Library Management System Project

Iteration 3 – FINAL ITERATION

Submitted by: Arshnoor Batra (434007284) and Varun Sahni (834006628)

Introduction:

This Server-Client project is a comprehensive implementation of a microservices architecture designed to efficiently manage and interact with various entities in a library system. The project is structured as a collection of microservices that handle distinct functionalities, ensuring scalability, flexibility, and ease of maintenance.

Server Components:

1. **User Server (Port 5057):**
    a. Manages user-related operations such as user authentication.
    b. Utilizes a microservice approach to handle user data retrieval.

2. **Book Server (Port 5058):**
    a. Manages book-related operations including retrieval, creation, and modification.
    b. Implements RESTful services for seamless integration with clients.

3. **OrderBook Server (Port 5059):**
    a. Handles order-related processes, such as order creation and receipt generation.
    b. Utilizes microservices for managing orders, students, and receipts.

Client Components:

The project includes two distinct clients, each tailored for specific interfaces:

1. **Desktop Application Client:**
    a. Developed in Java, the desktop client interacts with the microservices via HTTP requests.
    b. Facilitates functionalities such as updating book details, placing orders, and retrieving student information.
    c. Employs a socket connection to communicate with the main server, showcasing versatility in communication protocols.

2. **Web Browser Client:**
    a. Comprising HTML files (index.html, loadsave.html, login.html, manage.html, and orderbook.html) and a Node.js server (server.js).

b. Offers a user-friendly web interface for accessing library services.
c. Uses asynchronous HTTP requests to interact with the microservices, enabling seamless integration with modern web browsers.

DataAdapter for Remote Communication:

The **'RemoteDataAdapter'** class acts as a crucial bridge between the desktop application and the microservices. It provides methods for connecting to the main server, updating book and order details, and retrieving user and student information.

Key Features:

1. **Microservices Architecture:** The project adopts a microservices architecture, enhancing modularity and enabling independent development and deployment of each service.
2. **Java Desktop Client:** The desktop client, written in Java, showcases the versatility of the microservices by utilizing both HTTP and socket-based communication.
3. **Web Browser Client:** The web client provides an intuitive and accessible interface for users to interact with the library system.
4. **Data Access and Transfer:** The RemoteDataAdapter class ensures smooth communication between the desktop client and the microservices, employing HTTP requests for data retrieval and updates.

Conclusion:
This Server-Client project illustrates the power and adaptability of microservices in building a robust and scalable library management system. It combines a desktop application and a web client to cater to diverse user preferences, showcasing the flexibility and efficiency of the microservices architecture in modern software development.

ASSIGNMENT QUESTIONS:

Ques 1) Requirements: Description with UI sketch of all use cases. Use cases should be described with the standard format including a short description and a sequence of user actions and system reactions. Each user action or system reaction might include a UI sketch.

Answer 1) All use cases of the library management are described below:

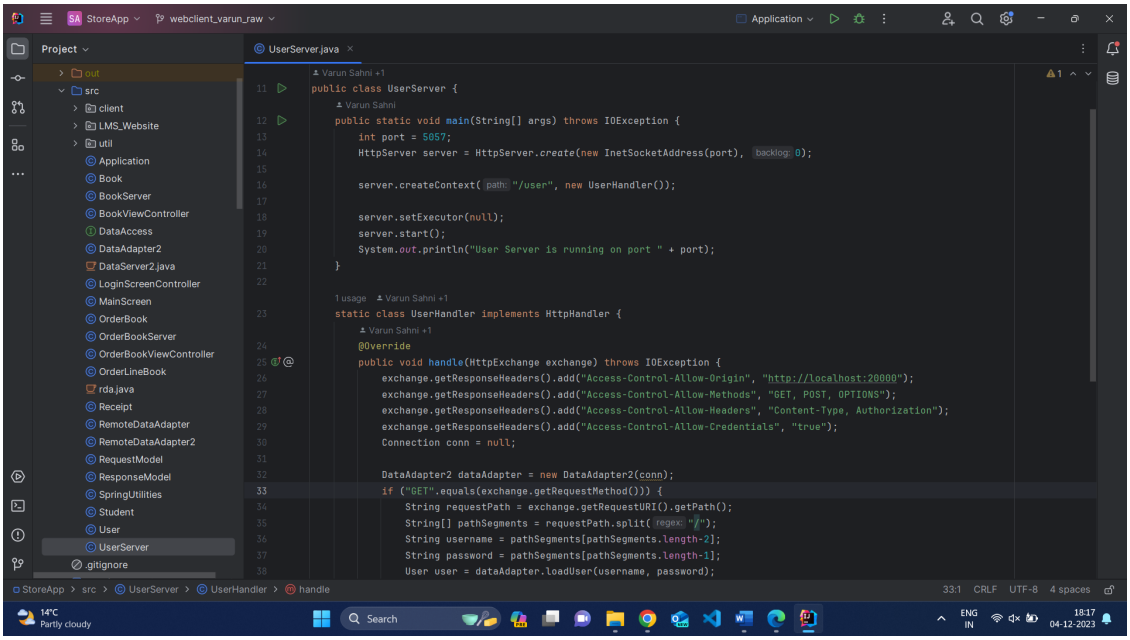
1st use case:

<table>
<tr><td>Use Case</td><td>Login</td></tr>
<tr><td>Actor</td><td>Administrator</td></tr>
<tr><td>Description</td><td>The administrator shall be able to log in to the library management system application.</td></tr>
<tr><td>Precondition</td><td>The username and password of the administrator asking to be logged in should be present in the Redis database so that it can be matched against it.</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The User server will be started to process the data.</td></tr>
<tr><td>2</td><td>The application will be started to load the UI.</td></tr>
<tr><td>3</td><td>A login screen window will appear on the screen.</td></tr>
<tr><td>4</td><td>The admin will enter login details in the login window.</td></tr>
<tr><td>5</td><td>The admin will click on the login button.</td></tr>
<tr><td>6</td><td>The login details of the admin will be verified against the information present in the database.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>The admin will be able to log into the application.</td></tr>
<tr><td>Comment</td><td>If the login details entered by the admin do not match the database, the “The user does not exist!” window pops up.</td></tr>
</table>

Actor Action	**System Response**
1 Admin enters username	
2 Admin enters a password	
3 Admin clicks on the login button	4 validates the user details
	5 takes admin to the main screen

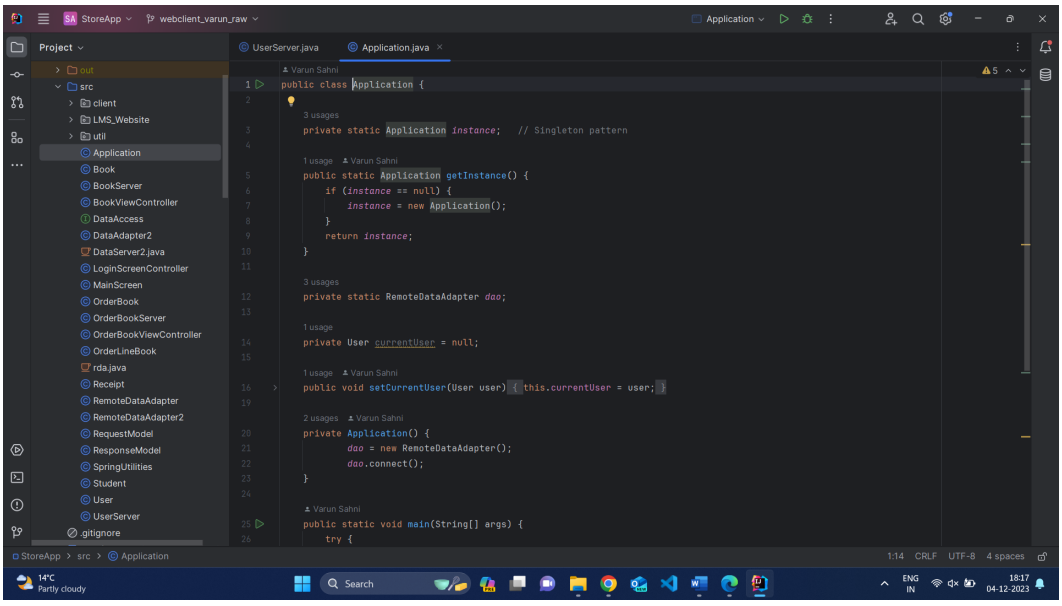
UI Sketches for all the Action Steps:

1)



The user server is loaded to start the application.

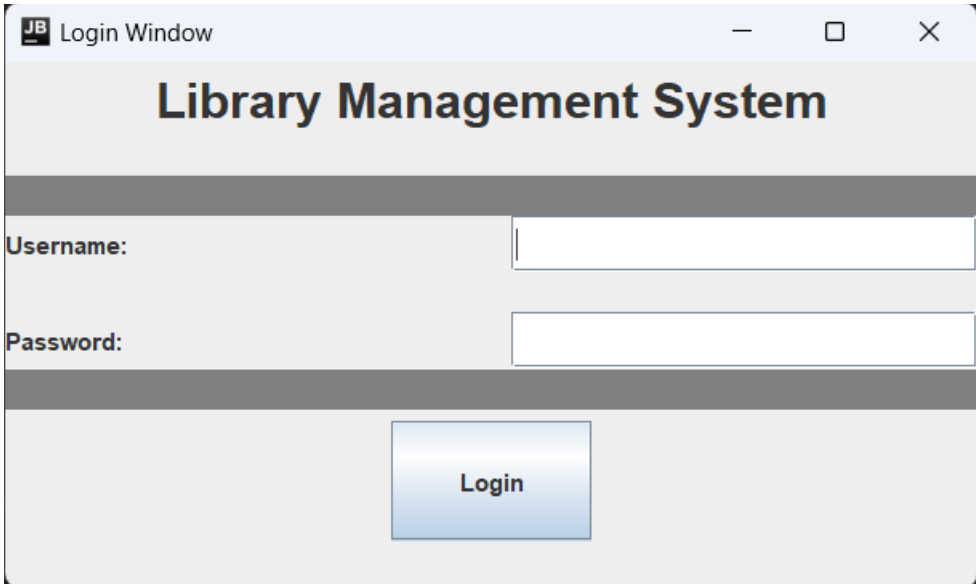
2)



The application class is run to load the User Interface

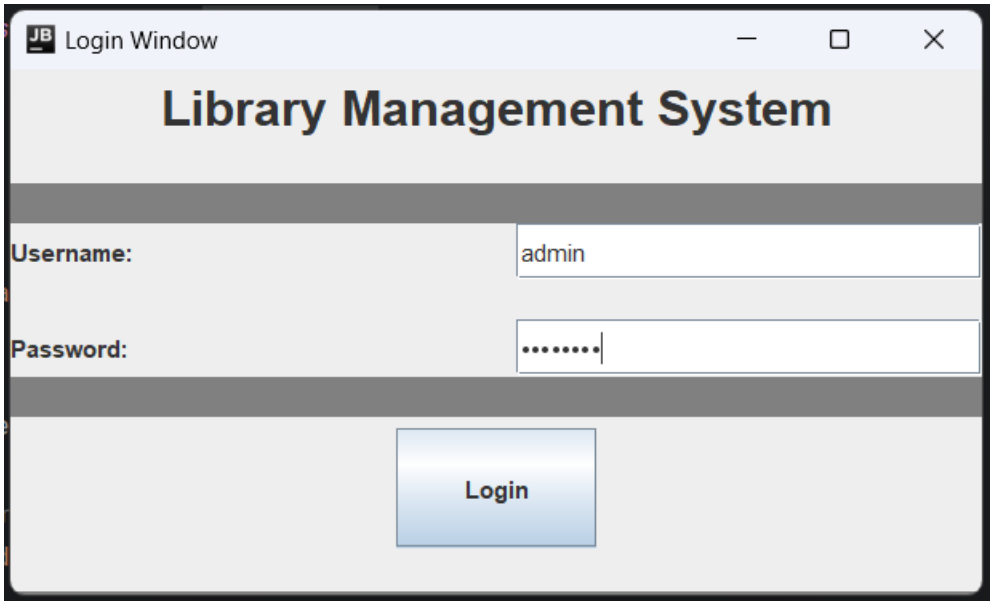
DESKTOP APPLICATION UI

3)



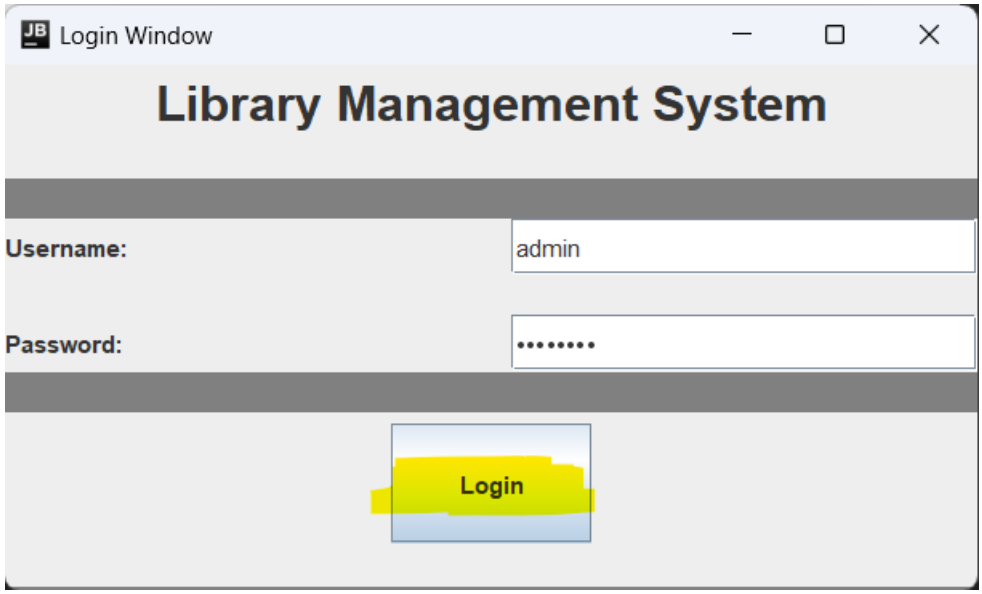
The login window pops up.

4)



The admin will enter login details in the window.

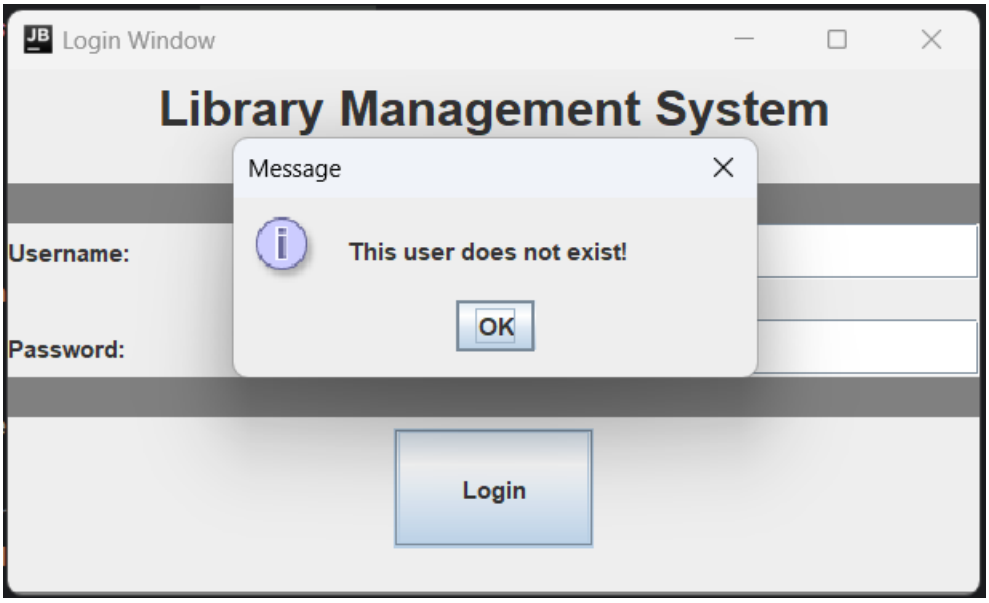
5)



The admin will click on the login button.

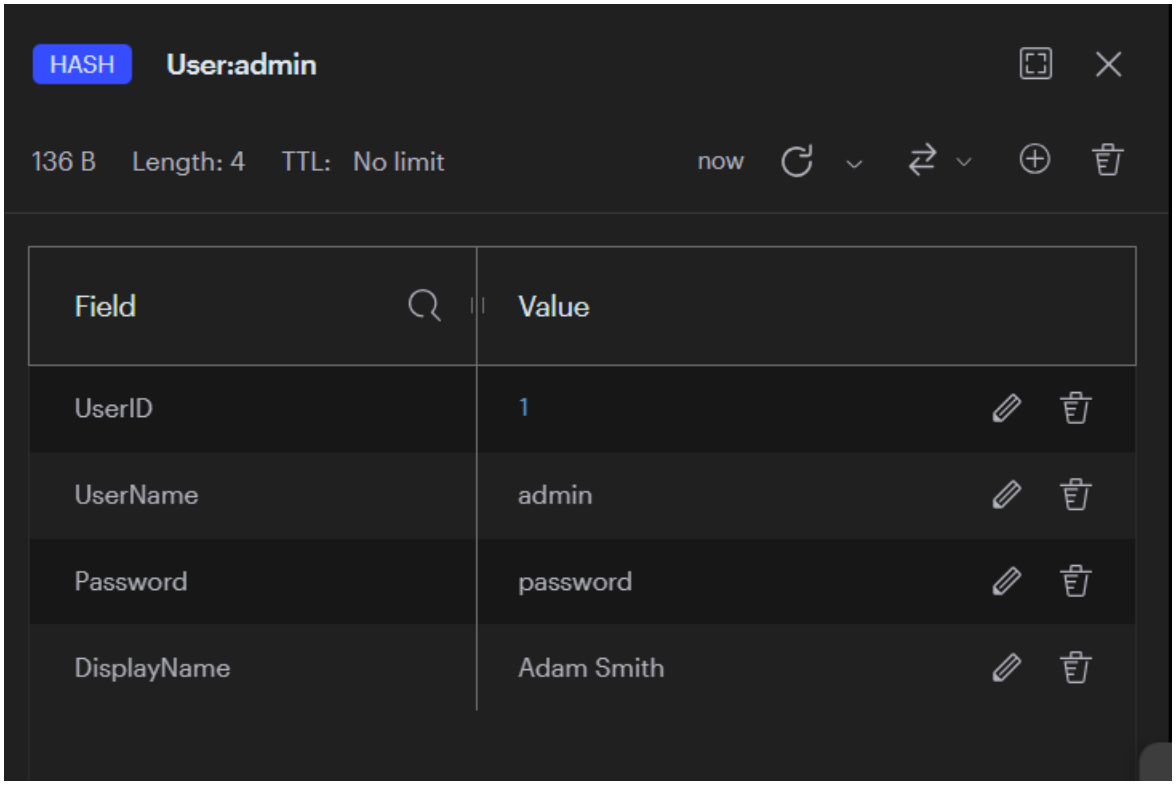
6)

Validation:



In case user detail does not match to the DB, “This user does not exist” window appears.

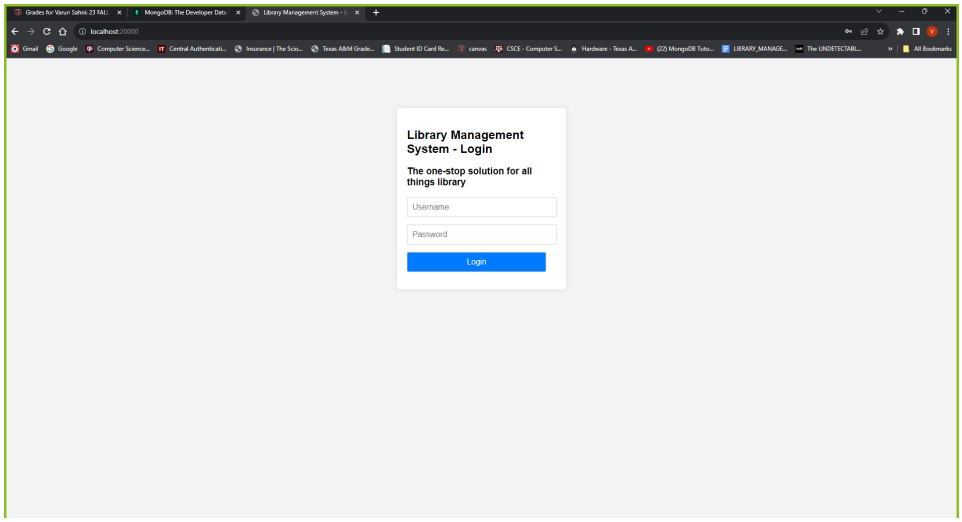
7)



Redis Database

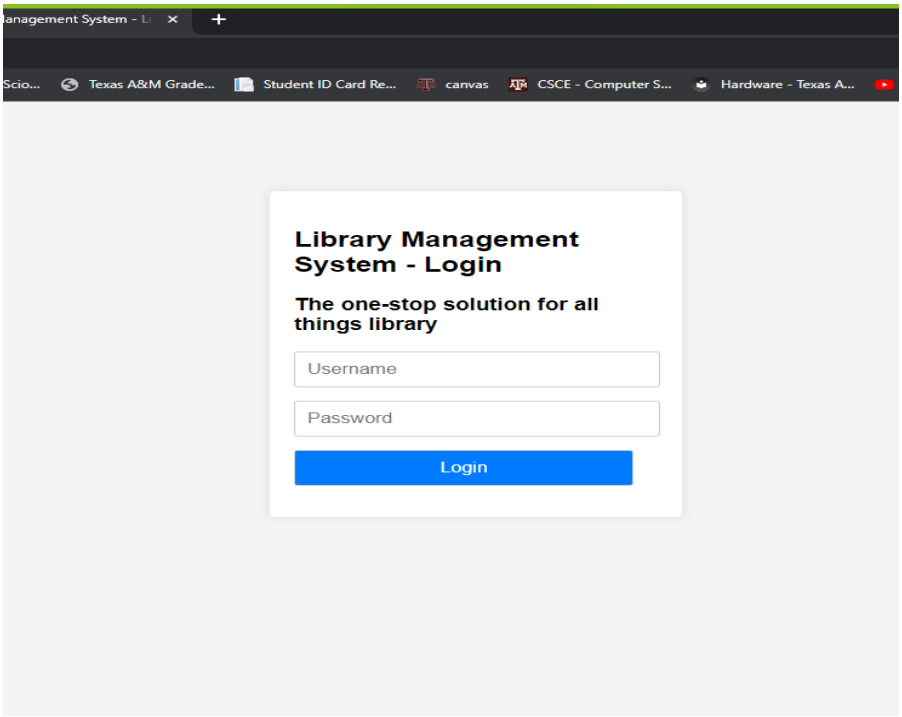
WEB APPLICATION UI

1)



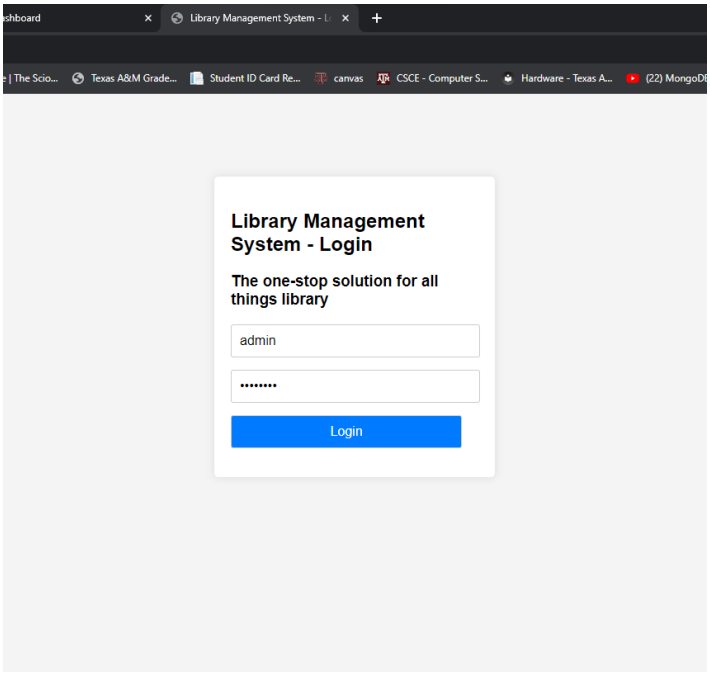
Full Screenshot of web browser

2)



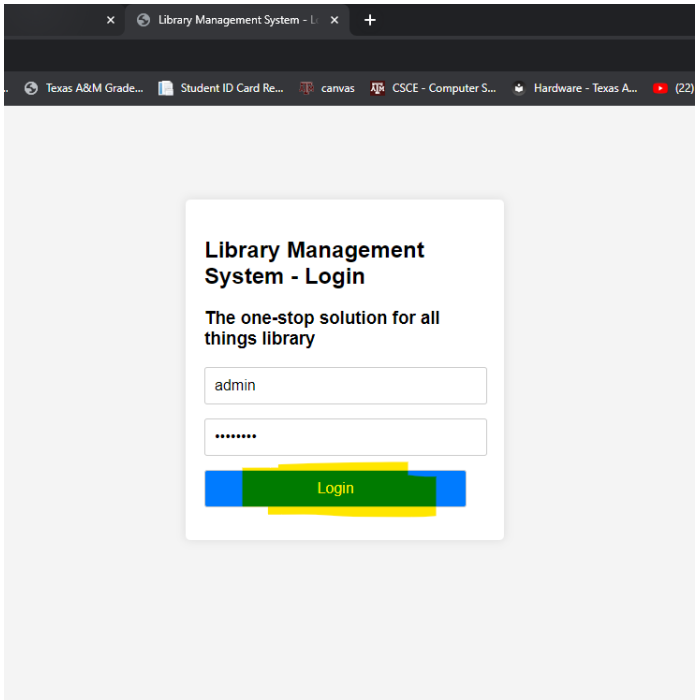
The login page.

3)



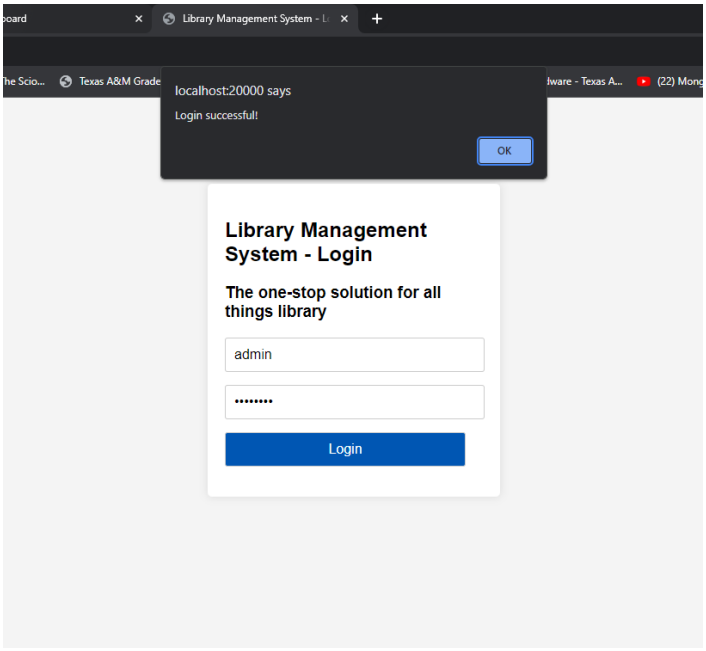
The admin will enter login details.

4)



The admin will click on the login button.

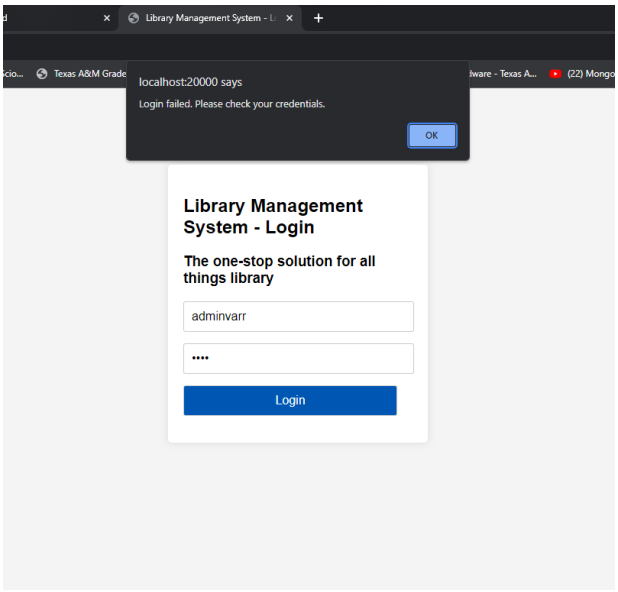
5)



Login Successful alert appears post user validation from UserServer

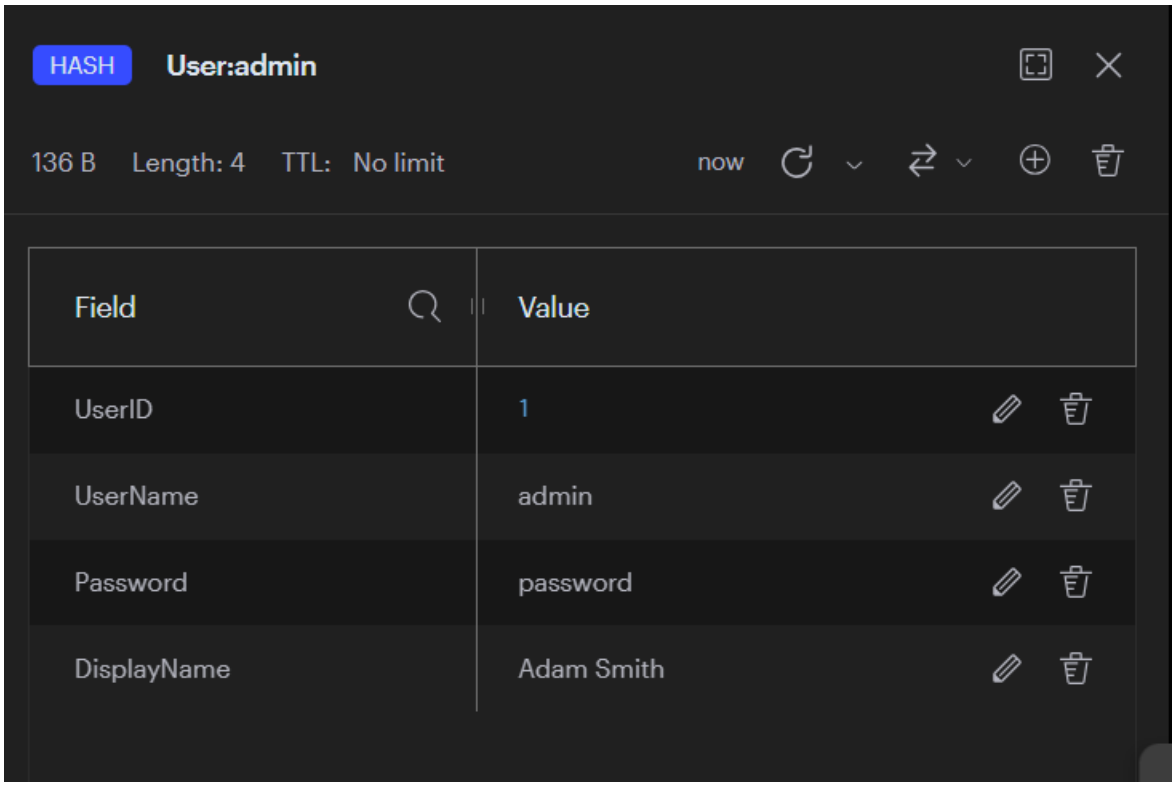
6)

Validation:



The entered login details are incorrect.

7)



Redis Database

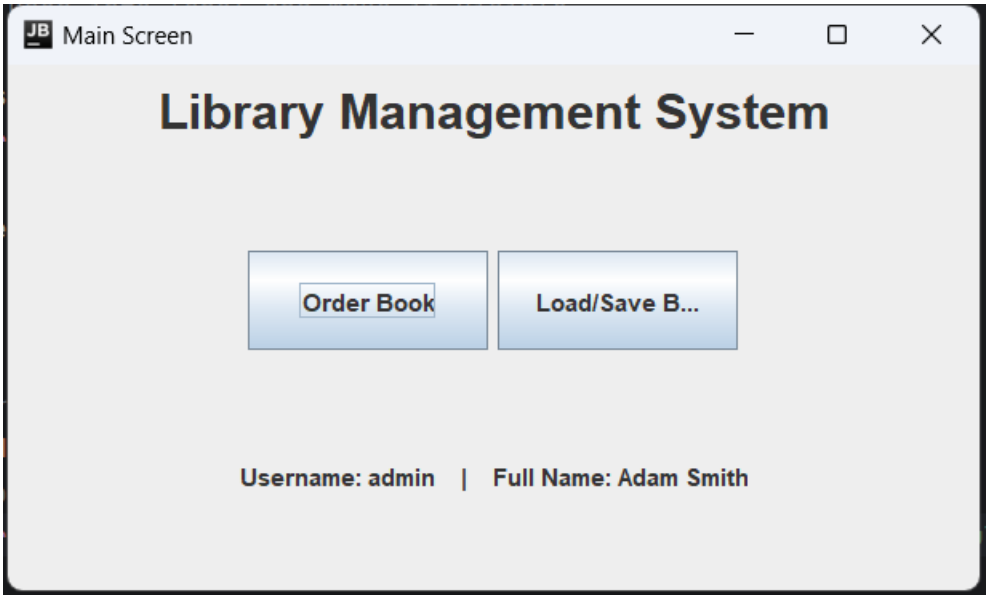
2nd Use Case:

Use Case	**Main Screen**
Description	The administrator shall be able to view the main screen of the library management system along with the details of the administrator. The details shown would contain the Username and full name.
Precondition	The administrator must be logged in to the application.
Ordinary Sequence	Step 1: The main screen will have 2 buttons. Step 2: The first button is for Order Book. Step 3: The second button is for Load/Save Book.
Postcondition	The admin can perform multiple functions in the library management system.

Actor Action	**System Response**
1 The admin can click on Load/Save Book and Order Book.	2 The system will open windows according to the user action.

DESKTOP APPLICATION UI

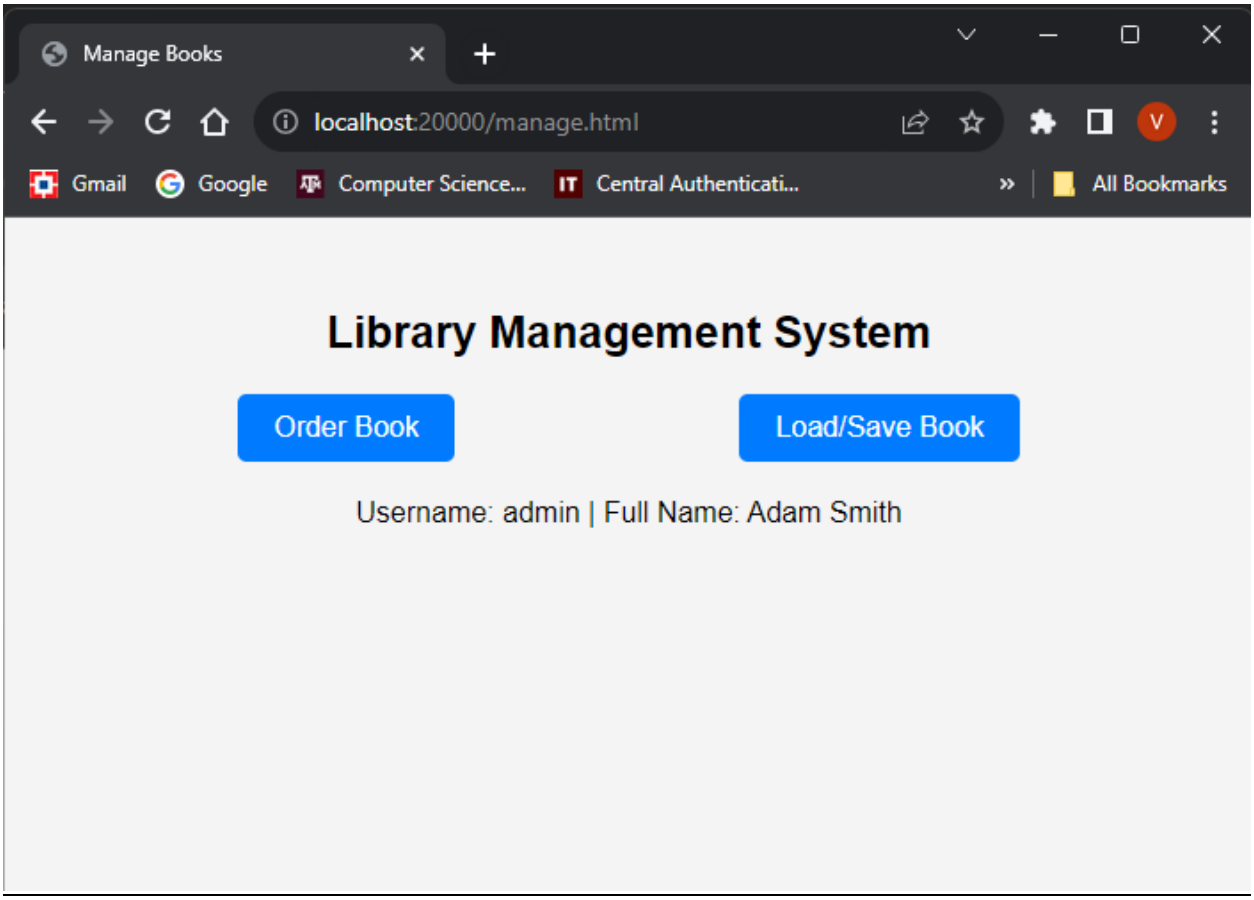
1)



The main screen has 2 buttons: Order Book and Load/Save Book.

WEB APPLICATION UI

1)



The manage books page has 2 buttons: Order Book and Load/Save Book.

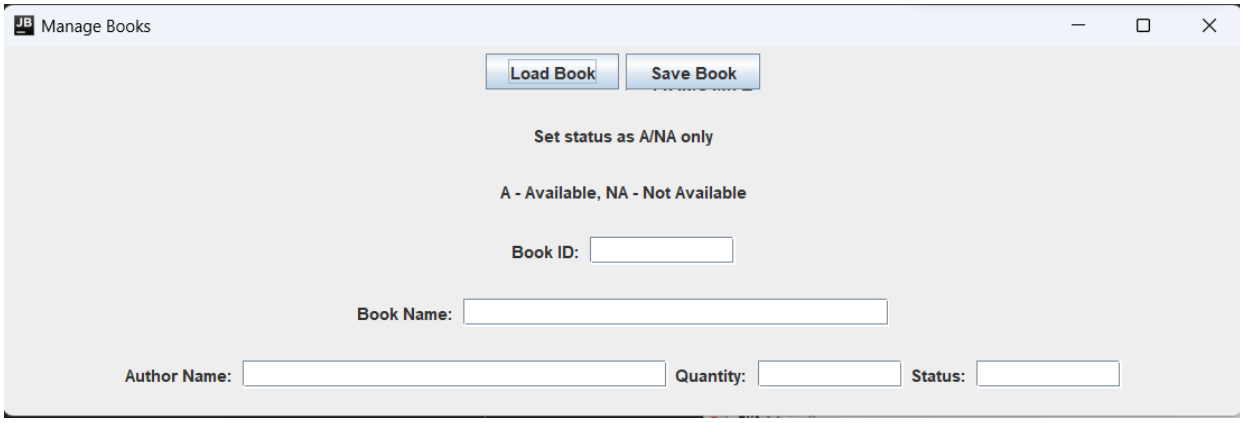
3rd Use Case:

Use Case	**Save Book**
Description	The administrator shall be able to save a book to the database of library management system.
Precondition	The main screen should be visible to the administrator.
Ordinary Sequence	<table><tr><td>Step</td><td>Action</td></tr><tr><td>1</td><td>The administrator will click on the Load/Save book button on main screen.</td></tr><tr><td>2</td><td>The administrator would enter all the book details on the Manage Books window.</td></tr><tr><td>3</td><td>The details include BookID, Book Name, Author Name, Quantity, and Status.</td></tr><tr><td>4</td><td>After entering the details, the administrator can save the book by clicking the Save Book button.</td></tr></table>
Postcondition	The entered book details will be saved in the Redis Database.
Comment	If the BookID corresponding to a particular book is present in the database, then that BookID would be overwritten.
Exceptions	If the BookID is negative or zero, a system pop-up will appear alerting the admin to enter a valid Book ID.

Actor Action	**System Response**
1 The admin clicks on the Save Book button from Main Screen window.	2 The “Manage Books” window appears.
3 Admin types in the book details like BookID, Book Name, Author Name, Quantity, Status.	
4 The admin clicks on “Save Book” button.	5 The system responds with “Book Saved Successfully” dialog box.
	6 The book details get saved in the database.

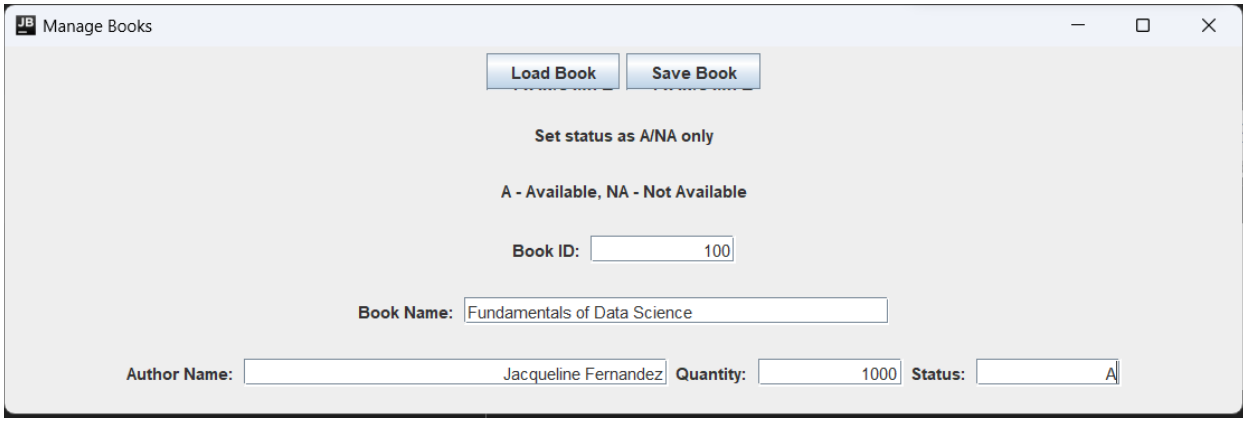
DESKTOP APPLICATION UI

1)



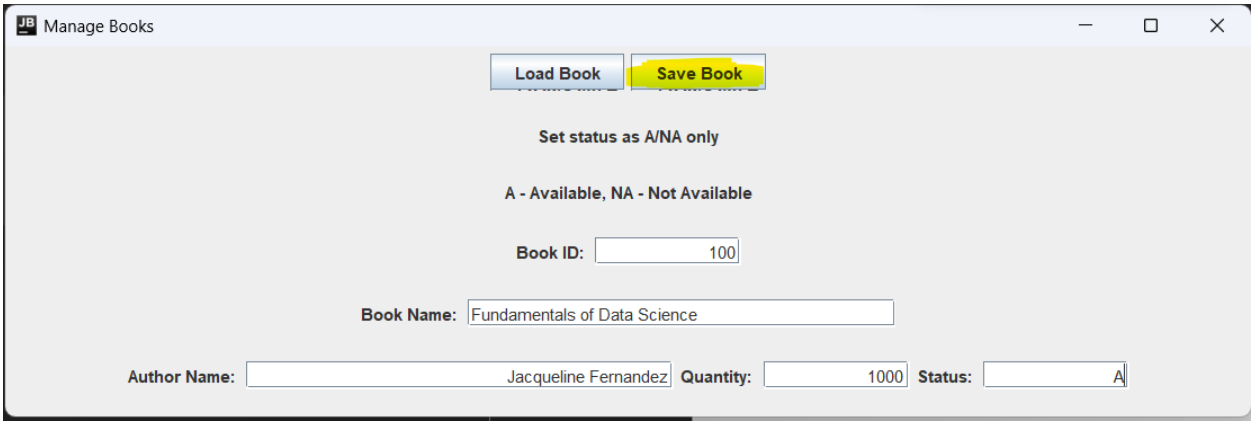
Manage Books window of Desktop Application.

2)



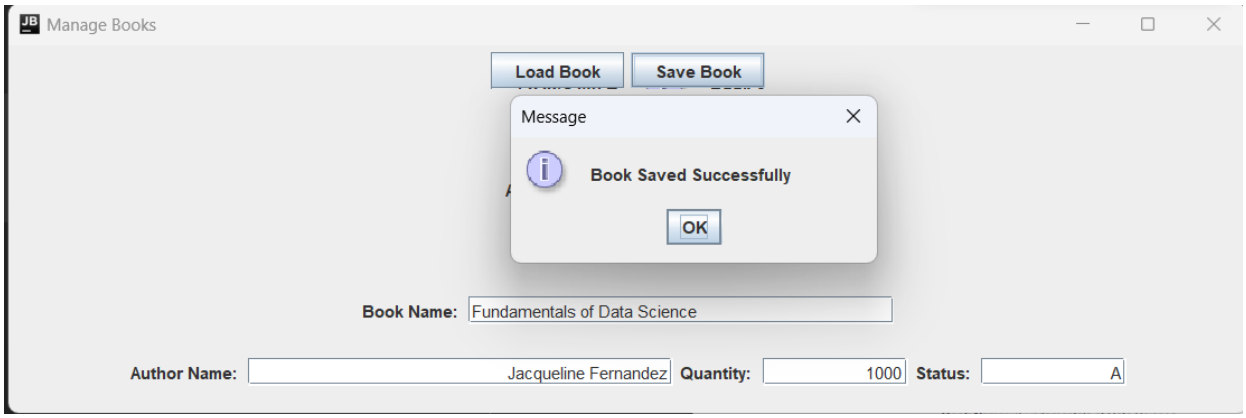
The details have been entered in the Manage Books window.

3)



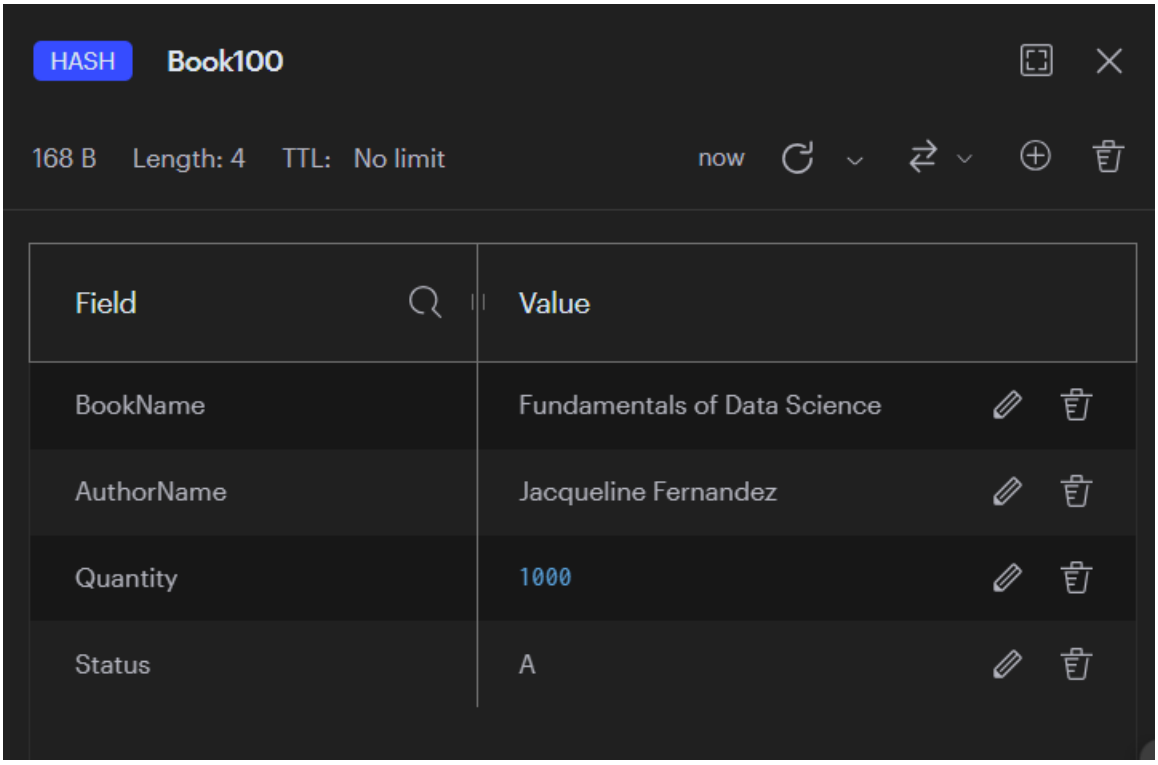
The save Book button is clicked.

4)



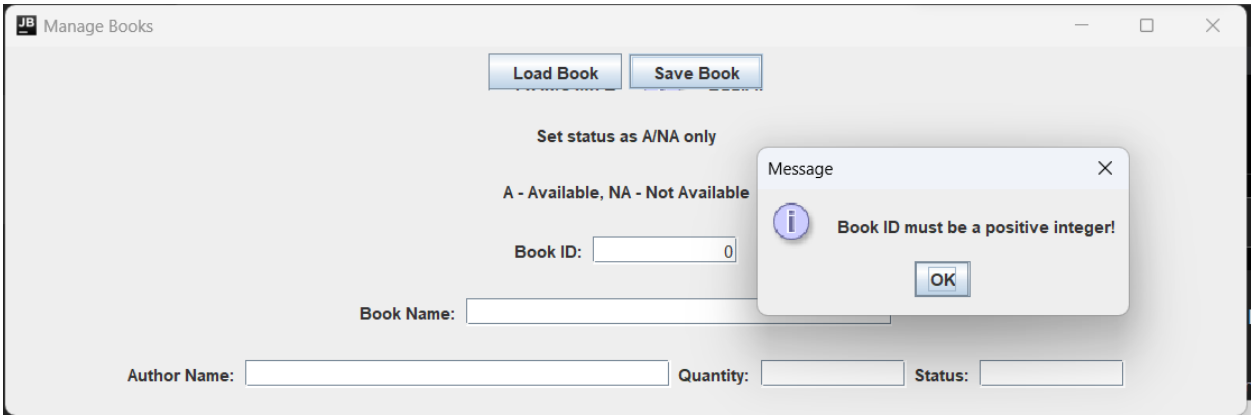
The “Book Saved Successfully” pop up window appears.

5)

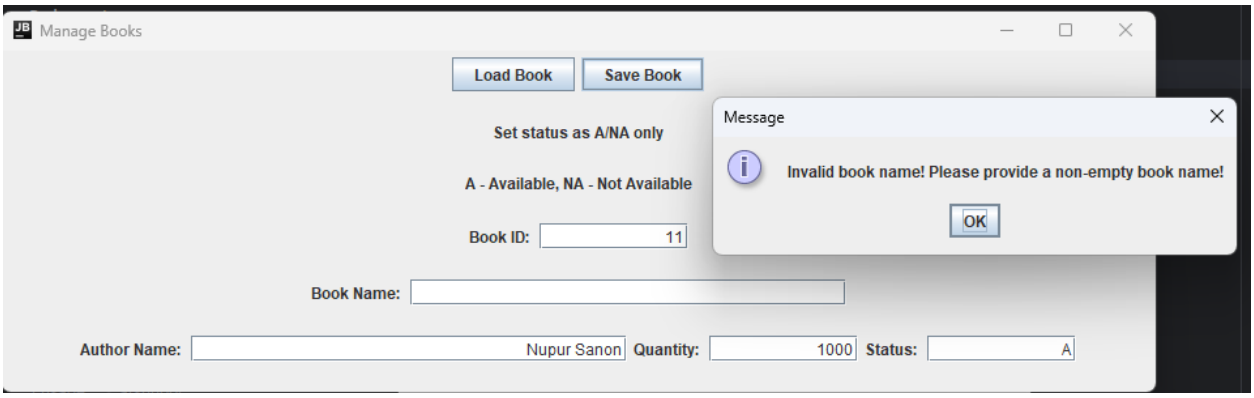


The book details are saved in the Redis database table.

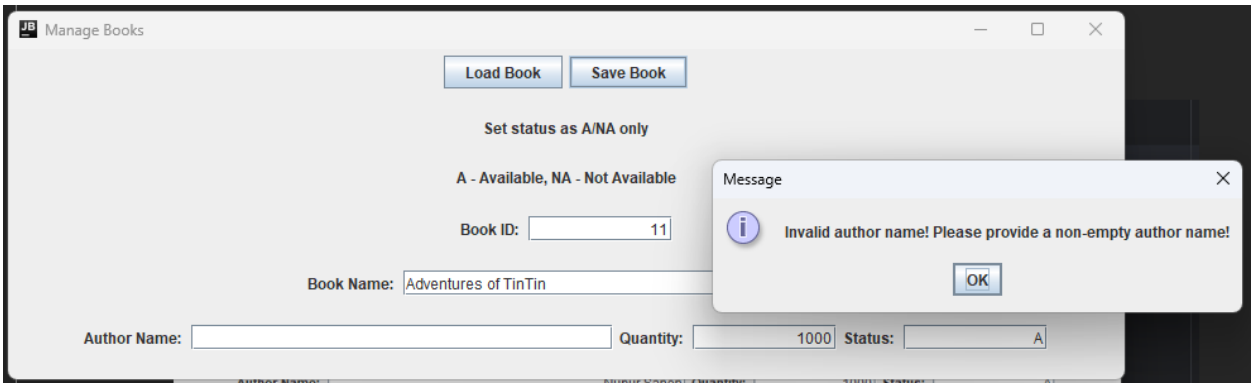
6) Validations:



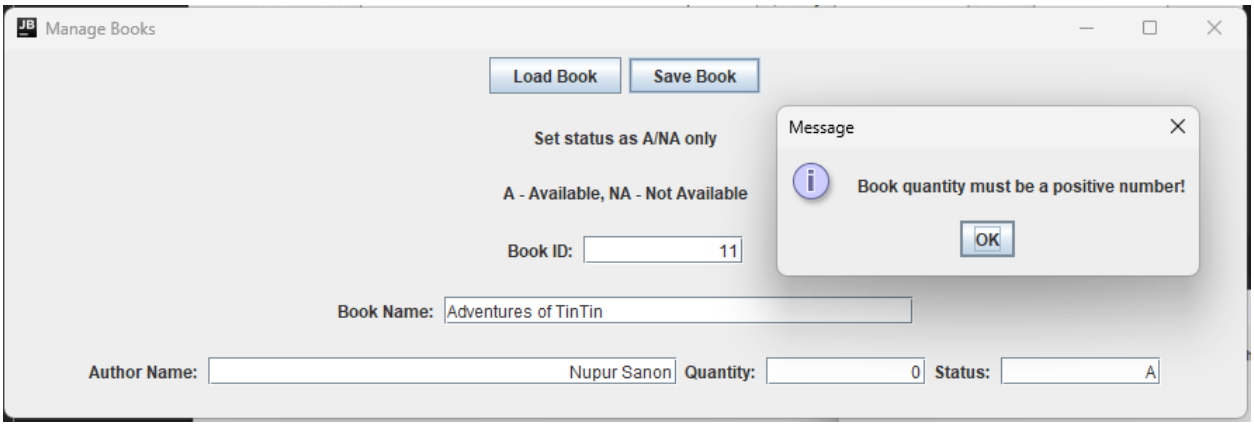
Book ID should be positive.



Book Name cannot be empty



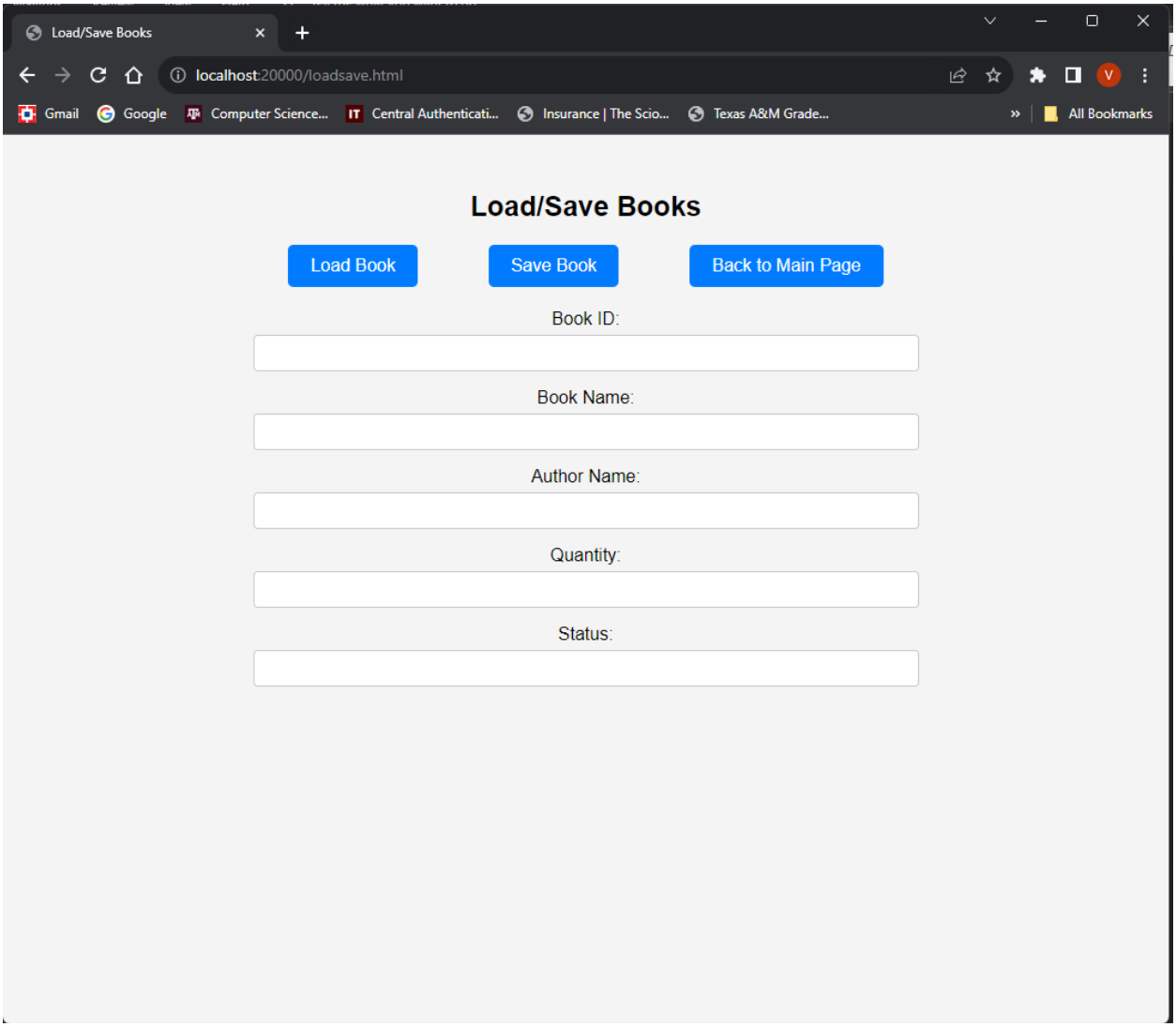
Author Name cannot be empty



Quantity must be positive

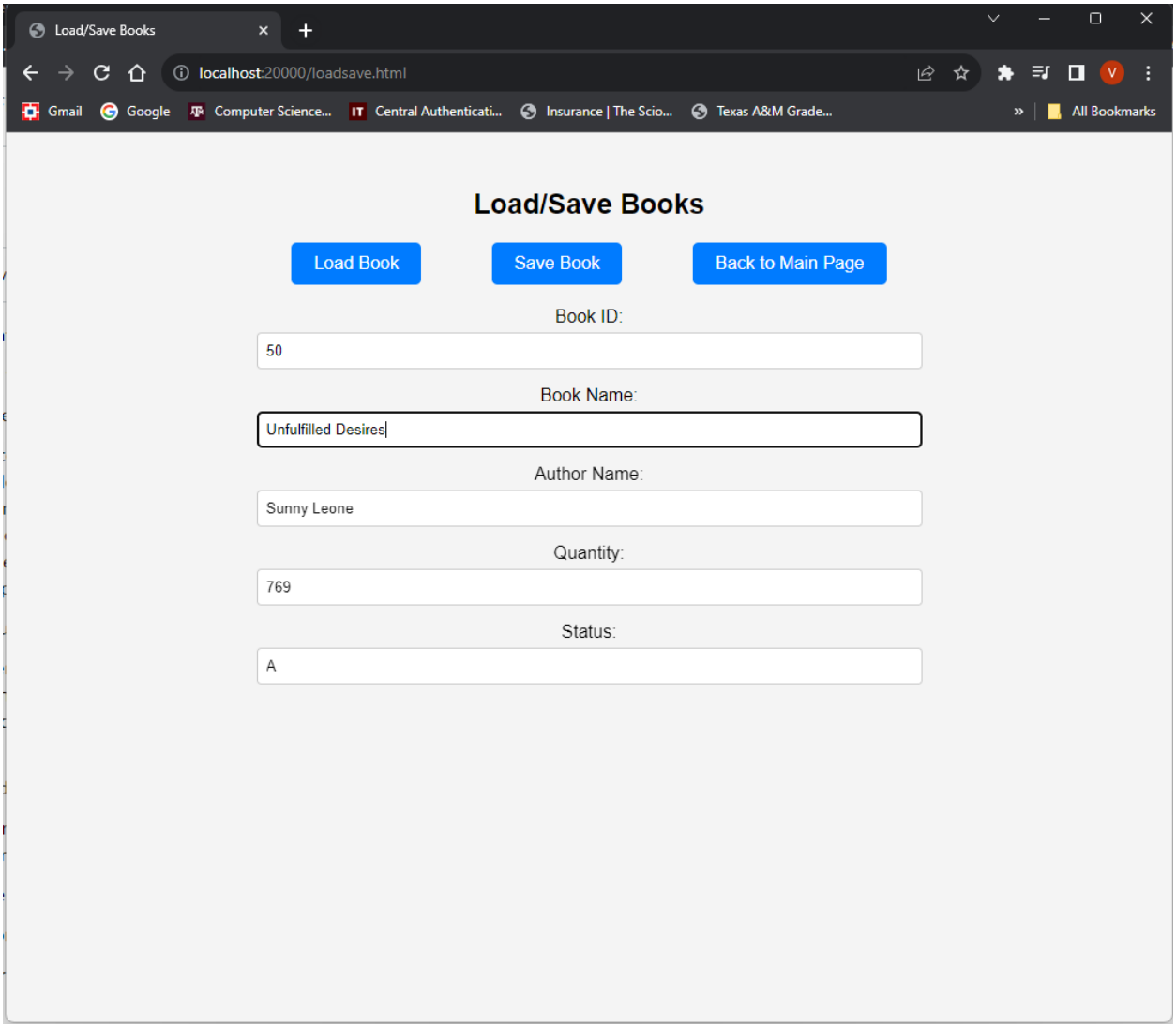
WEB APPLICATION UI

1)



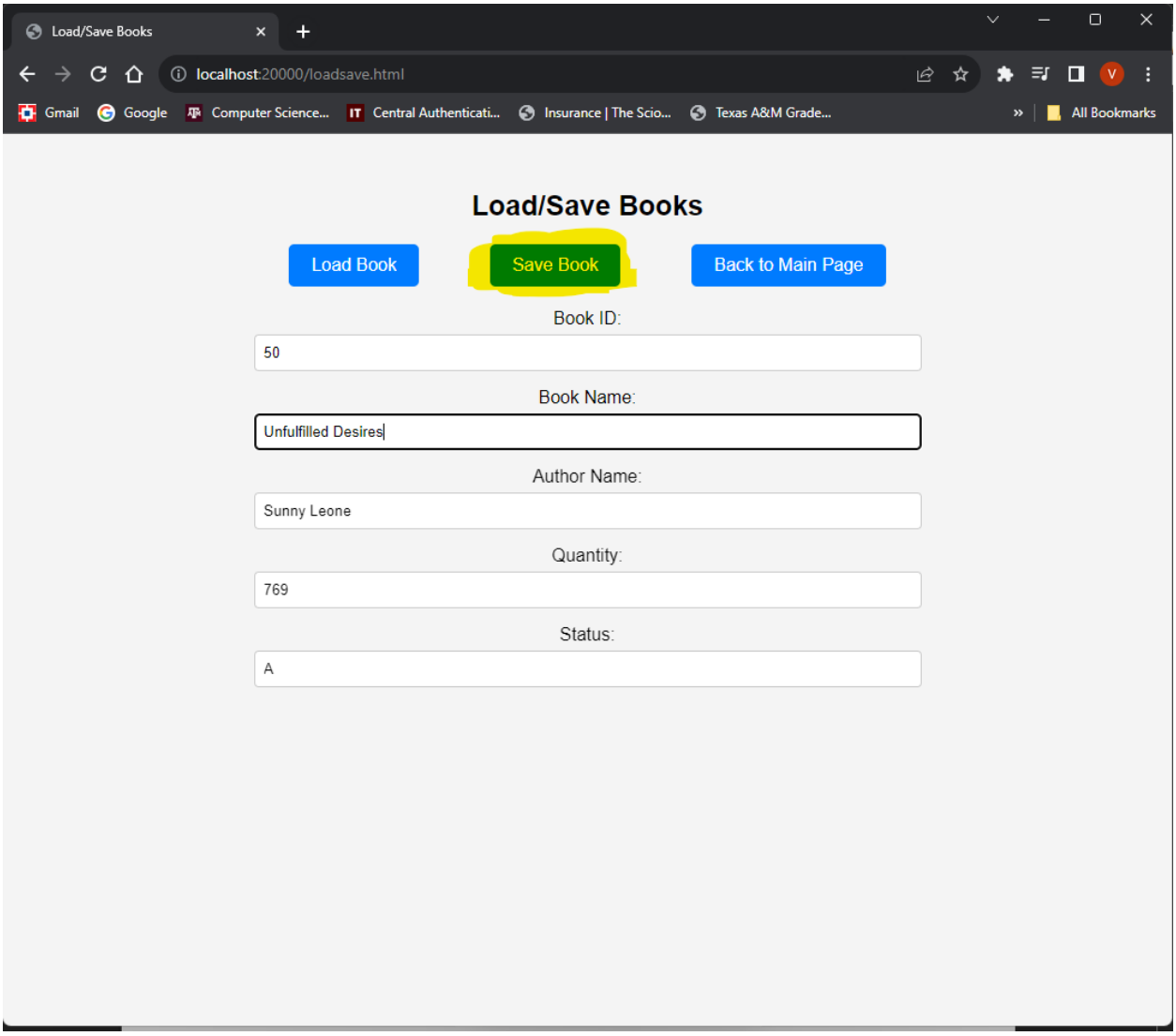
Load/Save Books page of Desktop Application.

2)



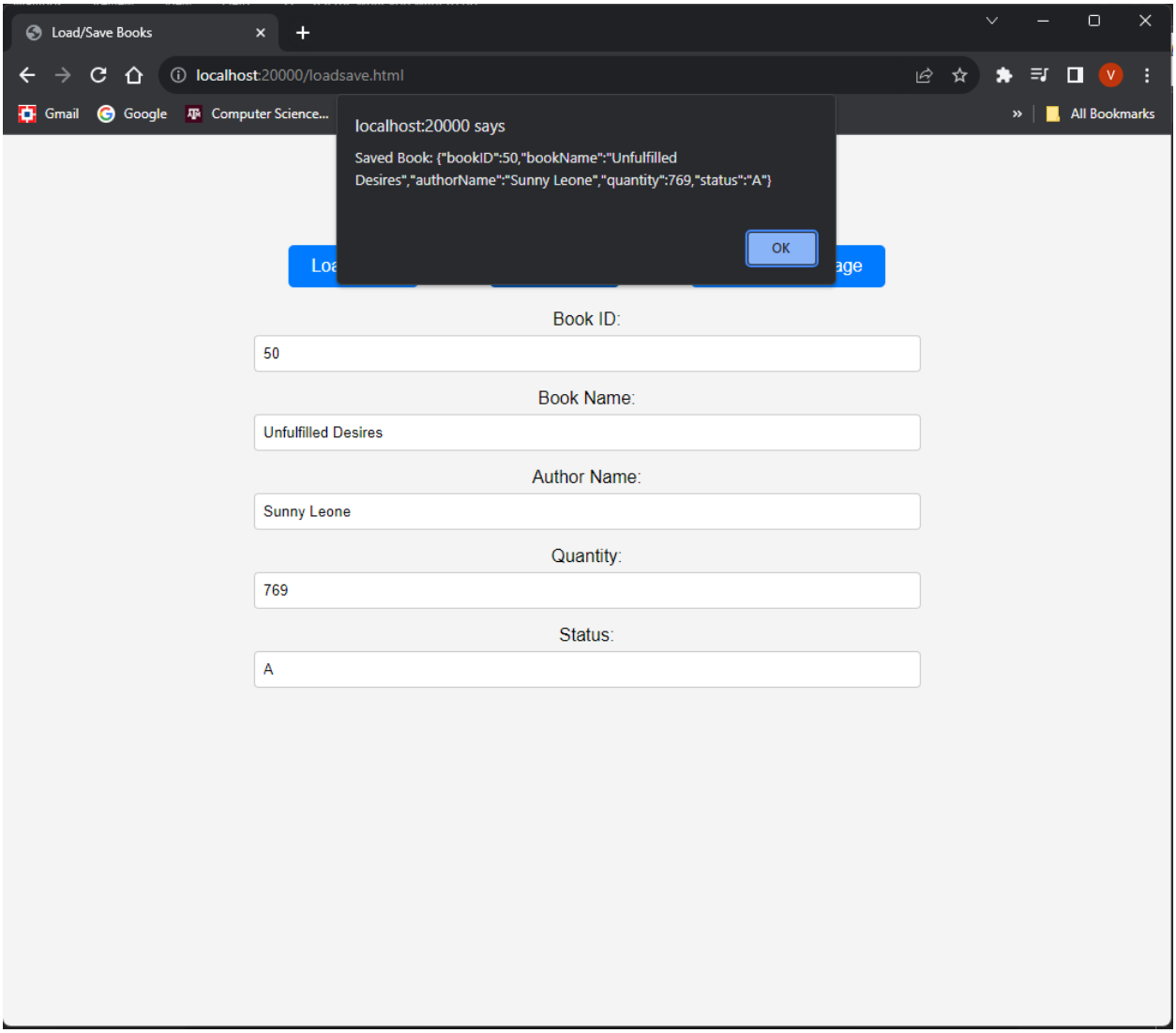
The details have been entered in the Load/Save Books page.

3)



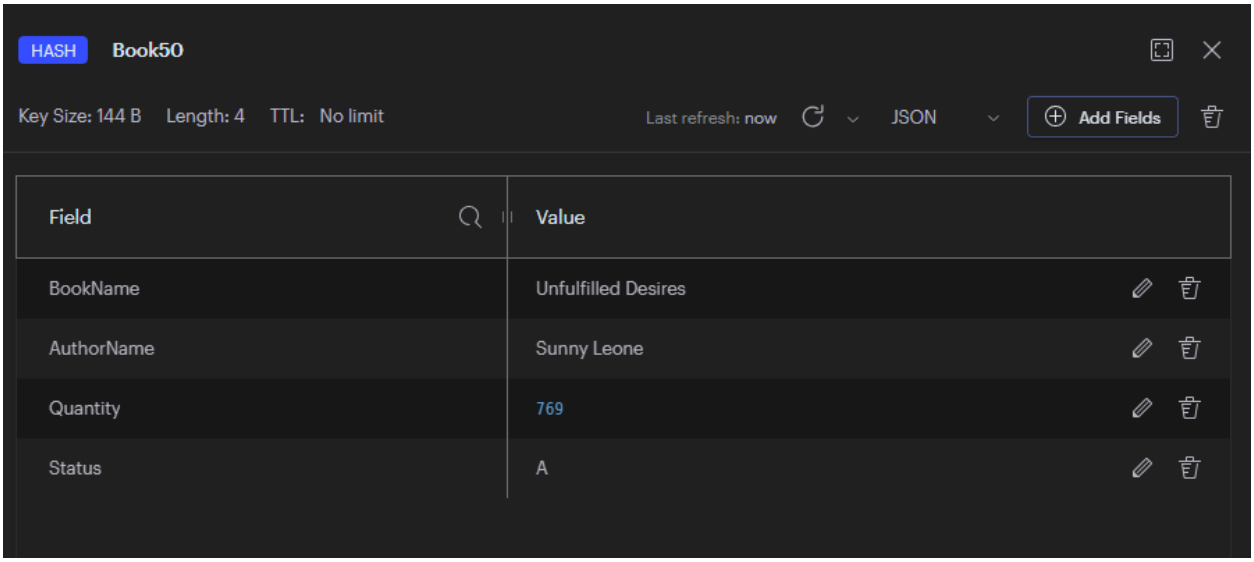
The save Book button is clicked.

4)



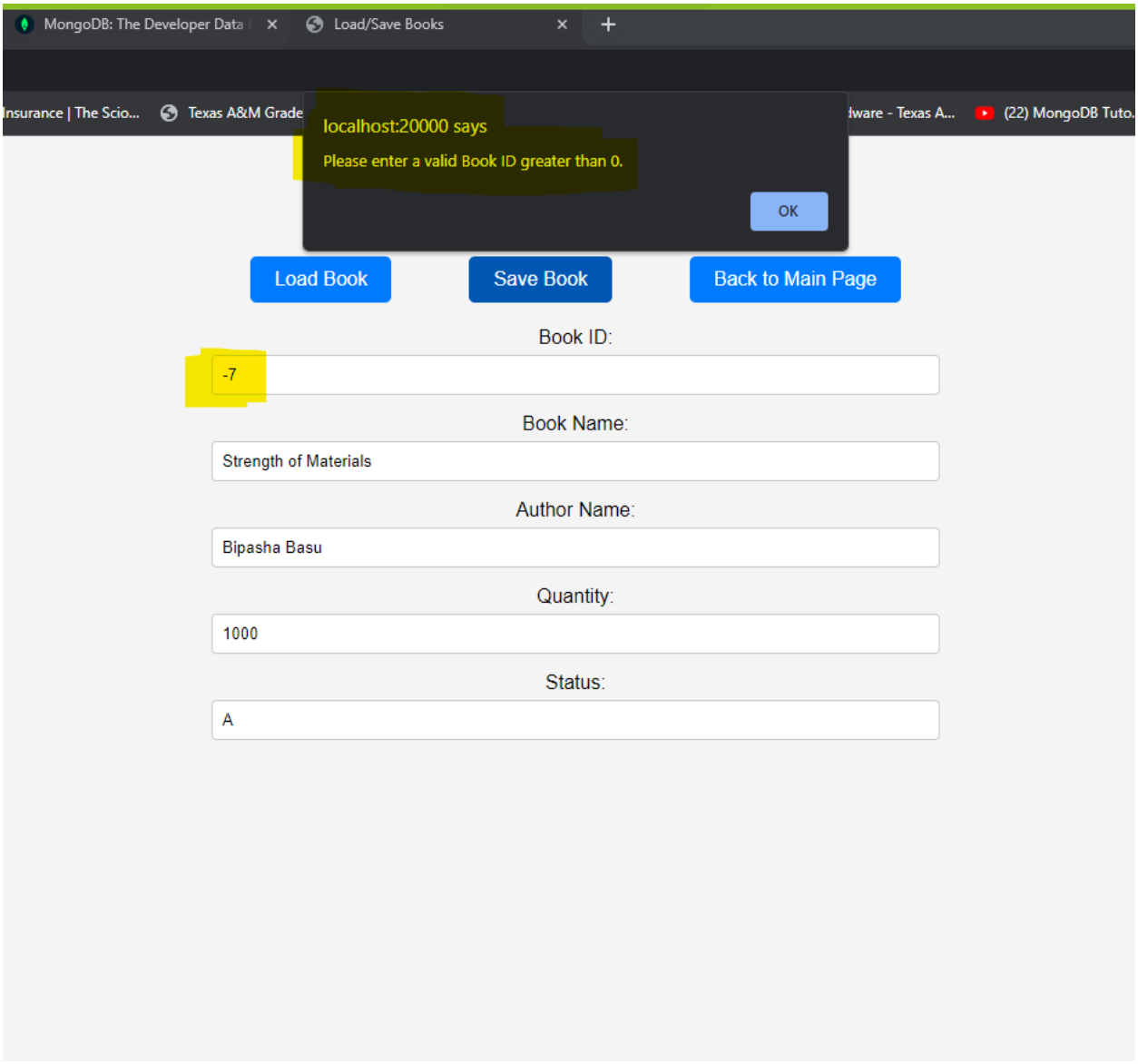
The “Book Saved Successfully” alert appears.

5)

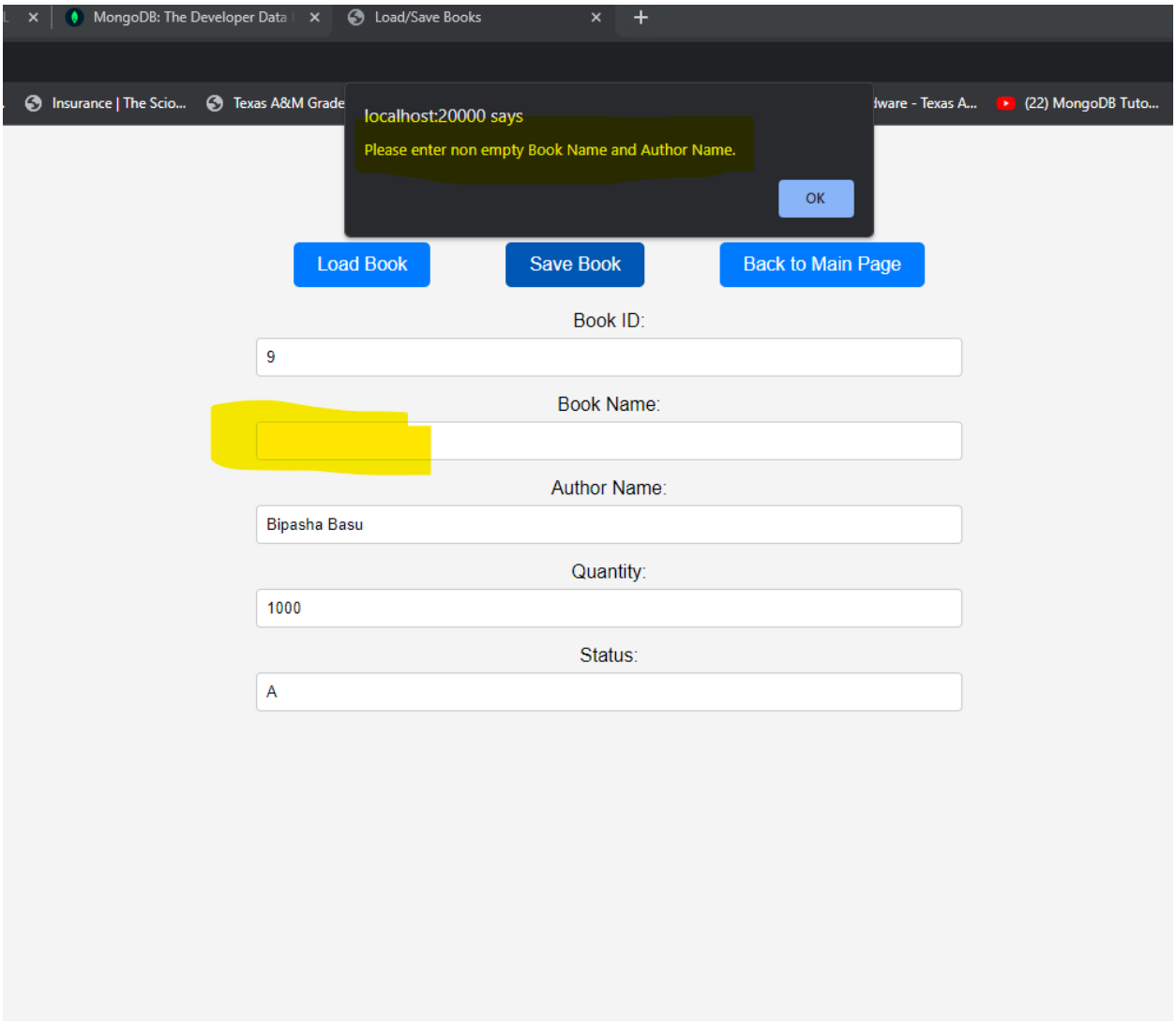


The book details are saved in the Redis database table.

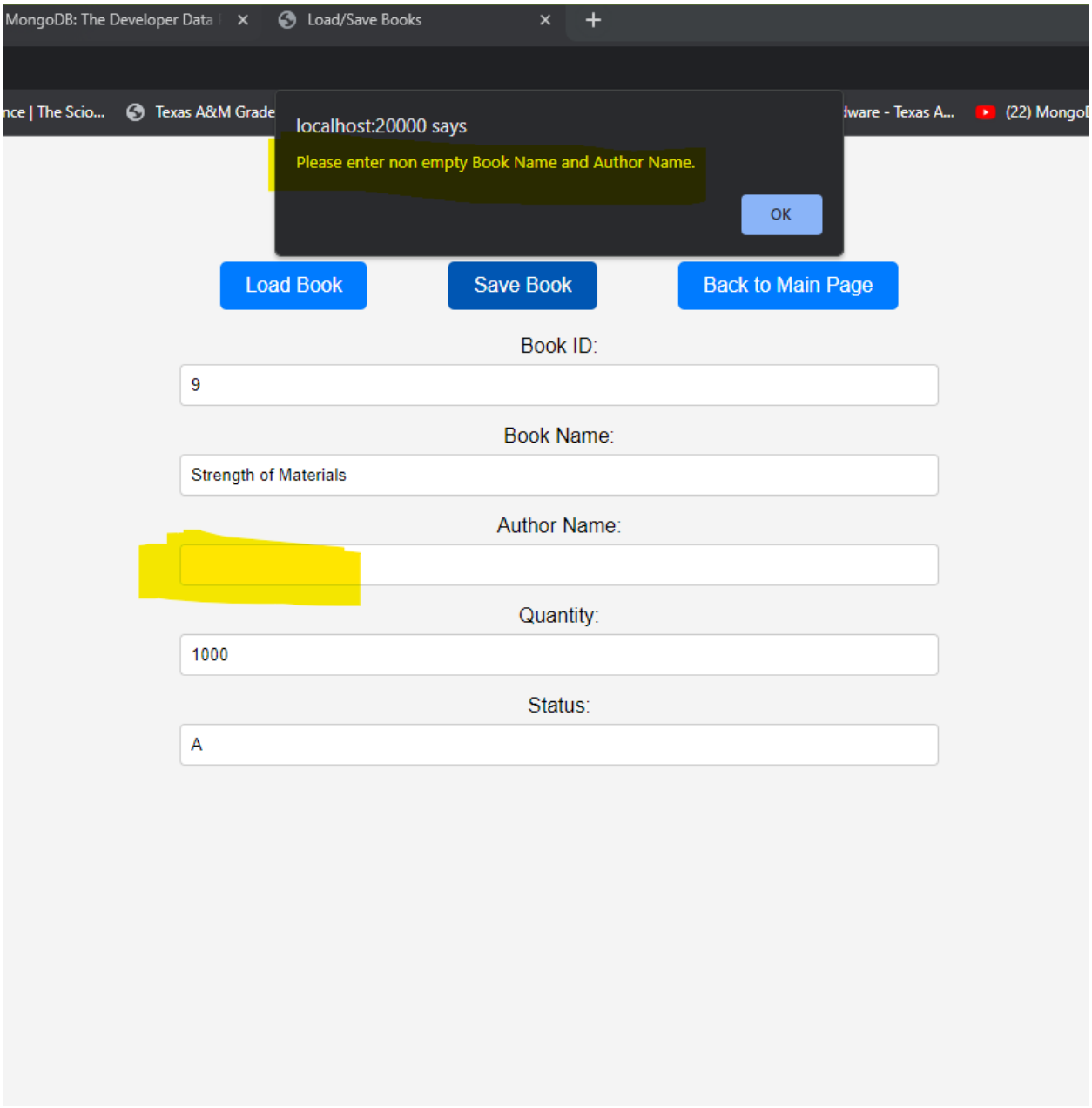
Validations:



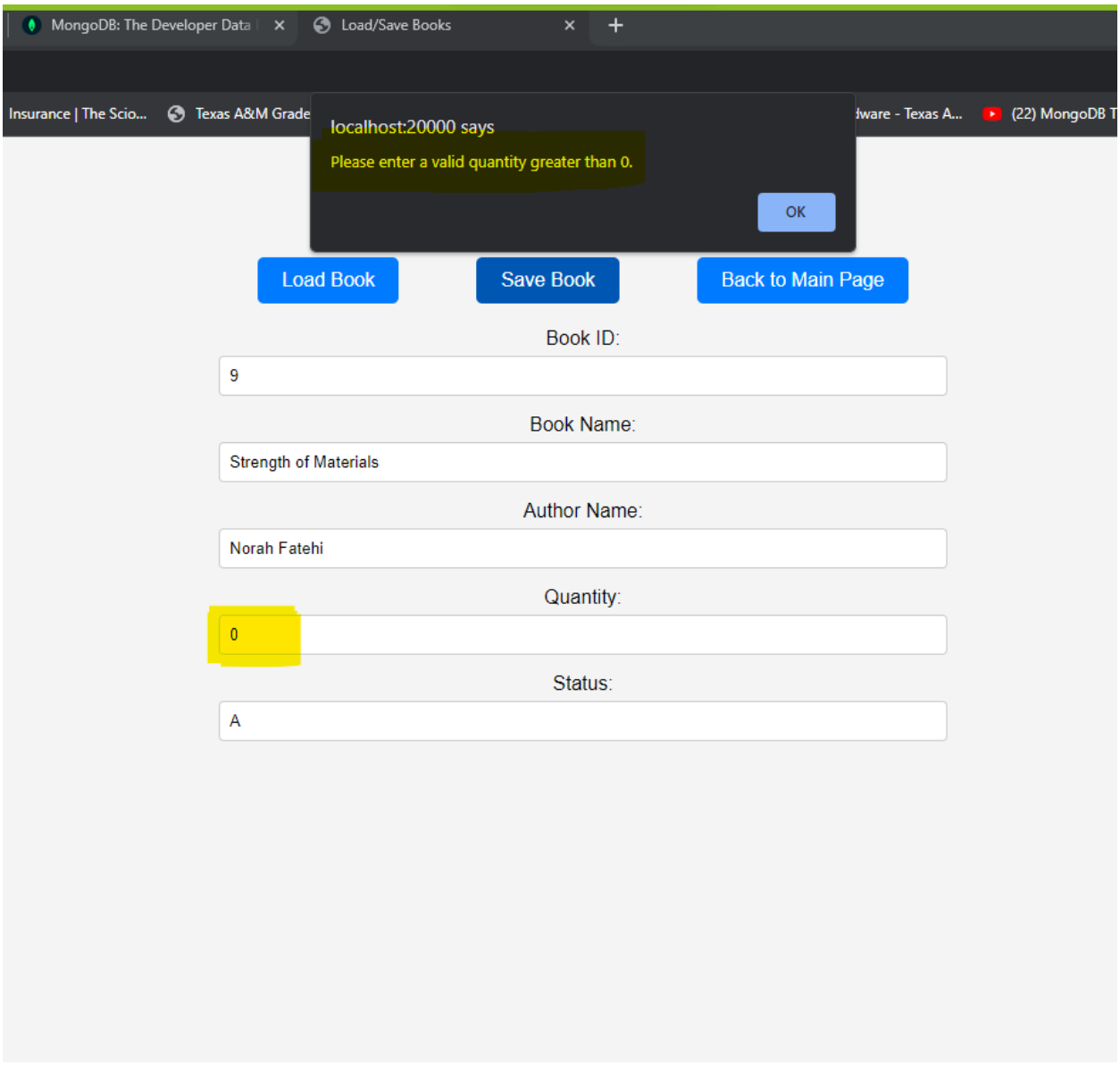
Book ID should be positive alert pops up on the page.



Book Name cannot be empty alert pops up on the page



Author Name cannot be empty alert pops up on the page



Quantity must be positive alert pops up on the page

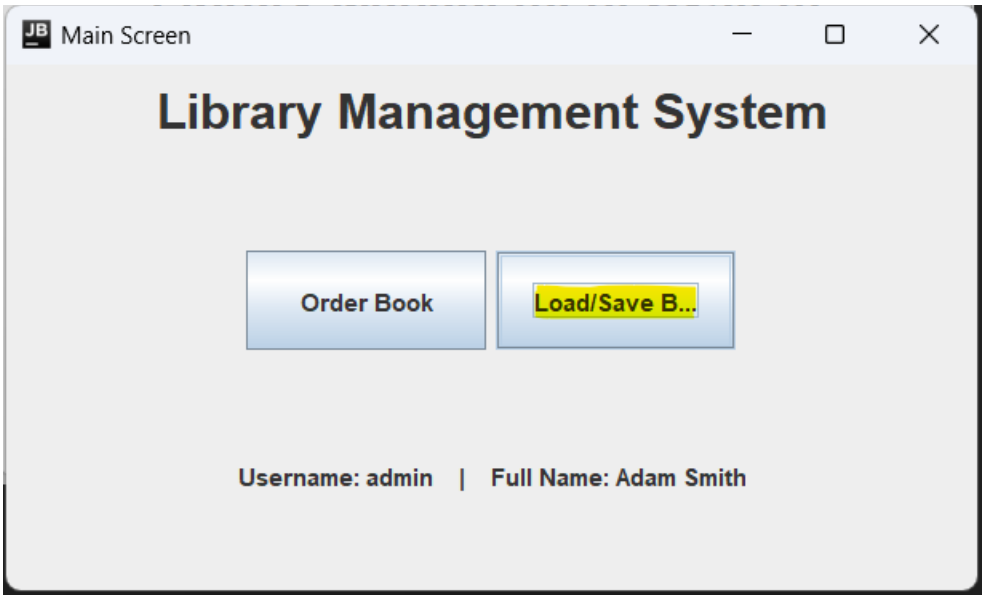
4th Use Case:

<table>
<tr><td>Use Case</td><td>Load Book</td></tr>
<tr><td>Description</td><td>The administrator shall be able to load a book to the database of the library management system</td></tr>
<tr><td>Precondition</td><td>The main screen should be visible to the administrator.
The book should be present in the Redis Database (with the same Book Id).</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The administrator will click on the Load/Save book button on the main screen.</td></tr>
<tr><td>2</td><td>The administrator would enter the Book ID against the “Book ID” label in the Manage Books window.</td></tr>
<tr><td>3</td><td>After entering the Book ID, the administrator will click on the “Load Book” button.</td></tr>
<tr><td>4</td><td>Library Management System application will automatically search for the book in the database, and auto-populate the Book Name, Author Name, Quantity, and Status if the Book exists in the Database. Else, a window will pop up that “No such book exists with this book id”.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>The entered book details will be displayed to the user.</td></tr>
<tr><td>Exceptions</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>If the Book ID corresponding to the particular book is not present in the database, then book details will not be auto-populated.</td></tr>
</table>
</td></tr>
</table>

Actor Action	**System Response**
1 The admin clicks on the Load Book button from Main Screen window.	2 The “Manage Books” window appears.
3 Admin types in the Book ID.	
4 The admin clicks on “Load Book” button.	5 The system responds by displaying the Book Name, Author Name, Quantity, and Status of the respective Book. In case the book is not present in the Redis Database, the system displays a pop-up window that “No book with this book id exists in Database”.
	6 The book details get saved in the database.

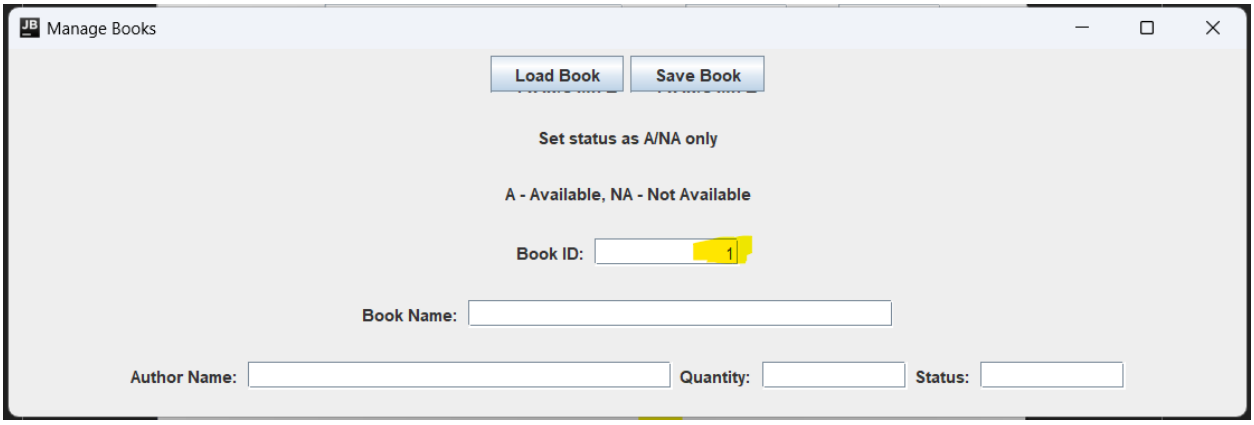
DESKTOP APPLICATION UI

1)



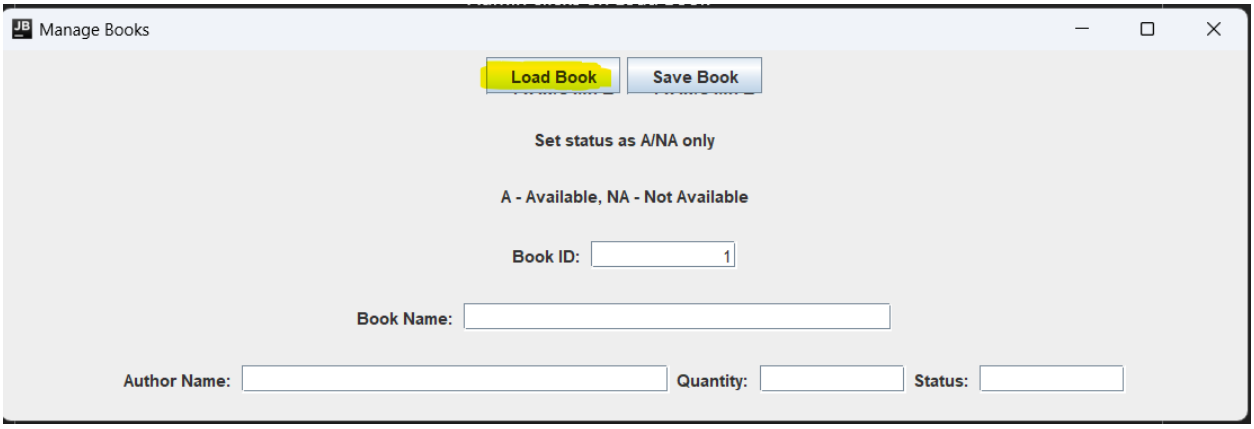
Main Screen window -> Load/Save Book button

2)



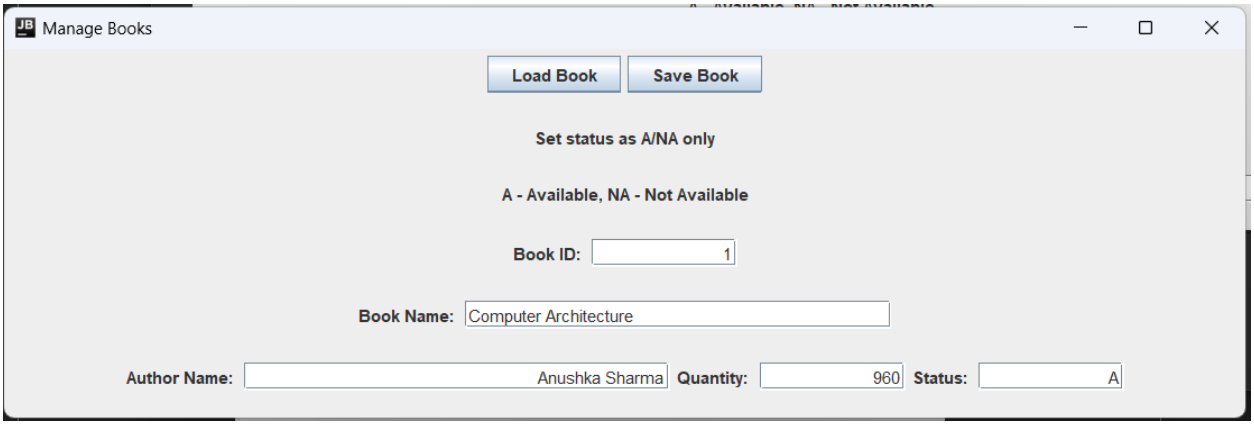
Manage Books screen **-> Admin enters Book ID 1**

3)

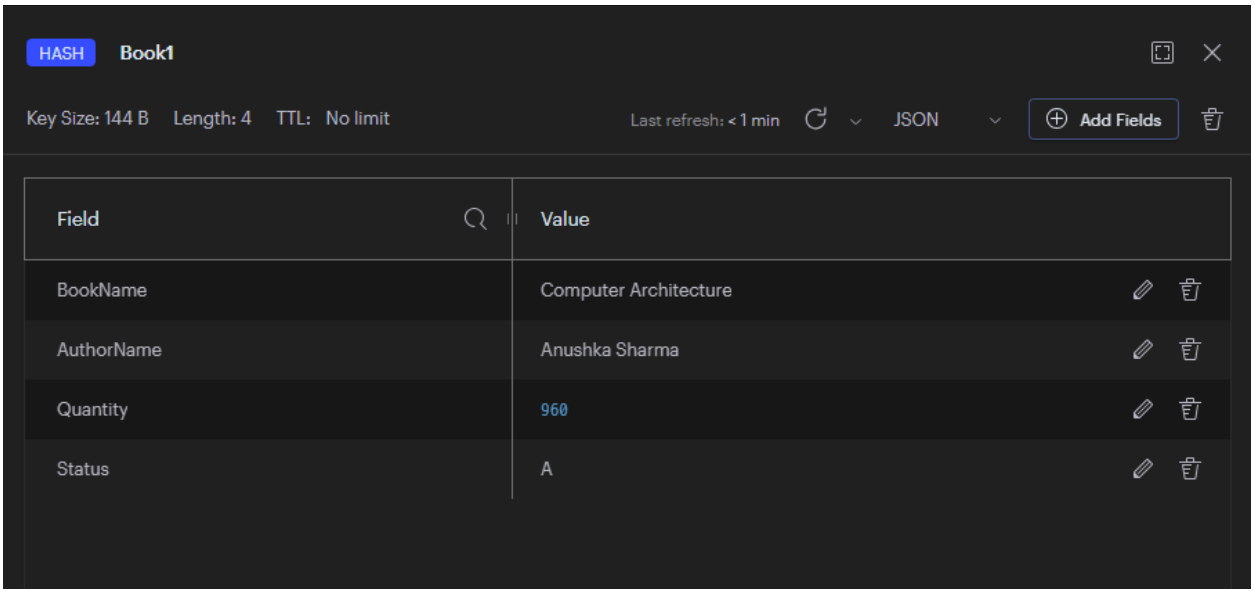


Admin clicks on Load Book button

4)

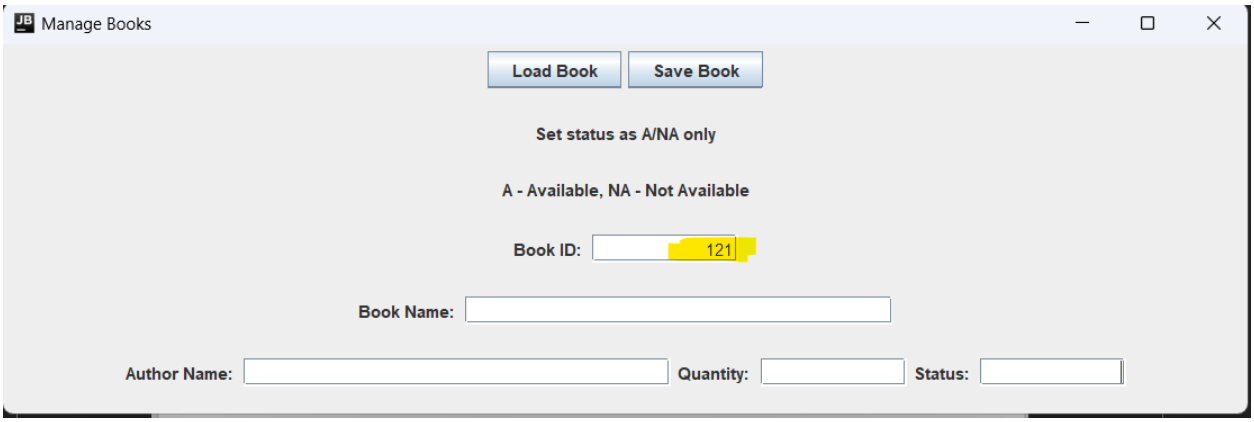


The system auto-fills the Book Name, Author Name, Quantity, Status from the Database in case the Book Id is available in the database.

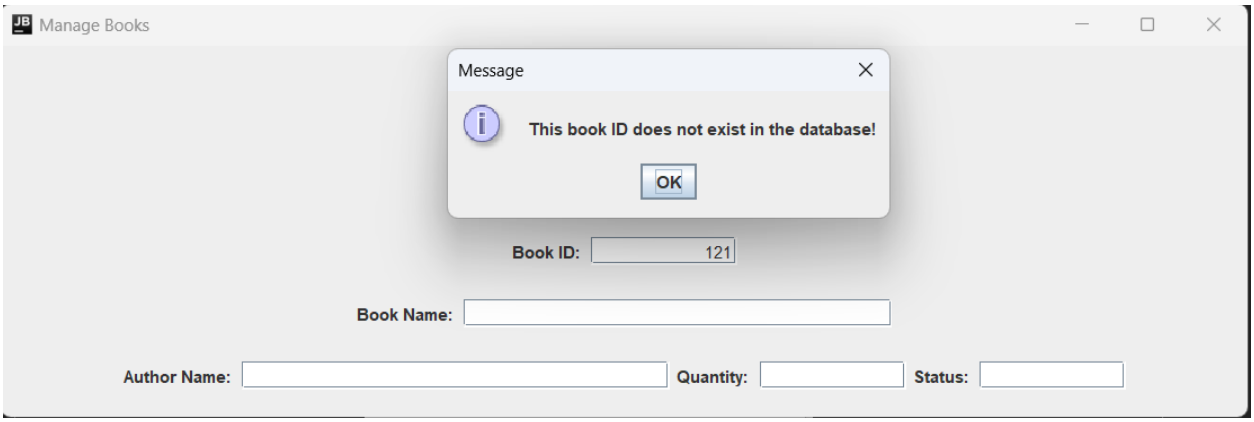


Redis Database

6)



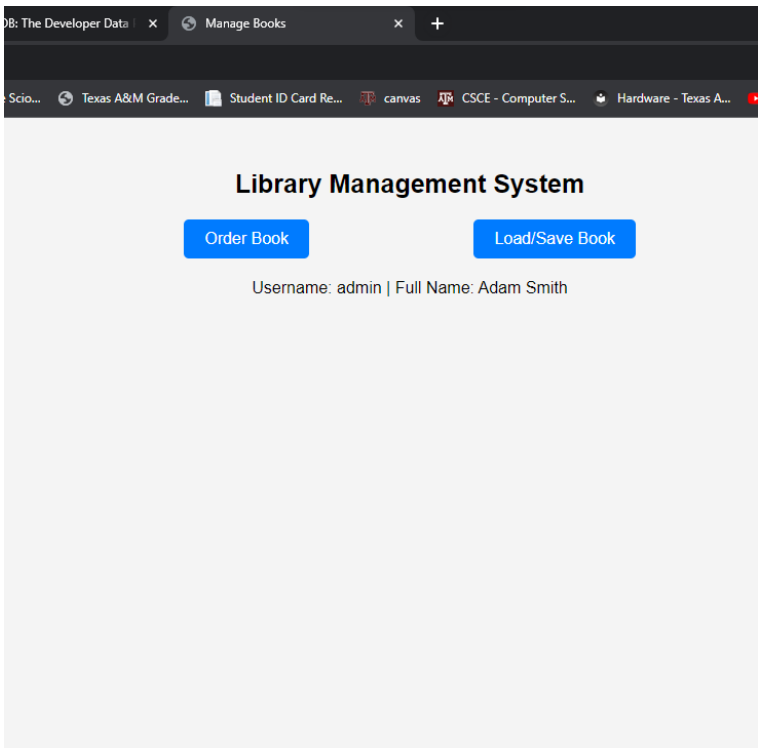
In case user enters invalid Book Id and clicks on Load Book Button.



A pop-up window appears stating that this book Id does not exist in database.

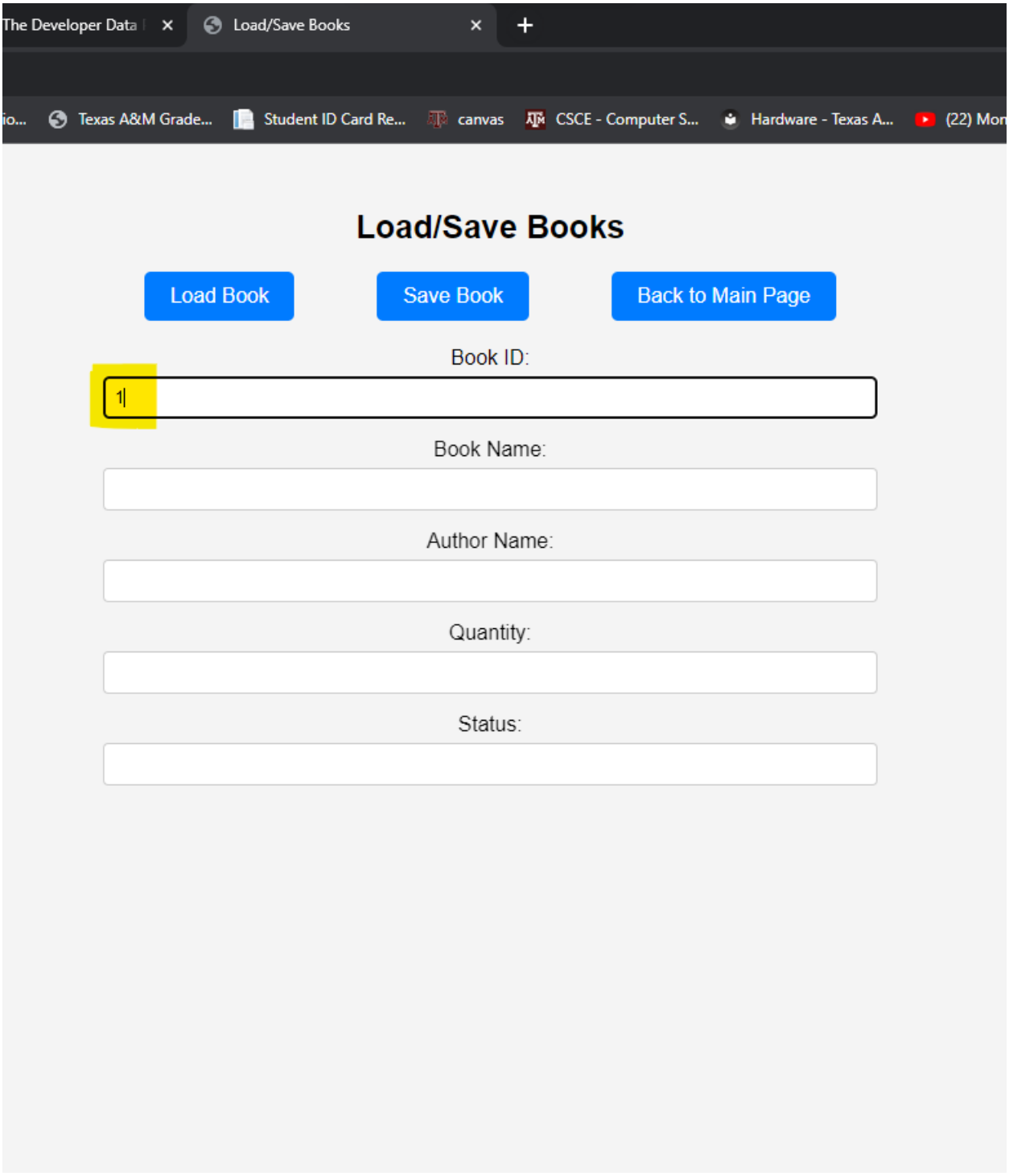
WEB APPLICATION UI

1)



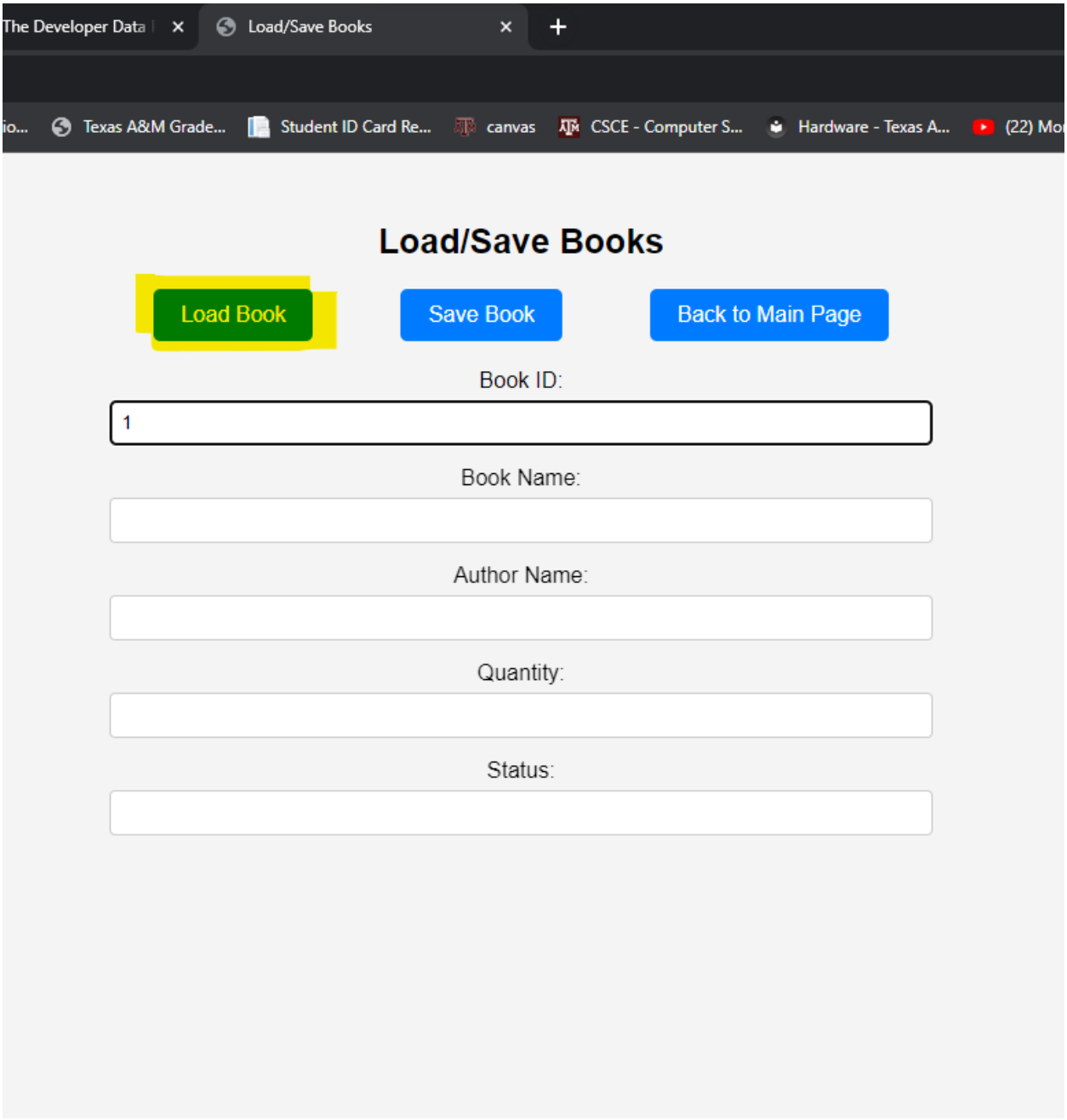
Main Screen window -> Load/Save Book button

2)



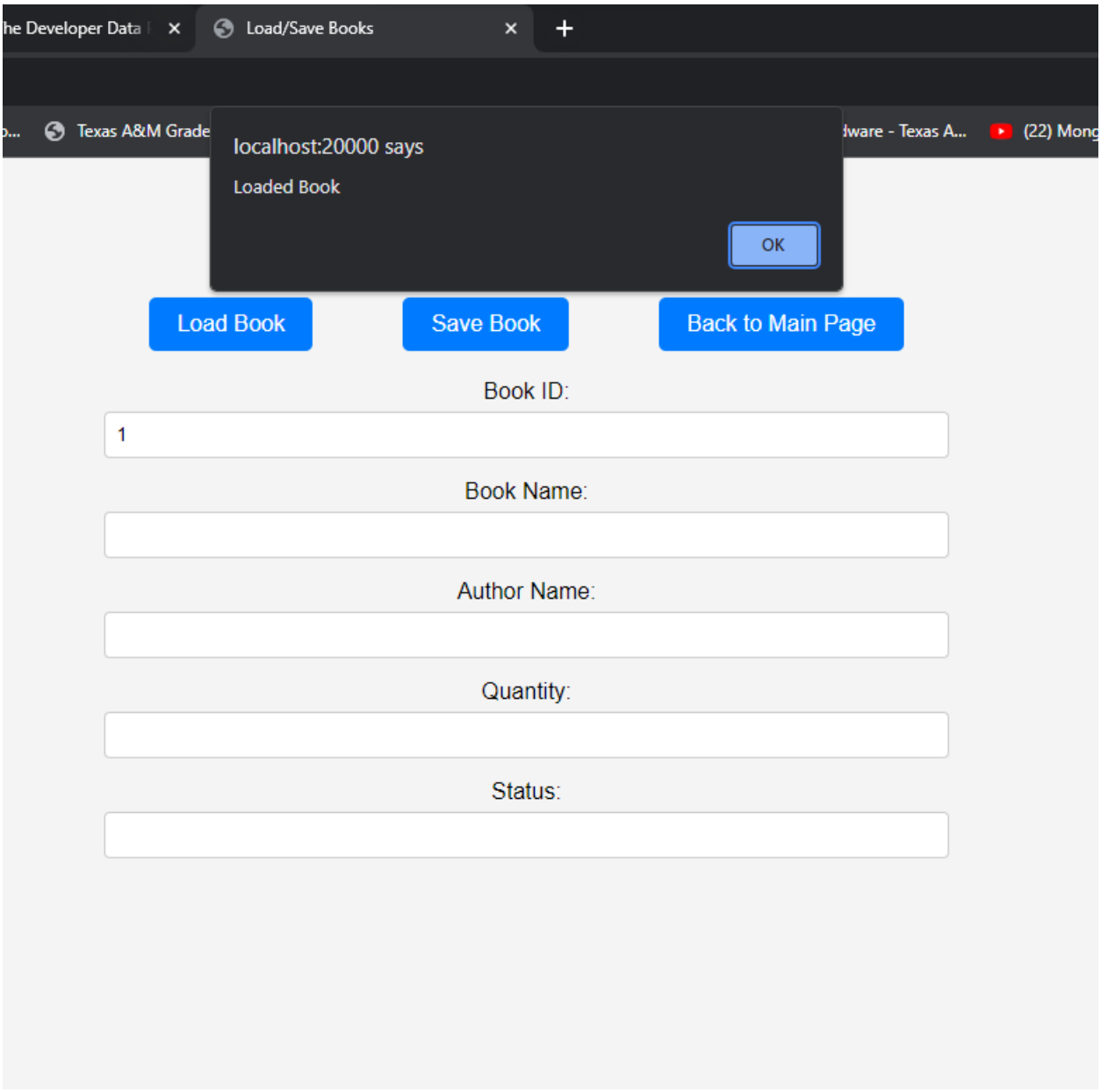
Manage Books screen **-> Admin enters Book ID 1**

3)



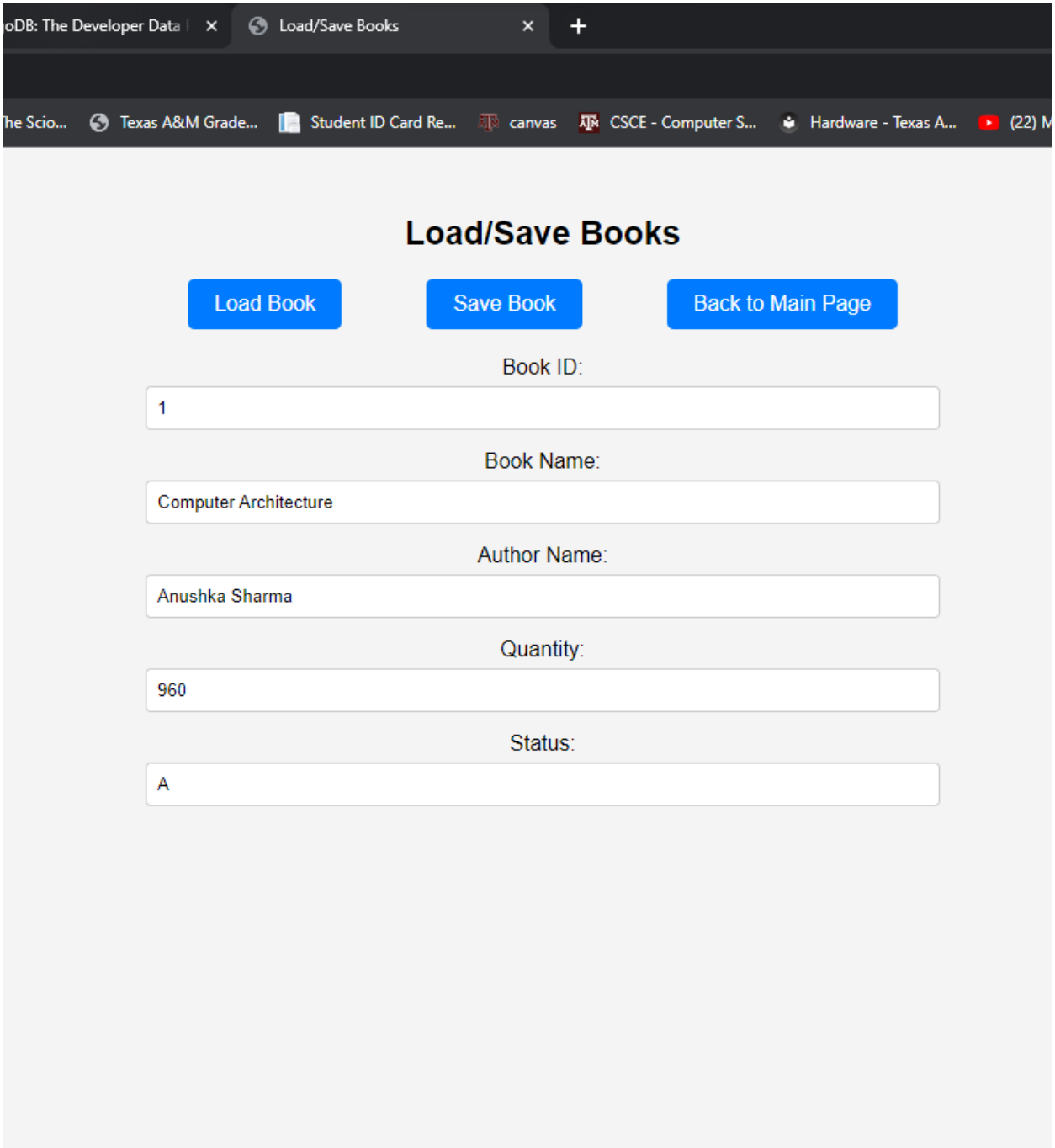
Admin clicks on Load Book button

4)

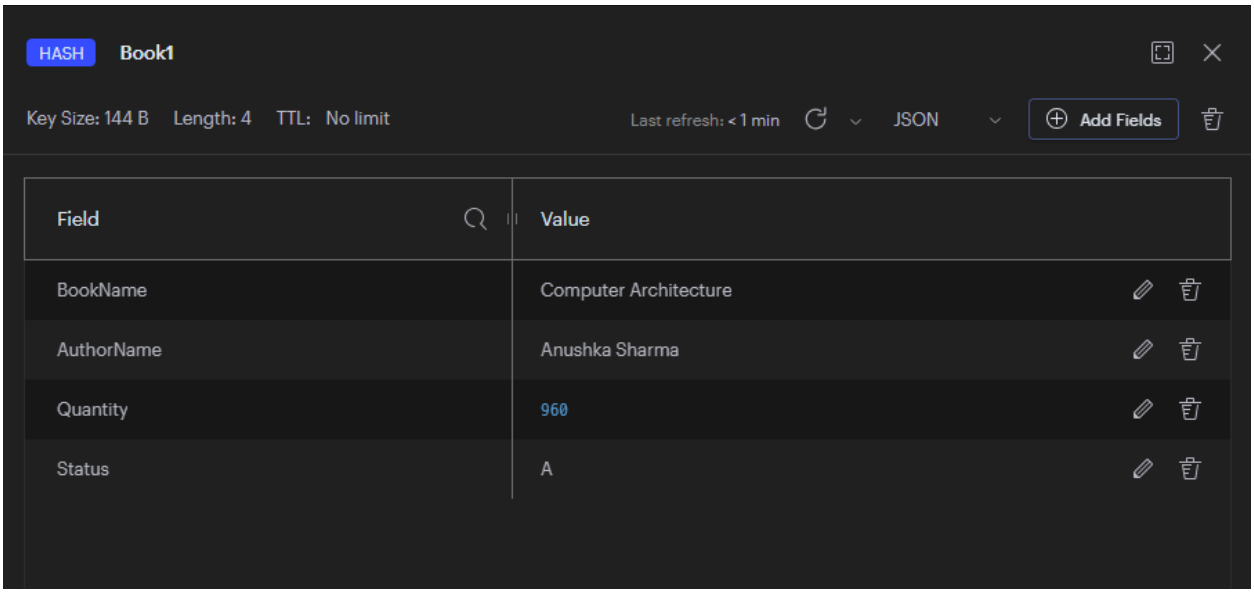


Loaded Book alert pops up

5)



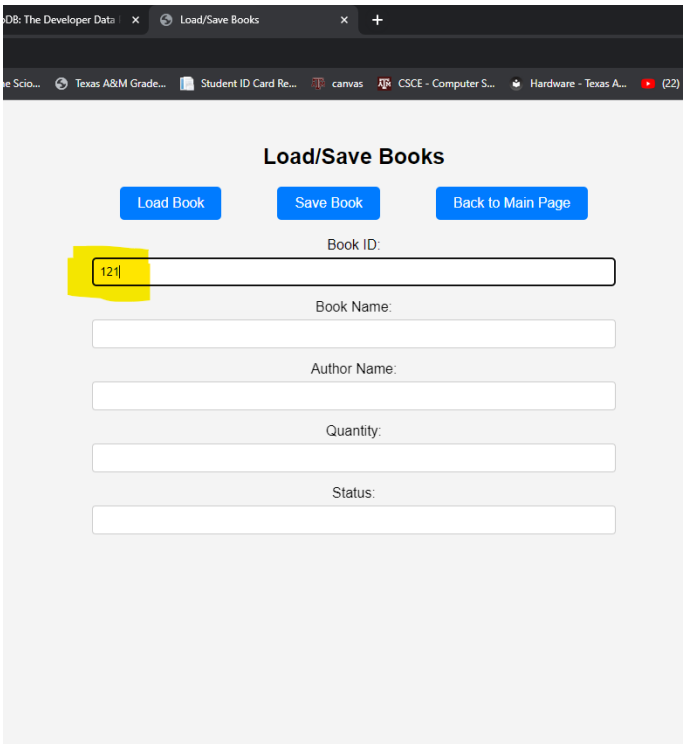
The system auto-fills the Book Name, Author Name, Quantity, Status from the Database in case the Book Id is available in the database.



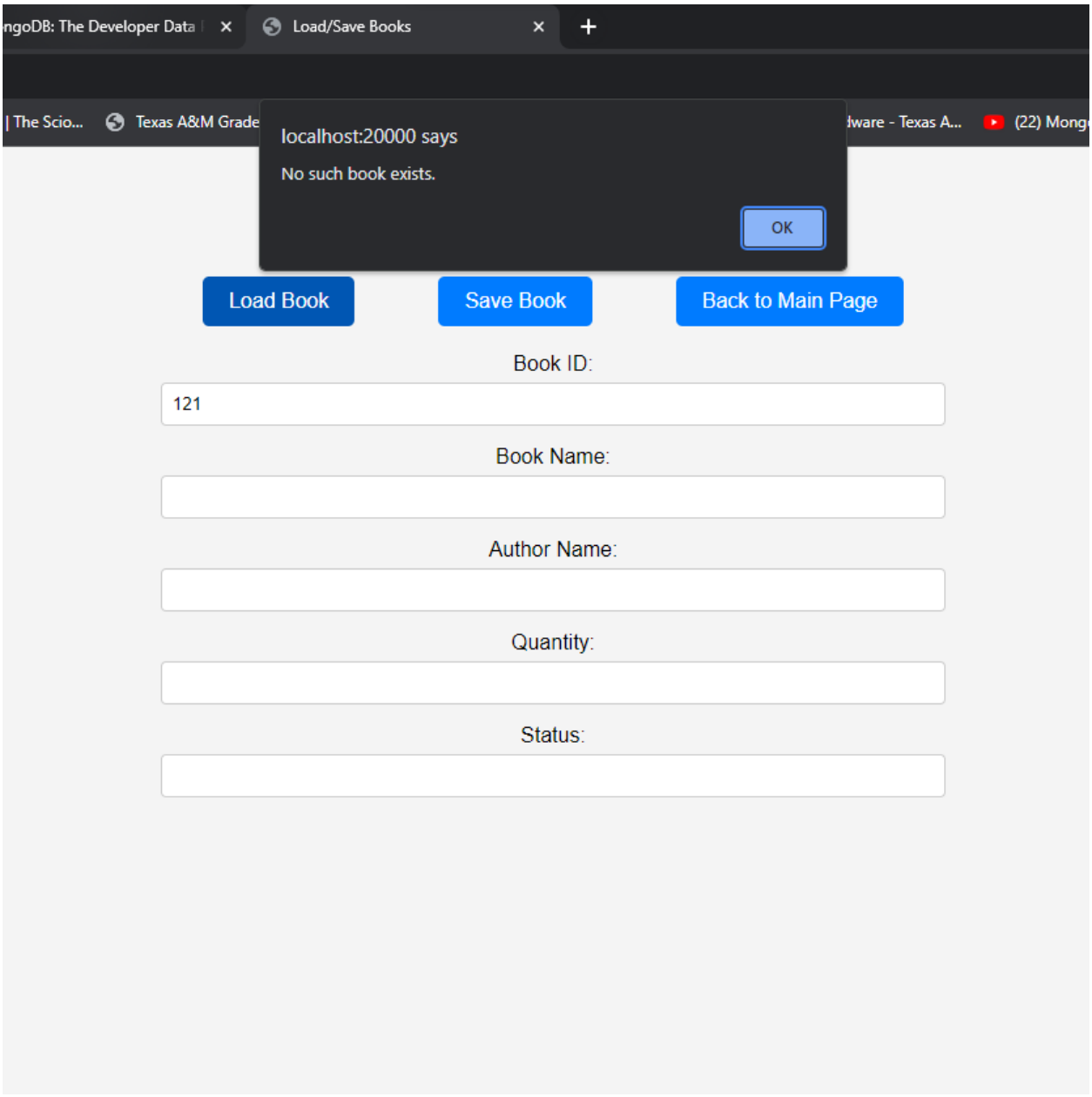
Redis Database

6)

Validations:



In case user enters invalid Book Id and clicks on Load Book Button.



“No such Book exists” alert pops up

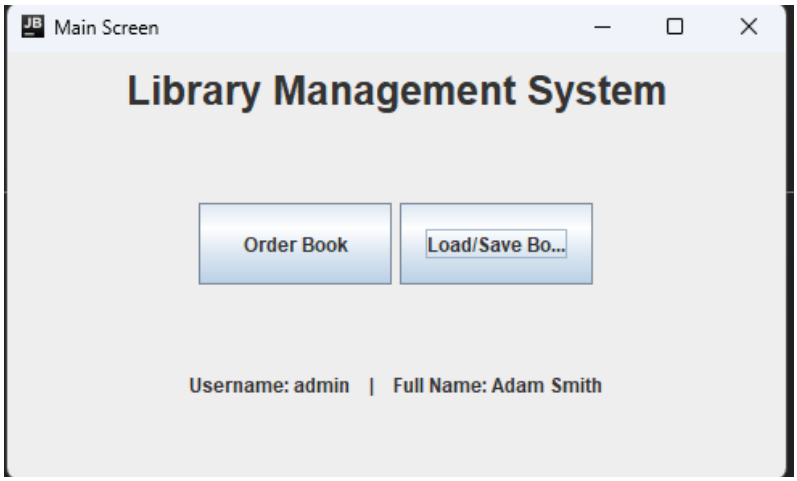
5th Use Case:

<table>
<tr><td>Use Case</td><td>Book Order View</td></tr>
<tr><td>Description</td><td>The administrator shall be able to access the Book Order View screen in the library management system.</td></tr>
<tr><td>Precondition</td><td>The administrator should have logged in to the application</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The administrator will click on the Order Book button on the main screen</td></tr>
<tr><td>2</td><td>Book Order View page appears on the screen.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>This screen shall also display two buttons - the “Add a new book” button, used to select books to order and the “Save book order” button, used to create an order
This screen shall display the Book ID, Book Name, Book Author, Order Date, and Return Date after entering the book details to current order</td></tr>
<tr><td>Comment</td><td>This screen shall be used to display the books that are being added to the current order.</td></tr>
</table>

Actor Action	**System Response**
1 The administrator will access the Login screen and enter his username and password	2 System will validate the user’s details and open the Main Screen window.
3 The administrator will click on the Order Book button on the main screen	4 System will display the Book Order view page.

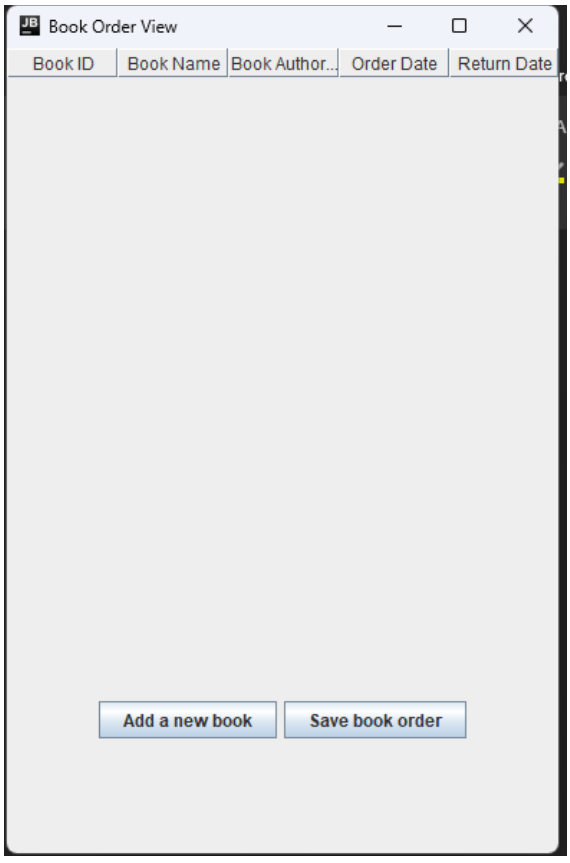
DESKTOP APPLICATION UI

1)



Main Screen -> Admin clicks on Order Book button

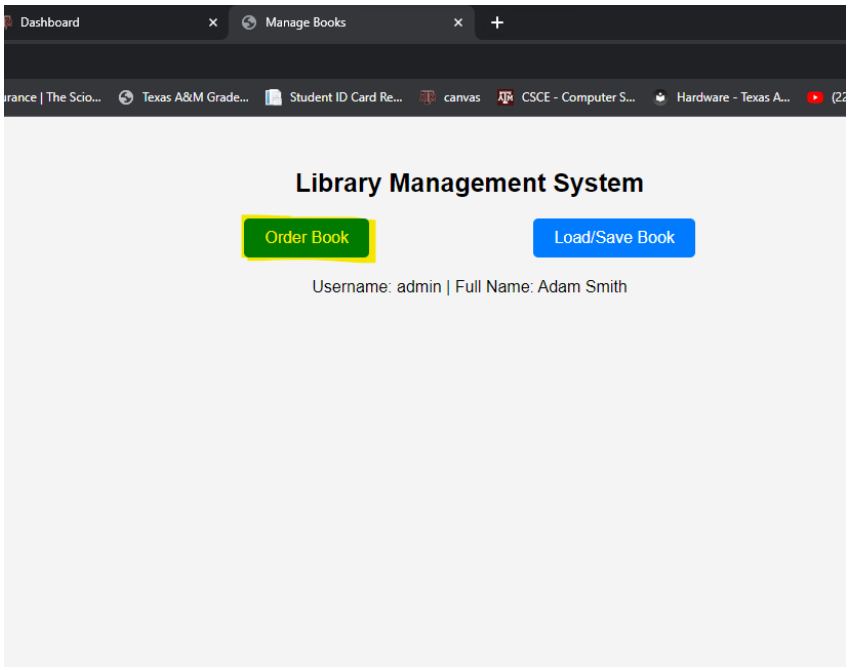
2)



Book Order View screen appears on the screen

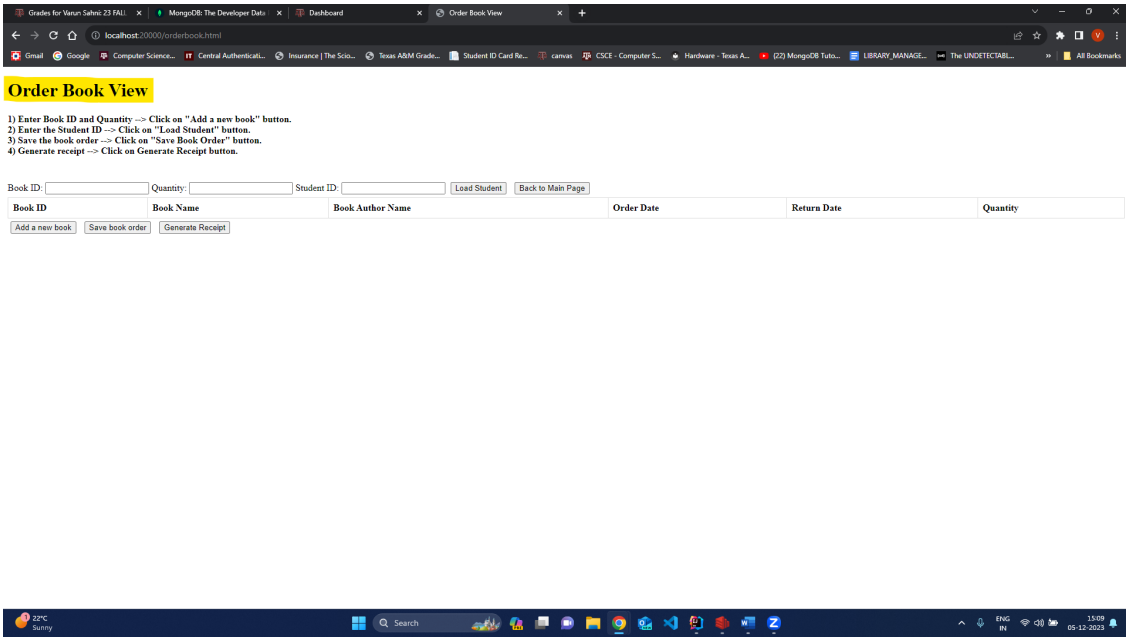
WEB APPLICATION UI

1)



Manage Books Page -> Admin clicks on Order Book button

2)



The book Order View page opens up and appears on the screen

6th Use Case:

<table>
<tr><td>Use Case</td><td>Add a new book to the Cart (Book Order View table)</td></tr>
<tr><td>Description</td><td>The administrator shall be able to add books to the current order</td></tr>
<tr><td>Precondition</td><td>The Book Order view screen should be visible to the administrator.</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The administrator will click on the “Add a new book” button on the Book Order View screen.</td></tr>
<tr><td>2</td><td>The book ID input dialog box appears</td></tr>
<tr><td>3</td><td>The administrator will enter the book ID.</td></tr>
<tr><td>4</td><td>The system will check if the respective book exists in the database and display the quantity input dialog box if the book exists. Otherwise, it shall display a pop-up window saying that this book does not exist in the database.</td></tr>
<tr><td>5</td><td>The administrator will enter the quantity</td></tr>
<tr><td>6</td><td>The system will check If the respective book has the respective quantity available in the database, and display the Book ID, Book Name, Author Name, Order Date, and Return Date on the Book Order View screen.</td></tr>
<tr><td>7</td><td>The administrator will click on the Save Book Order button on the Book Order View screen.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>The system will show a Student Information dialog box to capture Student details for completing further orders.</td></tr>
<tr><td>Comment</td><td>This shall validate the availability of the respective books from the books database.</td></tr>
</table>

Actor Action	**System Response**
1 The administrator will click on the Order Book button on the main screen.	2 The “Book Order View” window appears.
3 The administrator will click on “Add a new book”.	4 Book ID input dialog box appears
4 The administrator will enter the book ID.	5 System will check if the respective book exists in the database and display the quantity input dialog box if the book exists. Otherwise, it shall display a pop-up window saying that this book does not exist in the database.
6 The administrator will enter the quantity	6 System will check If the respective book has the respective quantity available in the database, and display the Book Id, Book Name, Author Name, Order Date, and Return Date on the Book Order View screen.

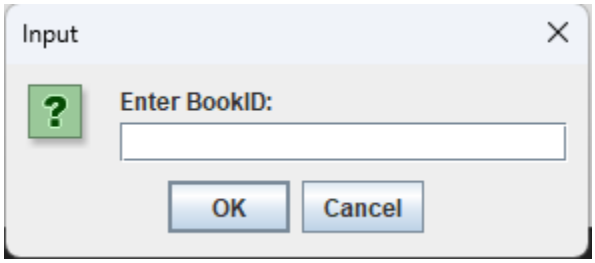
7 The administrator will click on the Save Book Order button on the Book Order View screen.	8 System will show a Student Information dialog box.

DESKTOP APPLICATION UI

1)

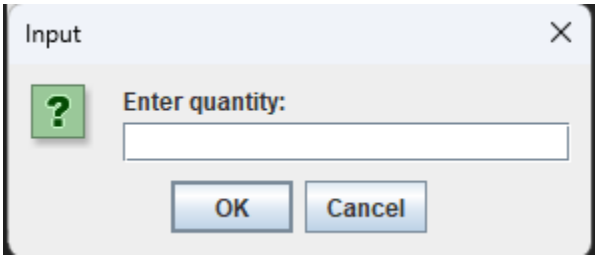
Book Order View screen appears and Admin clicks on “Add a new book” button

2)



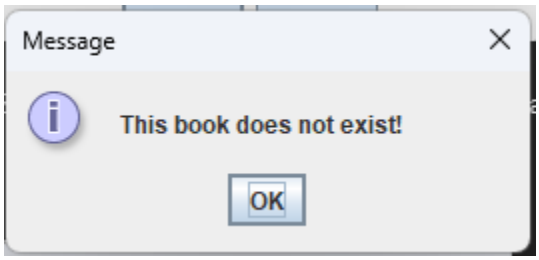
Enter Book Id dialog box appears

3)



Enter Quantity dialog box – Shown only if the book is available in Database -> user clicks on OK

Otherwise – “This book does not exist” dialog box appears



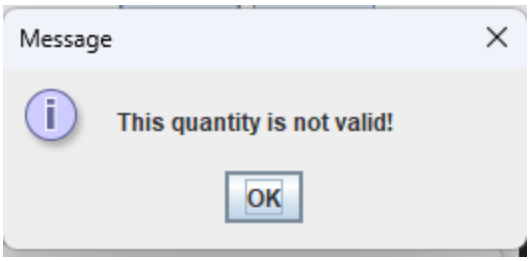
Book does not exist dialog box

4)

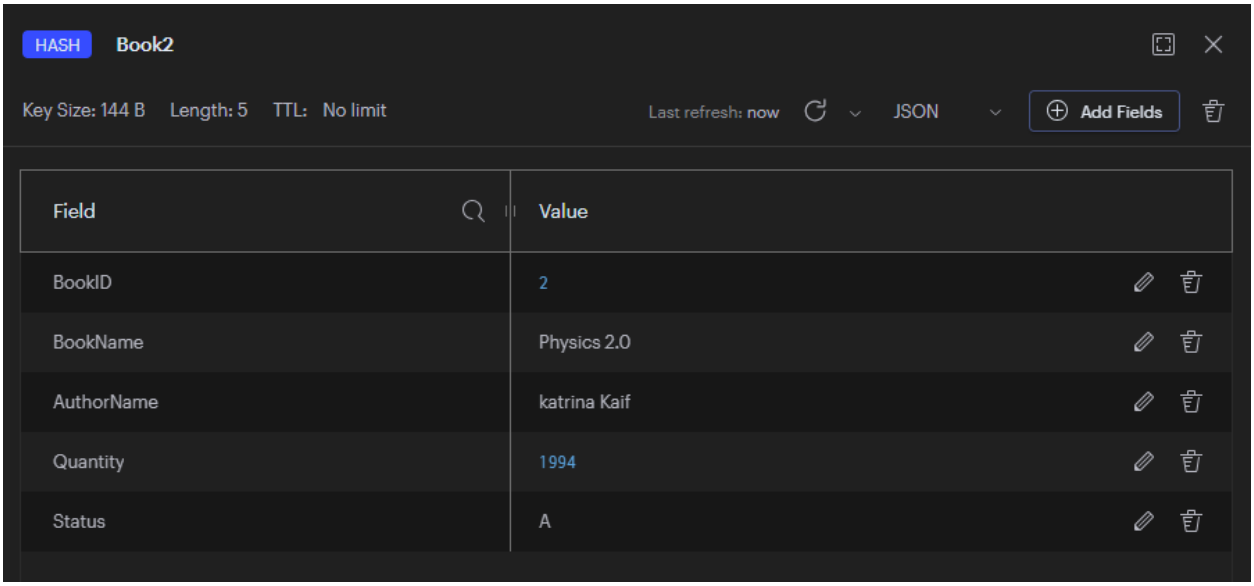
The entered book gets added to Book Order View table and all the fields – Book Name, Book Author, Order Date and Return Date gets auto-filled in case User enters valid quantity.

The entered book appears on Book Order View window

Otherwise – “Quantity is not valid” dialog box appears



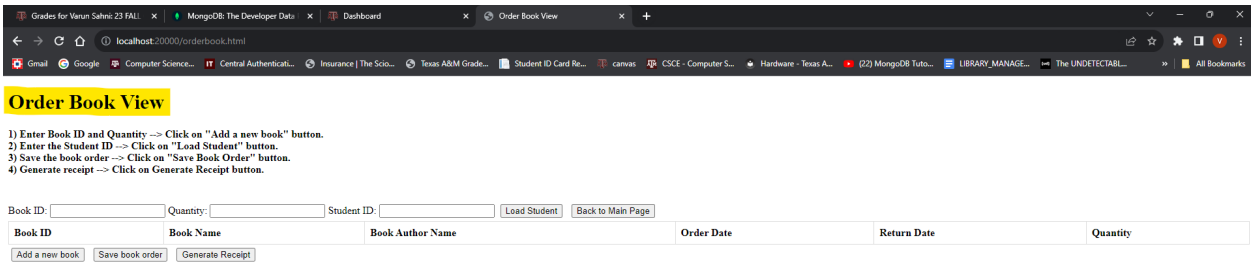
Quantity is not valid dialog box



This book exists in Redis Database

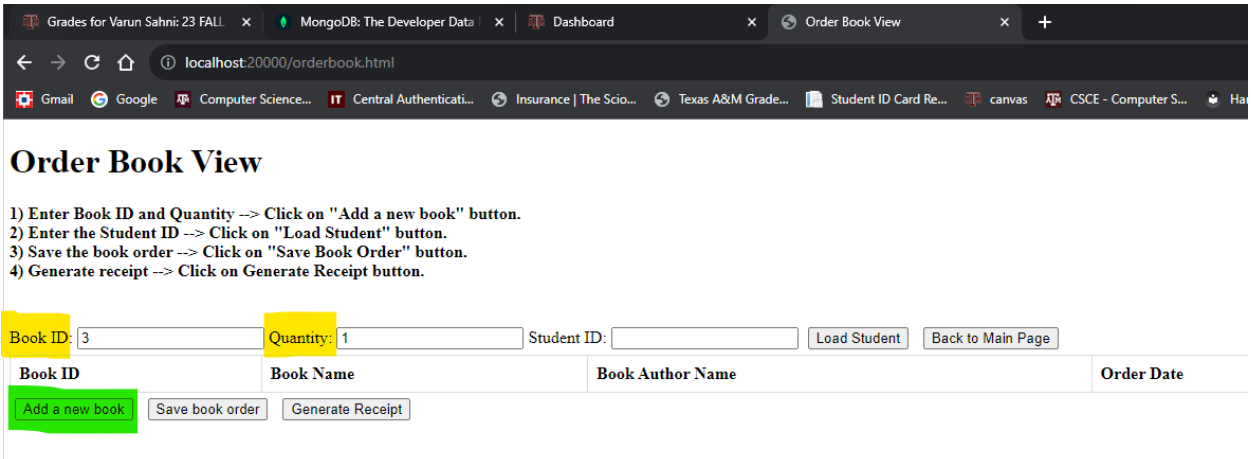
WEB APPLICATION UI

1)



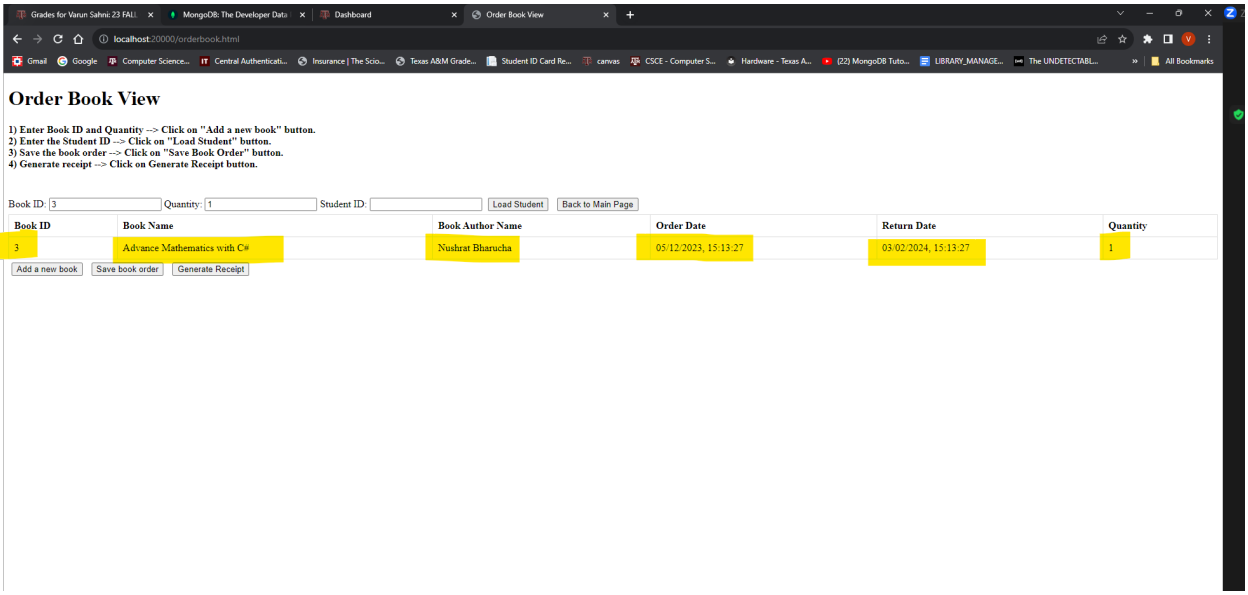
Book Order View screen appears

2)



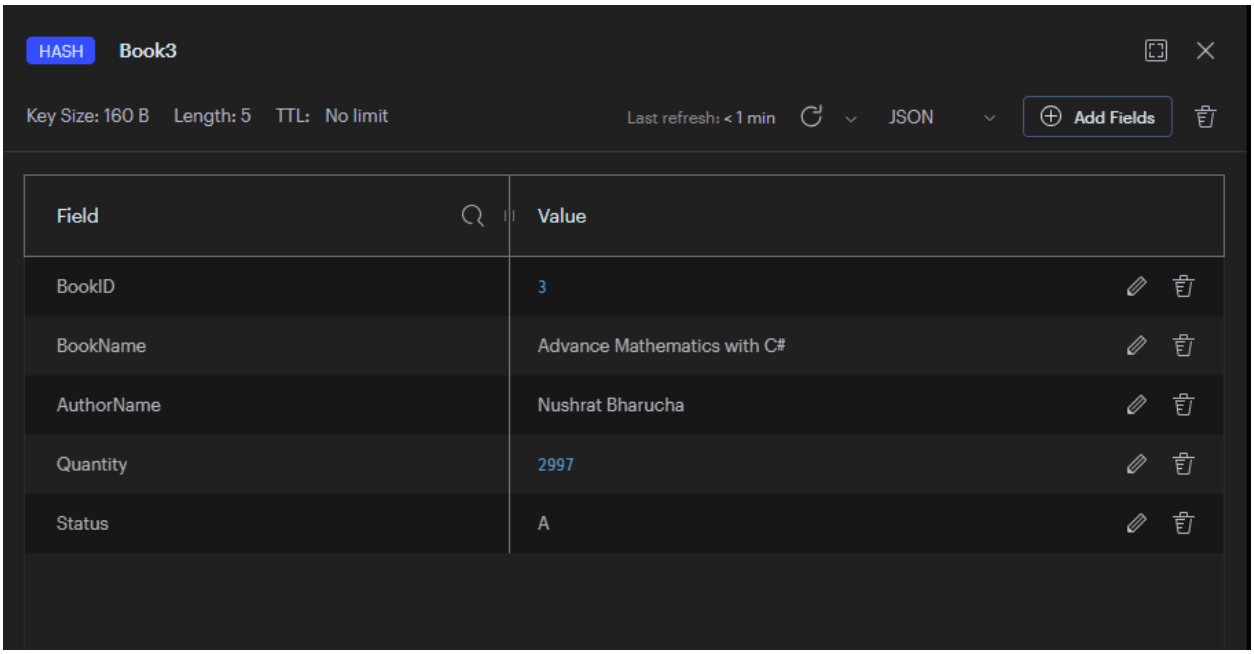
Admin enters Book ID and Quantity → Then clicks on Add a new Book button

3)



Book ID, Book Name, Book Author Name, Order Date, Return Date, and Quantity gets added to the Cart and auto-appears on the page in tabular format

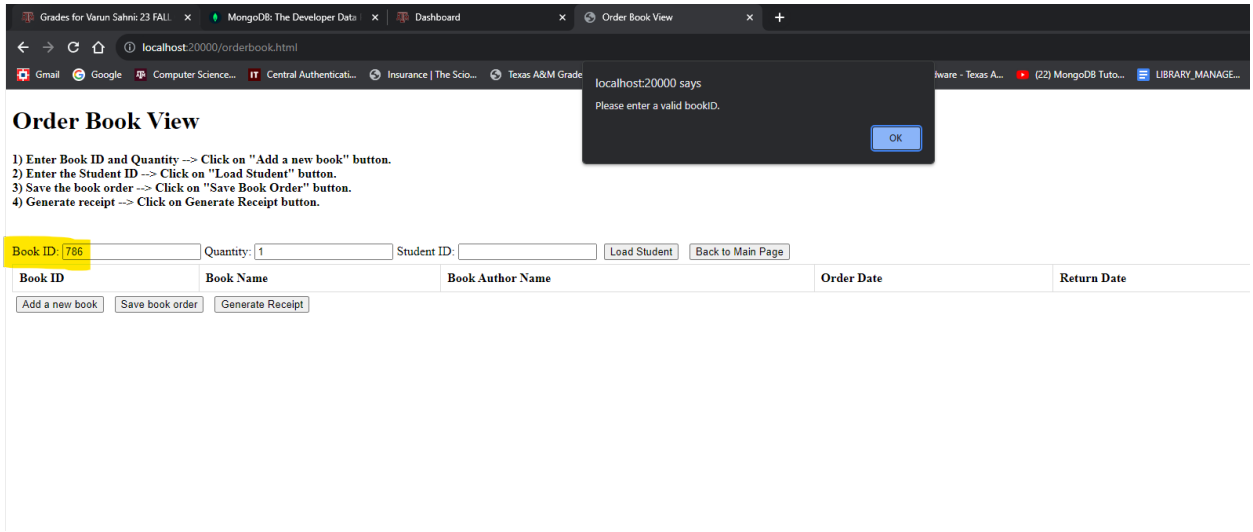
4)



This book exists in Redis Database

5)

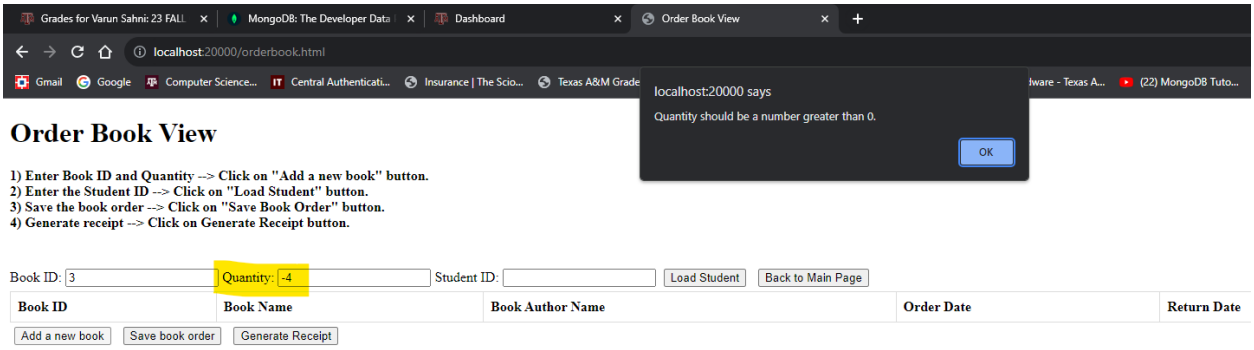
Validations:



Invalid Book ID entered alert pops up in case the Book does not exists in the Database

6)

Validations:



Invalid Quantity entered alert pops up in case Quantity <= 0

7th Use Case:

<table>
<tr><td>Use Case</td><td>Load student details for the current order</td></tr>
<tr><td>Description</td><td>The administrator shall be adding Student details to the current order</td></tr>
<tr><td>Precondition</td><td>Book Order View screen should be visible to the user.
User should have added books to the Book Order View screen.
Student Information dialog box should have been appeared on screen as in previous use case</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The administrator will enter Student Id, and Click on “Load Student Details” button.</td></tr>
<tr><td>2</td><td>System will validate the student’s details from the database and auto-populate the student’s name, email id and mobile number. In case, student does not exist in the database, a pop-up window will appear saying that this student does not exist in the database.</td></tr>
<tr><td>3</td><td>The administrator will click on OK button</td></tr>
<tr><td>4</td><td>System will generate an order.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>System will generate an order.</td></tr>
<tr><td>Comment</td><td>If the student ID corresponding is not present in the database, the application shall generate a pop up stating that this student does not exist in the database.</td></tr>
</table>

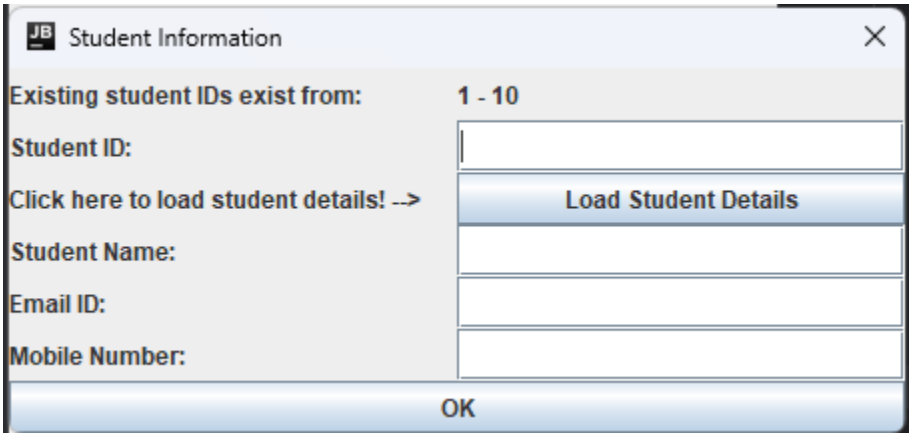
Actor Action	**System Response**
1 The administrator will enter Student Id, and Click on “Load Student Details” button.	2 System will validate the student’s details from the database and auto-populate the student’s name, email id and mobile number. In case, student does not exist in the database, a pop-up window will appear saying that this student does not exist in the database.
3 The administrator will click on OK button	4 System will generate an order.

DESKTOP APPLICATION UI

1)

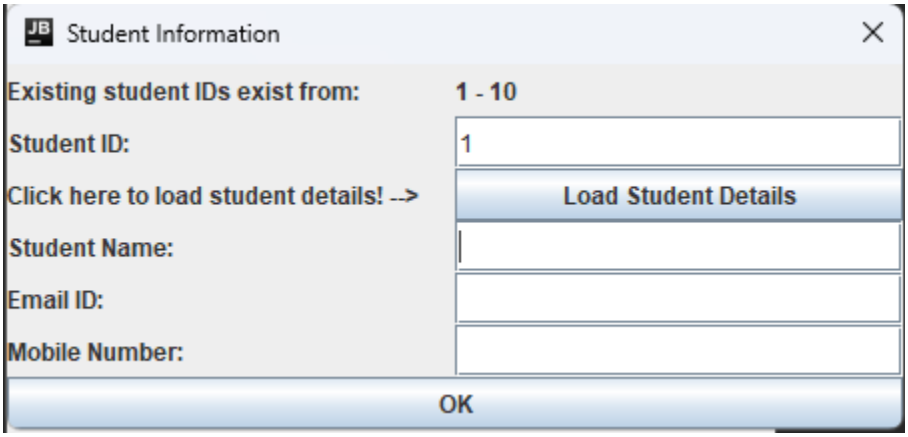
Book Order View page -> Admin clicks on Save book order button post adding the books

2)



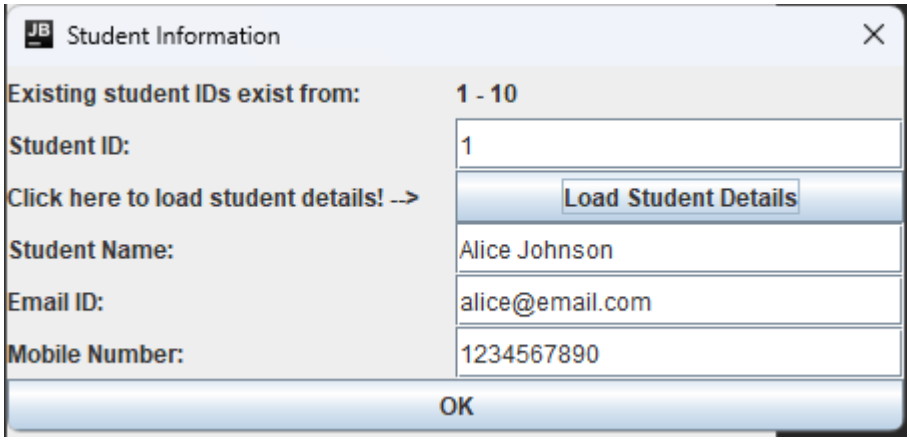
Student Information Dialog box appears

3)

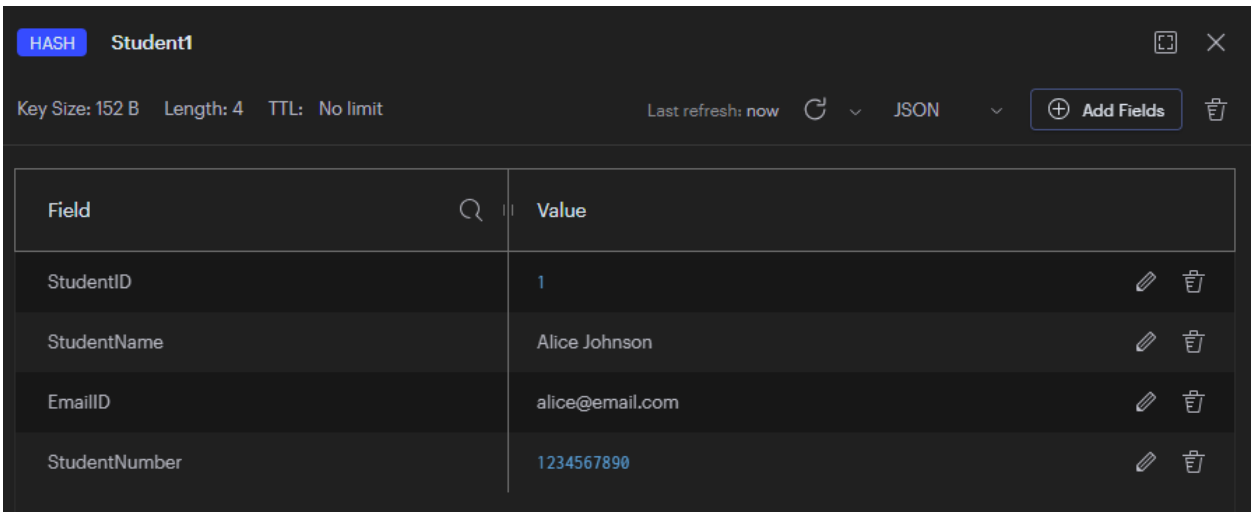


Admin enters student id -> Admin clicks on Load Student Details button

4)

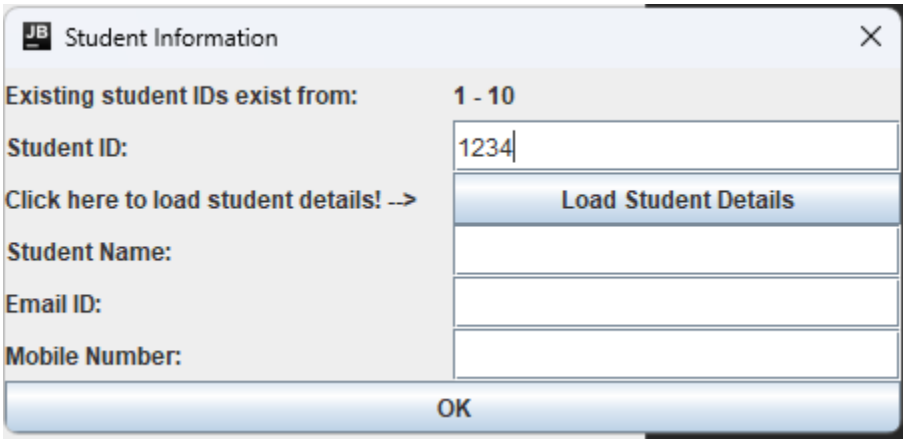


Student details gets updated from Database if student exists.



Student with Student ID: 1 in Redis Database

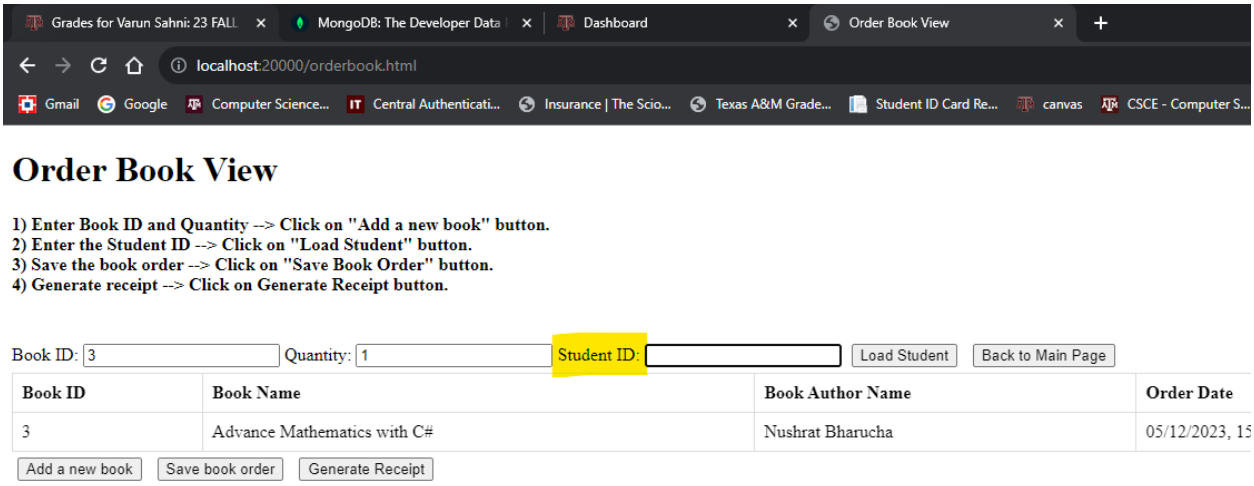
Validations:
If Admin enters an invalid student ID



Pop-up message appears stating that this student does not exists

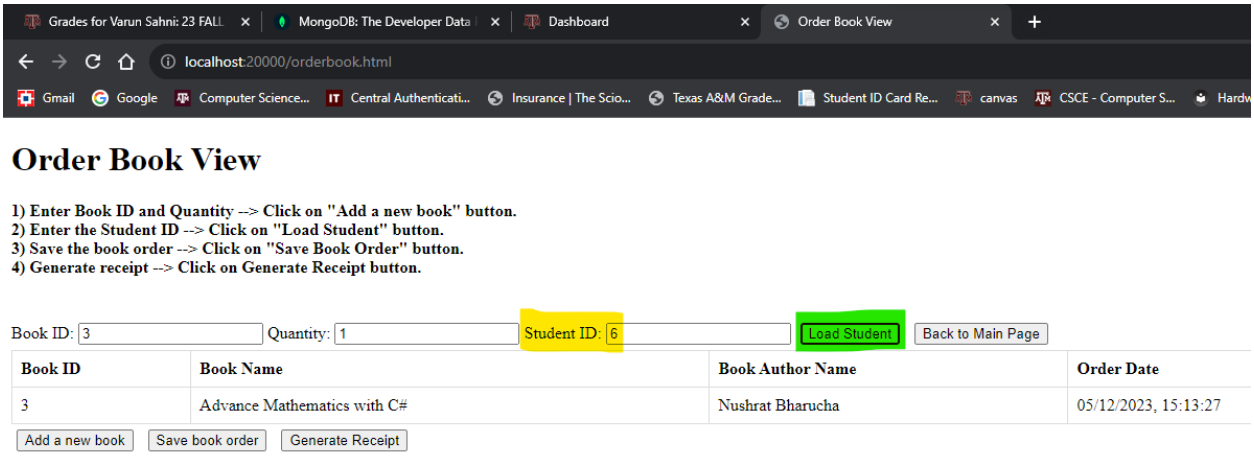
WEB APPLICATION UI

1)



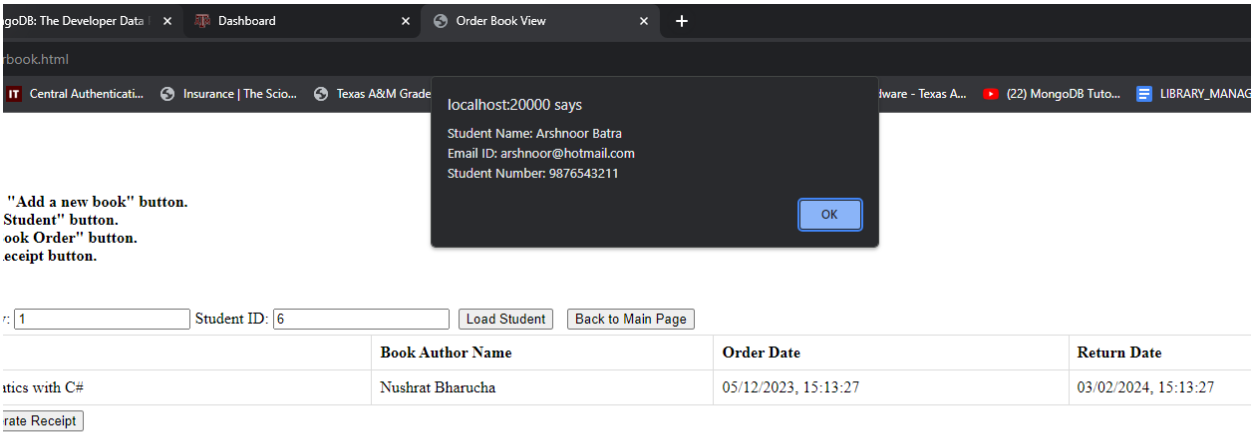
Order Book View page → Admin enters the Student ID

2)



Admin clicks on “Load Student” button

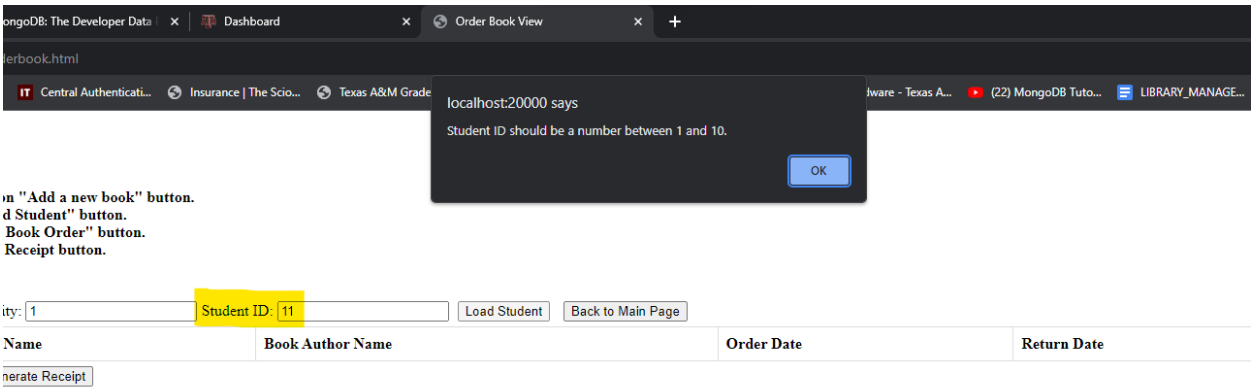
3)



Student Details alert pops up on the page and it shows the Student Name, Email ID, and Student Number

4)

Validations:



In case admin enters Student Id not between 1 to 10 → Invalid Student ID alert pops up on the page

8th Use Case:

<table>
<tr><td>Use Case</td><td>Create Order</td></tr>
<tr><td>Description</td><td>The administrator shall be able to create an order (OrderBook) for a student using library management system.</td></tr>
<tr><td>Precondition</td><td>The book should be available in the Database of the Library Management System.
The student should exist in the Library Database.
The administrator should have accessed to Book Order View page
The administrator must have added the books to Book Order View page
The administrator must have entered and loaded the student’s details and have clicked on Load student details button as done in previous use case</td></tr>
<tr><td>Ordinary Sequence</td><td>
<table>
<tr><td>Step</td><td>Action</td></tr>
<tr><td>1</td><td>The administrator will click on Save Book Order button on Book Order View screen. A “Student Information” dialog box will appear.</td></tr>
<tr><td>2</td><td>The administrator will enter Student Id, and Click on “Load Student Details” button.</td></tr>
<tr><td>3</td><td>System will auto-populate the student’s name, email id and mobile number. In case, student does not exist in the database, a pop-up window will appear saying that this student does not exist in the database.</td></tr>
<tr><td>4</td><td>After the student details are loaded, system will generate an order, save it to the database, and order confirmation window will appear on screen.</td></tr>
</table>
</td></tr>
<tr><td>Postcondition</td><td>The order will be saved in the database (Table: OrderBook).
Book quantity for respective book decreases in database.</td></tr>
<tr><td>Comment</td><td>The student Id field in each order shall contain the student Id of the respective student ordering the book.</td></tr>
<tr><td>Exceptions</td><td>The student and the book should be present in the database.</td></tr>
</table>

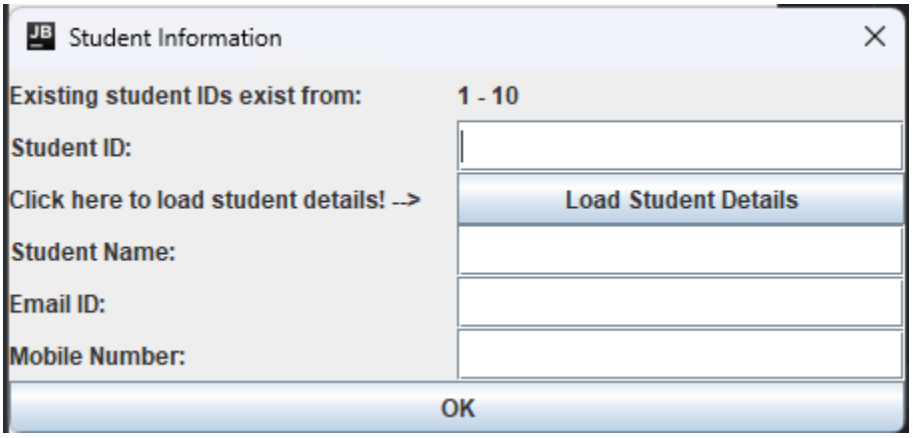
Actor Action	**System Response**
1 The administrator will click on Save Book Order button on Book Order View screen.	2 System will show a Student Information dialog box.
3 The administrator will enter the Student Id for which order needs to be made.	
4 The administrator will click on Load Student details button	5 System will validate the student details from the database and auto-populate the student’s name, email id and mobile number.
6 The administrator will click on OK button	6 System will save the order in database, and display order confirmation page

DESKTOP APPLICATION UI

1)

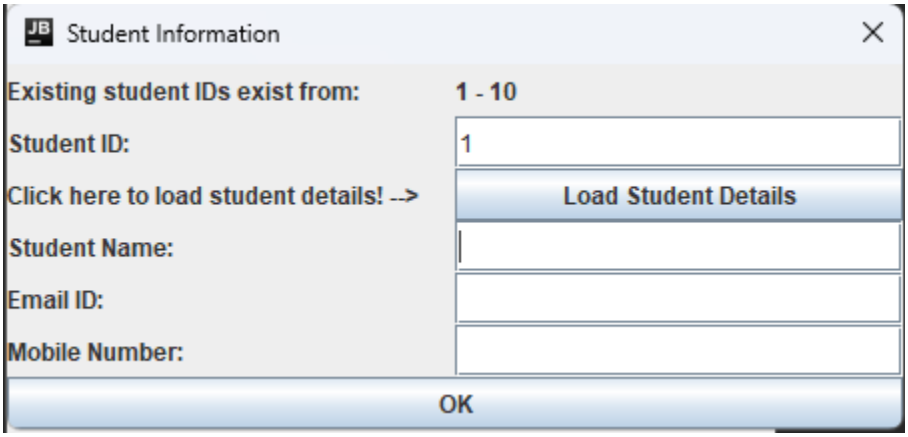
Book Order View page -> Admin clicks on Save book order button post adding the books

2)



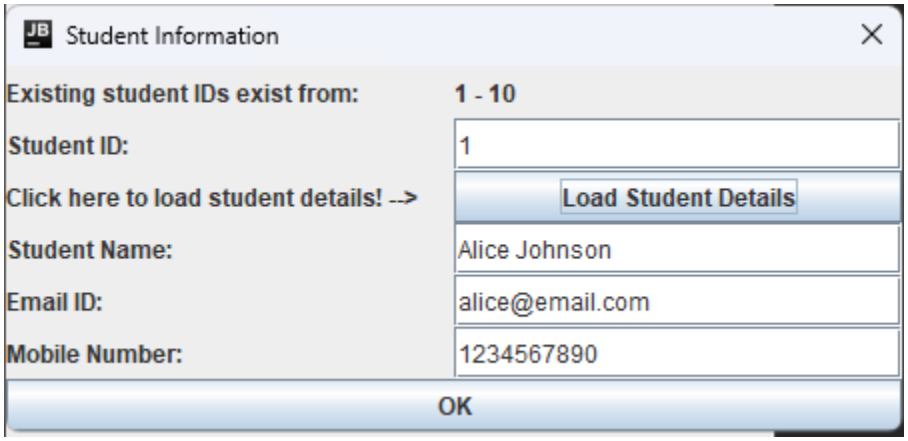
Student Information Dialog box appears

3)



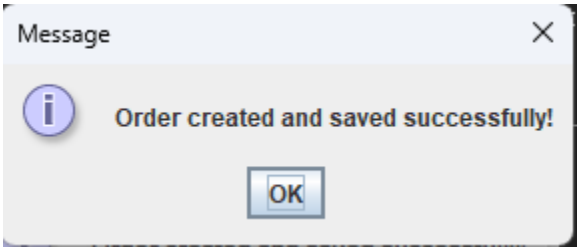
Admin enters student id -> Admin clicks on Load Student Details button

4)

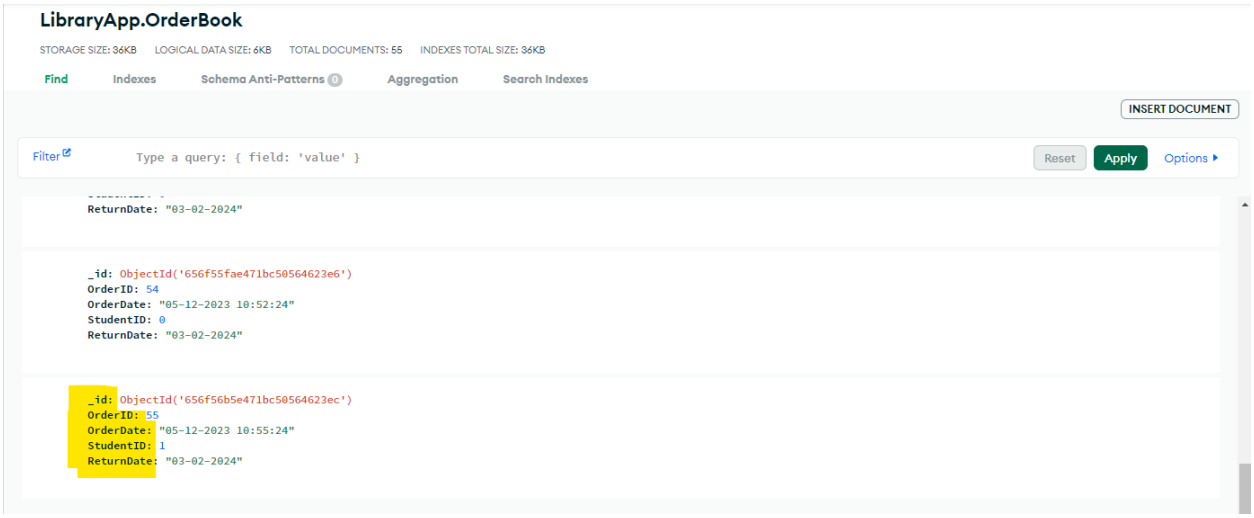


Student details gets updated from Database if student exists.

5)



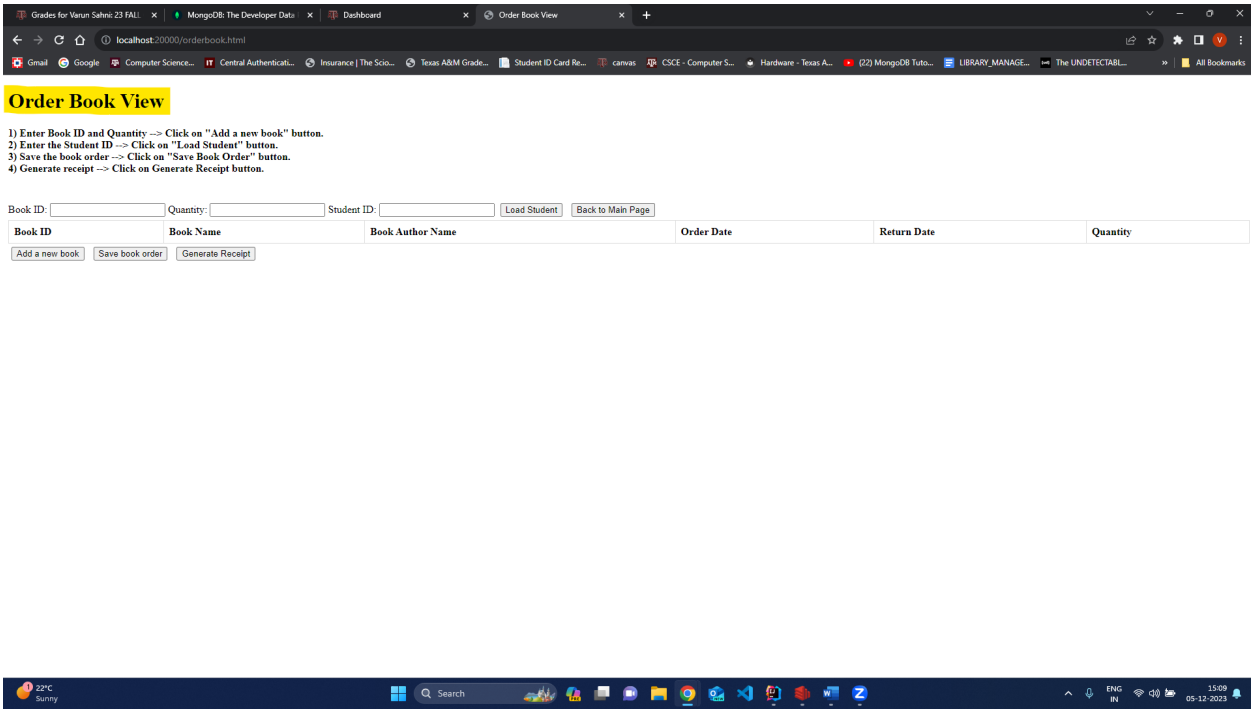
Order gets generated and saved to the MongoDB database**.**



MongoDB Database – OrderBook collection

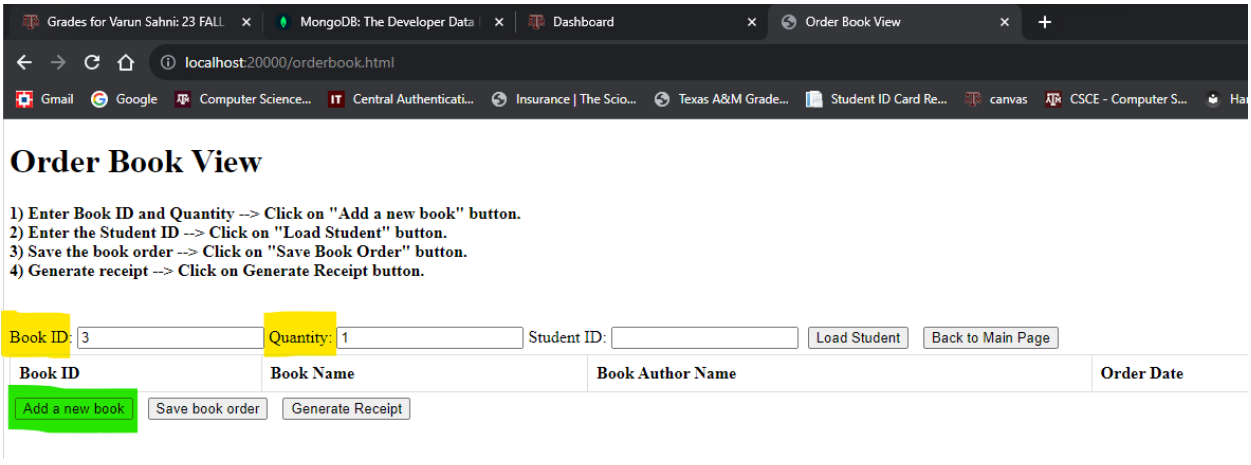
WEB APPLICATION UI

1)



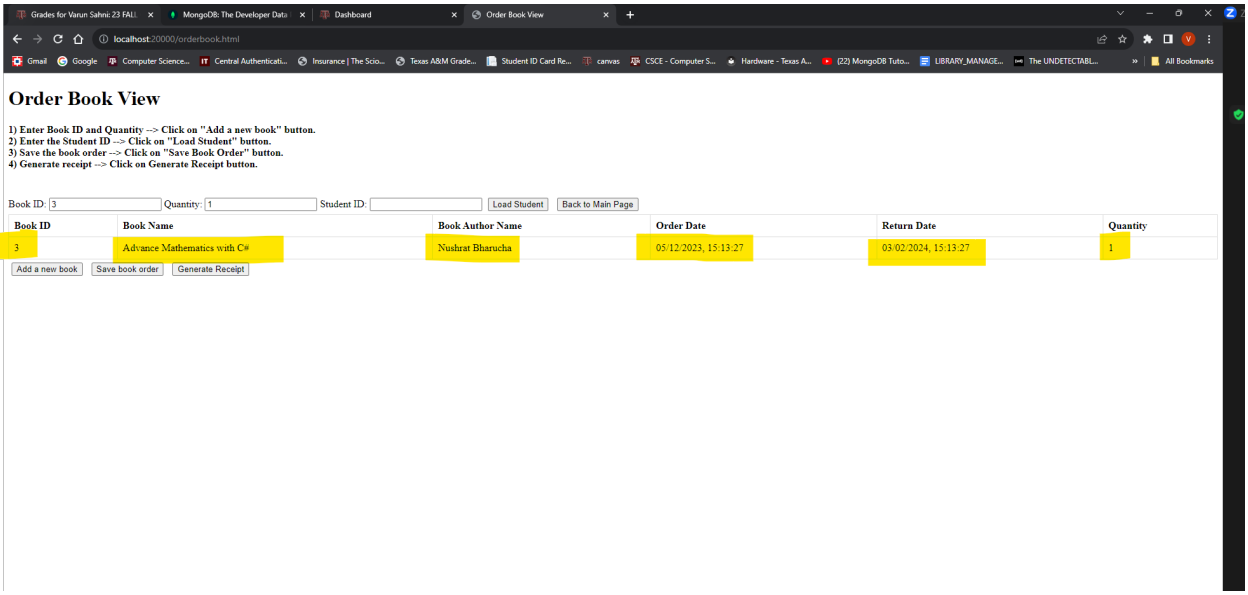
Book Order View screen appears

2)



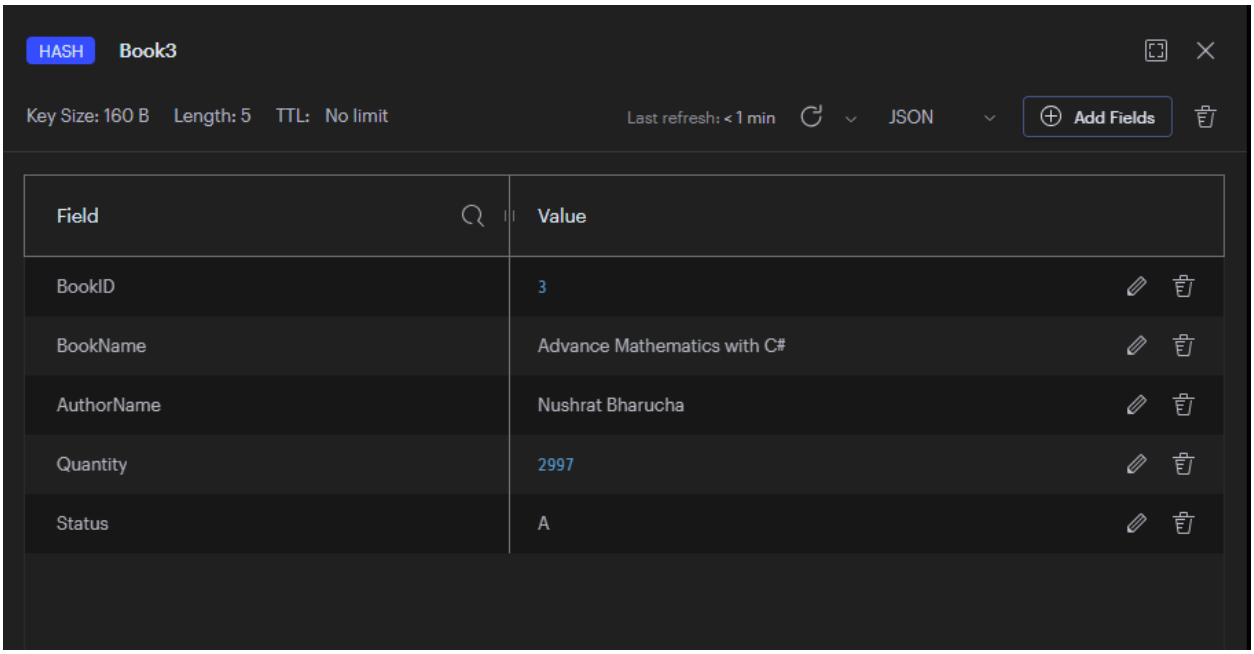
Admin enters Book ID and Quantity → Then clicks on Add a new Book button

3)



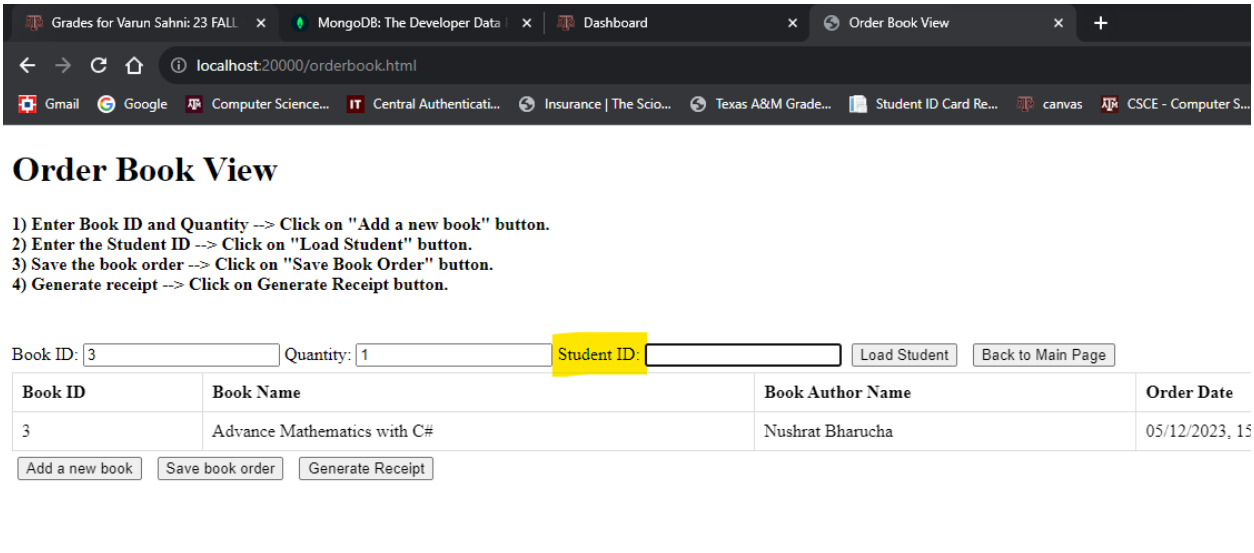
Book ID, Book Name, Book Author Name, Order Date, Return Date, and Quantity gets added to the Cart and auto-appears on the page in tabular format

4)



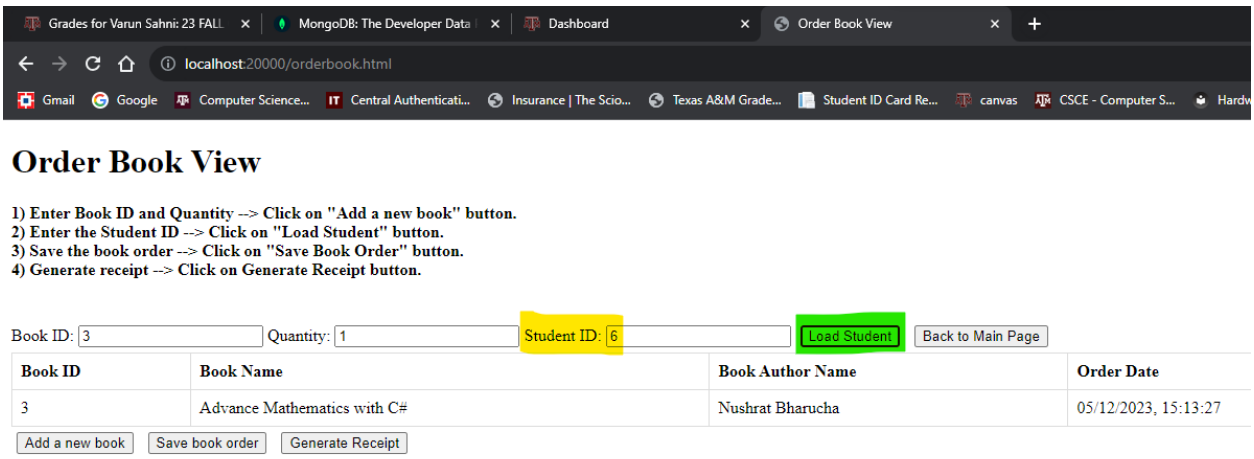
This book exists in Redis Database

5)



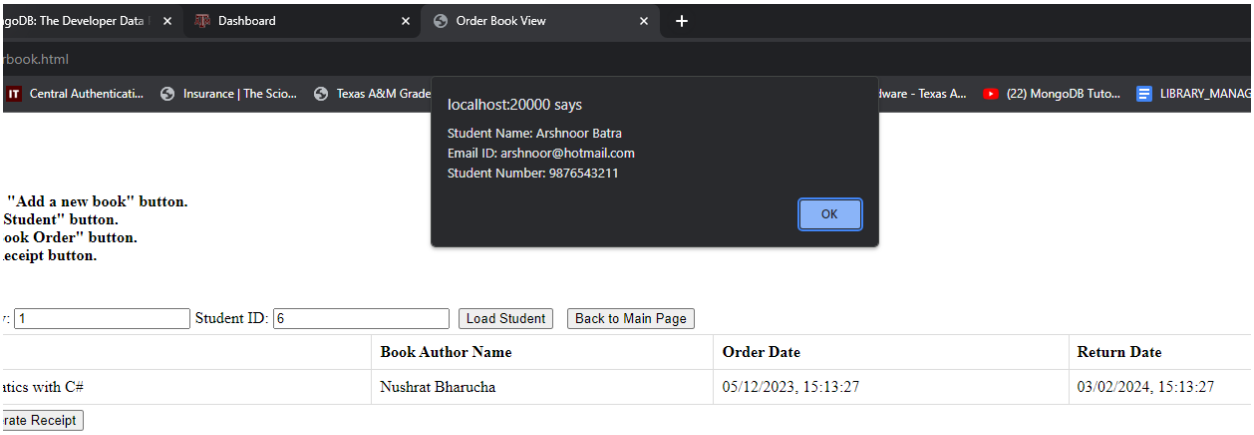
Admin enters the Student ID

6)



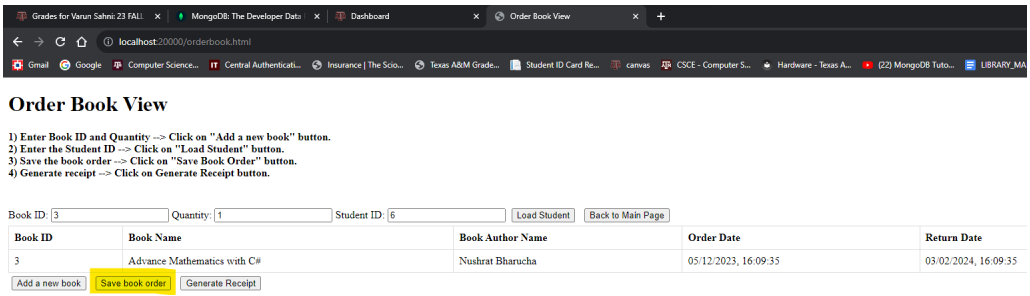
Admin clicks on “Load Student” button

7)



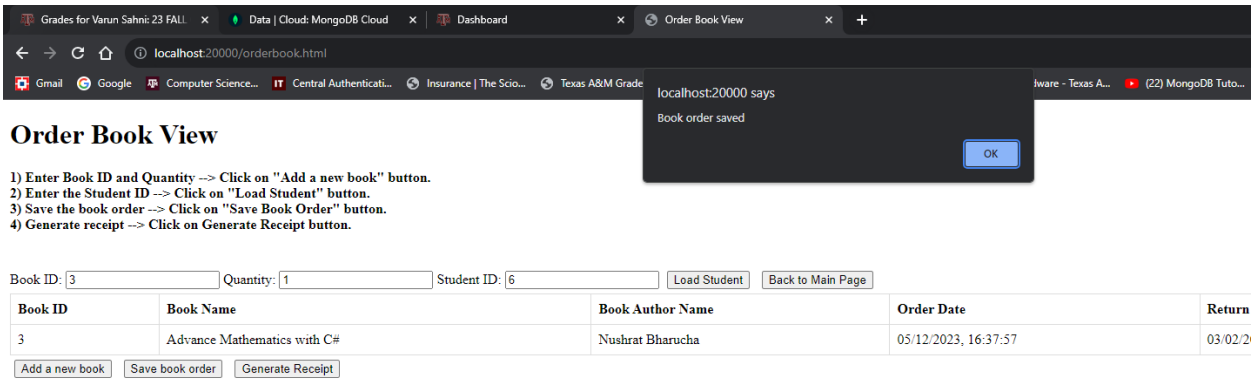
Student Details alert pops up on the page and it shows the Student Name, Email ID, and Student Number

8)



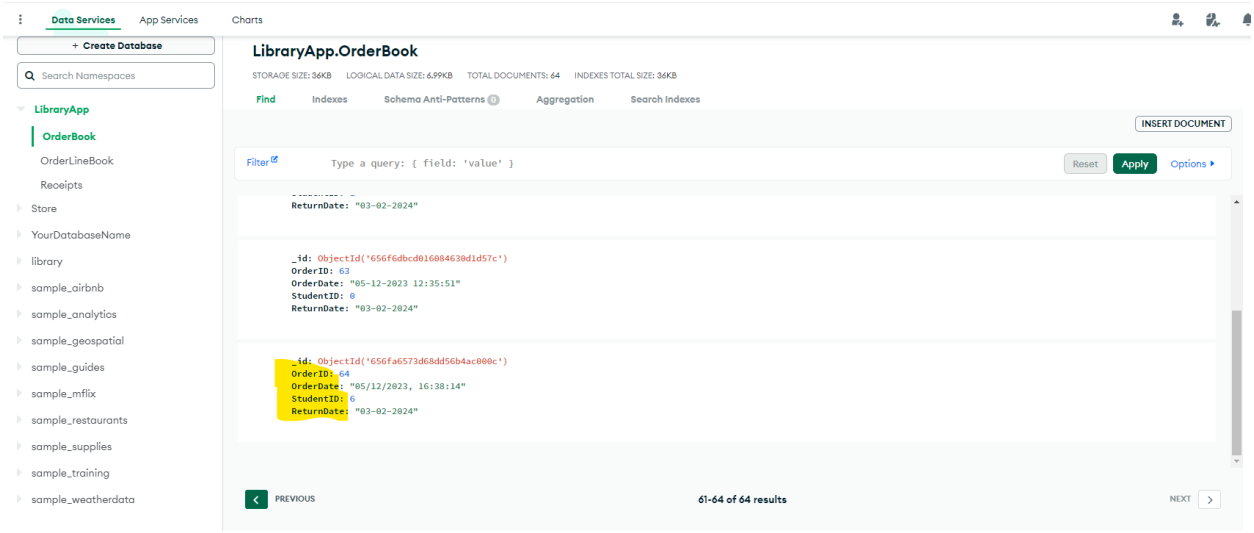
Admin clicks on the “Save book order” button

9)



Book Order gets generated and “Book order saved” alert pops up on the page → Admin to click on OK button in alert

10)



Book Order gets saved in MongoDB database OrderBook collection

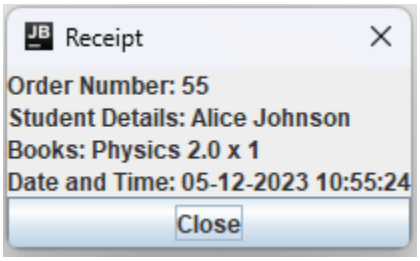
9th Use Case:

Use Case	**Generate Receipt**
Description	The administrator shall be able to generate a receipt.
Precondition	The order should have been generated and order confirmation page should have been appeared on screen
Ordinary Sequence	<table><tr><td>Step</td><td>Action</td></tr><tr><td>1</td><td>The administrator will click on OK button on the order generation confirmation age.</td></tr><tr><td>2</td><td>The system will generate a receipt and receipt page will be displayed on screen.</td></tr></table>
Postcondition	The receipt will be saved in database (Table: Receipt). The receipt will be saved in the database.
Comment	Receipt Table contains the books information.
Exceptions	The student and the book should be present in the database.

Actor Action	**System Response**
1 The administrator will click on OK button on order confirmation page	2 System will generate a receipt, save it to the database and display receipt dialog box.

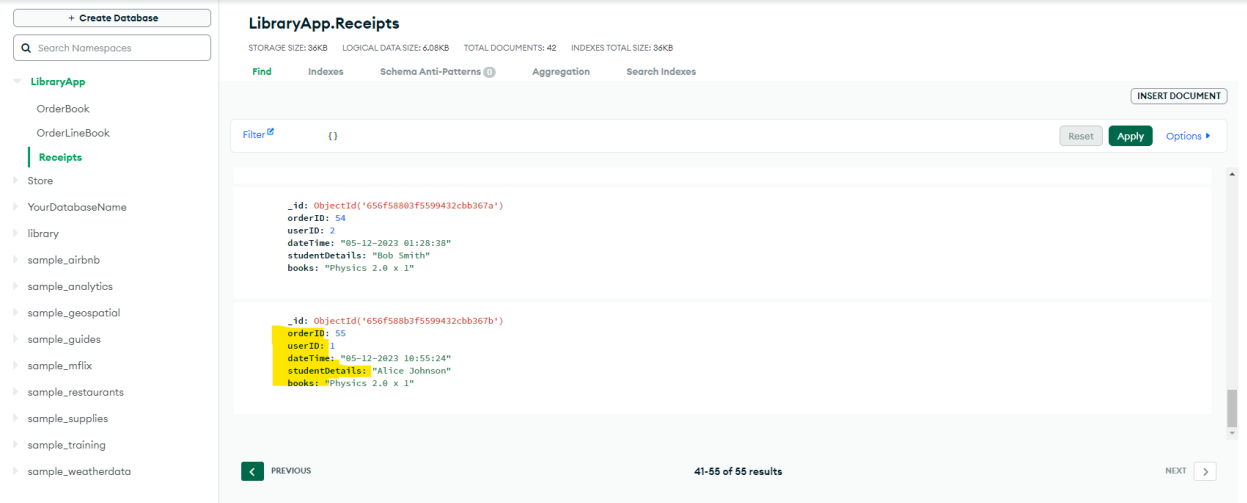
DESKTOP APPLICATION UI

1)



Receipt gets generated

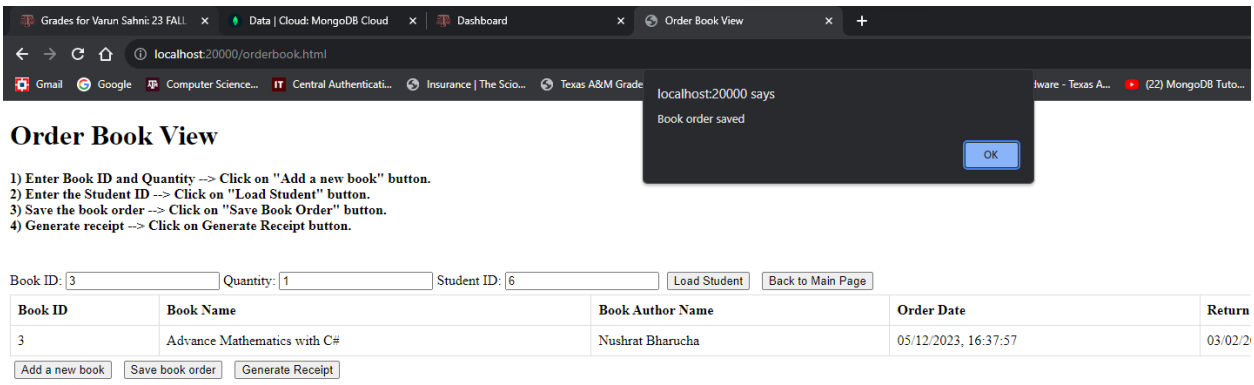
2)



Receipt gets saved in the MongoDB Database – Receipts collection

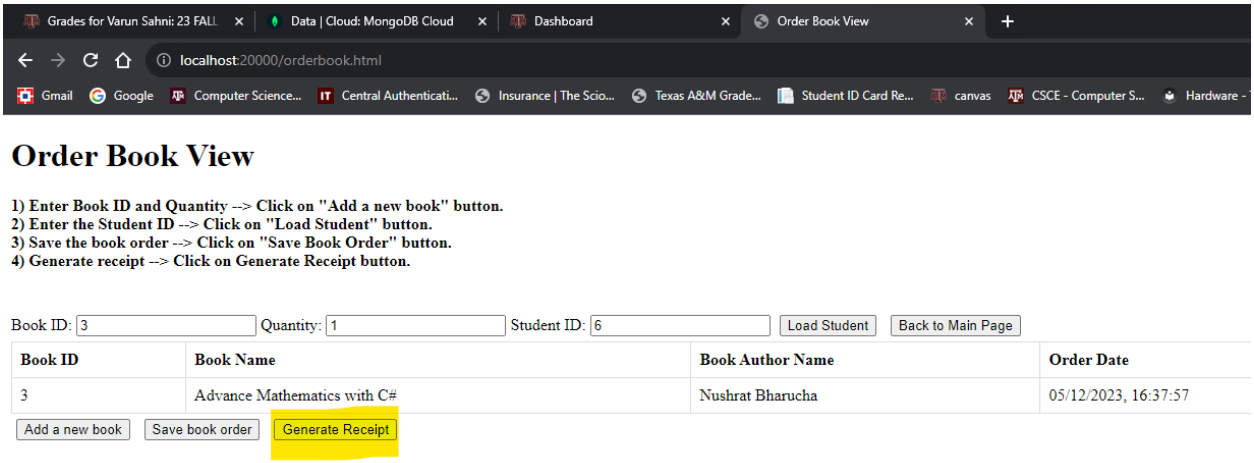
WEB APPLICATION UI

1)



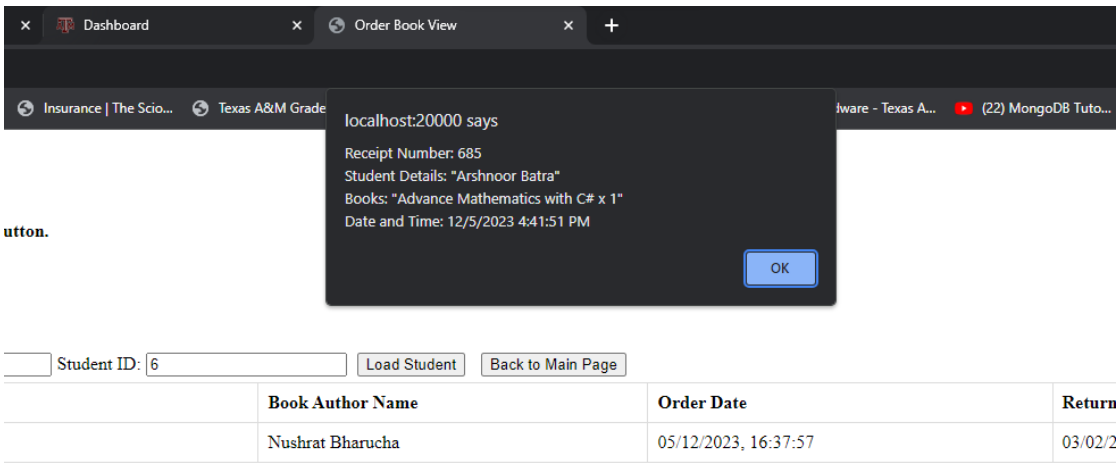
“Book order saved” alert pops up on the page → Admin to click on OK button in alert

2)



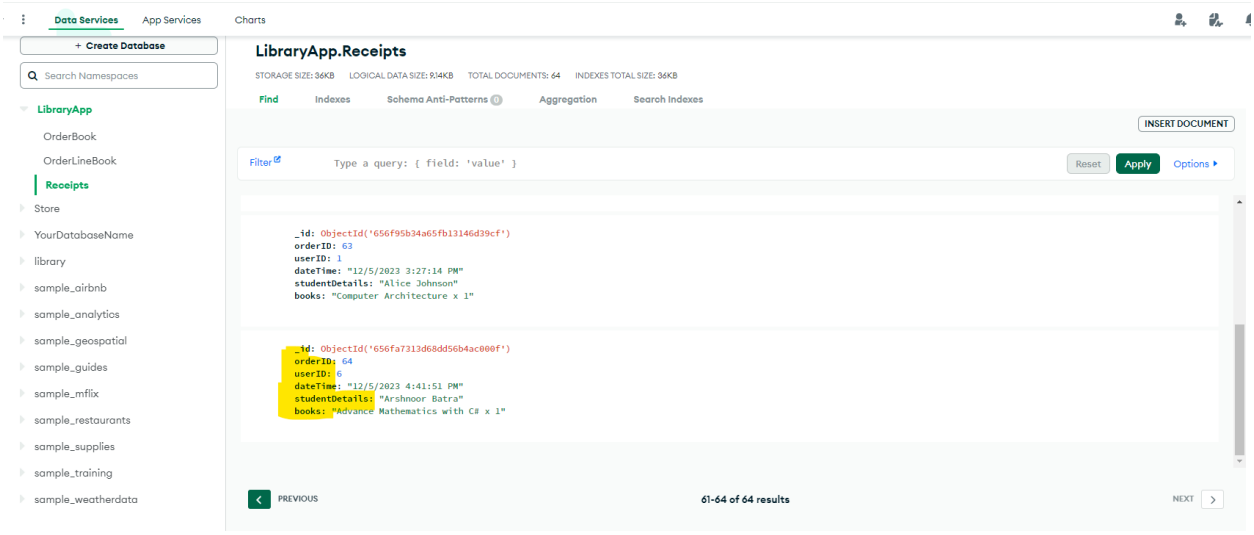
Admin to click on Generate Receipt button

3)



Receipt gets generated and Receipt Alert pops up on the page

4)



Receipt gets saved in the MongoDB Database – Receipts collection

Ques 2) Database design: Description of data entities and relationships, entity-relationship diagram, sample data.

A2. Below is the description of the data entities and relationships in our Library Management System App.

a. **Entity:** User
   Attributes: UserID (Type: Int) (Primary Key)
   UserName (Type: String)
   Password (Type: String)
   FullName (Type: String)

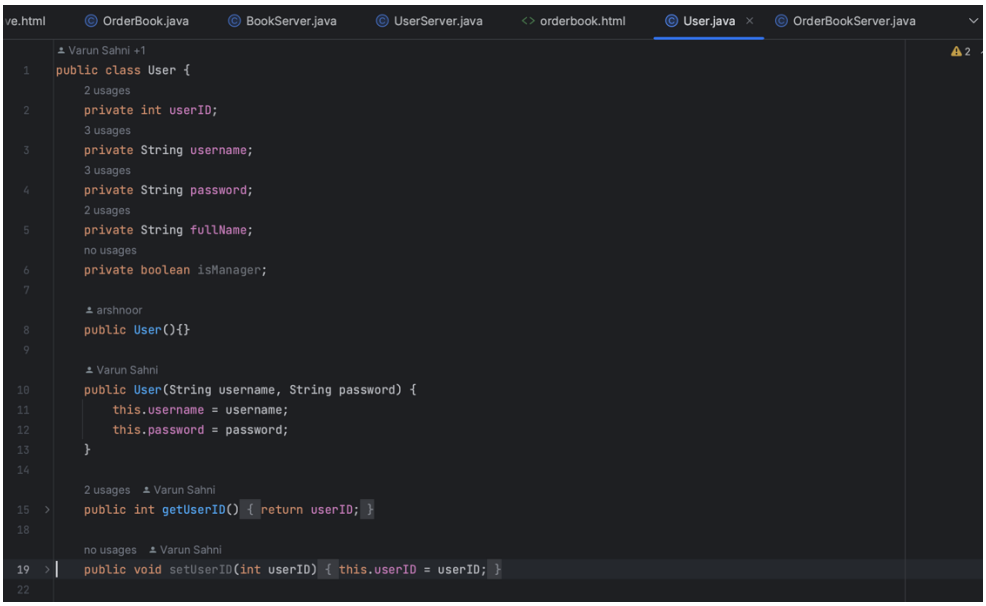
b. **Entity:** Book
   Attribute: BookID (Type: Int) (Primary Key)
   BookName (Type: String)
   AuthorName (Type: String)
   Quantity (Type: Int)
   Status (Type: String)

c. **Entity:** OrderBook
   Attribute: OrderID (Type: Int) (Primary Key)
   StudentID (Type: Int)
   OrderDate (Type: DATE)
   ReturnDate (Type: DATE)

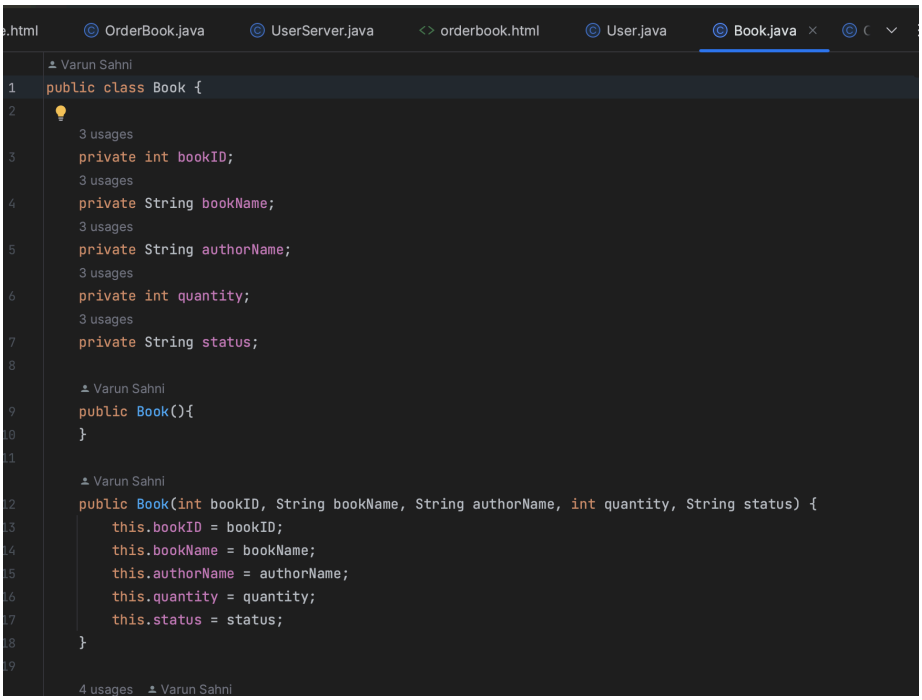
d. **Entity:** Student
   Attribute: StudentID (Type: Int) (Primary Key)
   StudentName (Type: String)
   EmailID (Type: String)
   StudentNumber (Type: String)

e. **Entity:** Receipt
   Attribute: ReceiptNumber (Type: Int) (Primary Key)
   OrderID (Type: Int)
   DateTime (Type: String)
   StudentDetails (Type: String)
   Books (Type: String)

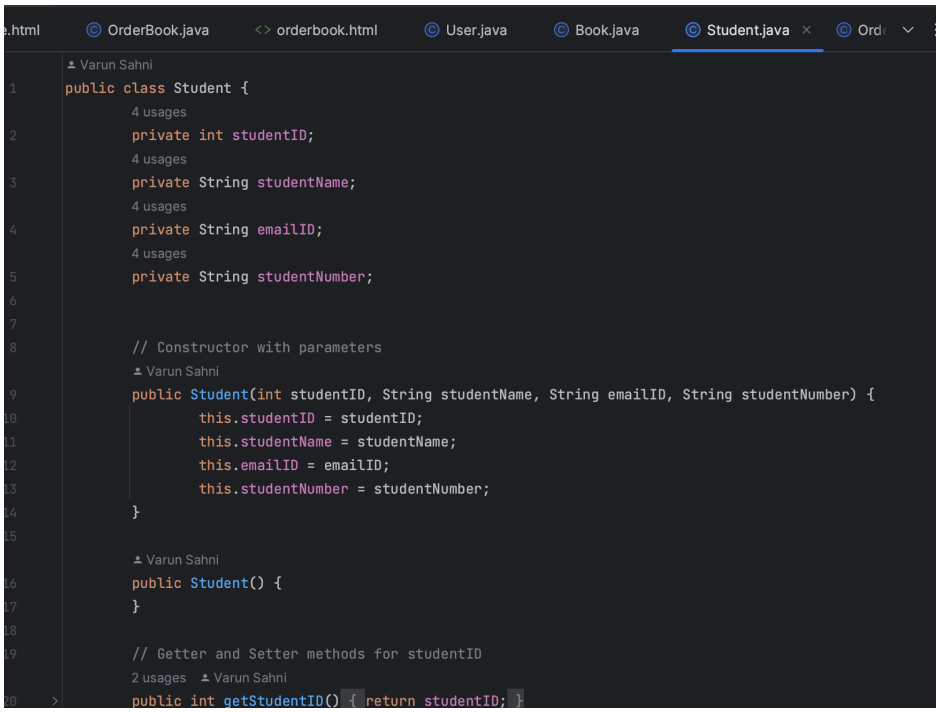
DATA ENTITIES SCREENSHOTS:



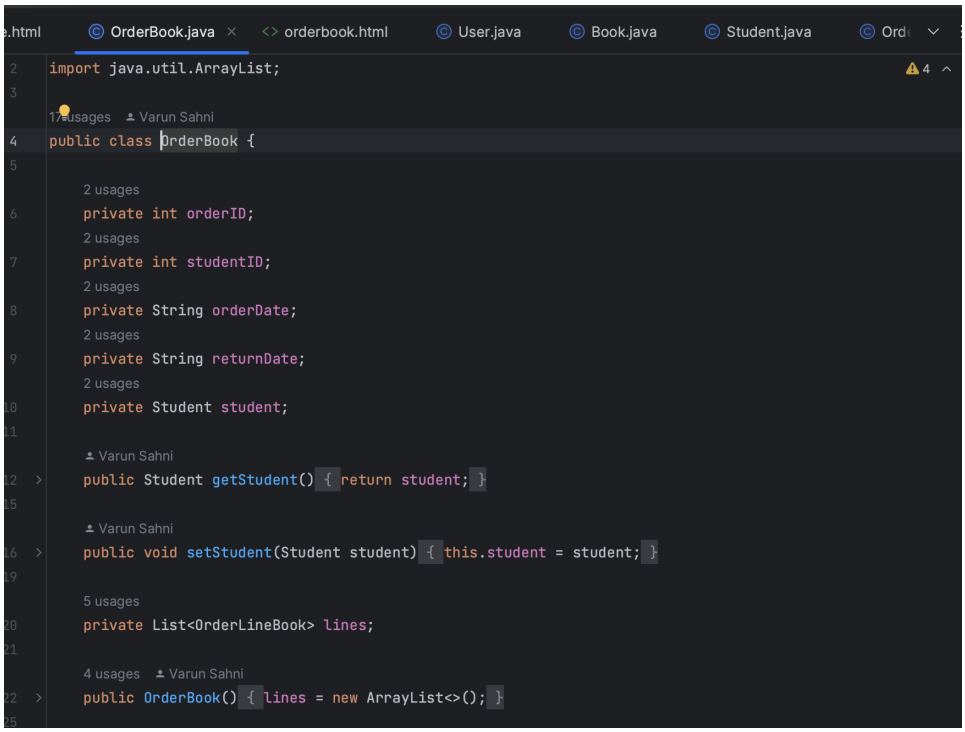
User Data Entity



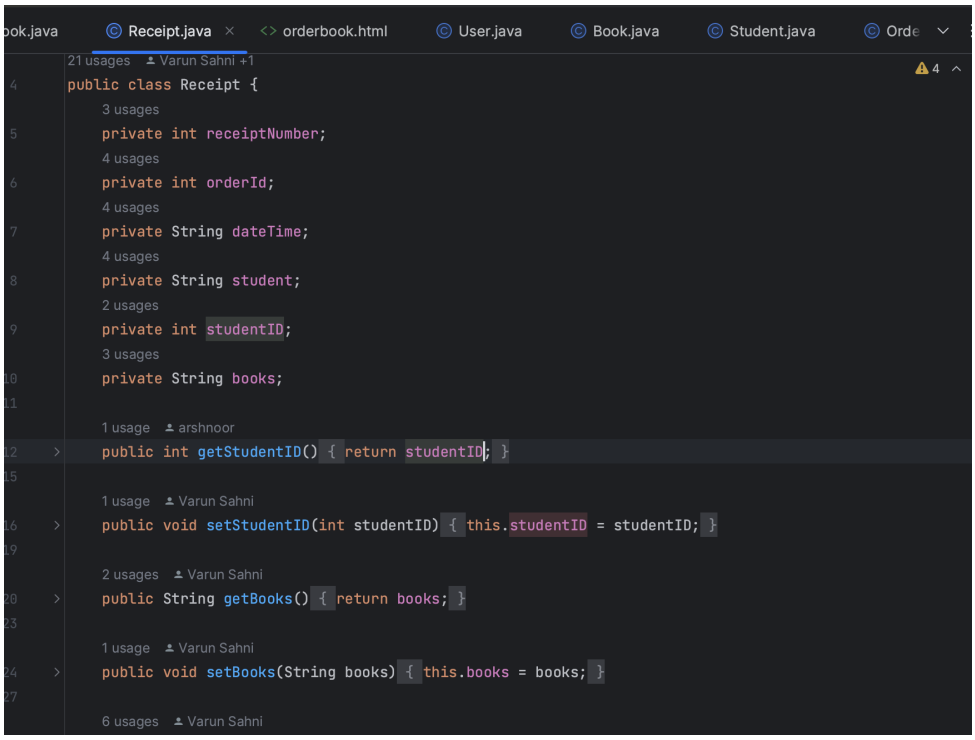
Book Data Entity



Student Data Entity



OrderBook Data Entity

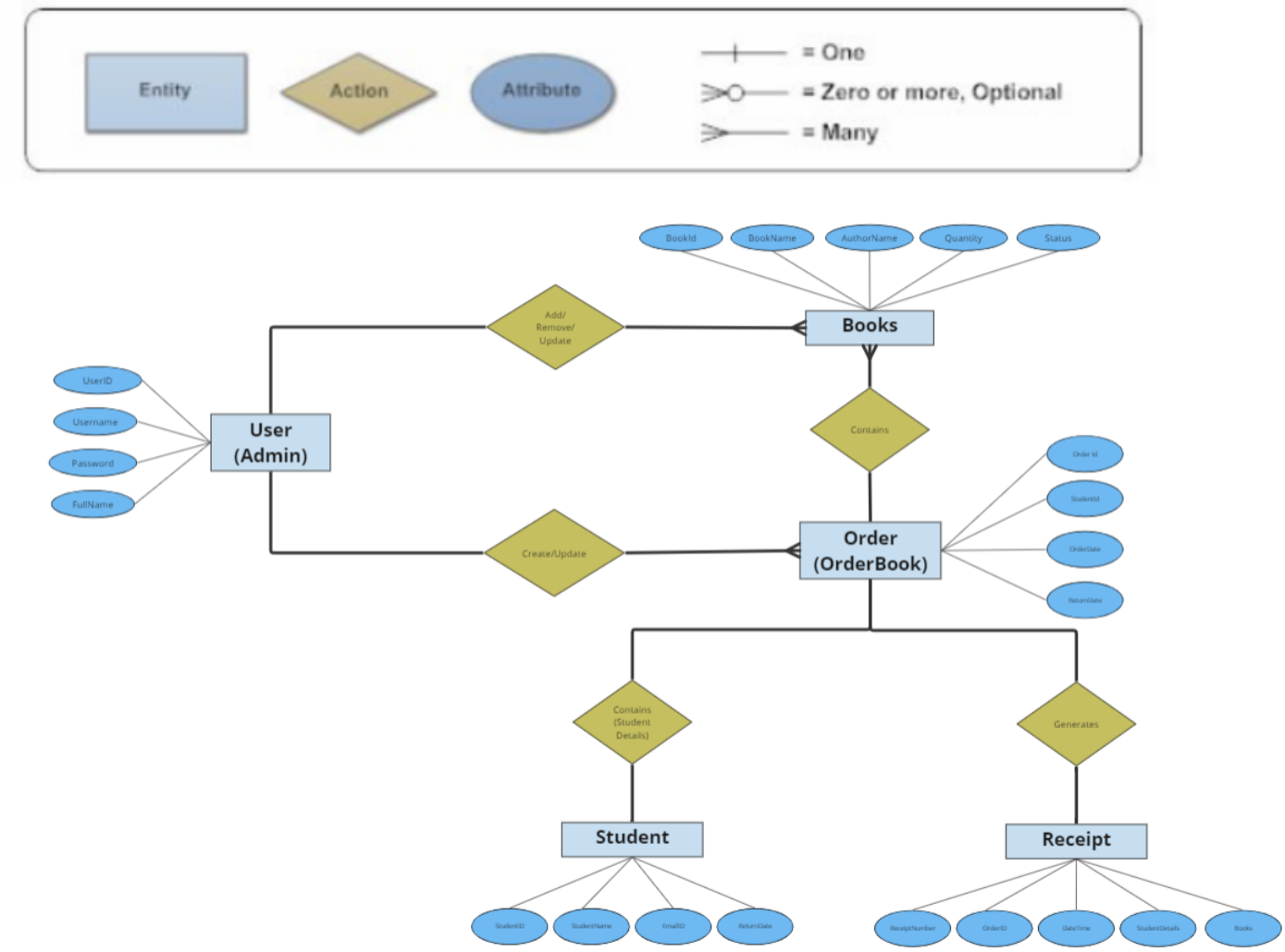


Receipt Data Entity

Relationships:

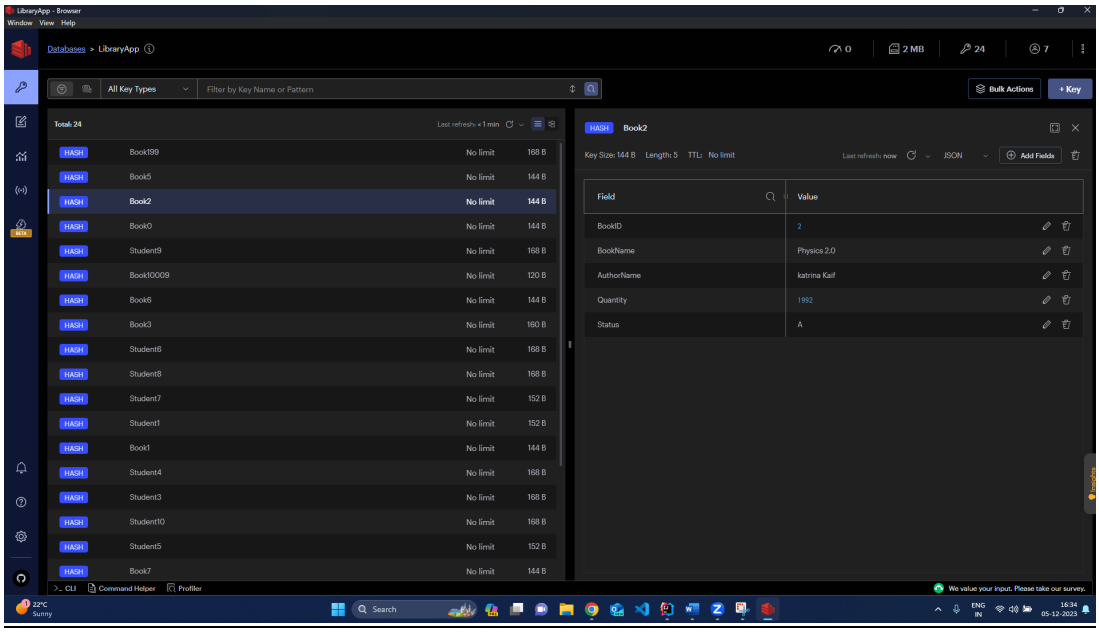
- User-Book: One-to-many (An admin or librarian can manage multiple books)
- User-OrderBook: One-to-many (An admin can place multiple orders)
- Student-OrderBook: One-to-many (A student can have multiple orders)
- OrderBook-Book: One-to-many (An order can have multiple books)
- OrderBook-Receipt: One-to-one (Every order will generate a new receipt)

Entity Relationship Diagram:

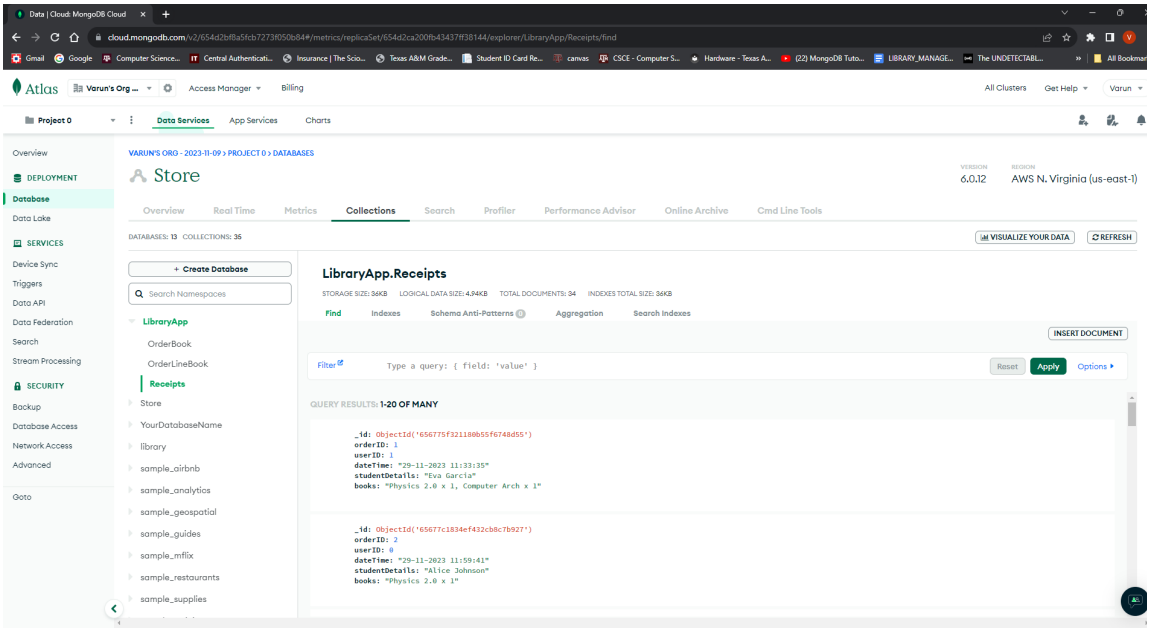


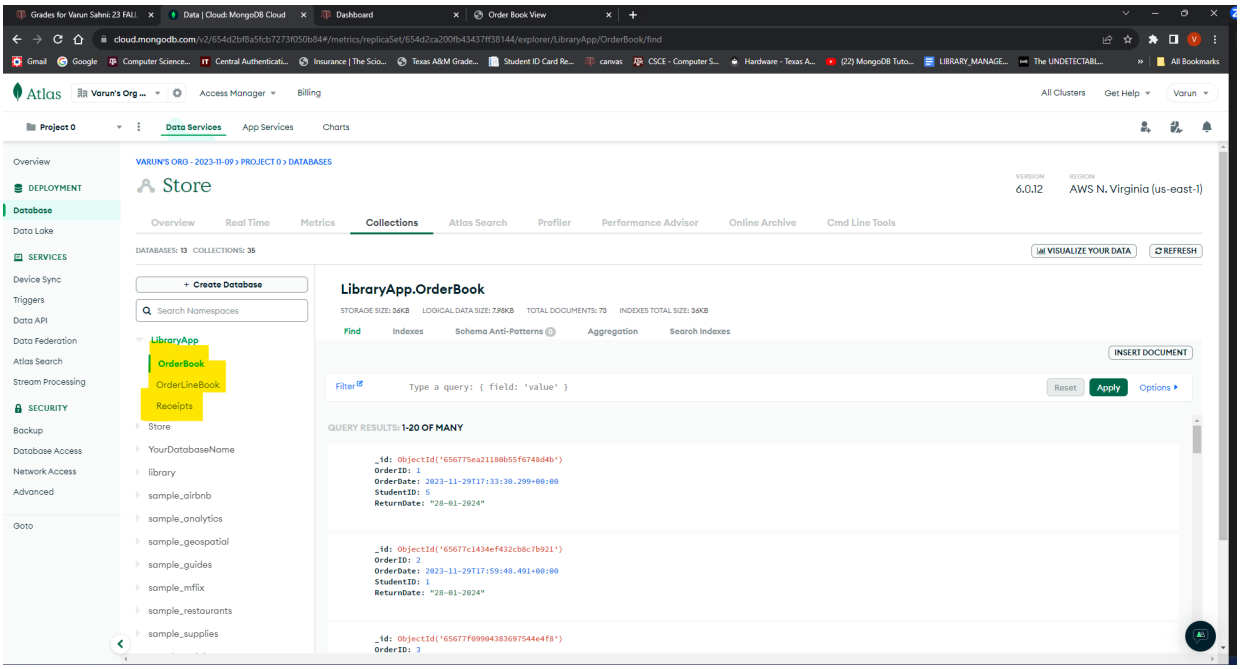
Sample Data:

Redis Database:



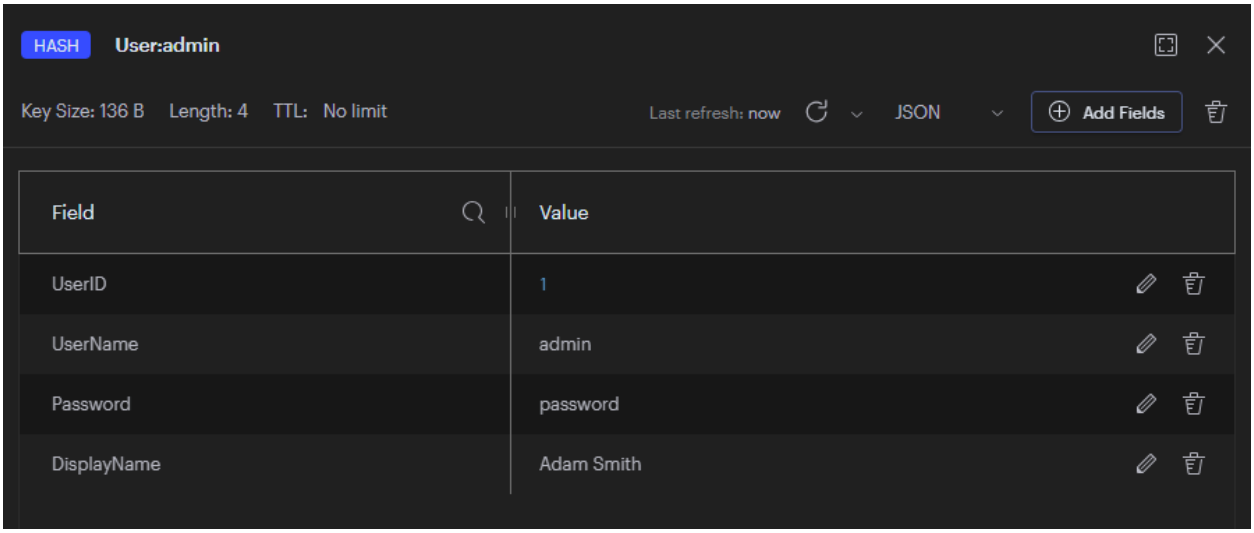
MongoDB Database:



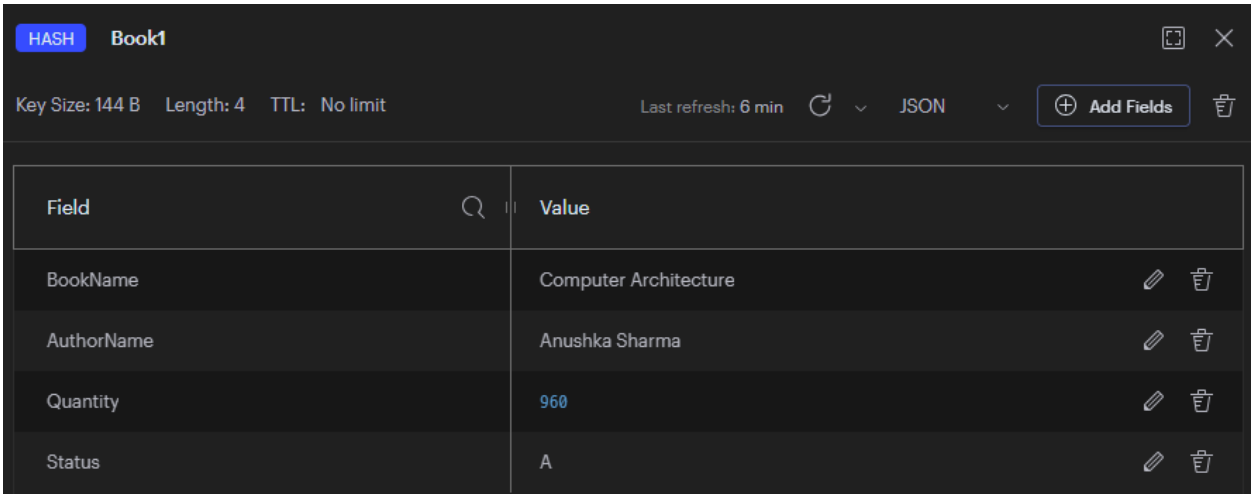


Specific Entities:

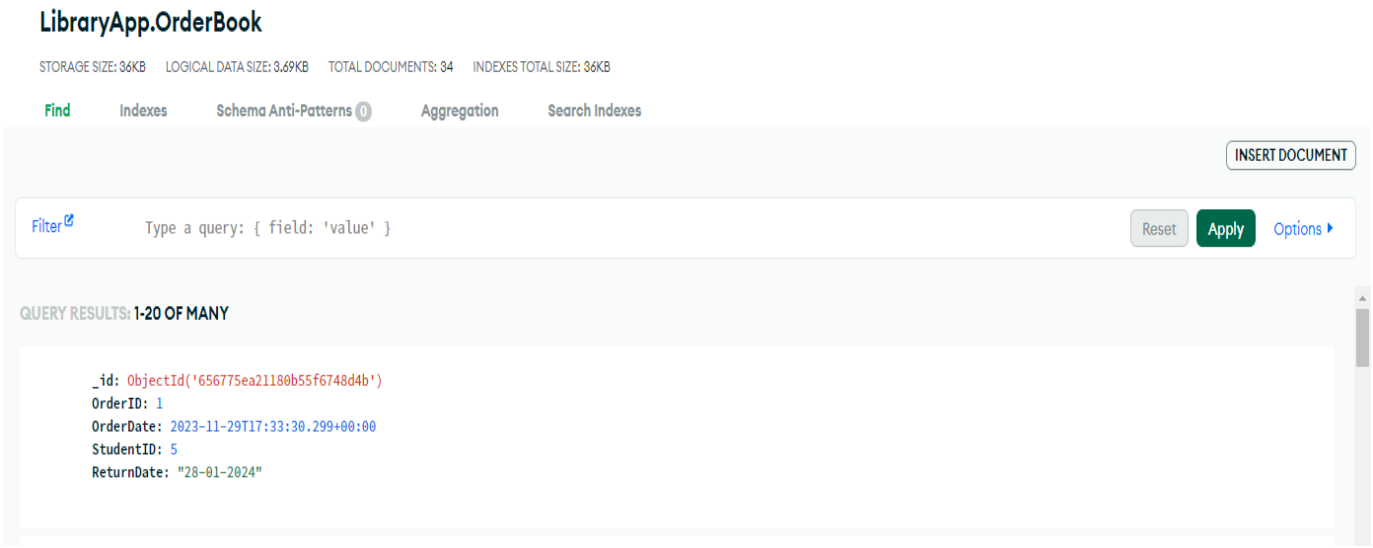
"Users" : Stored as HASH data type in Redis Database



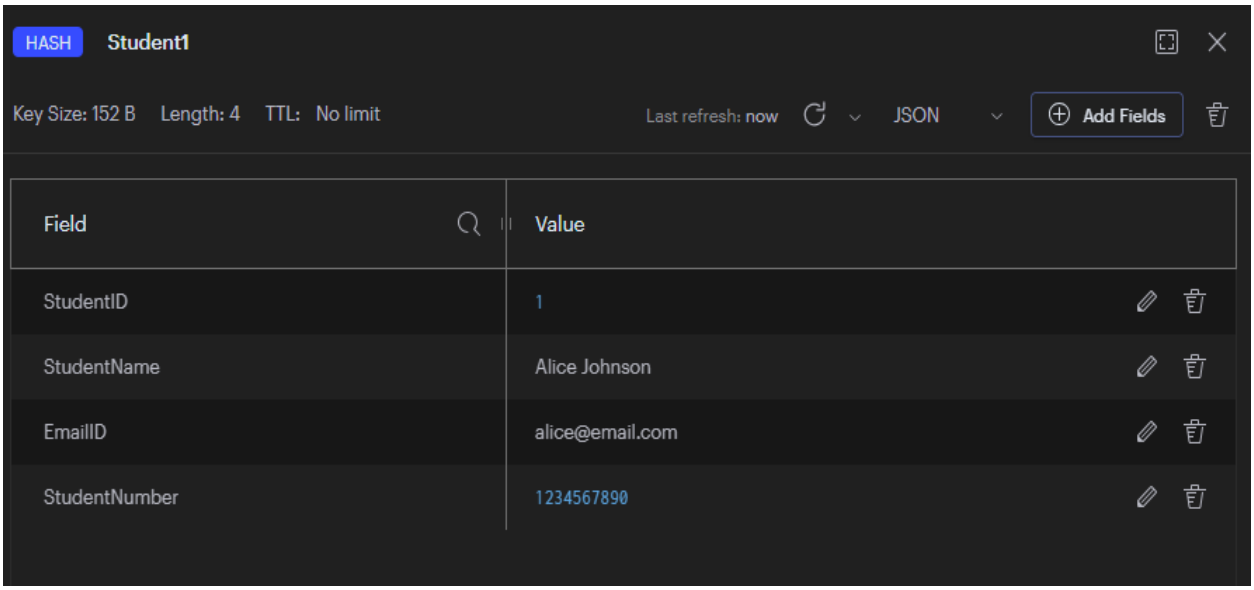
"Books" : stored as HASH datatype in Redis database. Redis is used for Books as books in the project scope of Librray Management Sytem is not a complicated entity as we only have books with single author and no reviews. Complicated data entities require Redis. But due to our project scope, we have made “Book” entity in Redis.



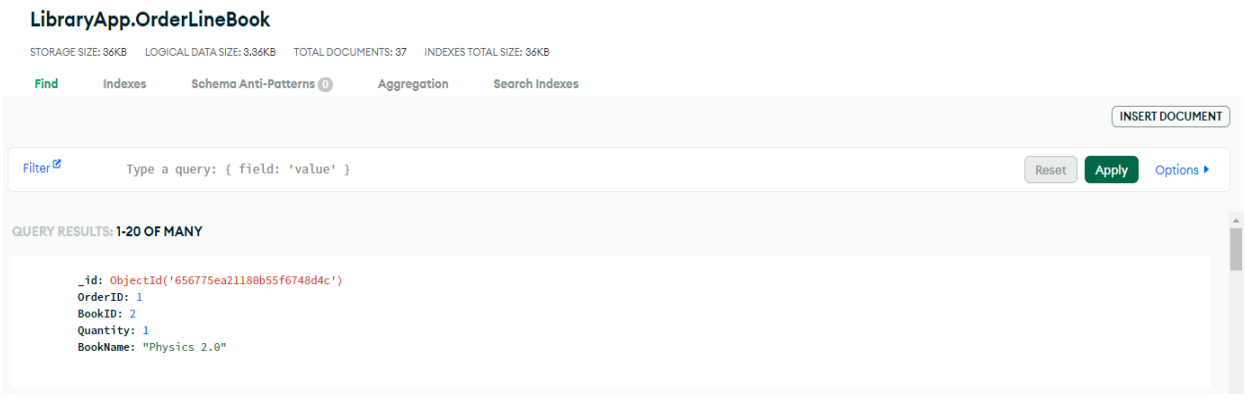
"OrderBook": Stored in MongoDB Atlas



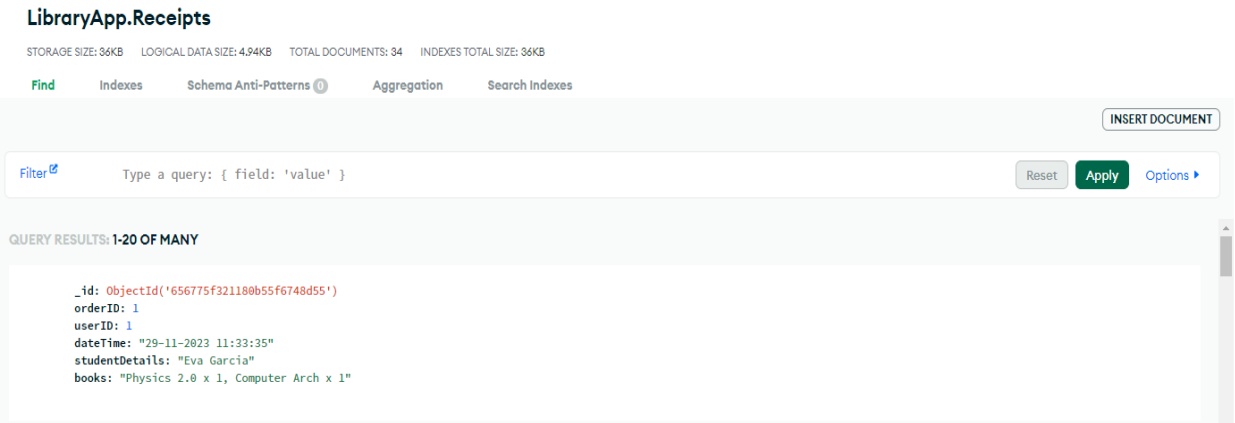
"Student" : stored as HASH datatype in Redis Database



“OrderLineBook” : Stored in MongoDB Atlas

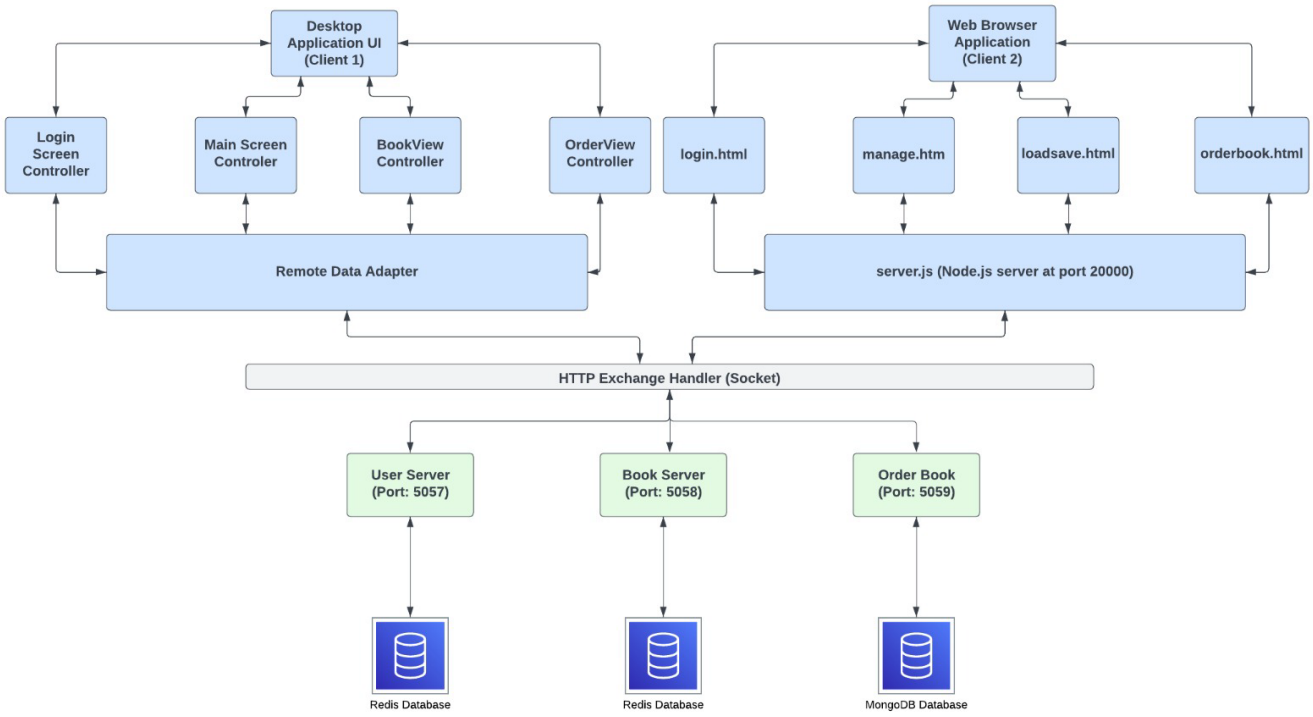


"Receipt": Stored in MongoDB Atlas



Ques 3) Architectural design: Description of client and server components; communication protocol (including data format), overall diagram of the system architecture.

Answer 3) **Client Server Architecture of Library Management System**



Client Components:

1. **Application Class:** This serves as the entry point for the client-side application. It orchestrates the interaction between various components. This class will call various other client components to load the user interfaces to interact with the user.

2. **LoginScreenController Class:** Responsible for displaying the login screen and handling user authentication. It communicates with the server for user validation through Remote Data Adapter class.

3. **MainScreen Class:** Once a user is authenticated, this class displays the main screen of the library management system, allowing users to choose between book management and order creation. This has load/save book and create order features.

4. **BookViewController Class:** This class is responsible for displaying book-related information, allowing users to add or load books. It communicates with the server to fetch

book data through Remote Data Adapter class. This sends response as json, receives response in json and converts it into book objects.

5. **OrderViewController Class:** Handles the order creation and management, enabling user to create new orders. It communicates with the server for order-related data (passing the books in the current order from UI to the server and displaying the response received from the server). This sends response as json, receives response in json and converts it into order objects.

6. **RemoteDataAdapter Class:** Handles requests that are received from the controller screens (i.e. UI) and forwards these requests to respective Servers i.e. UserServer, BookServer, OrderBookServer. Reads server response and displays it to the User Interface.

7. **Index.html file:** Facilitates updating product information in a web-based inventory system. Users input product details in a form, triggering a server update via a POST request for seamless inventory management.

8. **Loadsave.html file:** Enables users to efficiently load and save book details in a user-friendly interface. It provides functionalities for retrieving book information based on ID and saving new book details through asynchronous interactions with a server.

9. **Login.html file:** Serves as the login interface for a Library Management System, allowing users to securely access the system by submitting their credentials. Successful logins redirect users to the management dashboard.

10. **Manage.html file:** Functions as the central dashboard for managing library operations after a successful login. Users can seamlessly navigate to order books or load/save book details, with their user information dynamically displayed.

11. **Orderbook.html file:** Facilitates the process of ordering books in a library setting. Users input book and student details, and the interface dynamically updates a table, allowing for the efficient placement and tracking of book orders.

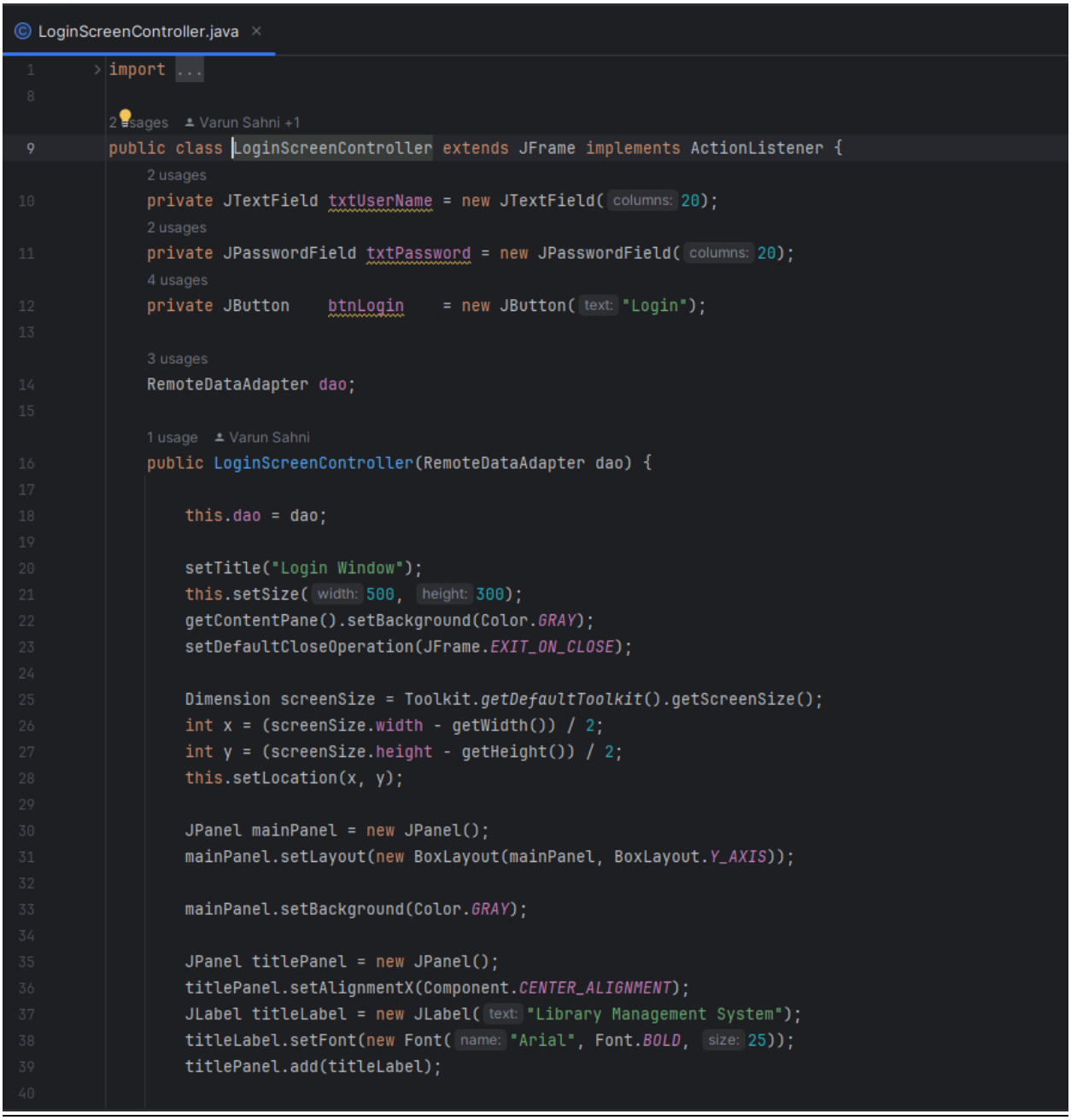
Below are the screenshots for various client components.

Application class

Application.java

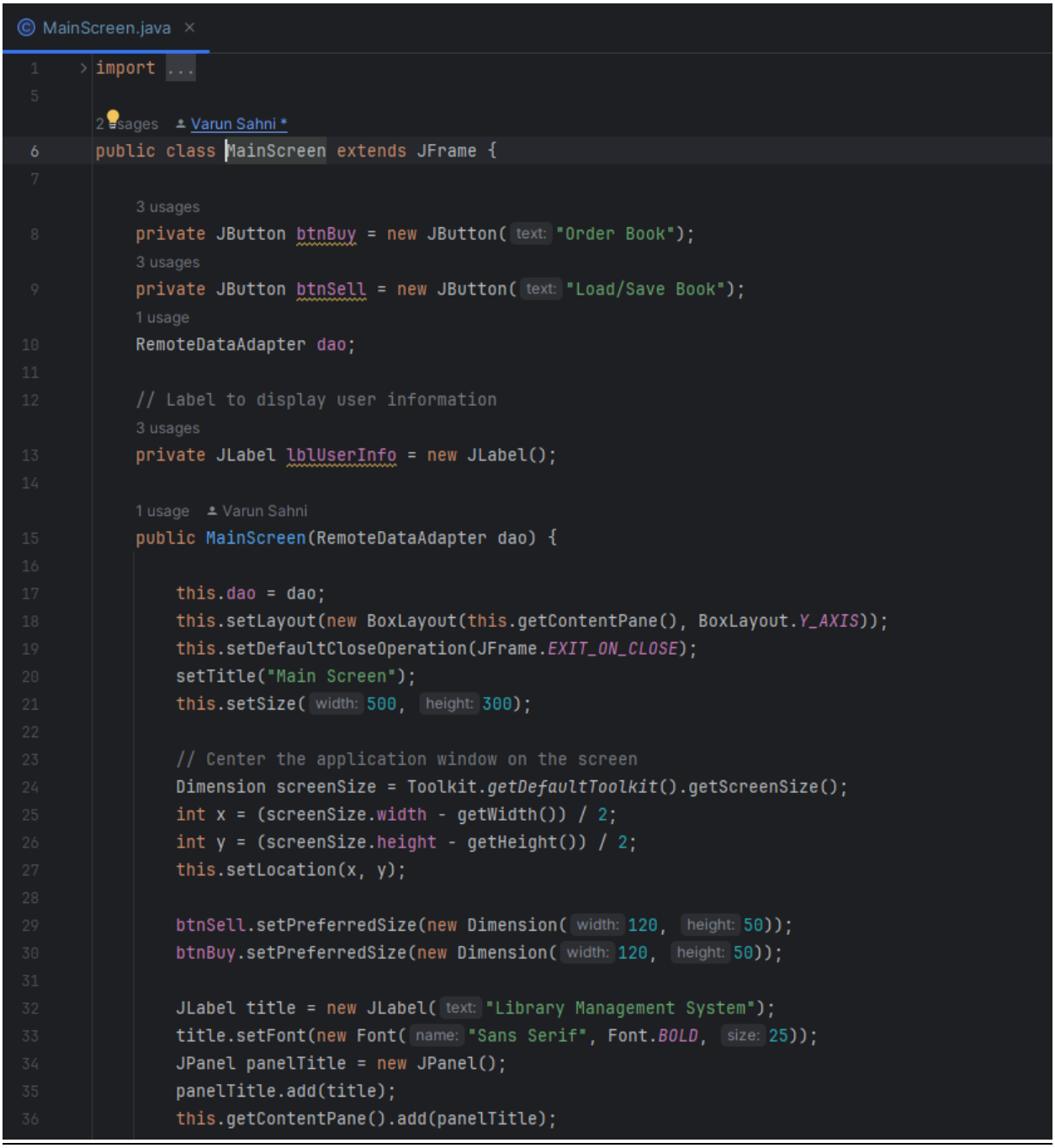
```
Varun Sahni
public class Application {

    3 usages
    private static Application instance;   // Singleton pattern

    1 usage  Varun Sahni
    public static Application getInstance() {
        if (instance == null) {
            instance = new Application();
        }
        return instance;
    }

    3 usages
    private static RemoteDataAdapter dao;

    1 usage
    private User currentUser = null;

    1 usage  Varun Sahni
    public void setCurrentUser(User user) { this.currentUser = user; }

    2 usages  Varun Sahni
    private Application() {
            dao = new RemoteDataAdapter();
            dao.connect();
    }

    Varun Sahni
    public static void main(String[] args) {
        try {
            Application application = new Application();

            LoginScreenController login = new LoginScreenController(dao);
            login.setVisible(true);

        } catch (Exception e) {
            System.out.println("e2: "+e.toString());
            e.printStackTrace();
```

Login screen controller

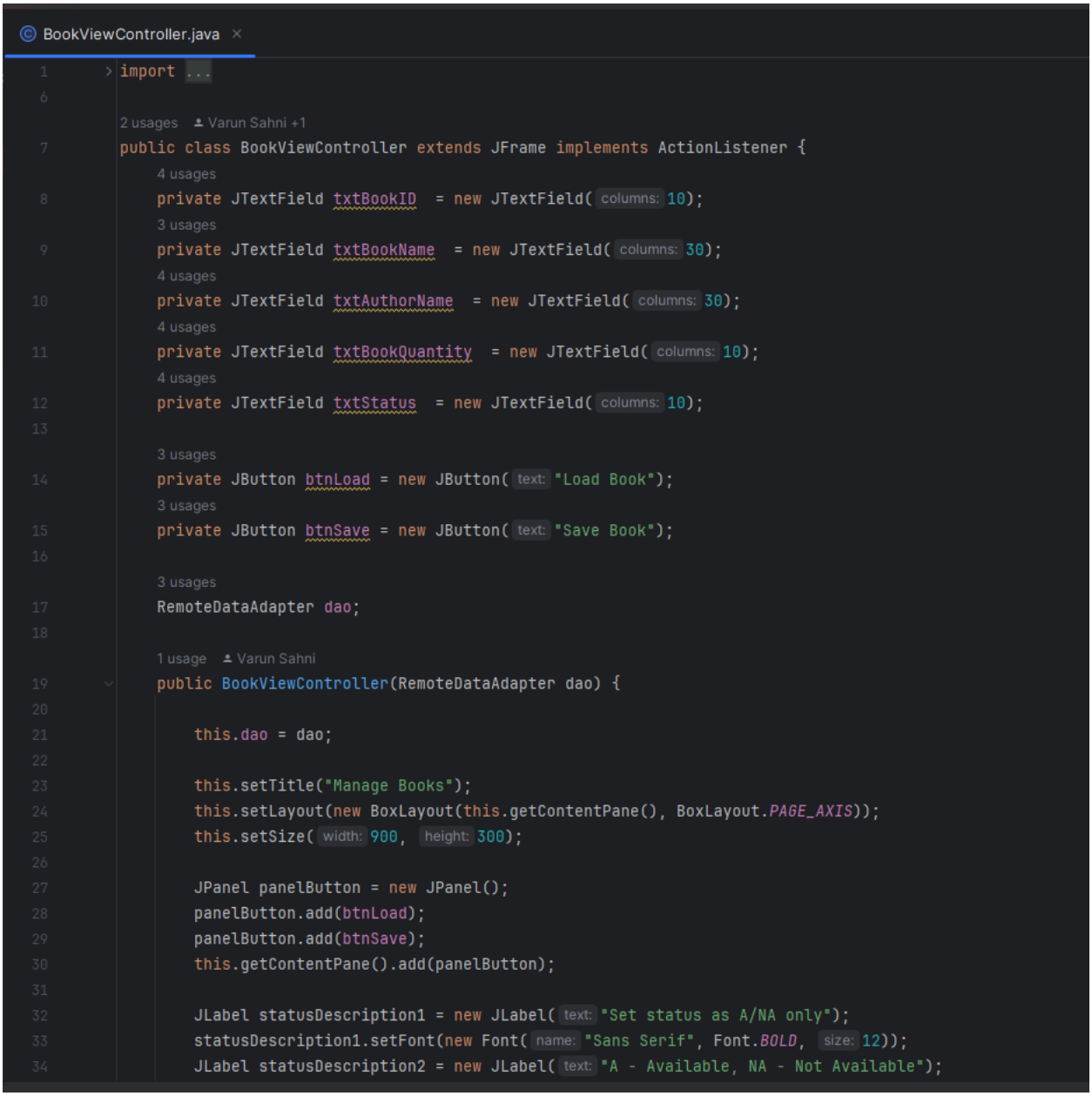


```java
import ...

2 usages  Varun Sahni +1
public class LoginScreenController extends JFrame implements ActionListener {
    2 usages
    private JTextField txtUserName = new JTextField( columns: 20);
    2 usages
    private JPasswordField txtPassword = new JPasswordField( columns: 20);
    4 usages
    private JButton    btnLogin    = new JButton( text: "Login");

    3 usages
    RemoteDataAdapter dao;

    1 usage  Varun Sahni
    public LoginScreenController(RemoteDataAdapter dao) {

        this.dao = dao;

        setTitle("Login Window");
        this.setSize( width: 500,  height: 300);
        getContentPane().setBackground(Color.GRAY);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int x = (screenSize.width - getWidth()) / 2;
        int y = (screenSize.height - getHeight()) / 2;
        this.setLocation(x, y);

        JPanel mainPanel = new JPanel();
        mainPanel.setLayout(new BoxLayout(mainPanel, BoxLayout.Y_AXIS));

        mainPanel.setBackground(Color.GRAY);

        JPanel titlePanel = new JPanel();
        titlePanel.setAlignmentX(Component.CENTER_ALIGNMENT);
        JLabel titleLabel = new JLabel( text: "Library Management System");
        titleLabel.setFont(new Font( name: "Arial", Font.BOLD,  size: 25));
        titlePanel.add(titleLabel);

```

Main screen controller

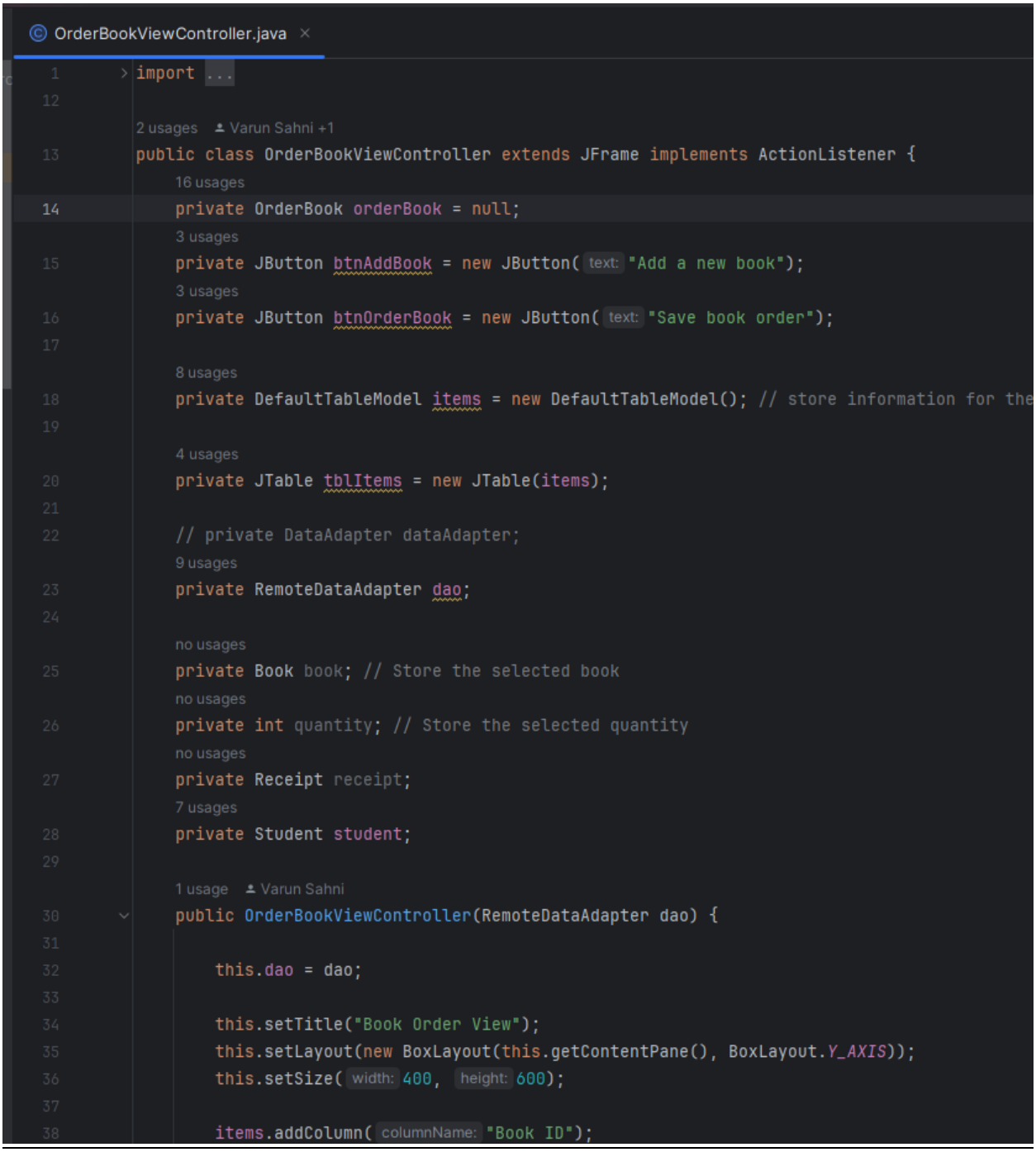


Book View controller

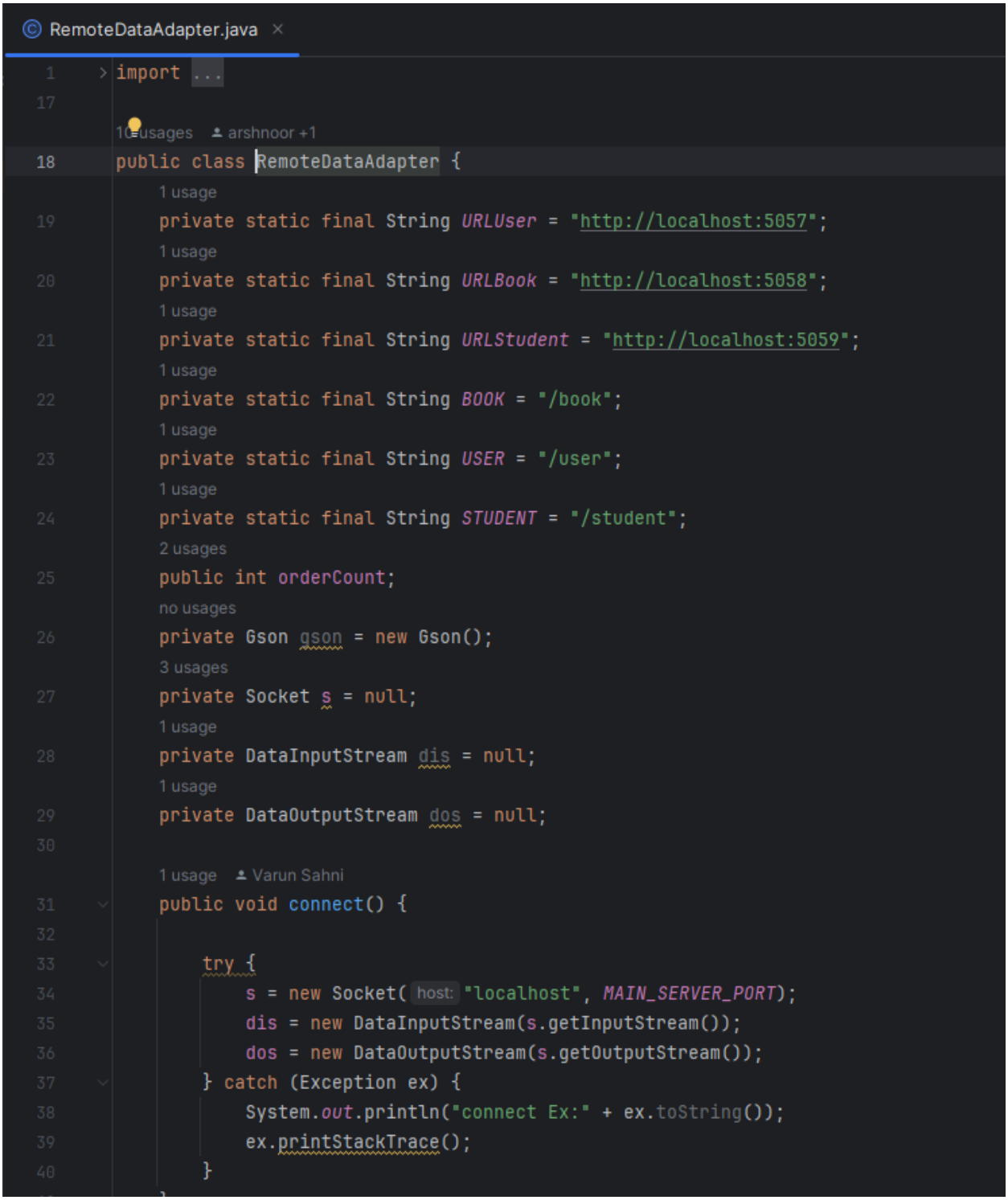


```java
import ...

2 usages  Varun Sahni +1
public class BookViewController extends JFrame implements ActionListener {
    4 usages
    private JTextField txtBookID  = new JTextField( columns: 10);
    3 usages
    private JTextField txtBookName  = new JTextField( columns: 30);
    4 usages
    private JTextField txtAuthorName  = new JTextField( columns: 30);
    4 usages
    private JTextField txtBookQuantity  = new JTextField( columns: 10);
    4 usages
    private JTextField txtStatus  = new JTextField( columns: 10);

    3 usages
    private JButton btnLoad = new JButton( text: "Load Book");
    3 usages
    private JButton btnSave = new JButton( text: "Save Book");

    3 usages
    RemoteDataAdapter dao;

    1 usage  Varun Sahni
    public BookViewController(RemoteDataAdapter dao) {

        this.dao = dao;

        this.setTitle("Manage Books");
        this.setLayout(new BoxLayout(this.getContentPane(), BoxLayout.PAGE_AXIS));
        this.setSize( width: 900,  height: 300);

        JPanel panelButton = new JPanel();
        panelButton.add(btnLoad);
        panelButton.add(btnSave);
        this.getContentPane().add(panelButton);

        JLabel statusDescription1 = new JLabel( text: "Set status as A/NA only");
        statusDescription1.setFont(new Font( name: "Sans Serif", Font.BOLD,  size: 12));
        JLabel statusDescription2 = new JLabel( text: "A - Available, NA - Not Available");
```

OrderBook view controller



Remote Data Adapter



```java
import ...

10 usages  arshnoor +1
public class RemoteDataAdapter {
    1 usage
    private static final String URLUser = "http://localhost:5057";
    1 usage
    private static final String URLBook = "http://localhost:5058";
    1 usage
    private static final String URLStudent = "http://localhost:5059";
    1 usage
    private static final String BOOK = "/book";
    1 usage
    private static final String USER = "/user";
    1 usage
    private static final String STUDENT = "/student";
    2 usages
    public int orderCount;
    no usages
    private Gson gson = new Gson();
    3 usages
    private Socket s = null;
    1 usage
    private DataInputStream dis = null;
    1 usage
    private DataOutputStream dos = null;

    1 usage  Varun Sahni
    public void connect() {

        try {
            s = new Socket( host: "localhost", MAIN_SERVER_PORT);
            dis = new DataInputStream(s.getInputStream());
            dos = new DataOutputStream(s.getOutputStream());
        } catch (Exception ex) {
            System.out.println("connect Ex:" + ex.toString());
            ex.printStackTrace();
        }
```

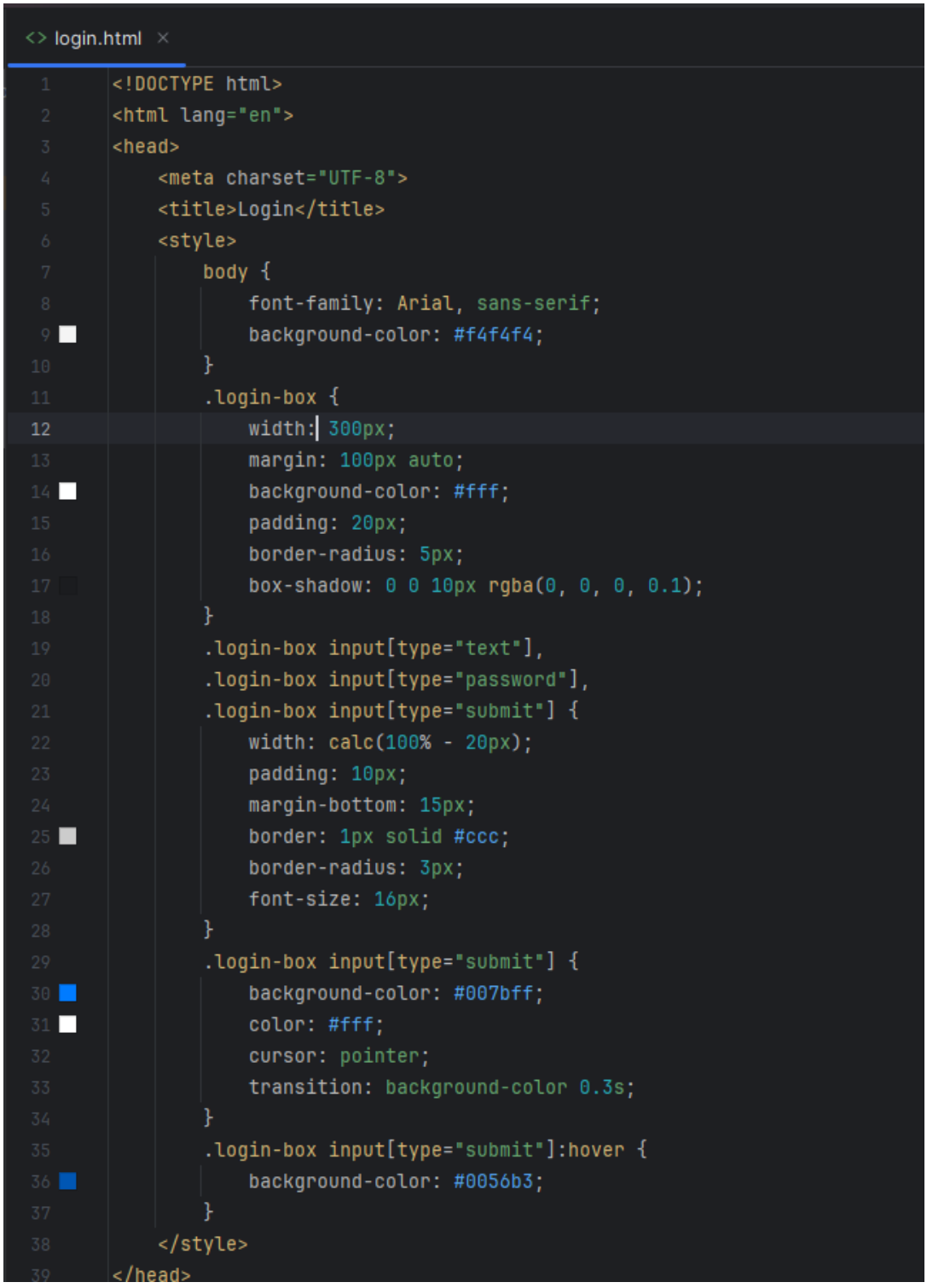
Index.html

index.html

```html
<!DOCTYPE html>
<html>
<head>
    <title>Update Product</title>
</head>
<body>
<h1>Update Product</h1>
<form id="productForm">
    Product ID:<br>
    <input type="text" id="productId" name="productId"><br>
    Name:<br>
    <input type="text" id="productName" name="productName"><br>
    Price:<br>
    <input type="text" id="productPrice" name="productPrice"><br>
    Quantity:<br>
    <input type="text" id="productQuantity" name="productQuantity"><br><br>
    Seller Id:<br>
    <input type="text" id="sellerId" name="sellerId"><br>

    <input type="button" value="Update" onclick="postData()">
</form>

<div id="responseMessage"></div>

<script>
    1 usage  arshnoor
    function postData() {
        const formData = {
            productID: document.getElementById('productId').value,
            name: document.getElementById('productName').value,
            price: document.getElementById('productPrice').value,
            quantity: document.getElementById('productQuantity').value,
            sellerID: document.getElementById('sellerId').value
        };

        fetch('http://localhost:8080/product/update', {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json'
```

Loadsave.html

Login.html


<> login.html ×
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>Login</title>
<style>
body {
font-family: Arial, sans-serif;
background-color: #f4f4f4;
}
.login-box {
width: 300px;
margin: 100px auto;
background-color: #fff;
padding: 20px;
border-radius: 5px;
box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}
.login-box input[type="text"],
.login-box input[type="password"],
.login-box input[type="submit"] {
width: calc(100% - 20px);
padding: 10px;
margin-bottom: 15px;
border: 1px solid #ccc;
border-radius: 3px;
font-size: 16px;
}
.login-box input[type="submit"] {
background-color: #007bff;
color: #fff;
cursor: pointer;
transition: background-color 0.3s;
}
.login-box input[type="submit"]:hover {
background-color: #0056b3;
}
</style>
</head>

Manage.html

manage.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Manage Books</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f4f4f4;
        }
        .main-container {
            text-align: center;
            margin: 50px auto;
            max-width: 600px;
        }
        .title {
            font-size: 25px;
            font-weight: bold;
            margin-bottom: 20px;
        }
        .button-container {
            display: flex;
            justify-content: space-around;
            margin-bottom: 20px;
        }
        .button {
            padding: 10px 20px;
            font-size: 16px;
            cursor: pointer;
            background-color: #007bff;
            color: #fff;
            border: none;
            border-radius: 5px;
        }
        .button:hover {
            background-color: #0056b3;
        }
    </style>
</head>
```

Orderbook.html

orderbook.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Order Book View</title>
    <style>
        /* Add your CSS styles here */
        table {
            border-collapse: collapse;
            width: 100%;
        }

        th, td {
            border: 1px solid #ddd;
            padding: 8px;
            text-align: left;
        }

        button {
            margin: 5px;
        }
    </style>
</head>
<body>
<h1>Book Order View</h1>

<!-- Input fields for Book ID and Quantity -->
<label for="bookID">Book ID:</label>
<input type="text" id="bookID">

<label for="quantity">Quantity:</label>
<input type="text" id="quantity">

<label for="studentID">Student ID:</label>
<input type="text" id="studentID">
```

Server Components:
Our project embraces a microservices architecture, featuring distinct servers for specialized operations:

1. **Book Microservice:**

    a. Employs the BookServer class, initializing an HTTP server on port 5058, with a dedicated '/book' endpoint handling GET and POST requests. The BookHandler class processes these requests, interacting with a database via DataAdapter2 for efficient book data management.
        i. HTTP Server Initialization: Establishes an HTTP server on port 5058, with a '/book' endpoint handling GET and POST requests for retrieving and saving book information, respectively.
        ii. BookHandler Class: Implements an HTTPHandler to process requests related to books, handling CORS headers, supporting GET (for retrieving book details) and POST (for saving new books), and interacting with a database through DataAdapter2.
        iii. Data Processing: Utilizes Jackson's ObjectMapper for JSON serialization/deserialization, processes incoming book data, and communicates with a database to load or save book information based on the HTTP method.

2. **User Microservices:**

    a. Utilizes the UserServer class, initializing an HTTP server on port 5057, equipped with a '/user' endpoint handling GET requests for retrieving user details. The UserHandler class manages these requests, communicating with the database through DataAdapter2 to load user information.
        i. HTTP Server Initialization: Initializes an HTTP server on port 5057, featuring a '/user' endpoint for handling GET requests to retrieve user details based on provided credentials.
        ii. UserHandler Class: Implements an HTTPHandler for handling user-related requests, supporting GET requests to load user details from a database using DataAdapter2, and handling CORS headers.
        iii. Data Processing: Utilizes Jackson's ObjectMapper for JSON processing, reads username and password from the request path, and communicates with a database to load user information.

3. **Order Microservices:**

    a. Incorporates the OrderBookServer class, establishing an HTTP server on port 5059, with '/order', '/student', and '/receipt' endpoints for handling POST requests related to order book, receipt, and student information. The OrderHandler,

StudentHandler, and ReceiptHandler classes process these requests, interacting with DataAdapter2 for efficient data storage and retrieval.

i. HTTP Server Initialization: Establishes an HTTP server on port 5059, creating '/order', '/student', and '/receipt' endpoints for handling POST requests to save order book, receipt, and student information.
ii. OrderHandler Class: Manages POST requests to '/order', processing incoming JSON data using Jackson's ObjectMapper, saving order book details to a database through DataAdapter2, and responding with JSON-formatted data.
iii. StudentHandler and ReceiptHandler Classes: Handle GET and POST requests for student and receipt information, respectively, interacting with a database through DataAdapter2, and employing Jackson's ObjectMapper for JSON processing.

Common Elements of Server Classes:

Database Interaction: Utilizes DataAdapter2 to interact with Redis database and MongoDB database for reading and writing information related to books, users, orders, students, and receipts.

CORS Headers: Ensures Cross-Origin Resource Sharing (CORS) compatibility by setting appropriate headers to handle requests from different origins.

Server Output: Provides console output indicating the successful initiation of each server on specific ports for monitoring purposes.

4. **DataAdapter2 Class:**

a. **Multi-Database Connectivity:** Connects seamlessly to both Redis and MongoDB, enabling comprehensive data retrieval and storage capabilities.
b. **Operations Handling:** Manages loading and saving operations efficiently for books, orders, receipts, users, and students. Includes potential sections for Redis database and MongoDB database interactions, enhancing adaptability.

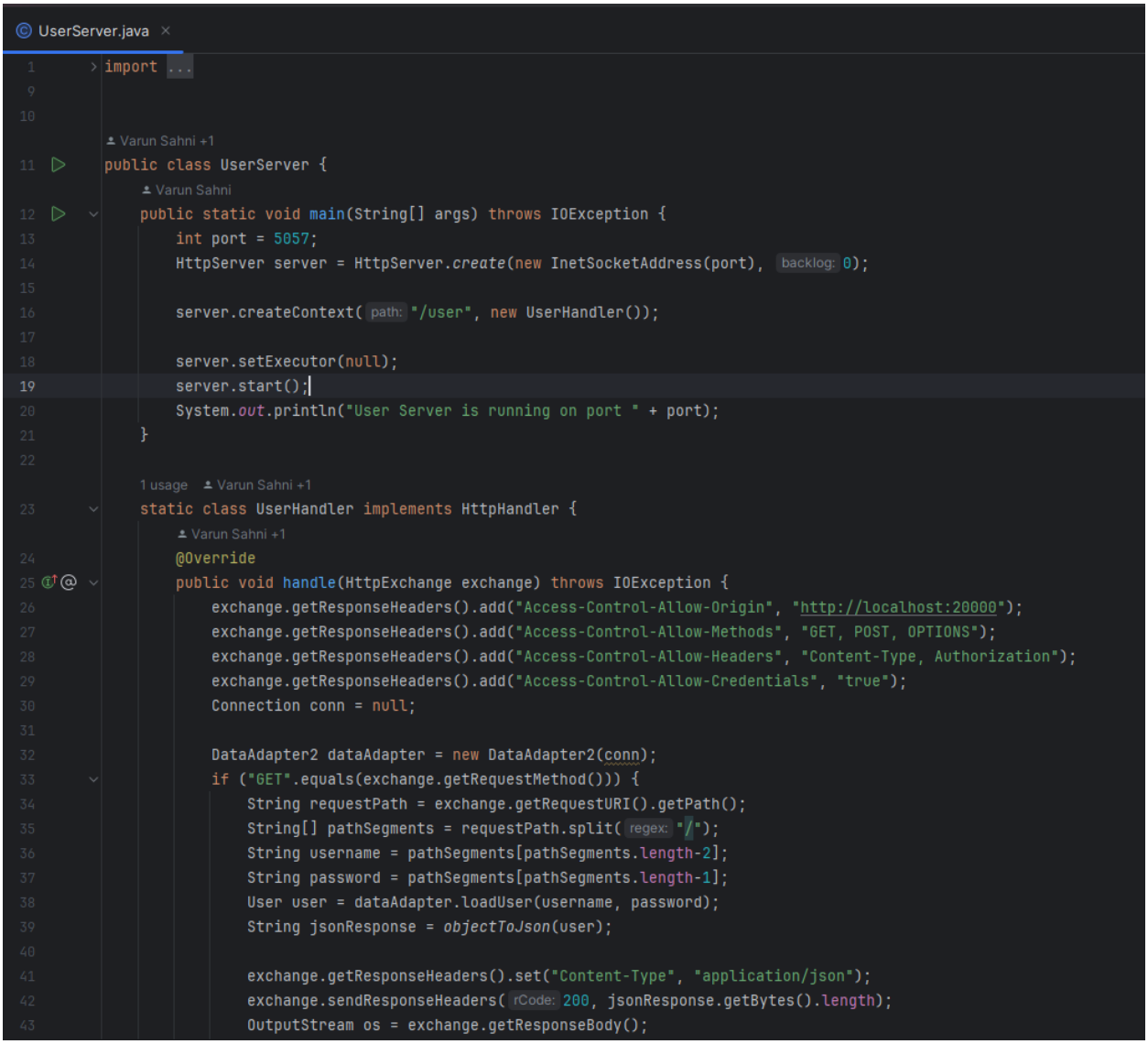
Below are the screenshots of the Server Components:

BookServer Class

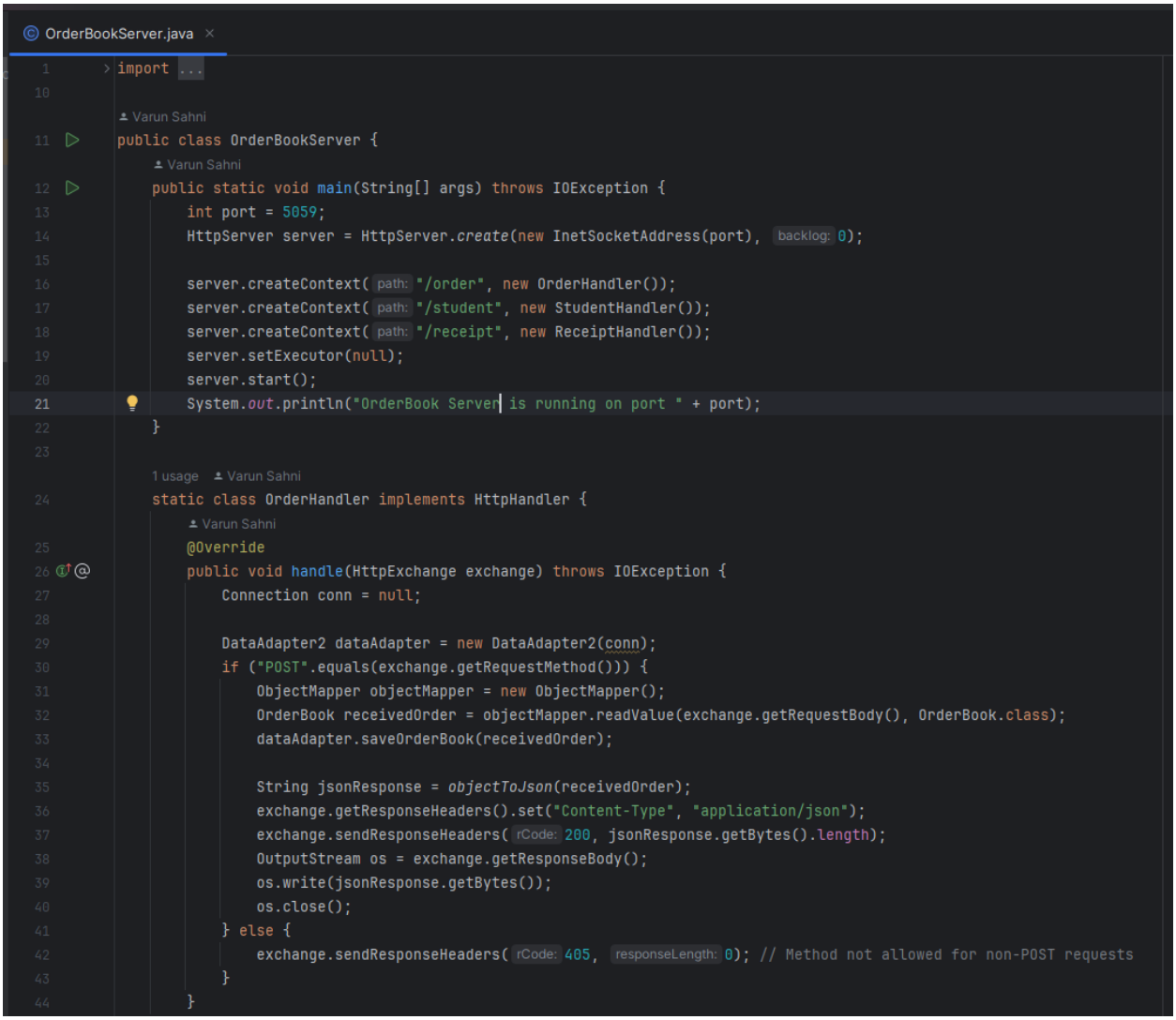
BookServer.java

```java
import com.sun.net.httpserver.HttpExchange;
import com.sun.net.httpserver.HttpHandler;
import com.sun.net.httpserver.HttpServer;
import org.codehaus.jackson.map.ObjectMapper;

import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.sql.*;

Varun Sahni
public class BookServer {
    Varun Sahni
    public static void main(String[] args) throws IOException {
        int port = 5058;
        HttpServer server = HttpServer.create(new InetSocketAddress(port),  backlog: 0);

        server.createContext( path: "/book", new BookHandler());

        server.setExecutor(null);
        server.start();
        System.out.println("Book Server is running on port " + port);
    }

    1 usage  Varun Sahni
    static class BookHandler implements HttpHandler {
        Varun Sahni
        @Override
        public void handle(HttpExchange exchange) throws IOException {

            // Set CORS headers for all requests
            exchange.getResponseHeaders().set("Access-Control-Allow-Origin", "*");
            exchange.getResponseHeaders().set("Access-Control-Allow-Methods", "GET, POST, OPTIONS");
            exchange.getResponseHeaders().set("Access-Control-Allow-Headers", "Content-Type");

            if ("OPTIONS".equals(exchange.getRequestMethod())) {
                exchange.sendResponseHeaders( rCode: 200,  responseLength: -1); // preflight request successful
                return;
```

UserServer Class

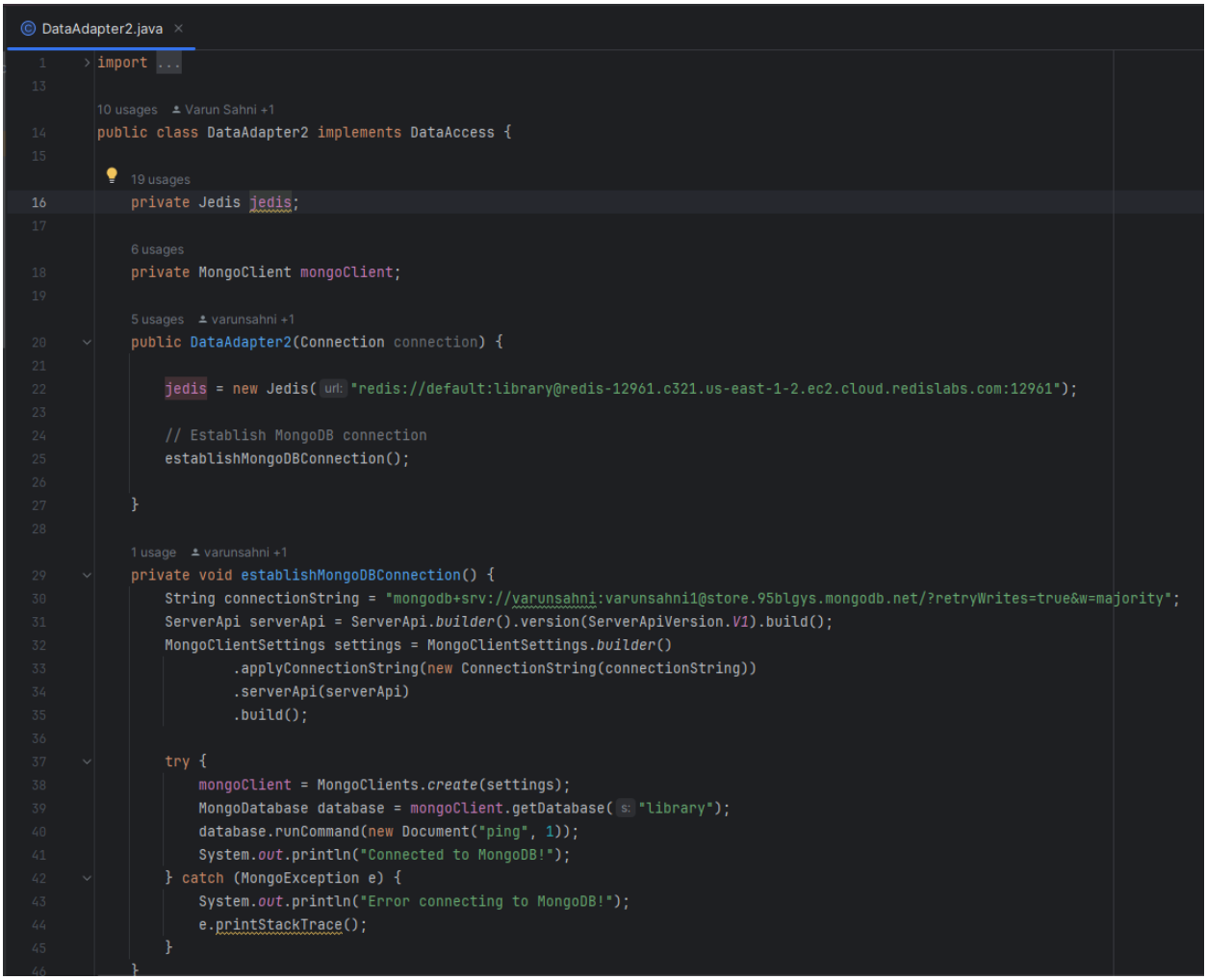


OrderBook Server



```java
import ...

Varun Sahni
public class OrderBookServer {
    Varun Sahni
    public static void main(String[] args) throws IOException {
        int port = 5059;
        HttpServer server = HttpServer.create(new InetSocketAddress(port), backlog: 0);

        server.createContext( path: "/order", new OrderHandler());
        server.createContext( path: "/student", new StudentHandler());
        server.createContext( path: "/receipt", new ReceiptHandler());
        server.setExecutor(null);
        server.start();
        System.out.println("OrderBook Server is running on port " + port);
    }

    1 usage  Varun Sahni
    static class OrderHandler implements HttpHandler {
        Varun Sahni
        @Override
        public void handle(HttpExchange exchange) throws IOException {
            Connection conn = null;

            DataAdapter2 dataAdapter = new DataAdapter2(conn);
            if ("POST".equals(exchange.getRequestMethod())) {
                ObjectMapper objectMapper = new ObjectMapper();
                OrderBook receivedOrder = objectMapper.readValue(exchange.getRequestBody(), OrderBook.class);
                dataAdapter.saveOrderBook(receivedOrder);

                String jsonResponse = objectToJson(receivedOrder);
                exchange.getResponseHeaders().set("Content-Type", "application/json");
                exchange.sendResponseHeaders( rCode: 200, jsonResponse.getBytes().length);
                OutputStream os = exchange.getResponseBody();
                os.write(jsonResponse.getBytes());
                os.close();
            } else {
                exchange.sendResponseHeaders( rCode: 405,  responseLength: 0); // Method not allowed for non-POST requests
            }
        }
```

DataAdapter2 Class



```java
import ...

10 usages  Varun Sahni +1
public class DataAdapter2 implements DataAccess {

    19 usages
    private Jedis jedis;

    6 usages
    private MongoClient mongoClient;

    5 usages  varunsahni +1
    public DataAdapter2(Connection connection) {

        jedis = new Jedis( url: "redis://default:library@redis-12961.c321.us-east-1-2.ec2.cloud.redislabs.com:12961");

        // Establish MongoDB connection
        establishMongoDBConnection();

    }

    1 usage  varunsahni +1
    private void establishMongoDBConnection() {
        String connectionString = "mongodb+srv://varunsahni:varunsahni1@store.95blgys.mongodb.net/?retryWrites=true&w=majority";
        ServerApi serverApi = ServerApi.builder().version(ServerApiVersion.V1).build();
        MongoClientSettings settings = MongoClientSettings.builder()
                .applyConnectionString(new ConnectionString(connectionString))
                .serverApi(serverApi)
                .build();

        try {
            mongoClient = MongoClients.create(settings);
            MongoDatabase database = mongoClient.getDatabase( s: "library");
            database.runCommand(new Document("ping", 1));
            System.out.println("Connected to MongoDB!");
        } catch (MongoException e) {
            System.out.println("Error connecting to MongoDB!");
            e.printStackTrace();
        }
    }
```

The client-server library management system uses the following communication protocol:

Transport Protocol: HTTP (Hyper Text Transfer Protocol) is used for reliable data transfer between the client and server.

Communication Protocol:
The communication between the client and server classes is based on the HTTP protocol. Both the client and server classes utilize the RESTful architectural style, where HTTP methods (GET, POST, OPTIONS) are employed for communication.

HTTP Methods:
GET: Used for retrieving data (e.g., fetching books, users, or students).
POST: Employed for creating or updating data (e.g., saving a book, order, or receipt).
OPTIONS: Utilized for preflight requests to check server capabilities.

Data Format:
Request Format: Clients send JSON-formatted data in the request body (e.g., creating or updating entities).
Response Format: Servers respond with JSON-formatted data containing the requested information or status of the operation.

System Architecture Overview:
The system architecture comprises multiple components, including client classes (BookClient, UserClient, OrderBookClient) and server classes (BookServer, UserServer, OrderBookServer). The overall architecture follows a client-server model with a RESTful design.

Client Classes:
BookClient: Interacts with the server to perform operations related to books.
UserClient: Manages interactions with the server for user-related tasks.
OrderBookClient: Handles interactions with the server regarding orders.

Server Classes:
BookServer: Listens for and handles HTTP requests related to books.
UserServer: Manages HTTP requests for user-related operations.
OrderBookServer: Handles HTTP requests for order-related functionalities.

Communication Flow:
Clients send HTTP requests with JSON-formatted data to the respective server endpoints (/book, /user, /order).
Servers process the requests, perform necessary actions (e.g., querying a database, saving data), and respond with JSON-formatted results.
The system utilizes the HTTP protocol and JSON format for data exchange, ensuring a standardized and interoperable communication process.

Data Storage:
DataAdapter2: Manages data storage and retrieval using Redis for quick key-value operations and MongoDB for more complex data structures.
This architecture promotes scalability, maintainability, and separation of concerns, allowing each component to handle specific functionalities. The adoption of RESTful principles simplifies integration and supports various client implementations.

Sample Server Response: An example of a server response in JSON format is as follows:

JSON object of ResponseModel:
{"code":0,"body":"{\"userID\":1,\"username\":\"admin\",\"password\":\"password\",\"fullNam
e\":\"Adam Smith\",\"isManager\":false}"}

Message from client:
{"code":33,"body":"{\"bookID\":1234,\"bookName\":\"yahoo\",\"authorName\":\"yy
zee\",\"quantity\":20.0,\"status\":\"A\"}"}

JSON object of ResponseModel: {"code":0,"body":"New book saved successfully."}
Message from client: {"code":3,"body":"1234"}

JSON object of ResponseModel:
{"code":0,"body":"{\"bookID\":1234,\"bookName\":\"yahoo\",\"authorName\":\"yy
zee\",\"quantity\":20.0,\"status\":\"A\"}"}

Message from client: {"code":3,"body":"1"}

JSON object of ResponseModel: {"code":0,"body":"{\"bookID\":1,\"bookName\":\"Engineering
Mechanics\",\"authorName\":\"Varun S.\",\"quantity\":927.0,\"status\":\"A\"}"}

Message from client: {"code":555,"body":"1"}

JSON object of ResponseModel: {"code":0,"body":"{\"studentID\":1,\"studentName\":\"Alice
Johnson\",\"emailID\":\"alice@email.com\",\"studentNumber\":\"1234567890\"}"}

Message from client:
{"code":22,"body":"{\"orderID\":0,\"studentID\":1,\"student\":{\"studentID\":1,\"studentName
\":\"Alice
Johnson\",\"emailID\":\"alice@email.com\",\"studentNumber\":\"1234567890\"},\"lines\":[{\"
orderID\":0,\"bookID\":1,\"quantity\":1.0,\"bookName\":\"Engineering Mechanics\"}]}"}
JSON object of ResponseModel: {"code":0,"body":"Order and order lines saved successfully."}
Message from client: {"code":3,"body":"1"}

JSON object of ResponseModel: {"code":0,"body":"{\"bookID\":1,\"bookName\":\"Engineering
Mechanics\",\"authorName\":\"Varun S.\",\"quantity\":927.0,\"status\":\"A\"}"}

Message from client: {"code":33,"body":"{\"bookID\":1,\"bookName\":\"Engineering
Mechanics\",\"authorName\":\"Varun S.\",\"quantity\":926.0,\"status\":\"A\"}"}

JSON object of ResponseModel: {"code":0,"body":"Book updated successfully."}
Message from client: {"code":3,"body":"1"}

JSON object of ResponseModel: {"code":0,"body":"{\"bookID\":1,\"bookName\":\"Engineering Mechanics\",\"authorName\":\"Varun S.\",\"quantity\":926.0,\"status\":\"A\"}"}

Message from client: {"code":7,"body":"{\"receiptNumber\":0,\"orderId\":0,\"dateTime\":\"29-10-2023 23:08:47\",\"student\":\"Student ID: 1 Name: Alice Johnson Email ID: alice@email.com Number: 1234567890\",\"books\":\"Engineering Mechanics x 1.0\"}"}

JSON object of ResponseModel: {"code":0,"body":"Receipt information saved successfully."}
Message from client: {"code":8}

JSON object of ResponseModel: {"code":0,"body":"100"}

This response includes a response code (0 for success) and a JSON body containing specific data related to the query.

This architectural design allows for a clear separation of concerns, making your library management system modular and scalable. It uses a robust communication protocol and data format to ensure reliable data exchange between the client and server components.

Ques 6) Manual for installation and usage with information on necessary libraries, frameworks, and running guidelines.

Answer 6)

INSTALLATION MANUAL:

System Requirements:

1. IntelliJ Idea – Java and Kotlin IDE
2. Java Development Kit (JDK) 8 or later
3. Node.js
4. Redis Server
5. MongoDB Server

JARs Required: Install all the JARs present in the Jars folder in a zip file. These are required to run the program.

JARs' Function:

com.fasterxml.jackson.databind: Jackson Databind, a general-purpose data-binding package for JSON and other data formats.

gson-2.9.0: Gson, a library for JSON serialization and deserialization in Java.

jackson-all-1.9.0: Jackson All-in-One, a bundle including various Jackson libraries for JSON processing.

jedis-5.0.2: Jedis, a Java client library for interacting with Redis, a key-value store.

mongodb-jdbc-2.0.3-all: MongoDB JDBC Driver, a JDBC-compliant driver for connecting to MongoDB.

These libraries provide essential functionality for data storage, serialization, and communication in the context of MongoDB, Redis, and general-purpose Java development. They are included as dependencies to enable seamless integration and smooth functioning of the code.

Installation Steps:

- Installation of any JAVA IDE is also very important. We have used IntelliJ IDE. Any other IDE can also be used for e.g. Eclipse.

- Java Development Kit (JDK): Ensure that you have Java installed. You can download it from Oracle JDK or use OpenJDK.

- Redis Server: Start the Redis server by running the classes UserServer and BookServer. The access of this is public so it is available.

- MongoDB Server: Start MongoDB server by running the classes OrderBookServer. The access of this is public so it is available.

- Node.js server: Start server.js by running server.js file in Terminal.

- Change the active directory to the path wherein node server file is present.
    - Write below command in Terminal:
    - cd <path_to_node_server_directory>
    - node server.js

Alternatively, you may write the below lines in Terminal to start server.js:

cd src
cd LMS_Website
node server.js

Terminal Screenshot:

Running the Clients:

1) Run User server at port 5057.
2) Run Book server at port 5058.
3) Run OrderBook server at port 5059.
    a. Desktop Client (Client 1): Run the “Application” class.
    b. Web Browser Client (Client 2) : Type this URL in web browser: http://localhost:20000/. This will lead to library management system website.