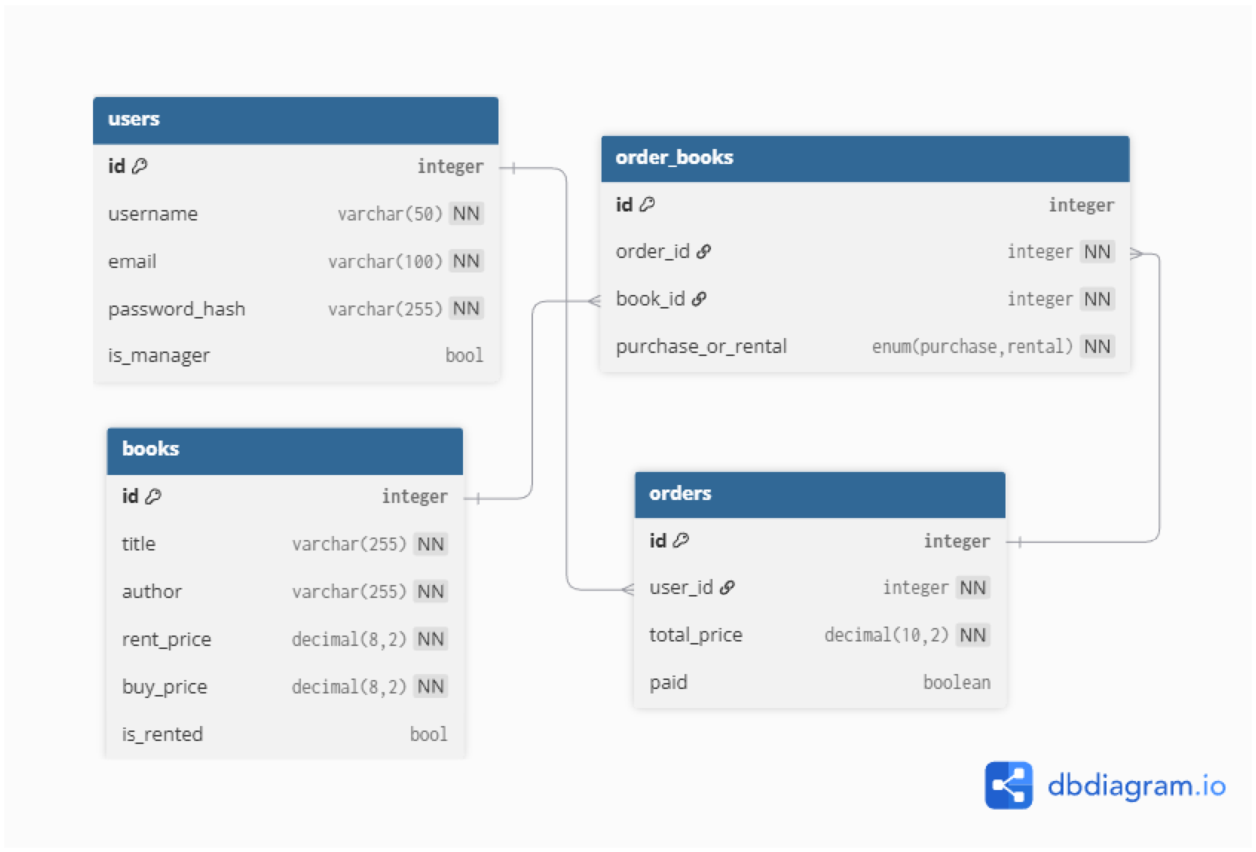
CSCE 310 Course Project

Database Design



To implement the functional requirements for this project, I created four tables for my database. The three main tables are users, books, and orders. There is also a join table between books and orders called order_books. Some notable relationships in my design are orders has a many-to-one relationship with users because one user can have multiple orders. The books and orders table have a many-to-one relationship, which is why there is a join table.

In the future, more attributes could be added to the books table, such as genre and year published, which would allow for more advanced searches. In addition, there could be an inventory attribute which would be an integer that tracks how many books are in the current inventory. Each time a book is ordered, this number increments down by one, and if the inventory is 0, the book will return but be displayed as “Not Available”.

Application Design

This application utilizes a client-server architecture, where the client was built using JavaFX and the server was built using Java Spring Boot. This allows the client to interface with the database through the server.

Beginning with the server, I decided to use MVC architecture to separate business logic from the endpoints. The database tables are represented as models, each being a Java object. All endpoints are stored in controllers, while all business logic is done by services. Database queries are done through repositories, further breaking down the logic of an API call. An example API call flow would be: The controller endpoint is called which calls the corresponding service. The service handles all the business logic and queries the database using the corresponding repository. Finally, the service returns the necessary information to the controller, which in turn sends this back to the client. In addition, DTOs are stored in a types directory, making API responses cleaner and uniform. All API responses use a general ApiResponse object, where the data attribute can be used to store the actual DTO being returned. All routes except for login and register are protected using a security filter chain, ensuring that unauthenticated users cannot access bookstore data.

The client handles the UI and calling the server for information. The UI was created using fxml files, which are like html, but allow elements to be manipulated by a corresponding Java controller. Controllers are used to both update frontend information and call the server API. Once the user is authenticated as either a manager or customer, the client stores the JWT token and whether the user is a manager or not in a global UserSession object. This way all controllers can pass the auth token in every controller and redirect the user to the correct page. All API calls are done as Tasks, allowing asynchronous calls that don’t freeze up the client UI. This ensures the client is responsive and user-friendly.

API Design

AuthController: /api/auth

The AuthController has two endpoints: *register* and *login*. There is no logout endpoint because the client will delete the JWT authentication token when the user clicks the logout button.

1. POST /register

- If the request is valid, the controller will call UserService.registerUser which will hash the password, save the user information to the database, and return the newly registered username.
- Request Body:

```
{
  "username": string,
  "email": string,
  "passwordHash": string,
  "manager": boolean
}
```

- Response:

```
{
  "success": boolean,
  "message": string,
  "data": {
    string (username)
   }
}
```

2. POST /login

- If the user credentials are valid, the jwtService will generate a auth token that will expire after 1 day. The endpoint returns the auth token and whether the user is a manager, so the client knows where to redirect the authenticated user.
- Request Body:

```
{
    “username”: string,
    “passwordHash”: string
}
```

- Response:

```
{
  "success": boolean,
  "message": string,
  "data": {
    “token”: string,
    “isManager”: boolean
   }
}
```

BookController: /api/books

The BookController handles searching by title or author keywords as well as returning a list of all books available for the user to purchase. All endpoints associated with the BookController require authentication.

1. GET /

- Queries all books from the database and returns them as a list of Book objects
- Request Body: none
- Response:

```
{
  "success": boolean,
  "message": string,
  "data": {
    List<Book>
   }
}
```

2. GET /search

- Returns all books that match the keyword by either title or author name.
- Request Body: keyword
- Response:

```
{
  "success": boolean,
  "message": string,
    "data": {
      List<Book>
    }
}
```

OrderController: /api/order

The OrderController handles placing orders. This includes single orders as well as being able to handle renting and buying multiple books at once. These are all done under a single endpoint, which is protected by authentication.

1. **POST /**

- Calls the OrderService object which adds the order to the orders table, adds an entry to the order_books join table, and updates books that are rented to reflect this change. Also sends an email using JavaMail with the order to the user's registered email. The controller returns the order made as a Java object.
- Request Body: Json object where the key is the bookId and the value will be either 0 or 1, 0 being buy and 1 being rent
- Response:

```
{
    "success": boolean,
    "message": string,
    "data": {
      BookOrderResponse
    }
}
```

ManageBookController: /api/books/manager

The ManageBookController handles all manager functions that relate to the book inventory, this includes: viewing all books, adding a new book, updating a book's information, and deleting a book from the database. All POST and DELETE endpoints require a book object to be passed in the body, and they all return that book's new information. All endpoints associated with the ManageBookController require authentication.

ManageOrderController: /api/order/manager

The ManageOrderController allows the manager to view all orders, including itemized lists of the items purchased, as well as updating an order's paid status. The update endpoint allows multiple orders to be updated at once. All endpoints associated with the ManageOrderController require authentication.