steps, which was the preference settings part of this phase. As I mentioned above, I didn’t have the time to finish, but had I had more time, I have shown in my development steps (p. 41) how I would implement it.

Evaluation:

This section will be an evaluation of my whole project. I will analyse what went well and what could have gone better as well as evaluating how well met the success criteria has been met. I will also consider maintenance and limitations of my solution, as well as discuss how the program would be developed if I hadn't been so restricted by time.

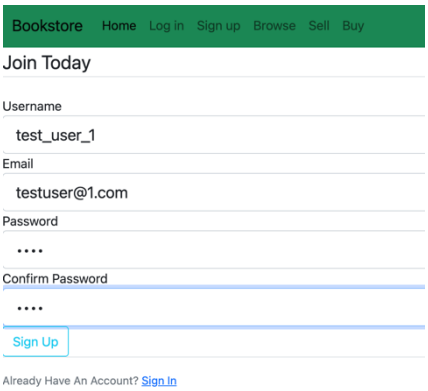
Testing to inform evaluation

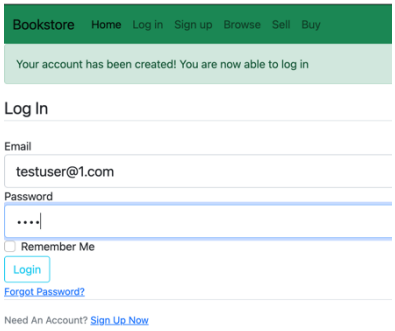
This section displays the post development testing I have done to test for **function** and **robustness** of my solution:

Test no.	What is being tested	Input data/actions	Expected outcomes	Actual outcome	reference
1	Creation of a user is functioning (1 on success criteria)	Username and password of new user signing up	Details are saved and user can now log in/log out	As expected	Test 1 evidence below
2	Login with password is functioning (2 on success criteria)	Log in to test_user_1 profile	User is successfully logged in and correct username displayed on account page	As expected	Test 2 evidence below, pg 23
3	Database is queried to make sure user is valid (3 on success criteria)	Input incorrect username and password	Displays error message and doesn't allow log in	As expected	Pg. 23
4	Navbar is functioning	See reference	See reference	As expected	Pg. 23
5	Post book for sale button only shown If user is logged in	See reference	Screenshot of homepage for logged in user showing button, and logged out user showing no button	As expected	Pg. 28
6	Books can be added/removed to booklist (4 and 5 on success criteria)	Book added to/removed booklist by clicking button	Book shows up in booklist page when button is pressed to add it, and doesn't show up anymore when button Is pressed to remove it	Failed – no outcome because didn't have enough time to implement booklist	Pg. 28
7	Dashboard view testing (6, 7, 8 on success criteria)	See reference	See reference	As expected	Pg. 46
8	Post book for sale testing (14, 15, 16 on success criteria)	See reference	See reference	As expected	Pg. 37-40

9	Picture of book is displayed for book posted for sale (11 on success criteria)	User inputs picture when entering book details	Book shows up on buy page for another user with the picture displayed	As expected	Test 9 evidence below
10	Books page has a functional search bar (19 on success criteria)	User inputs title of existing posted book into search bar	Book shows up under searched result	As expected	Test 10/11 evidence below
11	Books search bar still displays results despite spelling errors or capitalisation (12 on success criteria)	User inputs title of previously posted book, but with a spelling error	Book shows up under searched results, despite spelling error	Not as expected. Works for capitalisation and not fully finished words, but does not display for spelling errors	Test 10/11 evidence below
12	Suggestion system is working based on user's preference settings (9 on success criteria)	See reference	See reference	Failed	Pg. 47
13	A book can be posted to sell and details of book already existed in database can be loaded into book being posted to sell (10 and 13 on success criteria)	See reference	See reference	As expected	Pg 33
14	New activity is stored after logging out	User buys book and logs out, then logs back in	Bought book should be in past sales before and after logging out	As expected	

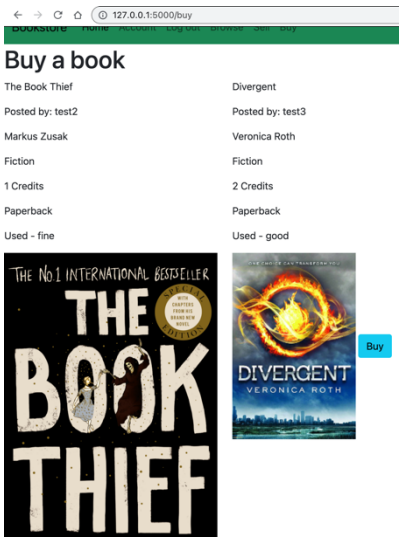
Test 1 evidence:



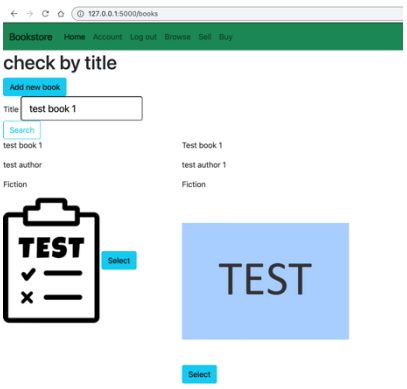


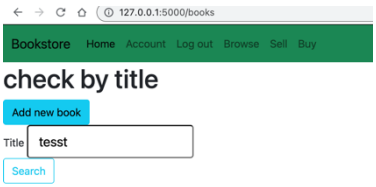
Test 2 evidence:

Test 9 evidence:



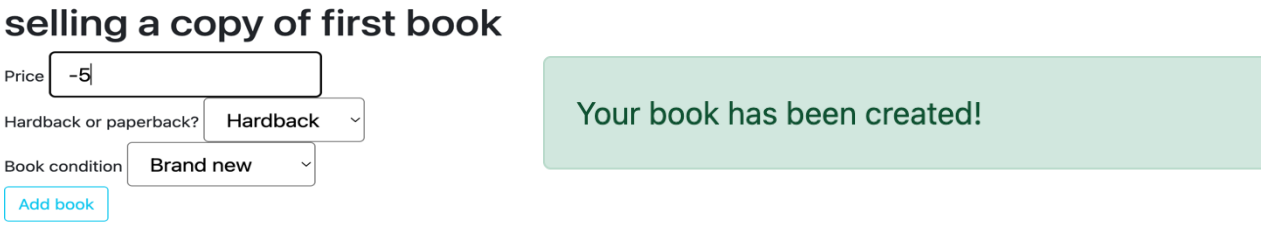
Test 10/11 evidence:



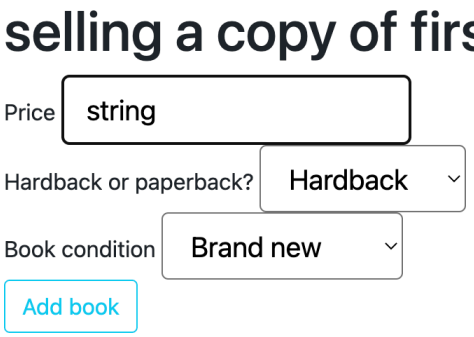


Destructive testing:

Test 3 in the test table is an example of destructive testing, because it is inputting knowingly wrong information in attempts to 'break' the code. To destructively test further, I inputted negative values into price when posting a book, and also inputted a string rather than an integer which is the data type it is supposed to accept:



My test failed for inputting a negative integer, because it was accepted and the book posted. I cannot change this now, but I discussed how I would change this if I had more time previously (pg 27).

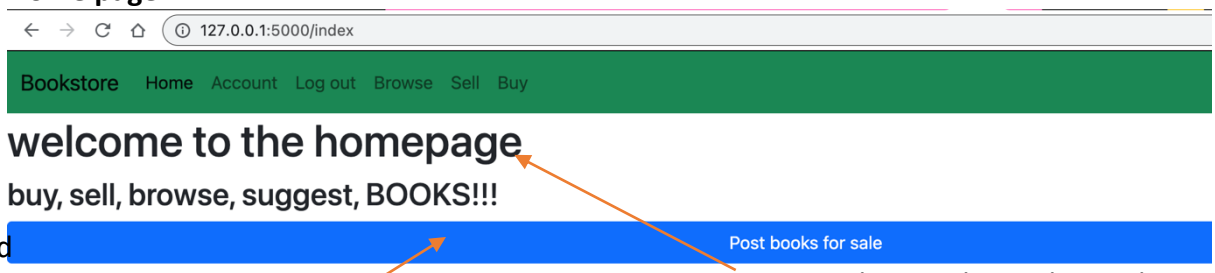


As for inputting a string, the book did not post, however, no flash message was shown, it just did nothing when the 'add book' button was pressed. This is very confusing to the user, and if I had more time, I would make sure that upon validation of credit, flash messages would be shown if an incorrect input was entered.

Usability testing:

To check my program is as user-friendly, easy to use, effective, and easy to understand as possible, I have provided annotated evidence for usability testing:

Home page:

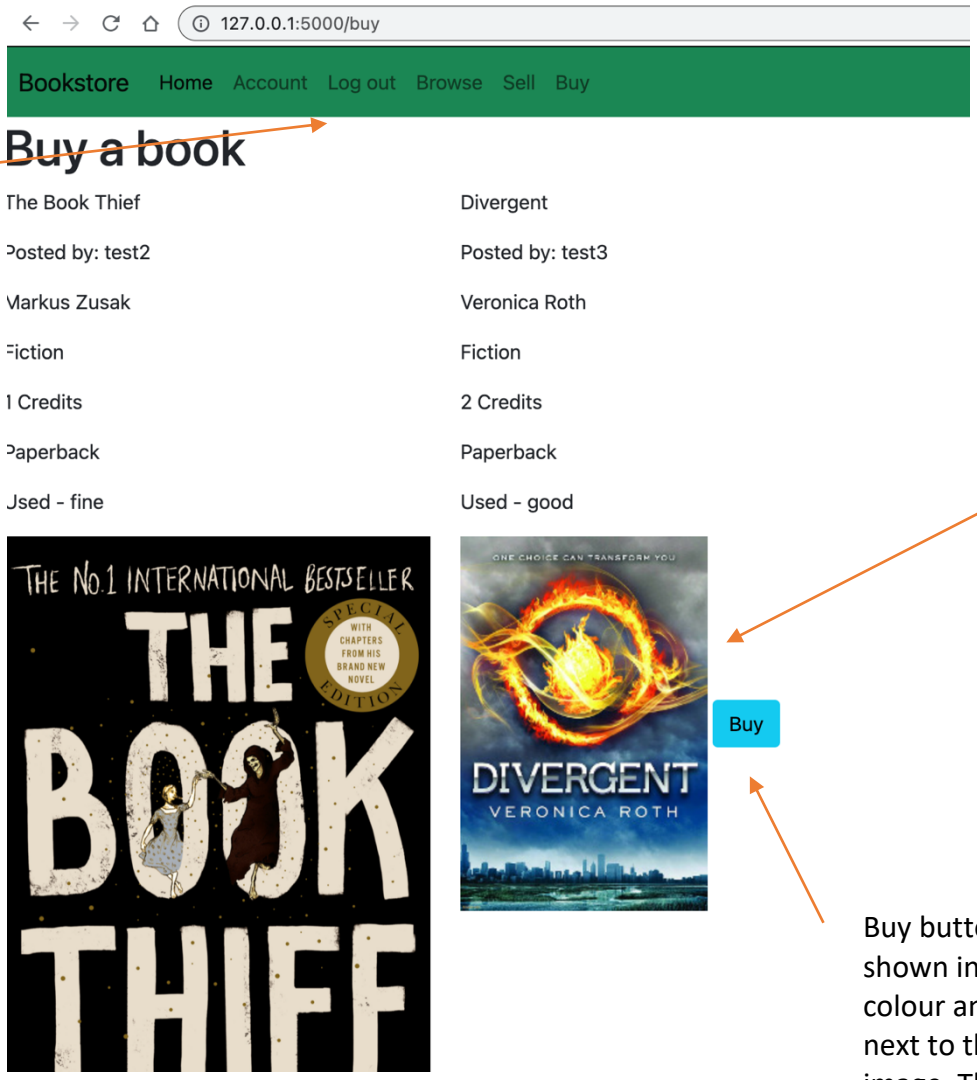


Text is large, easy to read, and there are no spelling mistakes

Button linking to sell page is user friendly and allows for quick access to a useful feature

Writing has not been changed since beginning of phase 1 – didn't have time to implement browsing system and suggest system for books – keeping this writing in the homepage can be confusing to the user – should leave it out until these steps have been implemented

Buy page:



Navbar is on every page and in the same format which it makes it easy and comfortable for the user to adjust to

Pictures of books displayed are helpful for the user to recognise a book and are pleasing to the eye. However, all the pictures are not the same size which makes the layout less even and therefore less visually aesthetic

Buy button is clearly shown in different colour and is right next to the book image. This makes it easily accessible to the user

Evaluation of solution:

Success criteria review:

I created my success criteria during the analysis stage at the very beginning of this project. At the time, I devised it in attempts of highlighting the key aspects of my solution and what would need to be completed in order to have a successful solution at the end of this project. However, at the time I didn't have much of an idea of how challenging some of the aspects of my solution would be and how time-consuming they would end up being. I underestimated the time it would take for me to solve problems in my code, and this resulted in me not having an accurate time scope.

Therefore, I was much too ambitious with my project within the analysis section, and I quickly realised I would have to scale back and cut out some features I had initially wanted to include if I was going to have a functional and manageable project by the end of this. Because of this, my project partially met the success criteria, with some things having to be abandoned, and others being developed further than I had initially thought they would be.

1. **Creation of a user is functioning**

In this case, I have met the success criteria in that a user is able to create an account via the signup form I created, and their inputted information is committed to the database via the signup route.

Tests: 1 – the test showed the user inputting details into the sign up form, and the flash message confirming their account creation has been successful once the user hass pressed the 'sign up' button

2. **Login with password is functioning**

This case has met the success criteria by allowing the user to input their login details into the login form, and then being logged in once submitting their login details

Tests: 2 – shows that once a user has successfully entered the correct details, they are logged on to their account and they have not been forgotten

3. **Database is queried to make sure user is valid**

My project met this success criteria because the user's login inputs are validated by comparing them against the data in the database

Tests: 3 – test shows that incorrectly inputted user details means the user is unable to log in and access their account

4. **Books can be added to booklist and**
5. **Books can be removed from booklist**

My project did not meet this part of the success criteria because the time restraint meant I was not able to complete it. If I had more time, this criteria would have been addressed by having a booklist table in the database, and a button for adding and removing a book posted for sale to your booklist. If the button was clicked under a book, the book id would be

committed to the user's booklist, and all the books added to booklist could be displayed on a booklist page that you can access from your account page.

Tests: 6 – shows how the testing would be if this feature could have been implemented

6. **Dashboard view displays user's current credit and**
7. **Dashboard view accurately displays user's past sales and**
8. **Dashboard view accurately displays user's past purchases**

These 3 aspects of the user's account have all been successfully met, and a user can view these details on account.html.

Tests: 7 – had some issues with queries and linking tables with foreign keys, but eventually figured it out

9. **Suggestion system is working based on user's preference settings**

This aspect of the user's account has not been successfully met, but I have shown how I would have successfully met it if I had more time on page 41

Tests: 12

10. **A book can be posted to sell**

This success criteria was partially met because my code does allow for books to be posted to sell, however, the success criteria specifies that I should evidence this by providing a screenshot on the test user's profile, and once a book is posted to sell, it is displayed on buy.html, not on a user's profile.

Tests: 13

11. **Picture of a book is displayed for book posted for sale**

This success criteria was fully met, and the book picture displays not only when looking for a book to buy on the book page, but also on the account page for past sales/purchases, and even on the books.html page where the user is looking to see if an existing book is the one they are trying to sell.

Tests: 9

12. **Books search bar still displays results despite spelling errors or capitalisation**

This success criteria was partially met, because as seen from the test evidence, results were displayed despite capitalisation but not despite spelling errors.

Tests: 11

13. **Details of book already existing in database can be loaded into new book being posted to sell**

This success criteria was fully met, and took longer to complete than I initially expected.

Tests: 13

14. Credit is added to user's total credit once a book is sold and
15. Book can be bought by user and
16. Credit is deducted from user's total credit once a book is purchased and
17. Book can be sold by user

These success criteria were met successfully, as the referenced tests show the user credit changing on total_credits_made and total_credits_spent accordingly, and the overall function of a book being able to be sold by a user is completed because all the decomposed steps of the overall step have been completed.

Tests: 8

18. New activity is stored after logging out

This success criteria was met, which is important since this is a vital criteria of the final solution, and many of the other success criteria would not be met without this one.

Tests: 14 – test demonstrates that any actions performed by the user are stored and remembered

19. Books page has a functional search bar

This success criteria was successfully met, and the steps to meet it could have been used to implement more features that I didn't have time for such as a search bar on the homepage that searched through books that can be bought by the user.

Tests: 10 – evidence shows search bar can be used to check for book by its title

20. Filtering system during browsing is functioning

This success criteria was not met as I didn't have enough time for it. This criteria is both a usability feature and a functionality feature, so it would have been a good one to have completed, but if it were to be implemented, it would have been done by using a query similar to the query used by the search bar in books.html, and the queries would be used to filter by price, genre, and more.

21. Pages include readable text and usability features such as buttons and navbars and
22. Navbar is functioning

I would say that this success criteria was partially met, because the navbar functions and all text is reasonable. At first glance, this success criteria was fully met, however, this criteria is focused on the usability project, and I feel as though it could have been improved if I had more time, by making layout of things such as books being displayed neater, by adding a logo, by having search by and filter by features when browsing for the book, and by being able to delete or edit books posted for sale. Overall though, usability is not bad in that my solution seems to be relatively easy to follow and use.

If I had more time:

If I had more time, I would have of course attempted to complete the unmet criteria within my success criteria. But there were also a few things I noticed as I went along the project that I had not thought of putting into the success criteria, but were still vital features of my solution:

1. **Resell feature**
   If I had more time, I would have implemented this idea by having a 'resell' button for every book under past purchases. If the user clicked the button, it would set the buyer id to be blank and set the seller id to be the current user. The problem with this is that it would no longer show up in the current user's past purchases and the previous seller's past sales, so maybe I would have to implement an array for past sales and past purchases, and append a book to the list when they are added, in which case the table in the database can be changed, but what is displayed in past sales and purchases won't change for each user.
2. **Edit/delete books posted for sale**
   As I mentioned above, this is definitely a feature I somehow forgot about when creating the success criteria, but something I would definitely include if I had more time. This is an extremely important feature, because if a user changes their mind about selling a book, or they lose the book they are trying to sell, there is no way to delete the post, so if someone buys the book and the seller can't deliver, it becomes a disaster. Therefore, by implementing the ability for the user to delete a post, this disaster is averted. Also, editing posts increases usability of my solution because if, for example, the user has set the price as high and the book therefore has not sold so the user wants to lower the price to help make it sell easier, they are not currently able to do this. But it is definitely a useful feature that I wish I had enough time to include because without it, it can lead to significant problems.

Maintainability of system:

Although I have some features that make the maintainability of the system good, there are things I could have added or implemented further to improve maintenance. For example, I annotated my code by adding a few comments - mainly at the start of a route to explain its function – in hopes that this would improve the maintainability of the system because a future developer would be able to read it and quickly understand what the code is trying to achieve and what the purpose of each part is. However, I could've improved my maintainability further by adding more comments because they are very limited and I found that even as I looked back on what I had written, it took me a while to fully grasp what the function of the code was. Next time, I should be sure to continuously add comments as I go through my iterative development cycles, keeping track of changes, functions, usability features, and more.

Other maintainability features I have thought about is having clear, well organised routes and meaningful table names, as well as attempted to have clear, different pages within my project. Although, it may be easy to get confused between buy.html and buybook.html, which have similar names for seemingly the same function, but actually complete different

things. This could be improved with comments at the start of each page to explain the difference, or by changing the name of one of them into something more distinguishable. These features will allow a future developer to quickly navigate the system, and helps avoid the mistake of changing the wrong section of code when debugging or adding something.

Limitations that could be improved in the future:

One of the limitations of my project is the lack of validation for pricing a book, which I mentioned earlier (pg 27). To improve on this in the future, I must ensure that the price entry box is validated before the sell form is allowed, by coding that a flash message should be shown saying 'price not valid' if they have entered a negative price, or 'price too high' If the user has entered a price higher than a certain value.

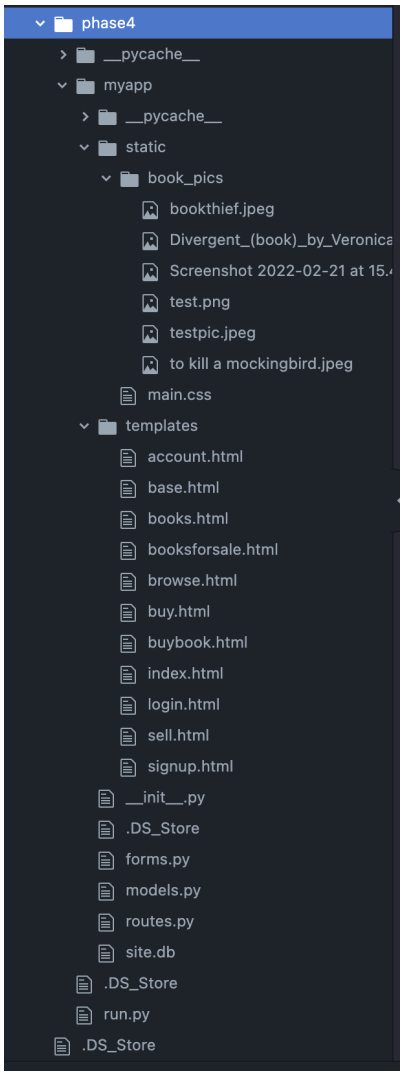
Another limitation of my solution is that it relies on users checking to see when they have sold a book so that they can bring it to the library and the swap can be made. However, since there is no notification set up for when a user sells a book, there is no way of them knowing they have sold it until they go on their account and see past sales. If I had more time in the future, I would improve this by firstly, having a check box for each book under past sales that reads 'delivered' so that the user can keep track of which past sales have been brought in and which have not. I would also set up a notification for when a user buys a book, that both notifies the buyer of their transaction via email, and notifies the seller that they have just sold a book and remind them to deliver it. If I wanted to take it a step further, an email notification could be sent weekly for each book that doesn't have the 'delivered' check box ticked, so that the user is reminded regularly and cannot forget about it. This improves the usability of my system, and lessens the responsibility on the users slightly.

Another key limitation is the fact that you cannot delete or edit a book, which I have discussed under the 'if I had more time' section above. The program could be developed to deal with this in the future by having a page linked from account.html that is called 'my posted books'. This page would display all of the user's current posted books, and for each of them, there would be an 'X' icon that would be clicked and would allow the user to delete the sale. This would just be done by deleting the book from the BooksForSale table. An edit button would also be under each book, and when clicked, the entry boxes from books.html and buybook.html would be available again for the user to change, and whichever was changed, it would update the database accordingly so that changes can be committed.

In conclusion:

Overall, my solution met a large amount of the success criteria, and although there were a few key features that slipped my mind or that I didn't have time to complete, the parts that I did complete were tested and proved to work successfully for the most part. There is a fair amount of usability features included which is important, and I have managed to discuss how I would implement the criteria I did not meet, so if I did have more time, it shouldn't be hard to meet all of my success criteria and develop further with things I had noticed that I hadn't included in my success criteria.

APPENDIX



In main.css:

main.css

```css
body {
  background: #fafafa;
  color: #333333;
  margin-top: 5rem;
}

h1, h2, h3, h4, h5, h6 {
  color: #444444;
}

.bg-steel {
  background-color: #5f788a;
}

.site-header .navbar-nav .nav-link {
  color: #cbd5db;
}

.site-header .navbar-nav .nav-link:hover {
  color: #ffffff;
}

.site-header .navbar-nav .nav-link.active {
  font-weight: 500;
}

.content-section {
  background: #ffffff;
  padding: 10px 20px;
  border: 1px solid #dddddd;
  border-radius: 3px;
  margin-bottom: 20px;
}

.article-title {
  color: #444444;
}

a.article-title:hover {
  color: #428bca;
  text-decoration: none;
}
```

```css
.article-content {
  white-space: pre-line;
}

.article-img {
  height: 65px;
  width: 65px;
  margin-right: 16px;
}

.article-metadata {
  padding-bottom: 1px;
  margin-bottom: 4px;
  border-bottom: 1px solid #e3e3e3
}

.article-metadata a:hover {
  color: #333;
  text-decoration: none;
}

.article-svg {
  width: 25px;
  height: 25px;
  vertical-align: middle;
}

.account-img {
  height: 125px;
  width: 125px;
  margin-right: 20px;
  margin-bottom: 16px;
}

.account-heading {
  font-size: 2.5rem;
}
```

In account.html:

account.html

```
{% extends 'base.html' %}

{% block content %}
<h1>Username: {{ current_user.username }}</h1>
<h3>Your credit: {{ current_user.credit }}</h3>

<br>
<br>
<h3>Your past purchases:</h3>
<div class="row">
    {% for book in purchases %}
        <div class="col-md-3">
            <p class="article-content">{{ book.Book.title }}</p>
            <p class="article-content">Sold by: {{ book.User.username}}</p>
            <p class="article-content">{{ book.Book.author }}</p>
            <p class="article-content">{{ book.Book.genre }}</p>
            <p class="article-content">Bought for {{ book.BooksForSale.price }} Credits</p>
            <p class="article-content">{{ book.BooksForSale.style }}</p>
            <p class="article-content">{{ book.BooksForSale.condition }}</p>
            <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">

        </div>

       {%  if loop.index is divisibleby 3 and not loop.last %}
       </div> <br> <div class="row">
         {% endif %}

    {% endfor %}
</div>

<br>
<p class="article-content">Total credits spent: {{ total_credits_spent }}</p>
<br>
<br>

<h3>Your past sales:</h3>
<div class="row">
    {% for book in sales %}
        <div class="col-md-3">
            <p class="article-content">{{ book.Book.title }}</p>
            <p class="article-content">Bought by: {{ book.User.username }}</p>
            <p class="article-content">{{ book.Book.author }}</p>
            <p class="article-content">{{ book.Book.genre }}</p>
            <p class="article-content">Sold for {{ book.BooksForSale.price }} Credits</p>
            <p class="article-content">{{ book.BooksForSale.style }}</p>
            <p class="article-content">{{ book.BooksForSale.condition }}</p>
            <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">


            <br>
        </div>

       {%  if loop.index is divisibleby 3 and not loop.last %}
       </div> <br> <div class="row">
         {% endif %}

    {% endfor %}
</div>

<br>
<p class="article-content">Total credits made: {{ total_credits_made }}</p>

{% endblock content %}
```

In base.html:

base.html

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-EVSTQN3/azprG1Anm3QD
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-MrcW6ZMFYlzcLA8Nl+NtUVF0sA7MsXs

  <head>
    <meta charset="utf-8">
    <title></title>

  </head>
  <body>
    <nav class="navbar navbar-expand-lg navbar-light bg-success">
      <div class="container-fluid">
        <a class="navbar-brand" href="index">Bookstore</a>
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarNav" aria-controls="navbarNav" aria-ex
          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarNav">
          <ul class="navbar-nav">
            <li class="nav-item">
              <a class="nav-link active" aria-current="page" href="{{ url_for('index') }}">Home</a>
            </li>
            {% if current_user.is_authenticated %}
            <li class="nav-item">
              <a class="nav-link" href="{{ url_for('account') }}">Account</a>
            </li>
            <li class="nav-item">
              <a class="nav-link" href="{{ url_for('logout') }}">Log out</a>
            </li>
            {% else %}
              <li class="nav-item">
                <a class="nav-link" href="{{ url_for('login') }}">Log in</a>
              </li>
              <li class="nav-item">
                <a class="nav-link" href="{{ url_for('signup') }}">Sign up</a>
              </li>
            {% endif %}
            <li class="nav-item">
              <a class="nav-link" href="{{ url_for('browse') }}">Browse</a>
            </li>
              <li class="nav-item">
                <a class="nav-link" href="{{ url_for('books') }}">Sell</a>
              </li>
              <li class="nav-item">
                 <a class="nav-link" href="{{ url_for('buy') }}">Buy</a>
                </li>
          </ul>
        </div>
      </div>
    </nav>
      {% with messages = get_flashed_messages(with_categories=true) %}
        {% if messages %}
          {% for category, message in messages %}
            <div class="alert alert-{{ category }}">
              {{ message }}
            </div>
          {% endfor %}
        {% endif %}
      {% endwith %}
      {% block content %} {% endblock %}
  </body>
</html>

```

In books.html:

books.html

```
{% extends 'base.html' %}

{% block content %}
<h1> check by title </h1>
<div class="content-section">
  <a class="btn btn-info" href="{{url_for('sell')}}">Add new book</a>
        <form method="POST" action="" enctype="multipart/form-data">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <div class="form-group">
                        {{ form.title.label(class="form-control-label") }}
                        {{ form.title(class="form-title form-control-lg") }}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
        </form>
</div>

<div class="row">
    {% for book in books %}
          <div class="col-md-3">
              <p class="article-content">{{ book.title }}</p>
              <p class="article-content">{{ book.author }}</p>
              <p class="article-content">{{ book.genre }}</p>
              <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.picture) }}">
              <a class="btn btn-info" href="{{url_for('booksforsale', selling_book_id=book.id)}}">Select</a>
          </div>

        {%  if loop.index is divisibleby 3 and not loop.last %}
        </div> <br> <div class="row">
          {% endif %}

    {% endfor %}
</div>
{% endblock content %}
<!-- will display pics of books and have select button that will take user to books for sale form -->
<!-- will have check by title form above display and when you press submit it shows the books that match -->

```

In booksforsale.html:

booksforsale.html

```
{% extends 'base.html' %}

{% block content %}
<h1> selling a copy of first book </h1>
<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <div class="form-group">
                        {{ form.price.label(class="form-control-label") }}
                        {{ form.price(class="form-price form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.style.label(class="form-control-label") }}
                    {{ form.style(class="form-style form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.condition.label(class="form-control-label") }}
                    {{ form.condition(class="form-condition form-control-lg") }}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
        </form>
    </div>
{% endblock %}

```

In buy.html:

buy.html

```
{% extends 'base.html' %}

{% block content %}
<h1> Buy a book </h1>

<div class="row">
    {% for book in books %}
          <div class="col-md-3">
              <p class="article-content">{{ book.Book.title }}</p>
              <p class="article-content">Posted by: {{ book.User.username }}</p>
              <p class="article-content">{{ book.Book.author }}</p>
              <p class="article-content">{{ book.Book.genre }}</p>
              <p class="article-content">{{ book.BooksForSale.price }} Credits</p>
              <p class="article-content">{{ book.BooksForSale.style }}</p>
              <p class="article-content">{{ book.BooksForSale.condition }}</p>
              <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">
              <a class="btn btn-info" href="{{url_for('buybook', buying_book_id=book.BooksForSale.id)}}">Buy</a>
          </div>

        {%  if loop.index is divisibleby 3 and not loop.last %}
        </div> <br> <div class="row">
          {% endif %}

    {% endfor %}
</div>


{% endblock content %}

```

In buybook.html:

buybook.html

```
{% extends 'base.html' %}

{% block content %}

<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <h1> Buy this book </h1>


            <p class="article-content">{{ book.Book.title }}</p>
            <p class="article-content">{{ book.Book.author }}</p>
            <p class="article-content">{{ book.Book.genre }}</p>
            <p class="article-content">{{ book.BooksForSale.price }} Credits</p>
            <p class="article-content">{{ book.Book.picture }}</p>

            <img class="img-fluid" src="{{ url_for('static', filename='book_pics/' + book.Book.picture) }}">

            <p class="article-content">You have {{ credit }} credits</p>


            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
            <br>
            <br>

            <a class="btn btn-info" href="{{url_for('index')}}">Cancel</a>

        </form>
    </div>
{% endblock %}

```

In index.html:

index.html

```
{% extends 'base.html' %}

{% block content %}
  <h1> welcome to the homepage </h1>
<h3> buy, sell, browse, suggest, BOOKS!!! </h3>
{% if current_user.is_authenticated %}
<div class="d-grid gap-2">
<a class="btn btn-primary" href="{{ url_for('books') }}">Post books for sale</a>
</div>
{% endif %}
{% endblock %}

```

In login.html:

login.html

```
{% extends 'base.html' %}

{% block content %}
<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <legend class="border-bottom mb-4">Log In</legend>
                <div class="form-group">
                    {{ form.email.label(class="form-control-label") }}
                    {% if form.email.errors %}
                        {{ form.email(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.email.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.email(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.password.label(class="form-control-label") }}
                    {% if form.password.errors %}
                        {{ form.password(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.password.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.password(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-check">
                    {{ form.remember(class="form-check-input") }}
                    {{ form.remember.label(class="form-check-label") }}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
            <small class="text-muted ml-2">
                <a href="#">Forgot Password?</a>
            </small>
        </form>
    </div>
    <div class="border-top pt-3">
        <small class="text-muted">
            Need An Account? <a class="ml-2" href="{{ url_for('signup') }}">Sign Up Now</a>
        </small>
    </div>
{% endblock %}
```

In sell.html:

sell.html

```
{% extends 'base.html' %}

{% block content %}
<h1> Welcome, sell a book :) </h1>
<div class="content-section">
        <form method="POST" action=" " enctype="multipart/form-data">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <div class="form-group">
                        {{ form.title.label(class="form-control-label") }}
                        {{ form.title(class="form-title form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.author.label(class="form-control-label") }}
                    {{ form.author(class="form-author form-control-lg") }}
                </div>
                <div class="form-group">
                    {{ form.genre.label(class="form-control-label") }}
                    {{ form.genre(class="form-genre form-control-lg") }}
                </div>
                <div class="media">
                    {{ form.picture.label() }}
                    {{ form.picture(class="form-control-file") }}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
        </form>
    </div>
{% endblock %}
```

In signup.html:

signup.html

```
{% extends 'base.html' %}

{% block content %}
<div class="content-section">
        <form method="POST" action="">
            {{ form.hidden_tag() }}
            <fieldset class="form-group">
                <legend class="border-bottom mb-4">Join Today</legend>
                <div class="form-group">
                    {{ form.username.label(class="form-control-label") }}

                    {% if form.username.errors %}
                        {{ form.username(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.username.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.username(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.email.label(class="form-control-label") }}
                    {% if form.email.errors %}
                        {{ form.email(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.email.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.email(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.password.label(class="form-control-label") }}
                    {% if form.password.errors %}
                        {{ form.password(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.password.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.password(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
                <div class="form-group">
                    {{ form.confirm_password.label(class="form-control-label") }}
                    {% if form.confirm_password.errors %}
                        {{ form.confirm_password(class="form-control form-control-lg is-invalid") }}
                        <div class="invalid-feedback">
                            {% for error in form.confirm_password.errors %}
                                <span>{{ error }}</span>
                            {% endfor %}
                        </div>
                    {% else %}
                        {{ form.confirm_password(class="form-control form-control-lg") }}
                    {% endif %}
                </div>
            </fieldset>
            <div class="form-group">
                {{ form.submit(class="btn btn-outline-info") }}
            </div>
        </form>
    </div>
    <div class="border-top pt-3">
        <small class="text-muted">
            Already Have An Account? <a class="ml-2" href="{{ url_for('login') }}">Sign In</a>
        </small>
    </div>
{% endblock %}

```

In __init__.py:

__init__.py

```python
from flask import Flask
from flask_sqlalchemy import SQLAlchemy
from flask_bcrypt import Bcrypt
from flask_login import LoginManager

app = Flask(__name__)
app.config['SECRET_KEY'] = '46eeb711778955cec2d6ce3853b003ff'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///site.db'
db = SQLAlchemy(app)
bcrypt = Bcrypt(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'

from myapp import routes

```

In forms.py:

forms.py

```python
from flask_wtf import FlaskForm
from flask_wtf.file import FileField, FileAllowed
from wtforms import StringField, PasswordField, SubmitField, BooleanField, IntegerField, SelectField, FileField
from wtforms.validators import DataRequired, Length, Email, EqualTo, ValidationError
from myapp.models import User, Book

class SignUpForm(FlaskForm):
    username = StringField('Username',
                           validators =[DataRequired(), Length(min=2, max=20)])
    email = StringField('Email',
                        validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    confirm_password = PasswordField('Confirm Password',
                                     validators=[DataRequired(), EqualTo('password')])
    submit = SubmitField('Sign Up')

    def validate_username(self, username):   #this function is checking to make sure that the same username that a user wants to create an a
        user = User.query.filter_by(username=username.data).first()
        if user:
            raise ValidationError('That username is taken. Please choose a different one ')

    def validate_email(self, email):   #this function is checking to make sure that the same username that a user wants to create an account
        user = User.query.filter_by(email=email.data).first()
        if user:
            raise ValidationError('That email is taken. Please choose a different one ')

class LoginForm(FlaskForm):
    email = StringField('Email',
                        validators=[DataRequired(), Email()])
    password = PasswordField('Password', validators=[DataRequired()])
    remember = BooleanField('Remember Me')
    submit = SubmitField('Login')

class BookForm(FlaskForm):
    title = StringField('Title',
                        validators =[DataRequired(), Length(min=2, max=100)])
    author = StringField('Author', validators = [DataRequired(), Length(min=3, max=75)])
    genre = SelectField('Genre', choices=['Fiction', 'Non-fiction', 'Fantasy', 'Sci-fi', 'Mystery', 'Thriller', 'Romance', 'Dystopian', 'Dra
    picture = FileField('Picture', validators = [DataRequired(), FileAllowed(['jpeg', 'png'])])
    submit = SubmitField('Add book')


class BooksForSaleForm(FlaskForm):
    price = IntegerField('Price', validators=[DataRequired()])
    style = SelectField('Hardback or paperback?', choices=['Hardback', 'Paperback'], validators=[DataRequired()])
    condition = SelectField('Book condition', choices=['Brand new', 'Used-perfect', 'Used - good', 'Used - fine', 'Used - poor'])
    submit = SubmitField('Add book')

class CheckByTitleForm(FlaskForm):
    title = StringField('Title')
    submit = SubmitField('Search')

class BuyBookForm(FlaskForm):
    submit = SubmitField('Confirm Purchase')

```

In models.py:

models.py

```python
from myapp import db, login_manager
from flask_login import UserMixin

@login_manager.user_loader
def load_user(user_id):
    return User.query.get(int(user_id))

#user table holds all data about the users created upon signup
class User(db.Model, UserMixin):
    __tablename__ = 'User'
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(20), unique=True, nullable=False)
    email = db.Column(db.String(120), unique=True, nullable=False)
    password = db.Column(db.String(50), nullable=False)
    credit = db.Column(db.Integer, nullable=False)

    def __repr__ (self):
        return f"User('{self.username}', '{self.email}', '{self.password}', '{self.credit}')"

#book table holds general info of books posted for sale for the first time, so that they may be reused for a different user wishing to sell
class Book(db.Model):
    __tablename__ = 'Book'
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(20), nullable=False)
    author = db.Column(db.String(50), nullable=False)
    genre = db.Column(db.String(50), nullable=False)
    picture = db.Column(db.String(50))

    def __repr__ (self):
        return f"Book('{self.title}', '{self.author}', '{self.genre}', '{self.picture}')"

#booksforsale table holds the information of all books ever posted for sale
class BooksForSale(db.Model):
    __tablename__ = 'BooksForSale'
    id = db.Column(db.Integer, primary_key=True)
    price = db.Column(db.Integer, nullable=False)
    style = db.Column(db.String(50), nullable=False)
    condition = db.Column(db.String(50), nullable=False)
    seller_id = db.Column(db.Integer, db.ForeignKey('User.id'), nullable=False)
    buyer_id = db.Column(db.Integer, db.ForeignKey('User.id'))
    book_id = db.Column(db.Integer, db.ForeignKey('Book.id'), nullable=False)

    seller = db.relationship('User', foreign_keys=[seller_id])
    buyer = db.relationship('User', foreign_keys=[buyer_id])
    book = db.relationship('Book', foreign_keys=[book_id])

    def __repr__ (self):
        return f"BooksForSale('{self.price}', '{self.condition}', '{self.seller_id}', '{self.buyer_id}', '{self.book_id}')"

#didn't have time to implement a booklist but this is what the table would look like in models.py
# class Booklist(db.Model):
#     id = db.Column(db.Integer, primary_key=True)
#     user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
#     book_id = db.Column(db.Integer, db.ForeignKey('book.id'), nullable=False)
#
#     def __repr__ (self):
#         return f"Booklist('{self.user_id}', '{self.book_id}')"
```

In routes.py:

routes.py

```python
import os
import secrets
from flask import render_template, url_for, flash, redirect, request
from myapp import app, db, bcrypt
from myapp.forms import SignUpForm, LoginForm, BookForm, BooksForSaleForm, CheckByTitleForm, BuyBookForm
from myapp.models import User, Book, BooksForSale
from flask_login import login_user, current_user, logout_user, login_required
# from PIL import Image

@app.route('/')
@app.route('/index')
def index():
    user = {'username': 'gaiag'}
    return render_template('index.html', title = 'title', user = user)

@app.route('/signup', methods=['GET', 'POST'])
def signup():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = SignUpForm()
    if form.validate_on_submit():
        hashed_password = bcrypt.generate_password_hash(form.password.data).decode('utf-8')
        user = User(username=form.username.data, email=form.email.data, password=hashed_password, credit=5)
        db.session.add(user)
        db.session.commit()
        flash(f'Your account has been created! You are now able to log in', 'success')
        return redirect(url_for('login'))
    return render_template('signup.html', title='Sign Up', form=form)

@app.route('/login', methods=['GET', 'POST'])
def login():
    if current_user.is_authenticated:
        return redirect(url_for('index'))
    form = LoginForm()
    if form.validate_on_submit():
        user = User.query.filter_by(email=form.email.data).first()
        if user and bcrypt.check_password_hash(user.password, form.password.data):
            login_user(user, remember=form.remember.data)
            next_page = request.args.get('next')
            return redirect(next_page) if next_page else redirect(url_for('index'))
        else:
            flash('Login unsuccessful. Please check email and password', 'danger')
    return render_template('login.html', title='Login', form=form)
```

```python


@app.route('/browse')
def browse():
    return render_template('browse.html')

#route for books.html - displays books so that user can select a previosuly posted book to sell if it exists in the database
@app.route('/books', methods=['GET', 'POST'])
def books():
    form = CheckByTitleForm()
    books = Book.query.all()
    if form.validate_on_submit():
        search_title = form.title.data
        search = "%{}%".format(search_title)
        books = Book.query.filter(Book.title.like(search)).all()
    return render_template('books.html', form=form, books=books)

def save_picture(form_picture):
    picture_path = os.path.join(app.root_path, 'static/book_pics', form_picture.filename)
    output_size = (125, 125)
    form_picture.save(picture_path)

#route for a new book being posted for sale - picture is saved to database during this using the procedure save_picture
@app.route('/sell', methods=['GET', 'POST'])
@login_required
def sell():
    if current_user.is_authenticated:
        form = BookForm()
        if form.validate_on_submit():
            flash('Please enter next details', 'success')
            book = Book(title=form.title.data, author = form.author.data, genre=form.genre.data, picture=form.picture.data.filename)
            db.session.add(book)
            #picture_file calls the procedure save_picture
            picture_file = save_picture(form.picture.data)
            print(book)
            db.session.commit()
            return redirect(url_for('booksforsale', selling_book_id = book.id))
        return render_template('sell.html', title='Post Book', form=form)
    else:
            flash('Please login to sell books.', 'danger')
            form = LoginForm()
            return render_template('sell.html', title='sell', form=form)

#final steps of a book being posted for sale - details of book entered and saved to BooksForSale table
@app.route('/booksforsale/<int:selling_book_id>', methods=['GET', 'POST'])
@login_required
def booksforsale(selling_book_id):
        form = BooksForSaleForm()
        if form.validate_on_submit():
            flash('Your book has been created!', 'success')
            bookforsale = BooksForSale(price=form.price.data, style=form.style.data,
                    condition=form.condition.data, seller_id = current_user.id, book_id = selling_book_id)
            db.session.add(bookforsale)
            db.session.commit()
            return redirect(url_for('index'))
        return render_template('booksforsale.html', title='Post book', form=form)


@app.route('/logout')
def logout():
    logout_user()
    return redirect(url_for('index'))

#route for account page - has the queries for displaying the users past sales and purchases, and has the code for calculating total credits
@app.route('/account')
@login_required
def account():
    purchases = db.session.query(BooksForSale, Book, User).filter(BooksForSale.buyer_id==current_user.id).join(Book, BooksForSale.book_id==
    total_credits_spent = 0
    for book in purchases:
        total_credits_spent = total_credits_spent + book.BooksForSale.price
    pass
    sales = db.session.query(BooksForSale, Book,User).filter(BooksForSale.seller_id==current_user.id, BooksForSale.buyer_id!=None).join(Boo
    total_credits_made = 0
    for book in sales:
        total_credits_made = total_credits_spent + book.BooksForSale.price
    pass
    return render_template('account.html', title='Account', purchases=purchases, sales=sales, total_credits_spent=total_credits_spent, tota

#buy route has query so that books can be displayed on buy.html that aren't posted by the current user and that have not already been sold
@app.route('/buy', methods=['GET', 'POST'])
def buy():
    books = db.session.query(BooksForSale, Book, User).filter(BooksForSale.seller_id!=current_user.id, BooksForSale.buyer_id==None).join(Bo
    return render_template('buy.html', title='Buy', books=books)

#once book is selected to be bought, buybook route ensures that user has enough credit to purchase it, and makes the changes to seller and
@app.route('/buybook/<int:buying_book_id>', methods=['GET', 'POST'])
@login_required
def buybook(buying_book_id):
        form = BuyBookForm()
        if request.method=='POST':
            book = BooksForSale.query.filter(BooksForSale.id==buying_book_id).first()
            print(book)
            sellerUser = User.query.filter(book.seller_id==User.id).first()
            new_seller_credit = sellerUser.credit + book.price
            new_credit = current_user.credit - book.price
            if new_credit < 0:
                flash('Insufficient credit', 'danger')
                return redirect(url_for('index'))
            else:
                book.buyer_id = current_user.id
                current_user.credit = new_credit
                sellerUser.credit = new_seller_credit
                db.session.commit()
                flash('Purchase successful', 'success')
                return redirect(url_for('index'))
        book = db.session.query(BooksForSale, Book).join(Book).filter(BooksForSale.id==buying_book_id).first()
        return render_template('buybook.html', title='Buy book', book=book, credit=current_user.credit, form=form)


```

In run.py:

run.py

```
1  from myapp import app
2
3  if __name__ == '__main__':
4      app.run(debug=True)
5
```