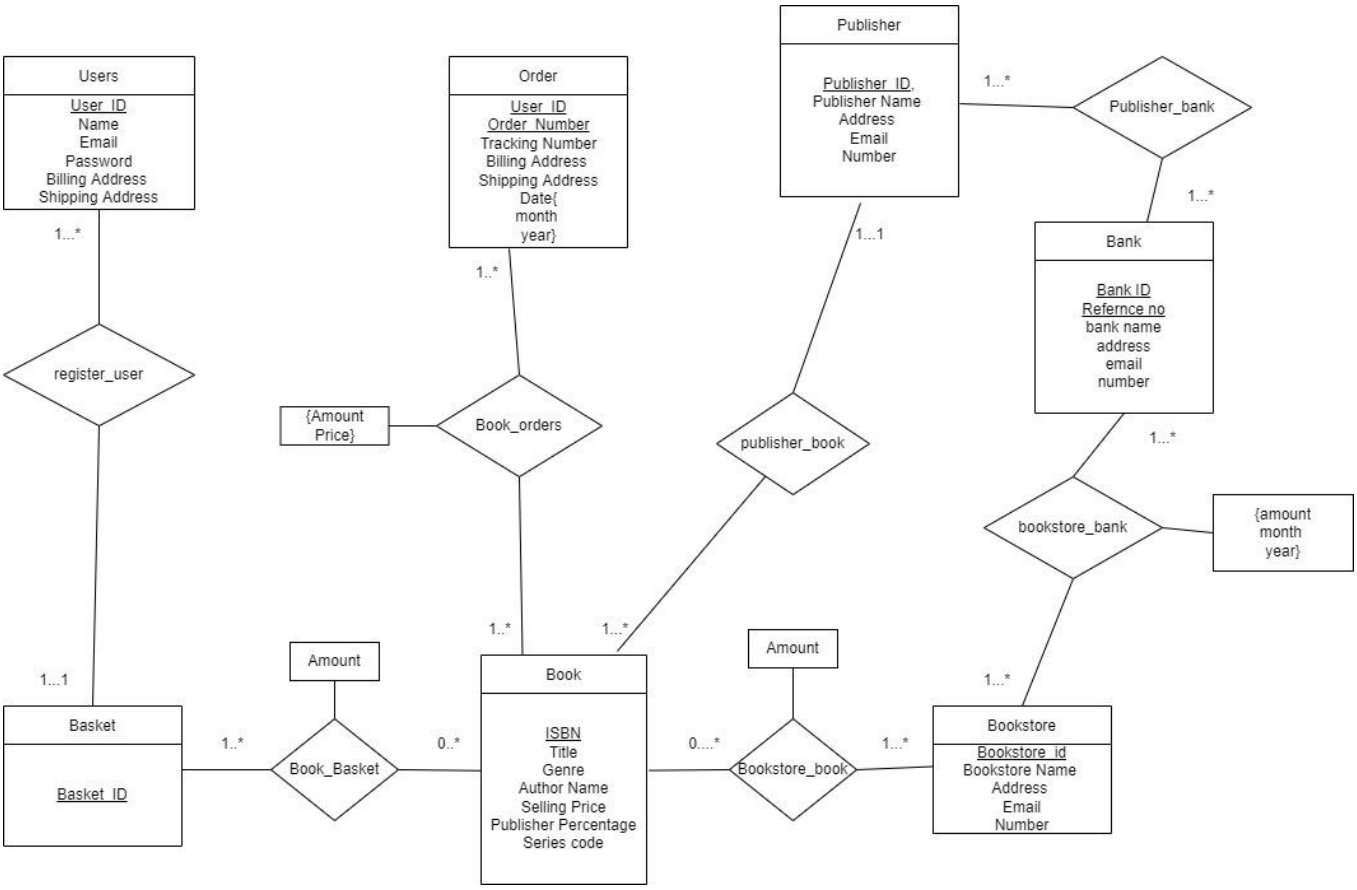
COMP 3005 Term Project
Fall 2021

Due: December 19th, 2021
Presented to: Ahmed El-Roby

Edward Akapo, 101095403
Sarah Abdallah, 101119716

Conceptual Design

The conceptual design of the database is represented by the following entity-relationship diagram:



The following assumptions apply:

- The user must sign in to their account using their email and password. The email must be unique.
- The owner does not need to sign in and instead has a specific allotted sign-in area.
- The user can sign up for an account with their username, password, email, shipping address, and billing address.
- Each book in the bookstore has a unique ISBN.
- Each basket has a unique ID value, and it is important to keep record of the amount of books that are in the basket.
- Each placed order has a unique order number.
- Each bookstore has a unique bookstore ID value.
- Book_orders shall store the total price of order after the publisher tax has been paid.
- When a user checks out books from their cart, the order will be separated into individual orders for each unique book.
- Bookstore_book functions as the collection of a bookstore owner's books (also known as the inventory). When an owner adds books to their collection, the system sees this as the owner buying books and adding them, thus this adds to the total expenditure. Removing books either reduces book amount or removes the book from the owner's collection.

- The user check-out basket holds the items the user wishes to buy and they will be saved until the user removes them or buys them. One purchased item will be removed from the check-out basket and added to the order relation.
- Upon registration, the user will gain a randomly-generated basket_ID and user_ID.
- Checkout baskets only exist when being filled.
- The bookstore must have at least one book in order to be considered a bookstore.
- The inventory keeps track of publisher_ID.
- Billing and Shipping Addresses are not assumed to be unique because two people could be roommates and have the same addresses for shipping, as well as people could share a credit card and have the same billing information.
- In terms of entities, a bookstore owner is synonymous to a bookstore.
- A user can exist without a check-out basket, but a check-out basket cannot exist without a user. Similarly, books can exist without a check-out basket, but a check-out basket cannot exist without books.
- Books can exist without a bookstore, but a bookstore cannot exist without books.

The following assumptions apply of the basis of cardinality:

- 1 user has 1 basket.
- 1 basket has zero to many books.
- 1 bookstore has 1 owner.
- 1 bookstore has 1 collection.
- 1 collection has 1 to many books.
- 1 user can have many orders.
- 1 order can have many books.
- 1 book has 1 publisher.
- 1 bookstore has many publishers.
- 1 publisher has many books.
- 1 publisher has 1 bank.
- 1 bank can manage many publishers.
- 1 bookstore has 1 bank.
- 1 bank can manage many bookstores.

Reduction to Relational Schemas

Users(ID, Basket_ID, Username, Email, User_Password, Billing_Address, Shipping_Address)
Basket(Basket_ID)
Orders(ID, Order Number, Tracking number, Billing_Address, Shipping_Address, Month, Year)
Book_Orders(ID, Order_Number, ISBN, Price, Amount)
Book(ISBN, Title, Publisher_ID, Genre, Author_name, Selling_Price, Publisher Percentage, Series Code)
Publisher(Publisher_ID, Publisher_Name, Address, Email, Number)
Bookstore(ID,Store_Name, Address, Email, Number)
Book_Basket(Basket_ID, ISBN, Amount)
Book_Bookstore(ID, ISBN, Amount)
Bookstore_Bank(BookStore_ID, Bank_ID, Reference_no, Amount, Month, Year)
Bank(ID, Reference_no, Bank_Name, Address, Email, Number)
Publisher_Bank(Publisher_ID, Bank_ID, Reference_no)

Normalization of Relation Schemas

For the normalizations and decompositions, we will be decomposing relations into 3NF then seeing if we can further decompose them into BCNF:

Users

R = {ID, Basket_ID, Username, Email, User_password, Billing_Address, Shipping_Address}

So our functional dependencies are;

F = {

ID → Basket_ID, Username, Email, User_password, Billing_Address, Shipping_Address

Basket_ID → ID
Basket_ID→ID,Username,Email,User_password,Billing_Address,Shipping_Address

Email→ID

Email →ID, Basket_ID, Username, User_password, Billing_Address, Shipping_Address

Shipping_Address, Billing_Address,Username→ID, Basket_ID, Email, User_password

}

Then we need to find the canonical cover in order to start reducing to 3nf

Fc = {

ID → Basket_ID, Username, Email, User_password, Billing_Address, Shipping_Address

Basket_ID → ID

Email→ID

Shipping_Address, Billing_Address,Username → ID, Basket_ID, Email, User_password

}

We removed Basket_ID → 'ALL' and Email→'ALL' because those dependencies can be gotten from ID → 'ALL', Basket_ID → ID & Email→ID.

now we attempt 3nf decomposition

3NF = {

R1 = (ID, Basket_ID, Username, Email, User_password, Billing_Address, Shipping_Address)

R2 = (Basket_ID, ID)

R3 = (Email, ID)

R4=(Shipping_Address,Billing_Address,Username, ID, Basket_ID, Email, User_password)

Because R1 contains all of R2 ,R3 and R4

so our final relation is

R(ID, Basket_ID, Username, Email, User_password, Billing_Address, Shipping_Address)

}

R is also in BCNF because the Functional dependencies are all superkeys.

Lastly because we used a 3NF decomposition we know that our relation is dependency preserving and lossless.

Orders

R = (ID, Order Number, Tracking number, Billing_Address, Shipping_Address, Month, Year)
our functional dependencies are;
F = {
ID, Order_Number → Tracking number, Billing_Address, Shipping_Address, Month, Year
Tracking_Number →ID, Order Number, Billing_Address, Shipping_Address, Month, Year
Tracking_Number →ID, Order Number
}
canonical cover is

Fc = {

ID, Order_Number → Tracking number, Billing_Address, Shipping_Address, Month, Year
Tracking_Number →ID, Order Number, Billing_Address, Shipping_Address, Month, Year
}

since we see that the canonical cover only contains superkeys we know that R is already in 3nf and Bcnf

Basket

R(Basket_ID)

since its only one attribute it is in 3nf and bcnf

Book

R(ISBN, Title, Publisher_ID, Genre, Author_name, Selling_Price, Publisher Percentage, Series Code)
our functional dependencies are;

F= {

ISBN→Title, Publisher_ID, Genre, Author_name, Selling_Price, Publisher Percentage, Series Code

Title, Author_name →ISBN, Publisher_ID, Genre, Selling_Price, Publisher Percentage, Series Code

Series Code → Author

Series Code → Genre

Series Code → Publisher

}

canonical cover is;

Fc = {

ISBN→Title, Publisher_ID, Genre, Author_name, Selling_Price, Publisher Percentage, Series Code

Title, Author_name →ISBN, Publisher_ID, Genre, Selling_Price, Publisher Percentage, Series Code

Series Code → Author, Genre, Publisher

}

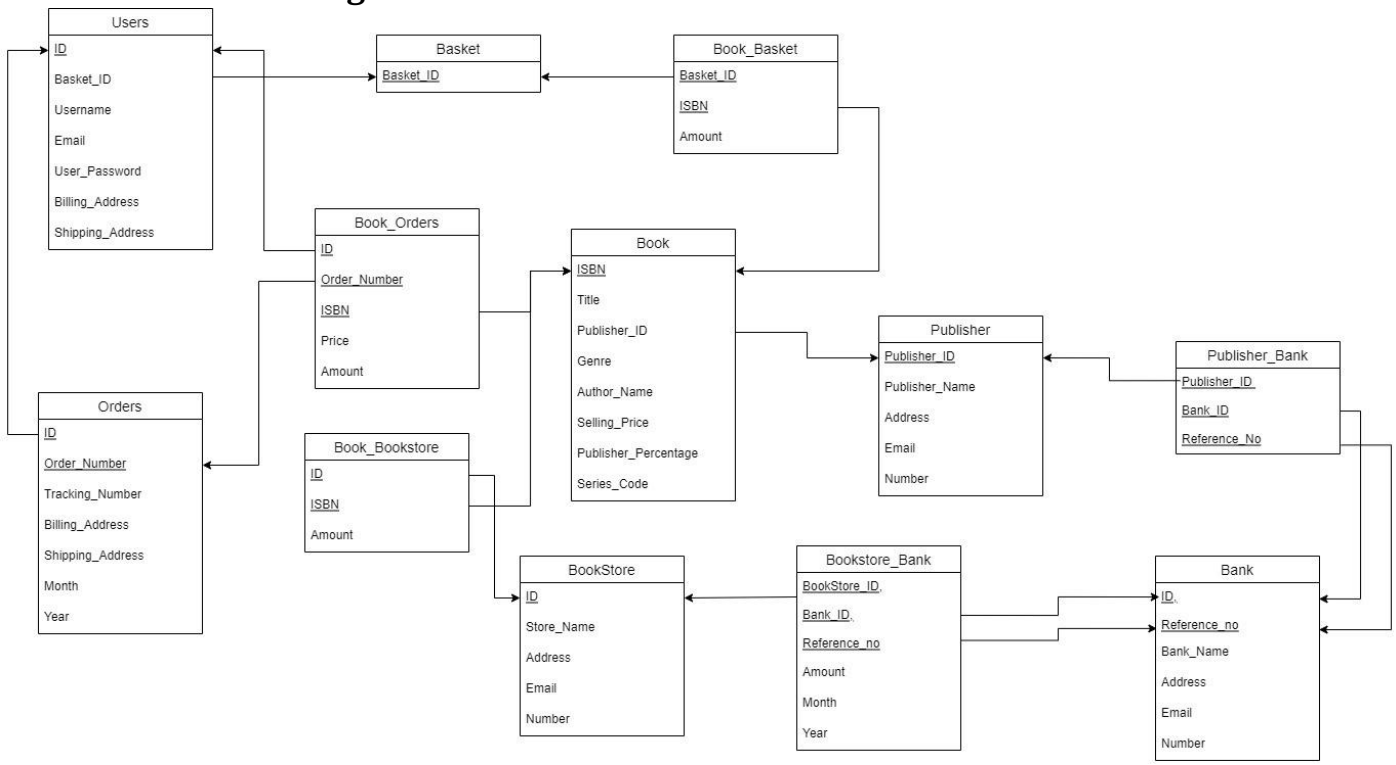
since we see that the canonical cover only contains superkeys we know that R is already in 3nf and Bcnf

Book_Orders

R(ID, Order_Number, ISBN, Price, Amount)
our functional dependencies are;
F ={
ID, Order_Number, ISBN → Price, Amount
}
The canonical cover is the same.
Since we see that the canonical cover only contains superkeys we know that R is already in 3nf and Bcnf

And similarly for the other Relations the functional dependencies are either trivial or all superkeys thus they will already be in 3NF and BCNF.

Database Schema Diagram



Implementation

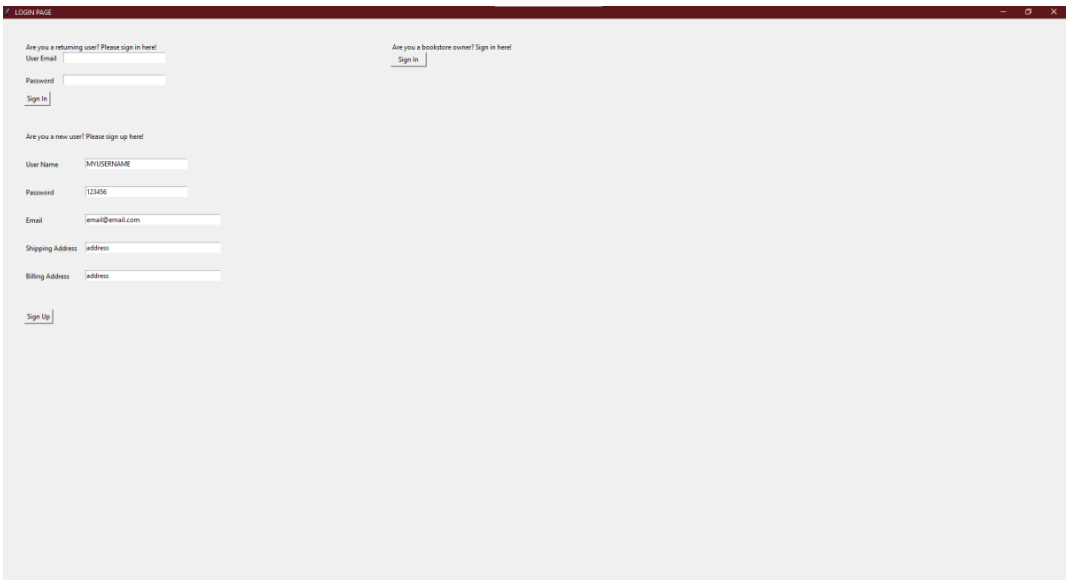
The application was implemented by using both SQL and Python. To define the relational database of the online bookstore, data-definition language (DDL) was written into MySQL. Once the database was initialized, it was populated and provided with queries that could add, retrieve, remove, and alter the data. In order to implement functionality, a connector was established to MySQL using Python, where the name and host of the database are provided, as well as the user and password information. Among the SQL files present, *DDL.sql* includes the DDL operations written to create and connect the tables of the "bookstore" database, as well as parametrize the datatypes of their attributes. The *relationsInsert.sql* file presents the user with insert statements that add data into the tables in the database for testing purposes, as well as the trigger and the views (which retrieves the tuples for a report which a function then aggregates for the report).. The *query.sql* file includes queries that retrieve and update data from the database. The *Triggers.sql* file contains the trigger that will notice when the book quantity is below a threshold and replenish the book supply. It is worth noting that the file *functions.sql* includes one function which was not compatible with MySQL and therefore could not be used.

This application was implemented to reflect the Model View Controller design pattern. One file contains the data and necessary retrieval functions (*data.py*) which will be used as the model. In terms of view, three user interface files exist (*InitialUI.py, UserUI.py,* and *OwnerUI.py*) to allow the users to interact with the system, and they were programmed using the Tkinter library in python. Firstly, the user will see *InitialUI,* which acts as a sign-in and sign-up screen for both

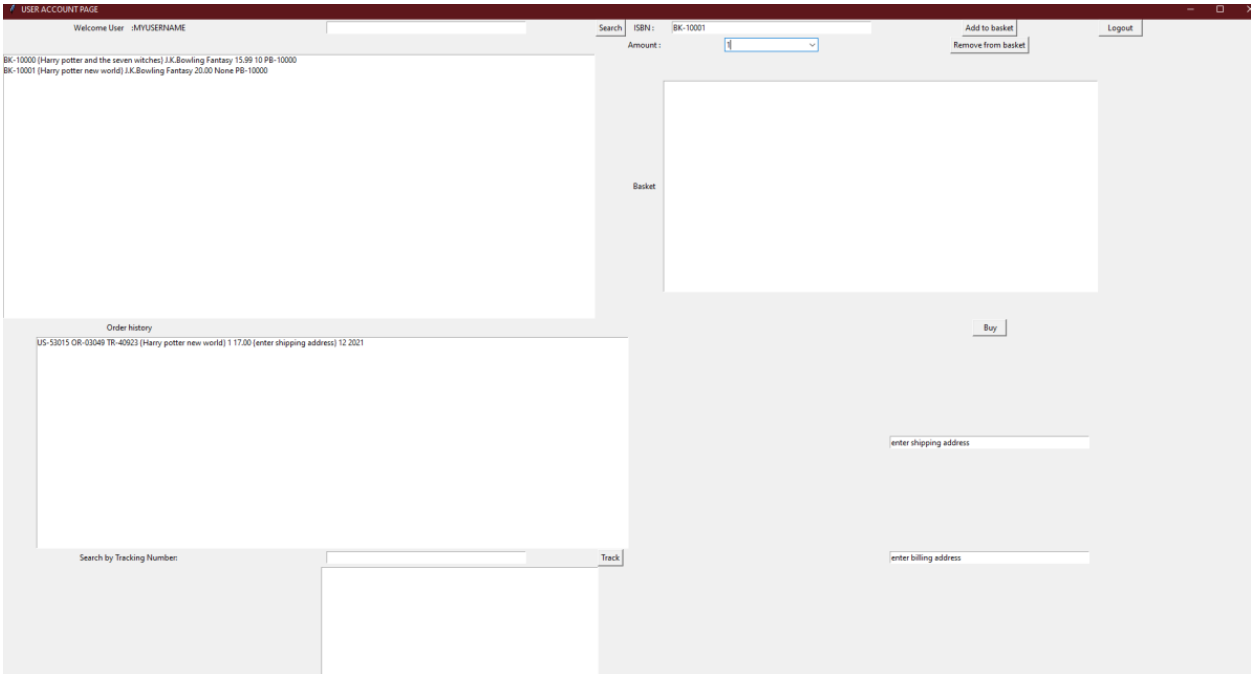
bookstore users and bookstore owners. If the user signs in as a bookstore user, they will be brought to the *UserUI* page. This page will allow a user to search for books, place an order by adding items to their basket, and track their order. When the user logs out, they will be brought back to the *InitialUI* screen. If the user signs in as a bookstore owner, they will be brought to the *OwnerUI* page. *OwnerUI* allows them to view their inventory, add a new book, add an existing book, remove a book, increase the amount of a book, generate a report, and view bank transfers. The controller in this scenario is the main.py file which listens to the events triggered by the user interacting with the view and executes the correct function to produce results.

The following screenshots demonstrate the sequence events that play out within the application:

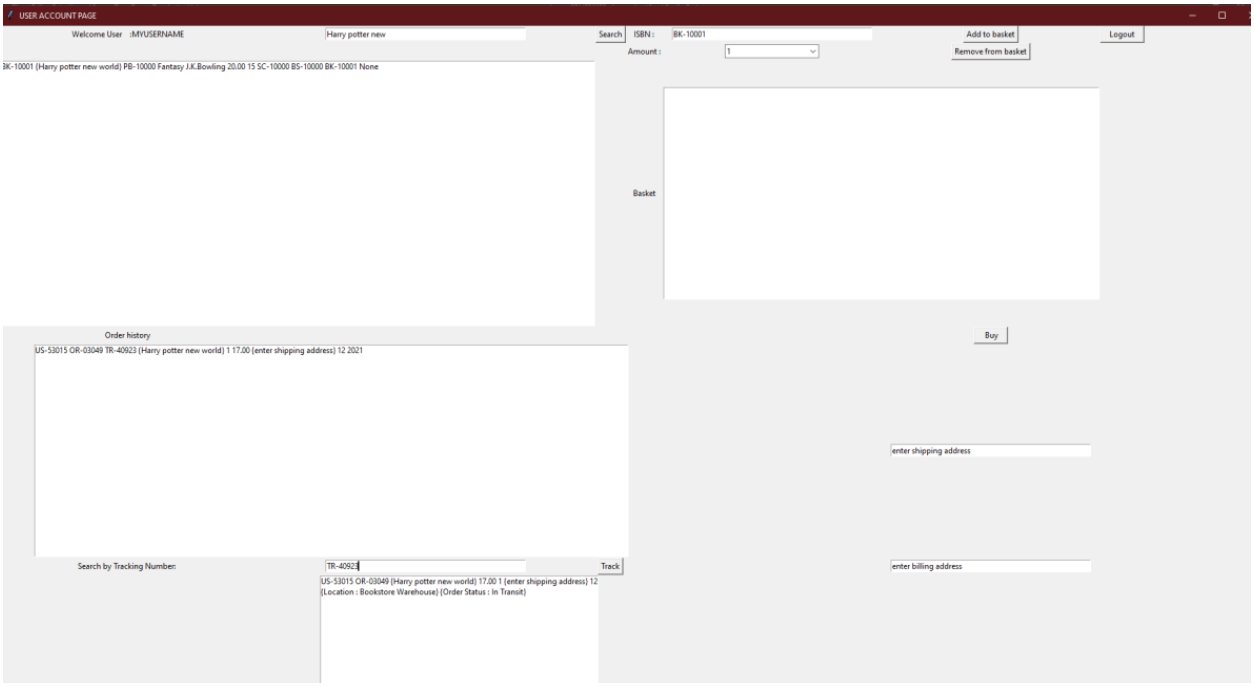
The initial sign-in and sign-up screen:



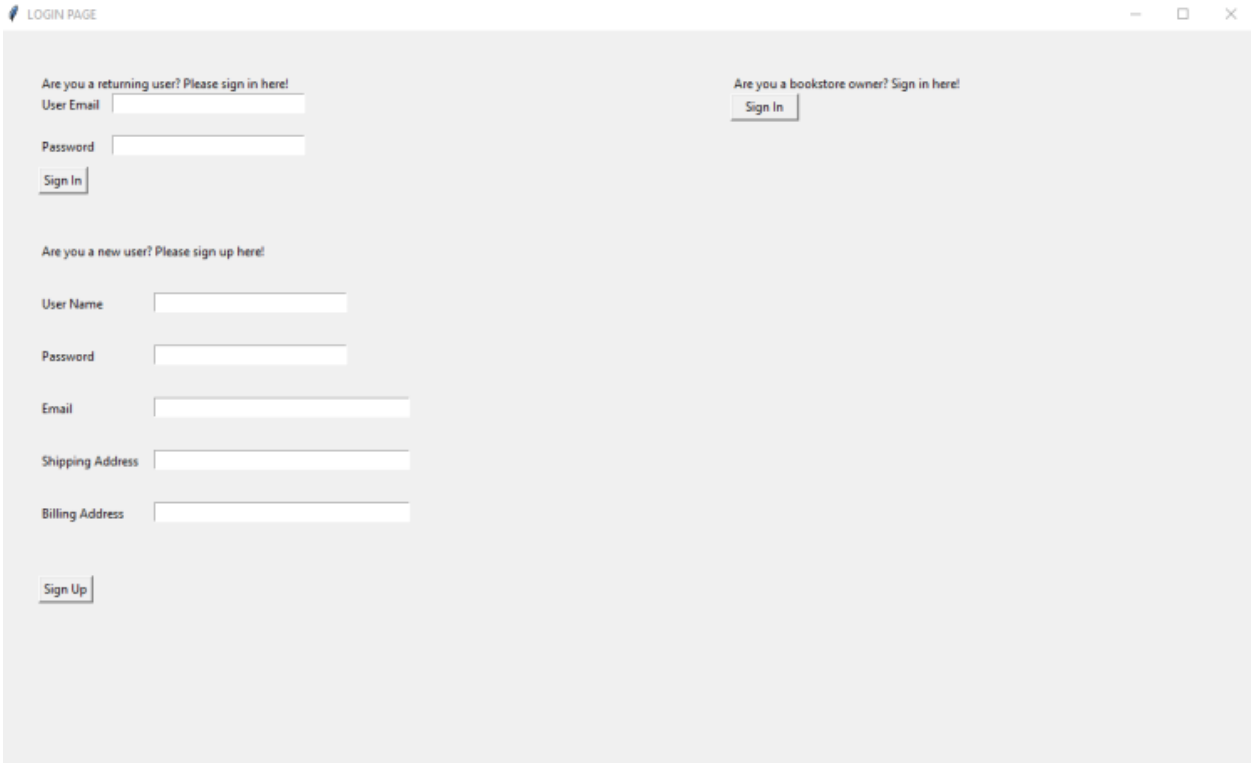
The user searches for a book via the top search bar, adds it to their basket and selects “Buy” to place their order:



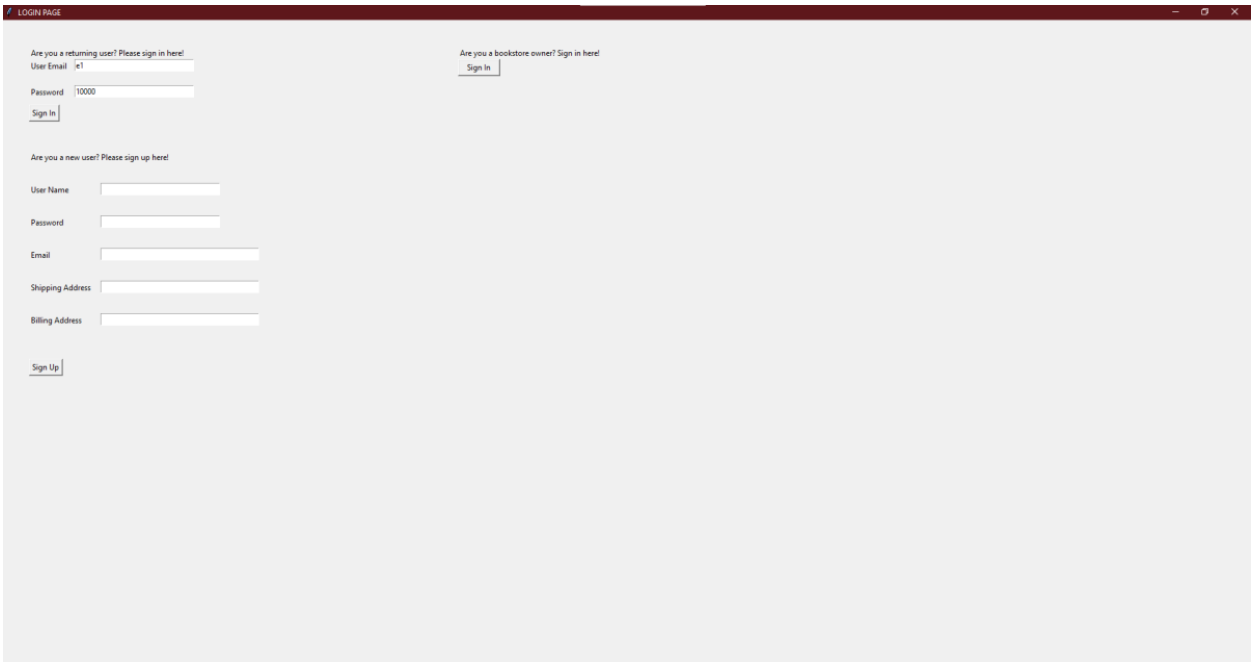
The user wishes to track their order:



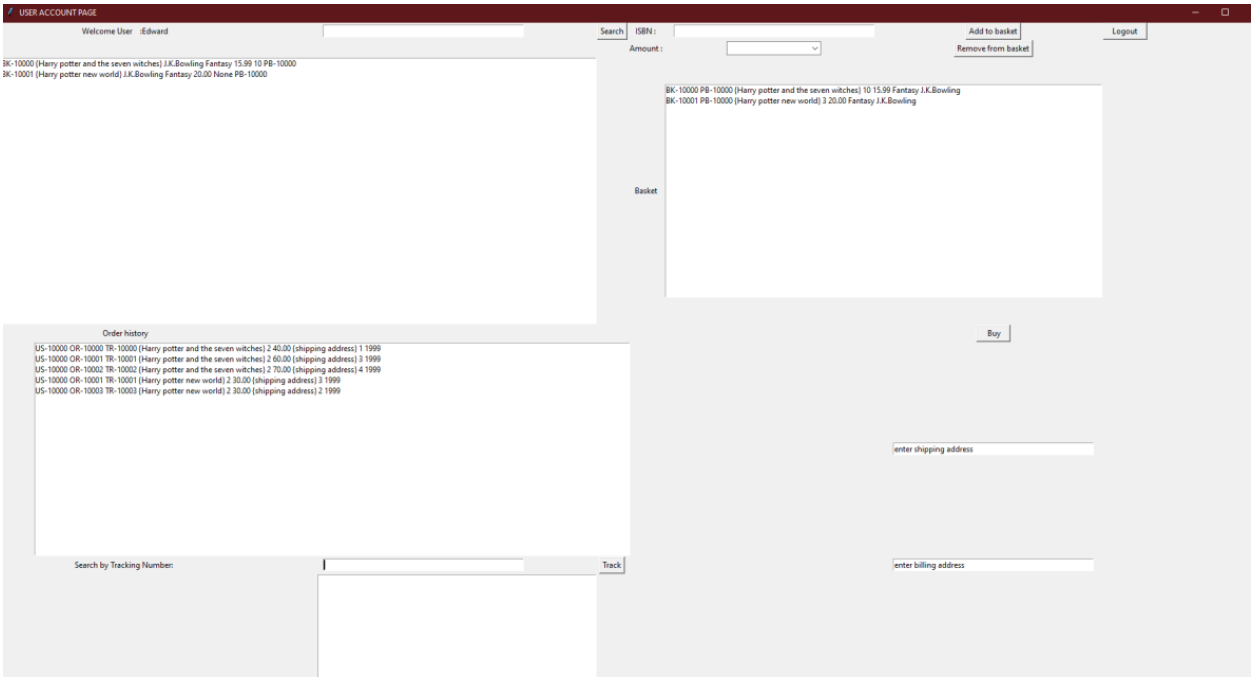
The user logs out of their account:



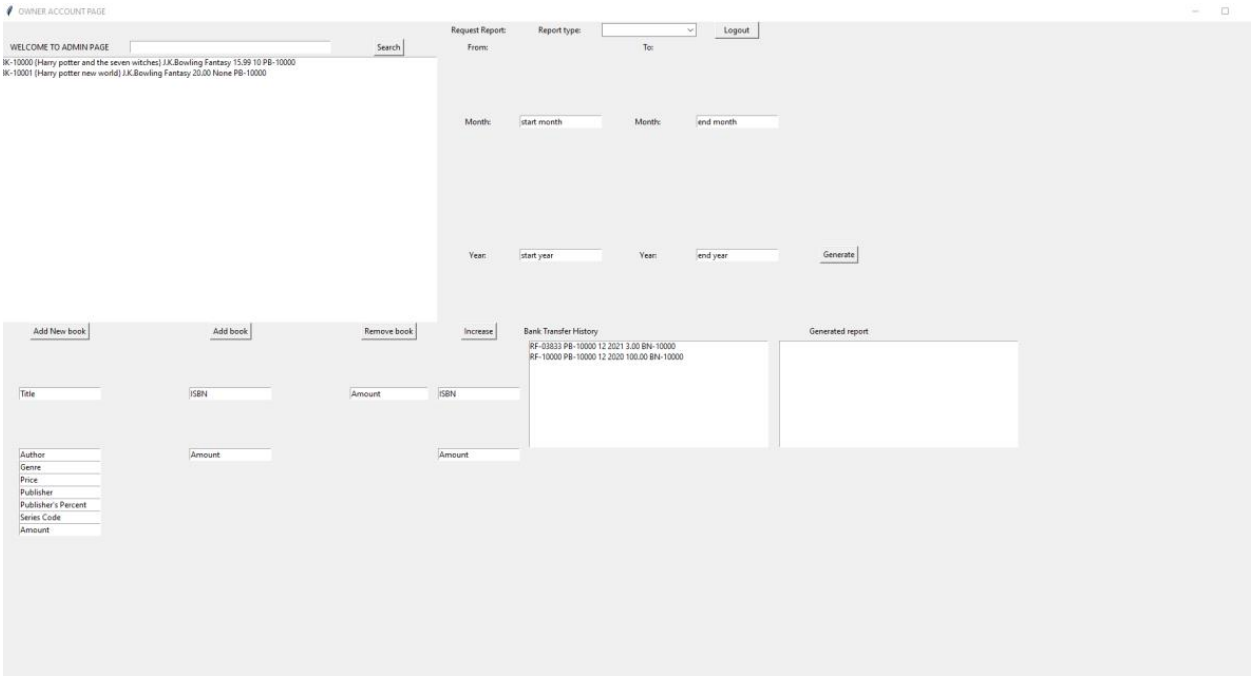
The user logs in with dummy/tester data:



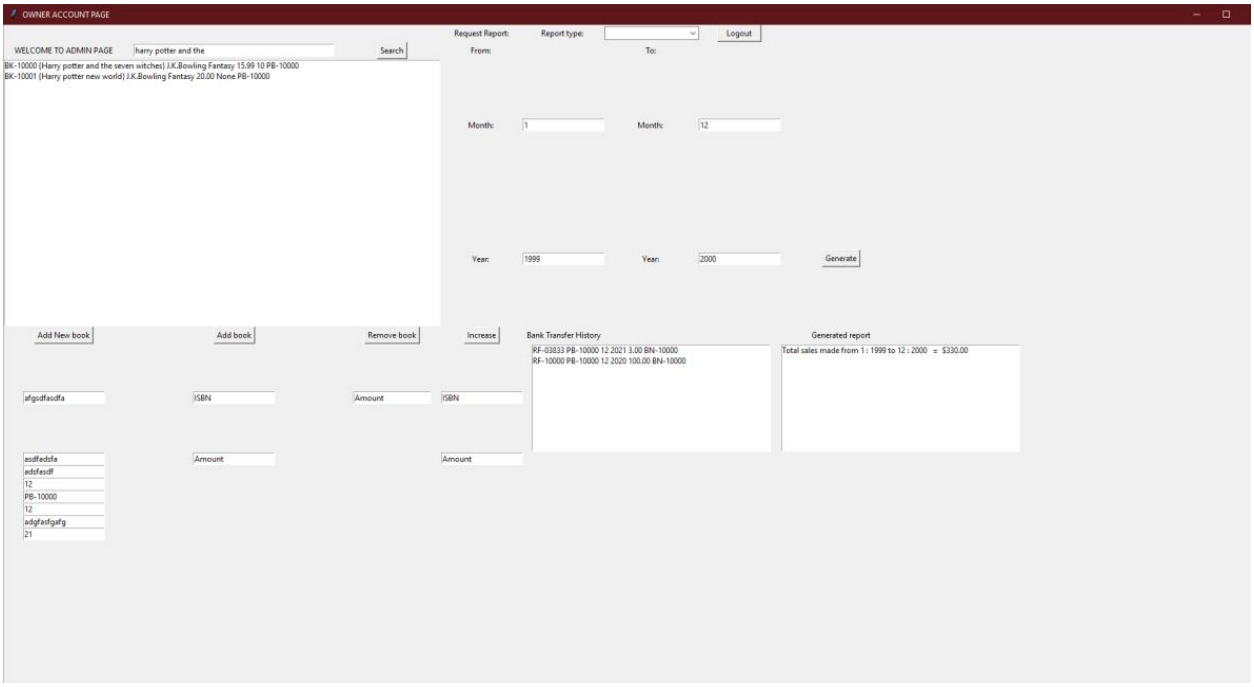
Successful login:



Bookstore owner signs in:



The owner search a book by partial name, then add new book, then generate sales report:



Bonus Features

Firstly, in order to ensure that the application provides a satisfactory user experience, three graphical user interfaces were created; one to represent the log-in screen, one specifically to allow the user to carry out procedures, and one to allow the owner to carry out procedures. Within the UserUI, order history can be displayed while in the OwnerUI, bank transfers can be displayed. We implemented additional functionality in terms of what users and owners are able to do with the data that is present in the individual entities. This data is explicitly output within the GUI in a clear format. Another bonus feature is that the bookstore owner is able to search via partial name (that is to say, the full title of the book does not need to be typed for it to be found).

GitHub Repository

This is the link to the Github Repository, please examine the **main** branch

Name: COMP3005Project

Link: https://github.com/sarahmadelia/COMP3005Project

If there are any access issues, please contact:

Sarah Abdallah (sarahabdallah@cmail.carleton.ca)

Edward Akapo (edwardakapo@cmail.carleton.ca)

Appendix I

Both team members are available on December 20th between 2:00 pm and 5:00 pm for any 20-minute slot within this interval.