zhejiang-yige

Java Programming

Course Project



Year & Semester 2024-2025-2

Major & Class 23计科全英

Name 代翔

Student No. 2023337621159

Phone No. 15700145894

Teacher 吴婷婷

6/2025

**MENU**

[Course Project 3](#_Toc24148)

[1.1. Project Introduction 3](#_Toc21746)

[1.1.1. Background 3](#_Toc4715)

[1.1.2. Objectives 3](#_Toc22582)

[1.2. Requirement Analysis 3](#_Toc9631)

[1.2.1 Functional Requirements 3](#_Toc21690)

[1.2.2. Non-functional Requirements 4](#_Toc26203)

[1.3. Database Design 4](#_Toc26064)

[1.3.1. Table Structure 4](#_Toc22232)

[1.3.2. ER Diagram 6](#_Toc20539)

[1.4. System Design 6](#_Toc28304)

[1.4.1. Overall Architecture 6](#_Toc25670)

[1.4.2. Module Design 7](#_Toc117)

[1.5. GUI Design (JavaFX) 84](#_Toc9638)

[1.5.1. Login and Registration FX 84](#_Toc13257)

[1.5.2. User Interface 85](#_Toc3950)

[1.5.3. Admin Interface 85](#_Toc11462)

[1.5.4. Recommend Interface. 86](#_Toc15513)

[1.5.5. Bookcard Panel and Css File 86](#_Toc11129)

[1.6. Testing & Evaluation 87](#_Toc16892)

[1.6.1. Server Test 87](#_Toc23163)

[1.6.2. Login Function 88](#_Toc11414)

[1.6.3. Register Function 91](#_Toc5229)

[1.6.4. Overdue Function 94](#_Toc28572)

[1.6.5. Return book function 95](#_Toc24167)

[1.6.6. Multiple Thread Test & Borrow Function 97](#_Toc10165)

[1.6.7. Recommend Function 100](#_Toc1712)

[1.6.8. Admin View Function 102](#_Toc1342)

[1.6.9. Add,Delete,Change Function 104](#_Toc4739)

[1.6.10. Frozen Function 105](#_Toc32149)

[1.7. Summary 107](#_Toc15020)

# Course Project

## Project Introduction

### Background

This project aims to develop a graphical library management system supporting multiple users. It incorporates essential features such as book borrowing and returning, admin-level management, and personalized book recommendations.

### Objectives

The goal is to apply core Java technologies—object-oriented programming, inheritance, multithreading, networking, JDBC, and recommendation algorithms—to build a complete and user-friendly library system.

## Requirement Analysis

### Functional Requirements

#### User:

Register and log in (identify role as normal user or admin),Search books by title, author, or category,View recommended books (content-based or collaborative filtering),Borrow/return books, view borrowing history,Receive overdue notifications

#### Admin:

Add/update/delete book records,View all borrow history,View top borrowed/popular books,Manage user accounts (freeze/unfreeze)

### Non-functional Requirements

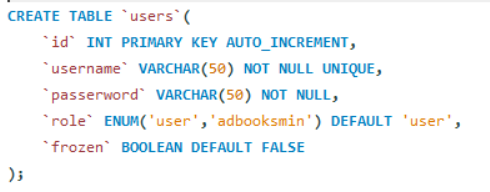
1. Stable multi-user support
2. Timely response to operations
3. Data consistency and integrity
4. Basic security (role verification)

## Database Design

### Table Structure

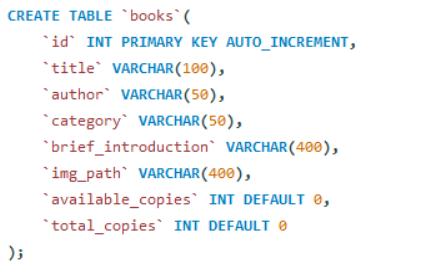
#### users (id, username, password, role, frozen)

When creating the users table, we need id, username, passerword, role, and frozen to describe, id is our primary key, and it can be self-incremented, username is not allowed to be repeated, password is not allowed to be empty, and the role can only be user or admin with an enumeration, and frozen is used to determine whether the current user is frozen.



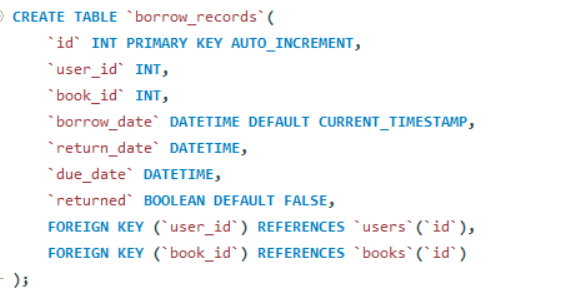
#### books(id,title,author,category,brief\_introduction,img\_path,available\_copies,total\_copie)

When creating the books table, I still choose id as the primary key and let him automatically achieve self-increment, title represents the title of the book, author represents the author, category represents the category, and the introduction, which is mainly used for the follow-up recommendation code as the basic variable, as well as the picture address beautification window (although the end is a picture), the number of remaining books and the total number of books.

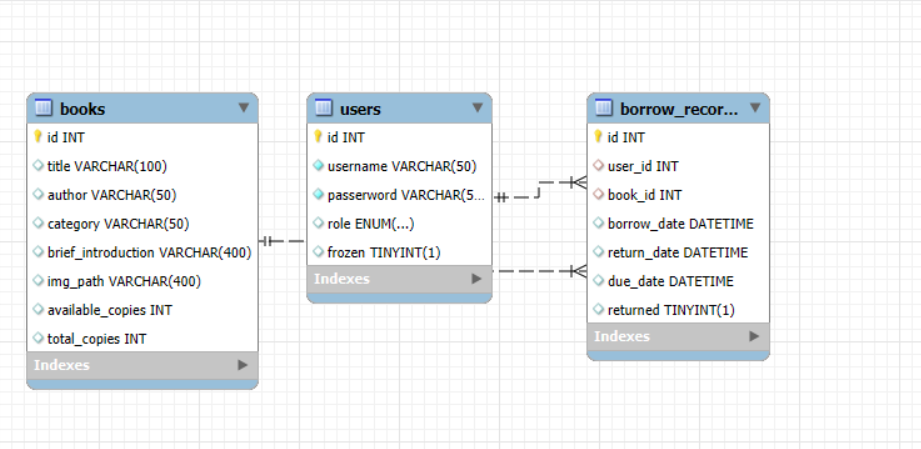


#### borrow\_records (id, user\_id, book\_id, borrow\_date, return\_date,due\_date, returned)

When creating this borrow\_records table, the id is still selected as the primary key and let it automatically achieve auto-increment, which uses user\_id and book\_id as foreign keys to connect with the other two tables, and the borrowing time in it defaults to the current time, and return defaults to false.

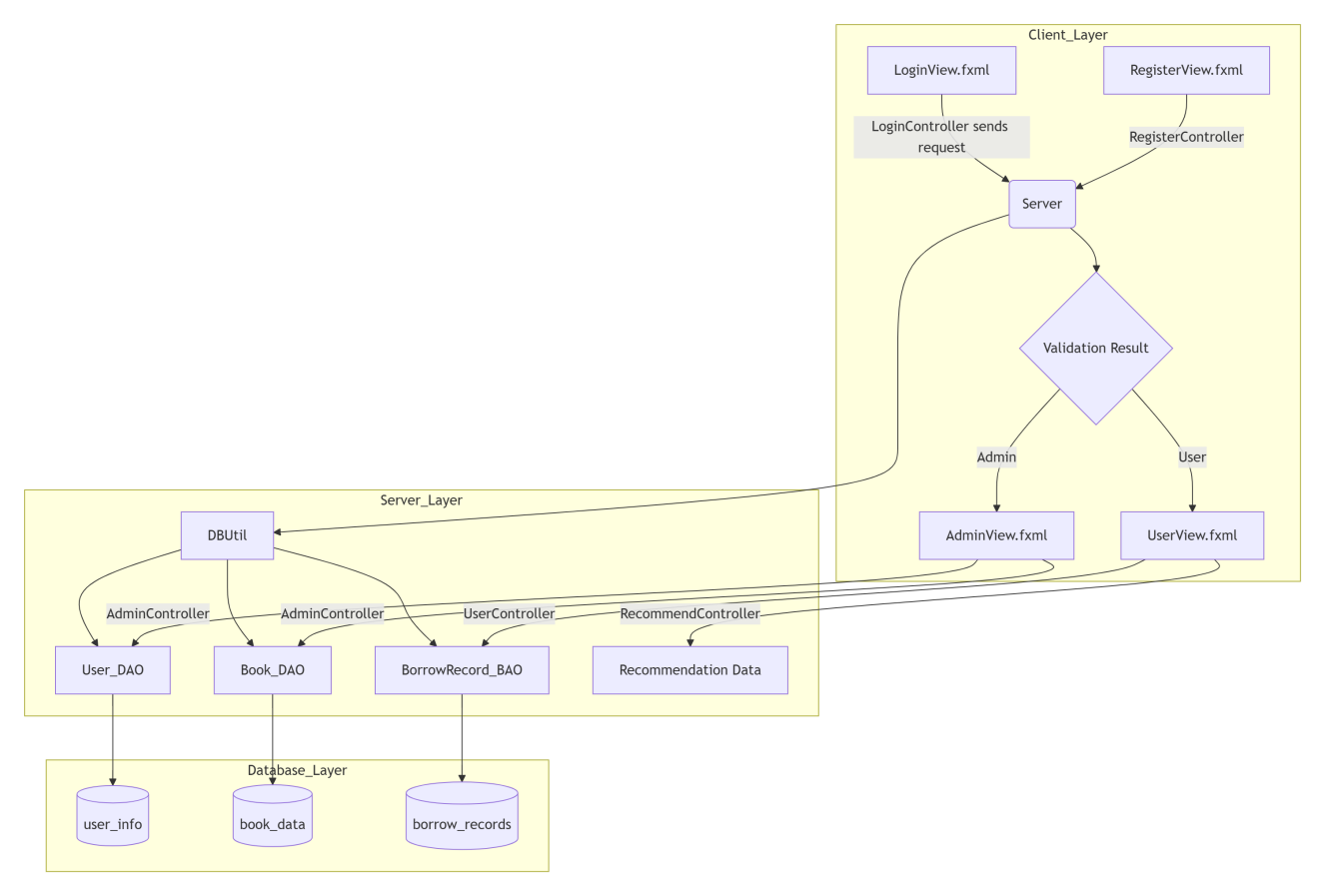


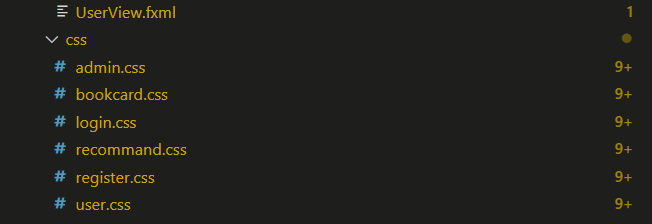
### ER Diagram



## System Design

### Overall Architecture





### Module Design

#### Data Access Object Module

The DbUtil class in this Java code is a utility class designed to facilitate the connection between a Java application and a MySQL database named library\_system. It defines static constants for the database URL, username, and password, and provides a static method getConnection() that attempts to establish and return a database connection using these credentials. If the connection is successful, it prints a confirmation message and returns the Connection object; if it fails, it catches the SQLException, prints an error message, and returns null. This class serves as a centralized and reusable component for managing database connectivity, allowing other parts of the application, such as data access objects (DAOs), to easily obtain a connection and interact with the database. The code also includes a commented-out main method that can be used to test whether the database connection works correctly. Overall, this class helps streamline and encapsulate the logic required to connect to the database, promoting cleaner and more maintainable code in the rest of the system.

package database;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class DbUtil {

    public static final String URL = "jdbc:mysql://127.0.0.1:3306/library\_system";

    public static final String USER = "root";

    public static final String PASSWORD = "050728";

    public static Connection getConnection(){

        try {

            Connection driver = DriverManager.getConnection(URL, USER, PASSWORD);

            System.out.println("Database connection established successfully.");

            return driver;

        } catch (SQLException e) {

            System.err.println("Database connection failed: " + e.getMessage());

            return null;

        }

    }

    // test the connection

    // public static void main(String[] args) {

    //     try (Connection connection = DriverManager.getConnection(URL, USER, PASSWORD)) {

    //         System.out.println("Database connection established successfully.");

    //     } catch (SQLException e) {

    //         System.err.println("Database connection failed: " + e.getMessage());

    //     }

    // }

}

This User\_DAO class is a Data Access Object (DAO) responsible for handling user-related operations with the MySQL database in a Java-based library management system. It interacts with the database to support functions such as user login, registration, username validation, user list retrieval, fuzzy search, and account status toggling. The Login method authenticates a user by checking the username and password against records in the users table, and returns a User object if the credentials match. The Register method allows new users or admins to register by inserting a new record into the users table; admin registration requires a correct activation code. The method isUsernameAvailable checks if a given username is already taken to ensure uniqueness during registration. getAllUsers retrieves and returns a list of all users in the database, while searchUser performs a fuzzy search on usernames using a keyword. The toggleFreezeStatus method flips the frozen status of a user account, which is useful for temporarily disabling or re-enabling user access. All methods use the DbUtil.getConnection() method to establish a connection with the database, and they handle exceptions gracefully, printing relevant error messages if something goes wrong during database interaction. This class encapsulates all user-related database logic, promoting code reuse and separation of concerns within the overall system.

package database;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

import models.User;

public class User\_DAO {

    public User Login(String *login\_name*, String *login\_password*){

        try (Connection driver = DbUtil.getConnection()){

            if (driver == null){

                System.err.println("Database connection is null.");

                return null;

            }else{

                String sql = "SELECT \* FROM users WHERE username = ? AND passerword = ?";

                PreparedStatement ps = driver.prepareStatement(sql);

                ps.setString(1, *login\_name*);

                ps.setString(2, *login\_password*);

                ResultSet rs = ps.executeQuery();

                if (rs.next()) {

                    return new User(

                        rs.getInt("id"),

                        rs.getString("username"),

                        rs.getString("passerword"),

                        rs.getString("role"),

                        rs.getBoolean("frozen")

                    );

                }

            }

        } catch (SQLException e) {

            System.err.println("Error during login: " + e.getMessage());

        }

        return null;

    }

    public *boolean* Register(String *register\_name*, String *register\_password*, String *role*, String *activationCode*) {

        try (Connection driver = DbUtil.getConnection()){

            if (driver == null){

                System.err.println("Database connection is null.");

                return false;

            }else{

                String sql = "INSERT INTO `users`(`username`, `passerword`, `role`) VALUES (?, ?, ?)";

                if(*role*.equals("user")){

                    PreparedStatement ps = driver.prepareStatement(sql);

                    ps.setString(1, *register\_name*);

                    ps.setString(2, *register\_password*);

                    ps.setString(3, *role*);

*int* rowsAffected = ps.executeUpdate();

                    if (rowsAffected > 0) {

                        System.out.println("Registration successful.");

                        return true;

                    } else {

                        System.out.println("Registration failed.");

                    }

                }else if(*role*.equals("admin") && *activationCode*.equals("050728")) {

                    PreparedStatement ps = driver.prepareStatement(sql);

                    ps.setString(1, *register\_name*);

                    ps.setString(2, *register\_password*);

                    ps.setString(3, *role*);

*int* rowsAffected = ps.executeUpdate();

                    if (rowsAffected > 0) {

                        System.out.println("Admin registration successful.");

                        return true;

                    } else {

                        System.out.println("Admin registration failed.");

                    }

                } else {

                    System.out.println("Invalid role or activation code.");

                }

            }

        }catch (SQLException e) {

            System.err.println("Error during registration: " + e.getMessage());

        }

        return false;

    }

    public *boolean* isUsernameAvailable(String *username*) {

        try (Connection driver = DbUtil.getConnection()) {

            if (driver == null) {

                System.err.println("Database connection is null.");

                return false;

            } else {

                String sql = "SELECT COUNT(\*) FROM users WHERE username = ?";

                PreparedStatement ps = driver.prepareStatement(sql);

                ps.setString(1, *username*);

                ResultSet rs = ps.executeQuery();

                if (rs.next()) {

                    return rs.getInt(1) == 0;

                }

            }

        } catch (SQLException e) {

            System.err.println("Error checking username availability: " + e.getMessage());

        }

        return false;

    }

    public List<User> getAllUsers() {

        List<User> userList = new ArrayList<>();

        try (Connection conn = DbUtil.getConnection()) {

            if (conn == null) {

                System.err.println("Database connection is null.");

                return userList;

            }

            String sql1 = "SELECT \* FROM `users`";

            try (PreparedStatement stmt = conn.prepareStatement(sql1);

                ResultSet rs = stmt.executeQuery()) {

                while (rs.next()) {

                    User user = new User(

                            rs.getInt("id"),

                            rs.getString("username"),

                            rs.getString("passerword"),

                            rs.getString("role"),

                            rs.getBoolean("frozen")

                    );

                    userList.add(user);

                }

            }

        } catch (SQLException e) {

            System.err.println("Error fetching user list: " + e.getMessage());

        }

        return userList;

    }

    public List<User> searchUser(String *keyword*){

        if(*keyword* == null){

           List<User> list = getAllUsers();

           return list;

        }else{

            List<User> list = new ArrayList<>();

            String sql = "SELECT \* FROM users WHERE `username` LIKE ?";

            try(Connection conn = DbUtil.getConnection();

                PreparedStatement ps = conn.prepareStatement(sql)){

                    String key =  "%" + *keyword* + "%";

                    ps.setString(1,key);

                    ResultSet rs = ps.executeQuery();

                    while (rs.next()){

                        User user = new User(

                            rs.getInt("id"),

                            rs.getString("username"),

                            rs.getString("passerword"),

                            rs.getString("role"),

                            rs.getBoolean("frozen")

                        );

                        list.add(user);

                    }

            } catch (SQLException e) {

                System.err.println("Error searching users: " + e.getMessage());

            }

            return list;

        }

    }

    public *boolean* toggleFreezeStatus(*int* *userId*) {

        String query = "SELECT `frozen` FROM users WHERE id = ?";

        String update = "UPDATE `users` SET `frozen` = ? WHERE id = ?";

        try (Connection conn = DbUtil.getConnection();

            PreparedStatement selectStmt = conn.prepareStatement(query)) {

            selectStmt.setInt(1, *userId*);

            ResultSet rs = selectStmt.executeQuery();

            if (rs.next()) {

*boolean* currentFrozen = rs.getBoolean("frozen");

*boolean* newStatus = !currentFrozen;

                try (PreparedStatement updateStmt = conn.prepareStatement(update)) {

                    updateStmt.setBoolean(1, newStatus);

                    updateStmt.setInt(2, *userId*);

*int* rows = updateStmt.executeUpdate();

                    return rows > 0;

                }

            }

        } catch (SQLException e) {

            System.err.println("Error toggling freeze status: " + e.getMessage());

        }

        return false;

    }

}

The Book\_DAO class functions as the comprehensive data access layer for all book-related operations in a library management system, providing complete CRUD functionality for book inventory management through methods like addBook(), updateBook(), deleteBook(), and getBookById(), while also handling the entire borrowing lifecycle—including transactional borrow operations with concurrency control via SELECT FOR UPDATE in borrowBook() to prevent duplicate loans and stock synchronization, and coordinated return processing in returnBook() that updates both records and inventory levels. It supports robust query capabilities including full inventory retrieval (getAllBooks()), keyword-based searches across titles/authors/categories (searchBooks()), and user-specific functions like fetching currently borrowed books (getBorrowedBooks()) and generating personalized recommendations by analyzing borrowing history to identify preferred categories before suggesting available unborrowed titles, with fallback to popular books for new users (getRecommendations(), searchRecommendedBooks), all while maintaining data integrity through atomic transactions, automatic rollbacks, and sanitized inputs within a connection-pooled environment.

package database;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.List;

import models.book;

public class Book\_DAO {

   public List<book> searchBorrowedBooks(*int* *userId*, String *keyword*) {

        List<book> books = new ArrayList<>();

        try (Connection conn = DbUtil.getConnection()) {

            String sql = "SELECT b.\* FROM books b " +

                        "JOIN borrow\_records br ON b.id = br.book\_id " +

                        "WHERE br.user\_id = ? AND (b.title LIKE ? OR b.author LIKE ? OR b.category LIKE ?) " +

                        "GROUP BY b.id";

            try (PreparedStatement ps = conn.prepareStatement(sql)) {

                String key = "%" + *keyword* + "%";

                ps.setInt(1, *userId*);

                ps.setString(2, key);

                ps.setString(3, key);

                ps.setString(4, key);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    books.add(new book(

                        rs.getInt("id"),

                        rs.getString("title"),

                        rs.getString("author"),

                        rs.getString("category"),

                        rs.getString("brief\_introduction"),

                        rs.getString("img\_path"),

                        rs.getInt("available\_copies"),

                        rs.getInt("total\_copies")

                    ));

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

        return books;

    }

   public List<book> searchRecommendedBooks(*int* *userId*, String *keyword*) {

        List<book> recommendations = new ArrayList<>();

        if (*keyword* != null && !*keyword*.trim().isEmpty()) {

            try (Connection driver = DbUtil.getConnection()) {

                if (driver == null) {

                    System.err.println("Database connection is null.");

                    return null;

                } else {

                    String sql = "SELECT \* FROM books WHERE title LIKE ? OR author LIKE ? OR category LIKE ?";

                    try (PreparedStatement ps = driver.prepareStatement(sql)) {

                        String key = "%" + *keyword*.trim() + "%";

                        ps.setString(1, key);

                        ps.setString(2, key);

                        ps.setString(3, key);

                        ResultSet rs = ps.executeQuery();

                        while (rs.next()) {

                            recommendations.add(new book(

                                rs.getInt("id"),

                                rs.getString("title"),

                                rs.getString("author"),

                                rs.getString("category"),

                                rs.getString("brief\_introduction"),

                                rs.getString("img\_path"),

                                rs.getInt("available\_copies"),

                                rs.getInt("total\_copies")

                            ));

                        }

                    }

                }

            } catch (Exception e) {

                e.printStackTrace();

            }

            return recommendations;

        }

        try (Connection conn = DbUtil.getConnection()) {

            String topCategoriesSql =

                "SELECT b.category, COUNT(\*) AS borrow\_count " +

                "FROM borrow\_records br " +

                "JOIN books b ON br.book\_id = b.id " +

                "WHERE br.user\_id = ? " +

                "GROUP BY b.category " +

                "ORDER BY borrow\_count DESC " +

                "LIMIT 3";

            List<String> topCategories = new ArrayList<>();

            try (PreparedStatement ps = conn.prepareStatement(topCategoriesSql)) {

                ps.setInt(1, *userId*);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    topCategories.add(rs.getString("category"));

                }

            }

            if (topCategories.isEmpty()) {

                String popularBooksSql =

                    "SELECT \* FROM books WHERE available\_copies > 0 ORDER BY total\_copies DESC LIMIT 10";

                try (PreparedStatement ps = conn.prepareStatement(popularBooksSql)) {

                    ResultSet rs = ps.executeQuery();

                    while (rs.next()) {

                        recommendations.add(new book(

                            rs.getInt("id"),

                            rs.getString("title"),

                            rs.getString("author"),

                            rs.getString("category"),

                            rs.getString("brief\_introduction"),

                            rs.getString("img\_path"),

                            rs.getInt("available\_copies"),

                            rs.getInt("total\_copies")

                        ));

                    }

                }

            return recommendations;

            }

            StringBuilder categoryPlaceholders = new StringBuilder();

            for (*int* i = 0; i < topCategories.size(); i++) {

                if (i > 0) categoryPlaceholders.append(",");

                categoryPlaceholders.append("?");

            }

            String sql = "SELECT \* FROM books WHERE " +

                        "category IN (" + categoryPlaceholders + ") " +

                        "AND available\_copies > 0 " +

                        "AND id NOT IN (SELECT book\_id FROM borrow\_records WHERE user\_id = ?) " +

                        "ORDER BY total\_copies DESC LIMIT 40";

            try (PreparedStatement ps = conn.prepareStatement(sql)) {

*int* idx = 1;

                for (String cat : topCategories) {

                    ps.setString(idx++, cat);

                }

                ps.setInt(idx++, *userId*);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    recommendations.add(new book(

                        rs.getInt("id"),

                        rs.getString("title"),

                        rs.getString("author"),

                        rs.getString("category"),

                        rs.getString("brief\_introduction"),

                        rs.getString("img\_path"),

                        rs.getInt("available\_copies"),

                        rs.getInt("total\_copies")

                    ));

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

        return recommendations;

    }

    public *boolean* borrowBook(*int* *userId*, *int* *bookId*){

        try (Connection driver = DbUtil.getConnection()){

            if(driver == null){

                System.err.println("Database connection is null.");

                return false;

            }

            driver.setAutoCommit(false);

            String checkRepeatSql = "SELECT COUNT(\*) FROM borrow\_records WHERE user\_id = ? AND book\_id = ? AND returned = FALSE";

            try (PreparedStatement psCheckRepeat = driver.prepareStatement(checkRepeatSql)) {

                psCheckRepeat.setInt(1, *userId*);

                psCheckRepeat.setInt(2, *bookId*);

                ResultSet rsCheck = psCheckRepeat.executeQuery();

                if (rsCheck.next() && rsCheck.getInt(1) > 0) {

                    driver.rollback();

                    System.err.println("User has already borrowed this book and not returned.");

                    return false;

                }

            }

            String sql\_query = "SELECT available\_copies FROM books WHERE id=? FOR UPDATE";

            try (PreparedStatement check = driver.prepareStatement(sql\_query)) {

                check.setInt(1, *bookId*);

                ResultSet rs = check.executeQuery();

                if(rs.next() && rs.getInt("available\_copies") > 0){

                    String sql\_insert = "INSERT INTO borrow\_records (user\_id, book\_id, borrow\_date, due\_date) VALUES (?, ?, NOW(), DATE\_ADD(NOW(), INTERVAL 14 DAY))";

                    try (PreparedStatement borrow = driver.prepareStatement(sql\_insert)) {

                        borrow.setInt(1, *userId*);

                        borrow.setInt(2, *bookId*);

                        borrow.executeUpdate();

                    }

                    String sql\_update = "UPDATE books SET available\_copies = available\_copies - 1 WHERE id = ?";

                    try (PreparedStatement update = driver.prepareStatement(sql\_update)) {

                        update.setInt(1, *bookId*);

                        update.executeUpdate();

                    }

                    driver.commit();

                    return true;

                } else {

                    driver.rollback();

                    System.err.println("No available copies to borrow.");

                    return false;

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

        return false;

    }

    public *boolean* returnBook(*int* *userId*, *int* *bookId*){

        try(Connection driver = DbUtil.getConnection()){

            if(driver == null){

                System.err.println("Database connection is null.");

                return false;

            }

            driver.setAutoCommit(false);

            String settime\_sql = "UPDATE borrow\_records SET returned = TRUE, return\_date = NOW() WHERE user\_id = ? AND book\_id = ? AND returned = FALSE LIMIT 1";

            try (PreparedStatement ps = driver.prepareStatement(settime\_sql)) {

                ps.setInt(1, *userId*);

                ps.setInt(2, *bookId*);

*int* updated = ps.executeUpdate();

                if(updated == 1){

                    String updateBook\_sql = "UPDATE books SET available\_copies = available\_copies + 1 WHERE id = ?";

                    try (PreparedStatement updateBook = driver.prepareStatement(updateBook\_sql)) {

                        updateBook.setInt(1, *bookId*);

                        updateBook.executeUpdate();

                    }

                    driver.commit();

                    return true;

                } else {

                    driver.rollback();

                    System.err.println("No matching borrow record found to return.");

                }

            }

        } catch(Exception e){

            e.printStackTrace();

        }

        return false;

    }

    public List<book> getRecommendations(*int* *userId*) {

        List<book> recommendations = new ArrayList<>();

        try (Connection conn = DbUtil.getConnection()) {

            if (conn == null) {

                System.err.println("Database connection is null.");

                return recommendations;

            }

            String topCategoriesSql =

                "SELECT b.category, COUNT(\*) AS borrow\_count " +

                "FROM borrow\_records br " +

                "JOIN books b ON br.book\_id = b.id " +

                "WHERE br.user\_id = ? " +

                "GROUP BY b.category " +

                "ORDER BY borrow\_count DESC " +

                "LIMIT 3";

            List<String> topCategories = new ArrayList<>();

            try (PreparedStatement ps = conn.prepareStatement(topCategoriesSql)) {

                ps.setInt(1, *userId*);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    topCategories.add(rs.getString("category"));

                }

            }

            if (topCategories.isEmpty()) {

                String popularBooksSql =

                    "SELECT \* FROM books WHERE available\_copies > 0 ORDER BY total\_copies DESC LIMIT 10";

                try (PreparedStatement ps = conn.prepareStatement(popularBooksSql)) {

                    ResultSet rs = ps.executeQuery();

                    while (rs.next()) {

                        recommendations.add(new book(

                            rs.getInt("id"),

                            rs.getString("title"),

                            rs.getString("author"),

                            rs.getString("category"),

                            rs.getString("brief\_introduction"),

                            rs.getString("img\_path"),

                            rs.getInt("available\_copies"),

                            rs.getInt("total\_copies")

                        ));

                    }

                }

                return recommendations;

            }

            StringBuilder categoryPlaceholders = new StringBuilder();

            for (*int* i = 0; i < topCategories.size(); i++) {

                if (i > 0) categoryPlaceholders.append(",");

                categoryPlaceholders.append("?");

            }

            String recommendSql =

                "SELECT \* FROM books " +

                "WHERE category IN (" + categoryPlaceholders + ") " +

                "AND available\_copies > 0 " +

                "AND id NOT IN ( " +

                "   SELECT book\_id FROM borrow\_records WHERE user\_id = ? " +

                ") " +

                "ORDER BY total\_copies DESC " +

                "LIMIT 40";

            try (PreparedStatement ps = conn.prepareStatement(recommendSql)) {

*int* idx = 1;

                for (String cat : topCategories) {

                    ps.setString(idx++, cat);

                }

                ps.setInt(idx, *userId*);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    recommendations.add(new book(

                        rs.getInt("id"),

                        rs.getString("title"),

                        rs.getString("author"),

                        rs.getString("category"),

                        rs.getString("brief\_introduction"),

                        rs.getString("img\_path"),

                        rs.getInt("available\_copies"),

                        rs.getInt("total\_copies")

                    ));

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

        return recommendations;

    }

    public List<book> getBorrowedBooks(*int* *userId*) {

        List<book> books = new ArrayList<>();

        String sql = "SELECT b.\* FROM books b " +

                 "JOIN borrow\_records r ON b.id = r.book\_id " +

                 "WHERE r.user\_id = ? AND r.returned = 0";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement stmt = conn.prepareStatement(sql)) {

            stmt.setInt(1, *userId*);

            ResultSet rs = stmt.executeQuery();

            while (rs.next()) {

                book bk = new book(

                    rs.getInt("id"),

                    rs.getString("title"),

                    rs.getString("author"),

                    rs.getString("category"),

                    rs.getString("brief\_introduction"),

                    rs.getString("img\_path"),

                    rs.getInt("available\_copies"),

                    rs.getInt("total\_copies")

                );

                books.add(bk);

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return books;

    }

    public *boolean* addBook(String *title*, String *author*, String *category*, *int* *copies*) {

        String sql = "INSERT INTO `books` (`title`, `author`, `category`, `total\_copies`, `available\_copies`) VALUES (?, ?, ?, ?, ?)";

        try (Connection conn = DbUtil.getConnection()) {

            if (conn == null) {

                System.err.println("Database connection is null.");

                return false;

            }

            try (PreparedStatement stmt = conn.prepareStatement(sql)) {

                stmt.setString(1, *title*);

                stmt.setString(2, *author*);

                stmt.setString(3, *category*);

                stmt.setInt(4, *copies*);

                stmt.setInt(5, *copies*);

                return stmt.executeUpdate() == 1;

            }

        } catch (SQLException e) {

            System.err.println("Error adding book: " + e.getMessage());

            return false;

        }

    }

    public *boolean* deleteBook(*int* *bookId*) {

        String sql = "DELETE FROM books WHERE id = ?";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setInt(1, *bookId*);

            return ps.executeUpdate() == 1;

        } catch (SQLException e) {

            System.err.println("Error deleting book: " + e.getMessage());

            return false;

        }

    }

    public *boolean* updateBook(*int* *id*, String *title*, String *author*,

                              String *category*, *int* *totalCopies*, *int* *availableCopies*) {

        String sql = "UPDATE books SET title = ?, author = ?, category = ?, " +

                     "total\_copies = ?, available\_copies = ? WHERE id = ?";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setString(1, *title*);

            ps.setString(2, *author*);

            ps.setString(3, *category*);

            ps.setInt(4, *totalCopies*);

            ps.setInt(5, *availableCopies*);

            ps.setInt(6, *id*);

            return ps.executeUpdate() == 1;

        } catch (SQLException e) {

            System.err.println("Error updating book: " + e.getMessage());

            return false;

        }

    }

    public book getBookById(*int* *id*) {

        String sql = "SELECT \* FROM books WHERE id = ?";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setInt(1, *id*);

            ResultSet rs = ps.executeQuery();

                if (rs.next()) {

                    return new book(

                    rs.getInt("id"),

                    rs.getString("title"),

                    rs.getString("author"),

                    rs.getString("category"),

                    rs.getString("brief\_introduction"),

                    rs.getString("img\_path"),

                    rs.getInt("available\_copies"),

                    rs.getInt("total\_copies")

                );

            }

        } catch (SQLException e) {

            System.err.println("Error fetching book: " + e.getMessage());

        }

        return null;

    }

    public List<book> searchBooks(String *keyword*) {

        if(*keyword* == null){

            List<book> books = getAllBooks();

            return books;

        }else{

            List<book> list = new ArrayList<>();

            String sql = "SELECT \* FROM books WHERE " +

                         "title LIKE ? OR author LIKE ? OR category LIKE ?";

            try (Connection conn = DbUtil.getConnection();

                 PreparedStatement ps = conn.prepareStatement(sql)) {

                String key = "%" + *keyword* + "%";

                ps.setString(1, key);

                ps.setString(2, key);

                ps.setString(3, key);

                ResultSet rs = ps.executeQuery();

                while (rs.next()) {

                    book b = new book(

                        rs.getInt("id"),

                        rs.getString("title"),

                        rs.getString("author"),

                        rs.getString("category"),

                        rs.getString("brief\_introduction"),

                        rs.getString("img\_path"),

                        rs.getInt("available\_copies"),

                        rs.getInt("total\_copies")

                    );

                    list.add(b);

                }

            } catch (SQLException e) {

                System.err.println("Error searching books: " + e.getMessage());

            }

            return list;

        }

    }

    public List<book> getAllBooks() {

        List<book> books = new ArrayList<>();

        String sql = "SELECT \* FROM books";

        try (Connection conn = DbUtil.getConnection();

            PreparedStatement stmt = conn.prepareStatement(sql);

            ResultSet rs = stmt.executeQuery()) {

            while (rs.next()) {

                book bk = new book(

                    rs.getInt("id"),

                    rs.getString("title"),

                    rs.getString("author"),

                    rs.getString("category"),

                    rs.getString("brief\_introduction"),

                    rs.getString("img\_path"),

                    rs.getInt("available\_copies"),

                    rs.getInt("total\_copies")

                );

                books.add(bk);

            }

        } catch (SQLException e) {

            System.err.println("Error fetching all books: " + e.getMessage());

        }

        return books;

    }

}

The BorrowRecordDAO class serves as the data access layer for all borrowing operations in the library management system, providing essential methods to check for overdue books via hasOverdueBooks() using SQL timestamp comparisons, retrieve specific overdue book IDs with getOverdueBookIds(), and perform comprehensive searches across borrowing records through searchBorrowRecords() and getAllBorrowRecords() methods that join user and book tables to return detailed loan histories including usernames, titles, borrow/due dates, and return statuses in map structures, while also offering administrative statistical insights via getBorrowStats() that aggregates total borrows, current loans, overdue counts, and identifies the most active users through complex grouping queries, ensuring complete oversight of library lending activities and user accountability within the borrowing lifecycle.

package database;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class BorrowRecordDAO {

    public static *boolean* hasOverdueBooks(*int* *userId*) {

        String sql = "SELECT COUNT(\*) FROM borrow\_records WHERE user\_id = ? AND returned = false AND due\_date < NOW()";

        try (Connection conn = DbUtil.getConnection();

            PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setInt(1, *userId*);

            try (ResultSet rs = ps.executeQuery()) {

                if (rs.next()) {

                    return rs.getInt(1) > 0;

                }

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return false;

    }

    public static List<Map<String, Object>> searchBorrowRecords(String *keyword*) {

        List<Map<String, Object>> records = new ArrayList<>();

        String sql = "SELECT br.id, u.username, b.title, br.borrow\_date, br.due\_date, br.returned " +

                "FROM borrow\_records br " +

                "JOIN users u ON br.user\_id = u.id " +

                "JOIN books b ON br.book\_id = b.id " +

                "WHERE u.username LIKE ? OR b.title LIKE ? " +

                "ORDER BY br.borrow\_date DESC";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql)) {

            String like = "%" + *keyword* + "%";

            ps.setString(1, like);

            ps.setString(2, like);

            try (ResultSet rs = ps.executeQuery()) {

                while (rs.next()) {

                    Map<String, Object> map = new HashMap<>();

                    map.put("id", rs.getInt("id"));

                    map.put("username", rs.getString("username"));

                    map.put("title", rs.getString("title"));

                    map.put("borrow\_date", rs.getTimestamp("borrow\_date"));

                    map.put("due\_date", rs.getTimestamp("due\_date"));

                    map.put("returned", rs.getBoolean("returned"));

                    records.add(map);

                }

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return records;

    }

    public static List<Integer> getOverdueBookIds(*int* *userId*) {

        List<Integer> overdueBookIds = new ArrayList<>();

        String sql = "SELECT book\_id FROM borrow\_records WHERE user\_id = ? AND returned = false AND due\_date < NOW()";

        try (Connection conn = DbUtil.getConnection();

            PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setInt(1, *userId*);

            try (ResultSet rs = ps.executeQuery()) {

                while (rs.next()) {

                    overdueBookIds.add(rs.getInt("book\_id"));

                }

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return overdueBookIds;

    }

    public static List<String> getBorrowStats() {

        List<String> stats = new ArrayList<>();

        try (Connection conn = DbUtil.getConnection()) {

            if (conn == null) {

                stats.add("Database connection error.");

                return stats;

            }

            String totalSql = "SELECT COUNT(\*) FROM borrow\_records";

            try (PreparedStatement ps = conn.prepareStatement(totalSql);

                 ResultSet rs = ps.executeQuery()) {

                if (rs.next()) stats.add("Total borrows: " + rs.getInt(1));

            }

            String currentSql = "SELECT COUNT(\*) FROM borrow\_records WHERE returned = false";

            try (PreparedStatement ps = conn.prepareStatement(currentSql);

                 ResultSet rs = ps.executeQuery()) {

                if (rs.next()) stats.add("Currently borrowed: " + rs.getInt(1));

            }

            String overdueSql = "SELECT COUNT(\*) FROM borrow\_records WHERE returned = false AND due\_date < NOW()";

            try (PreparedStatement ps = conn.prepareStatement(overdueSql);

                 ResultSet rs = ps.executeQuery()) {

                if (rs.next()) stats.add("Overdue borrows: " + rs.getInt(1));

            }

            String topUserSql = "SELECT u.username, COUNT(\*) as total FROM borrow\_records br JOIN users u ON br.user\_id = u.id GROUP BY u.username ORDER BY total DESC LIMIT 1";

            try (PreparedStatement ps = conn.prepareStatement(topUserSql);

                 ResultSet rs = ps.executeQuery()) {

                if (rs.next()) {

                    stats.add("Most active user: " + rs.getString("username") + " (borrows: " + rs.getInt("total") + ")");

                }

            }

        } catch (SQLException e) {

            stats.add("Error: " + e.getMessage());

        }

        return stats;

    }

    public static List<Map<String, Object>> getAllBorrowRecords() {

        List<Map<String, Object>> records = new ArrayList<>();

        String sql = "SELECT br.id, u.username, b.title, br.borrow\_date, br.due\_date, br.returned FROM borrow\_records br JOIN users u ON br.user\_id = u.id JOIN books b ON br.book\_id = b.id ORDER BY br.borrow\_date DESC";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql);

             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {

                Map<String, Object> map = new HashMap<>();

                map.put("id", rs.getInt("id"));

                map.put("username", rs.getString("username"));

                map.put("title", rs.getString("title"));

                map.put("borrow\_date", rs.getTimestamp("borrow\_date"));

                map.put("due\_date", rs.getTimestamp("due\_date"));

                map.put("returned", rs.getBoolean("returned"));

                records.add(map);

            }

        } catch (SQLException e) {

            e.printStackTrace();

        }

        return records;

    }

}

#### Models Module

The User class is a model entity in the library management system, designed to represent user accounts with properties including a unique integer identifier (id), authentication credentials (username and password), role-based access control classification (role), and account status flag (frozen). It implements Java's Serializable interface to support object persistence and network transfer capabilities. This POJO (Plain Old Java Object) serves as the fundamental data container for user information throughout the application, with getter/setter methods providing controlled access to its private fields while maintaining encapsulation principles required for secure user management operations such as authentication, authorization, and administrative account control functions.

package models;

import java.io.Serializable;

public class User implements Serializable {

    private static final *long* serialVersionUID = 1L;

    private *int* id;

    private String username;

    private String password;

    private String role;

    private *boolean* frozen;

    public User(*int* *id*, String *username*, String *password*, String *role*, *boolean* *frozen*) {

        this.id = *id*;

        this.username = *username*;

        this.password = *password*;

        this.role = *role*;

        this.frozen = *frozen*;

    }

    public *int* getId() {

        return id;

    }

    public *void* setId(*int* *id*) {

        this.id = *id*;

    }

    public String getUsername() {

        return username;

    }

    public *void* setUsername(String *username*) {

        this.username = *username*;

    }

    public String getPassword() {

        return password;

    }

    public *void* setPassword(String *password*) {

        this.password = *password*;

    }

    public String getRole() {

        return role;

    }

    public *void* setRole(String *role*) {

        this.role = *role*;

    }

    public *boolean* isFrozen() {

        return frozen;

    }

    public *void* setFrozen(*boolean* *frozen*) {

        this.frozen = *frozen*;

    }

}

The book class functions as the primary data model representing individual book entities within the library management system, encapsulating properties for unique identification (id), descriptive metadata (title, author, category, brief\_introduction), visual representation (img\_path), inventory management (available\_copies, total\_copies), and advanced text processing capabilities through a tfidfMap for recommendation algorithms. Implemented as a serializable Java Bean with getter/setter methods for all attributes, it facilitates object-relational mapping by mirroring database structure while providing domain-specific functionality like generating combined search text via getCombinedText() that concatenates category, author, and introduction fields for enhanced information retrieval operations within the system's search and recommendation subsystems.

package models;

import java.io.Serializable;

import java.util.Map;

public class book implements Serializable{

    private static final *long* serialVersionUID = 1L;

    private *int* id;

    private String title;

    private String author;

    private String category;

    private String brief\_introduction;

    private String img\_path;

    private *int* available\_copies;

    private *int* total\_copies;

    private Map<String, Double> tfidfMap;

    public book(*int* *id*, String *title*, String *author*, String *category*, String *brief\_introduction*, String *img\_path*, *int* *available\_copies*, *int* *total\_copies*) {

        this.id = *id*;

        this.title = *title*;

        this.author = *author*;

        this.category = *category*;

        this.available\_copies = *available\_copies*;

        this.total\_copies = *total\_copies*;

        this.brief\_introduction = *brief\_introduction*;

        this.img\_path = *img\_path*;

    }

    public *int* getId() {

        return id;

    }

    public *void* setId(*int* *id*) {

        this.id = *id*;

    }

    public String getTitle() {

        return title;

    }

    public *void* setTitle(String *title*) {

        this.title = *title*;

    }

    public String getAuthor() {

        return author;

    }

    public *void* setAuthor(String *author*) {

        this.author = *author*;

    }

    public String getCategory() {

        return category;

    }

    public *void* setCategory(String *category*) {

        this.category = *category*;

    }

    public String getBrief\_introduction() {

        return brief\_introduction;

    }

    public *void* setBrief\_introduction(String *brief\_introduction*) {

        this.brief\_introduction = *brief\_introduction*;

    }

    public String getImg\_path() {

        return img\_path;

    }

    public *void* setImg\_path(String *img\_path*) {

        this.img\_path = *img\_path*;

    }

    public *int* getAvailable\_copies() {

        return available\_copies;

    }

    public *void* setAvailable\_copies(*int* *available\_copies*) {

        this.available\_copies = *available\_copies*;

    }

    public *int* getTotal\_copies() {

        return total\_copies;

    }

    public *void* setTotal\_copies(*int* *total\_copies*) {

        this.total\_copies = *total\_copies*;

    }

    public String getCombinedText() {

        return category + " " + author + " " + brief\_introduction;

    }

    public Map<String, Double> getTfidfMap() {

        return tfidfMap;

    }

    public *void* setTfidfMap(Map<String, Double> *tfidfMap*) {

        this.tfidfMap = *tfidfMap*;

    }

}

The borrow\_record class is a serializable entity modeling library loan transactions between users and books, tracking temporal elements (borrow\_date, return\_date, due\_date), participating entities (user\_id, book\_id), and lifecycle status (returned). Using Java 8's LocalDateTime for precise temporal handling, this POJO provides comprehensive getter/setter methods to encapsulate all loan lifecycle states within the library management ecosystem while supporting data persistence and transmission requirements through Java serialization mechanisms.

package models;

import java.io.Serializable;

import java.time.LocalDateTime;

public class borrow\_record implements Serializable {

    private static final *long* serialVersionUID = 1L;

    private *int* id;

    private *int* user\_id;

    private *int* book\_id;

    private LocalDateTime borrow\_date;

    private LocalDateTime return\_date;

    private LocalDateTime due\_data;

    private *boolean* returned;

    public borrow\_record(*int* *id*, *int* *user\_id*, *int* *book\_id*, LocalDateTime *borrow\_date*, LocalDateTime *return\_date*, LocalDateTime *due\_date*, *boolean* *returned*) {

        this.id = *id*;

        this.user\_id = *user\_id*;

        this.book\_id = *book\_id*;

        this.borrow\_date = *borrow\_date*;

        this.return\_date = *return\_date*;

        this.due\_data = *due\_date*;

        this.returned = *returned*;

    }

    public *int* getId() {

        return id;

    }

    public *int* getUser\_id() {

        return user\_id;

    }

    public *int* getBook\_id() {

        return book\_id;

    }

    public LocalDateTime getBorrow\_date() {

        return borrow\_date;

    }

    public LocalDateTime getReturn\_date() {

        return return\_date;

    }

    public LocalDateTime getDue\_data() {

        return due\_data;

    }

    public *boolean* isReturned() {

        return returned;

    }

    public *void* setId(*int* *id*) {

        this.id = *id*;

    }

    public *void* setUser\_id(*int* *user\_id*) {

        this.user\_id = *user\_id*;

    }

    public *void* setBook\_id(*int* *book\_id*) {

        this.book\_id = *book\_id*;

    }

    public *void* setBorrow\_date(LocalDateTime *borrow\_date*) {

        this.borrow\_date = *borrow\_date*;

    }

    public *void* setReturn\_date(LocalDateTime *return\_date*) {

        this.return\_date = *return\_date*;

    }

    public *void* setDue\_data(LocalDateTime *due\_data*) {

        this.due\_data = *due\_data*;

    }

    public *void* setReturned(*boolean* *returned*) {

        this.returned = *returned*;

    }

}

#### Networking & Request Handling Module

#### The ClientHandler class is the core request processing engine of the library management system's server, implementing a multi-threaded socket handler that manages all client-server communication through serialized Java object streams. It asynchronously processes diverse client actions including user authentication (login, register), inventory operations (add\_book, update\_book, delete\_book, search\_books), borrowing lifecycle management (borrow, return, get\_borrowed\_books),personalized recommendations(get\_recommendations, search\_recommended\_books),administrative functions (toggle\_freeze, get\_users, get\_borrow\_stats), and comprehensive record tracking (get\_all\_borrow\_records, search\_borrow\_records, check\_overdue\_for\_user). This handler enforces strict transaction boundaries with 30-second timeouts, implements error resilience through structured exception handling and socket cleanup protocols, and utilizes domain-specific Data Access Objects (DAOs) to execute database operations while maintaining thread safety for concurrent client connections within the library service architecture.

package server.handlers;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.net.SocketTimeoutException;

import java.util.List;

import java.util.Map;

import database.Book\_DAO;

import database.BorrowRecordDAO;

import database.User\_DAO;

import models.User;

import models.book;

import network.request;

import network.response;

public class ClientHandler implements Runnable {

    private final Socket clientSocket;

    private static final *int* MAX\_RECOMMENDATIONS = 50;

    public ClientHandler(Socket *clientSocket*) {

        this.clientSocket = *clientSocket*;

    }

    @Override

    public *void* run() {

        try (ObjectOutputStream outputStream = new ObjectOutputStream(clientSocket.getOutputStream());

             ObjectInputStream inputStream = new ObjectInputStream(clientSocket.getInputStream())) {

            clientSocket.setSoTimeout(30000);

            System.out.println("Client connected: " + clientSocket.getRemoteSocketAddress());

            while (!clientSocket.isClosed() && clientSocket.isConnected()) {

                try {

                    Object requestObj = inputStream.readObject();

                    if (!(requestObj instanceof request)) {

                        throw new IllegalArgumentException("Invalid request type");

                    }

                    request clientRequest = (request) requestObj;

                    System.out.println("Processing: " + clientRequest.getAction());

                    response serverResponse = handleRequest(clientRequest);

                    outputStream.writeObject(serverResponse);

                    outputStream.flush();

                    outputStream.reset(); // 清除对象缓存

                } catch (SocketTimeoutException e) {

                    System.out.println("Client timeout: " + e.getMessage());

                    break;

                } catch (ClassCastException | ClassNotFoundException e) {

                    System.err.println("Invalid request format: " + e.getMessage());

                    break;

                } catch (Exception e) {

                    System.err.println("Request handling error: " + e.getMessage());

                    e.printStackTrace();

                    // 发送错误响应

                    response errorResponse = new response(false, null, "Error: " + e.getMessage());

                    outputStream.writeObject(errorResponse);

                    outputStream.flush();

                    if (e instanceof IllegalArgumentException) {

                        break;

                    }

                }

            }

        } catch (IOException e) {

            System.err.println("Connection error: " + e.getMessage());

        } finally {

            closeQuietly(clientSocket);

            System.out.println("Client disconnected");

        }

    }

    private *void* closeQuietly(Socket *socket*) {

        try {

            if (*socket* != null && !*socket*.isClosed()) {

*socket*.close();

            }

        } catch (IOException e) {

            System.err.println("Error closing socket: " + e.getMessage());

        }

    }

    private response handleRequest(request *clientRequest*) {

        try {

            switch (*clientRequest*.getAction()) {

                case "login":

                    String login\_username = (String) *clientRequest*.getData().get("username");

                    String login\_password = (String) *clientRequest*.getData().get("password");

                    User\_DAO userDao = new User\_DAO();

                    User user = userDao.Login(login\_username, login\_password);

                    if(user != null) {

                        if(user.isFrozen()) {

                            return new response(false, null, "Account is frozen.");

                        } else {

                            return new response(true, user, "Login successful.");

                        }

                    } else {

                        return new response(false, null, "Invalid username or password.");

                    }

                case "register":

                    String register\_username = (String) *clientRequest*.getData().get("username");

                    String register\_password = (String) *clientRequest*.getData().get("password");

                    String register\_role = (String) *clientRequest*.getData().get("role");

                    String register\_code = (String) *clientRequest*.getData().get("activationCode");

                    User\_DAO user\_DAO = new User\_DAO();

                    if (!user\_DAO.isUsernameAvailable(register\_username)) {

                        return new response(false, null, "Username already exists.");

                    }

*boolean* regSuccess = user\_DAO.Register(register\_username, register\_password, register\_role, register\_code);

                    if (regSuccess) {

                        return new response(true, null, "Registration successful.");

                    } else {

                        return new response(false, null, "Registration failed. Check activation code or database error.");

                    }

                case "search\_borrowed\_books": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    String keyword = (String) *clientRequest*.getData().get("keyword");

                    Book\_DAO dao = new Book\_DAO();

                    List<book> result = dao.searchBorrowedBooks(userId, keyword);

                    return new response(true, result, "The search user has successfully borrowed the book");

                }

                case "search\_recommended\_books": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    String keyword = (String) *clientRequest*.getData().get("keyword");

                    Book\_DAO dao = new Book\_DAO();

                    List<book> result = dao.searchRecommendedBooks(userId, keyword);

                    return new response(true, result, "The search for recommended books is successful");

                }

                case "borrow": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

*int* bookId = (*int*) *clientRequest*.getData().get("bookId");

                    Book\_DAO book\_DAO = new Book\_DAO();

*boolean* success = book\_DAO.borrowBook(userId, bookId);

                    return new response(success, null, success ? "Success borrow" : "Fail borrow");

                }

                case "return": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

*int* bookId = (*int*) *clientRequest*.getData().get("bookId");

                    Book\_DAO book\_DAO = new Book\_DAO();

*boolean* success = book\_DAO.returnBook(userId, bookId);

                    return new response(success, null, success ? "Success return" : "Fail return");

                }

                case "get\_recommendations": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    Book\_DAO book\_DAO = new Book\_DAO();

                    List<book> recommendedBooks = book\_DAO.getRecommendations(userId);

                    // 限制返回数量

                    if (recommendedBooks.size() > MAX\_RECOMMENDATIONS) {

                        recommendedBooks = recommendedBooks.subList(0, MAX\_RECOMMENDATIONS);

                    }

                    return new response(true, recommendedBooks, "Recommendations retrieved");

                }

                case "get\_borrowed\_books": {

                    try {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                        Book\_DAO book\_DAO = new Book\_DAO();

                        List<book> borrowedBooks = book\_DAO.getBorrowedBooks(userId);

                        return new response(true, borrowedBooks, "Borrowed books retrieved");

                    } catch (Exception e) {

                        e.printStackTrace();

                        return new response(false, null, "Failed to retrieve borrowed books");

                    }

                }

                case "get\_users": {

                    User\_DAO dao = new User\_DAO();

                    List<User> users = dao.getAllUsers();

                    System.out.println("数据库返回用户数量: " + users.size());

                    return new response(true, users, "Users loaded.");

                }

                case "toggle\_freeze": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    User\_DAO dao = new User\_DAO();

*boolean* success = dao.toggleFreezeStatus(userId);

                    if (success) {

                        return new response(true, null, "User freeze status toggled.");

                    } else {

                        return new response(false, null, "Failed to toggle freeze status.");

                    }

                }

                case "get\_all\_books": {

                    Book\_DAO dao = new Book\_DAO();

                    List<book> allBooks = dao.getAllBooks();

                    return new response(true, allBooks, "All books loaded.");

                }

                case "add\_book": {

                    try {

                        String title = (String) *clientRequest*.getData().get("title");

                        String author = (String) *clientRequest*.getData().get("author");

                        String category = (String) *clientRequest*.getData().get("category");

                        Object copiesObj = *clientRequest*.getData().get("copies");

*int* copies = (copiesObj instanceof Number) ? ((Number) copiesObj).intValue() : Integer.parseInt(copiesObj.toString());

                        Book\_DAO dao = new Book\_DAO();

*boolean* ok = dao.addBook(title, author, category, copies);

                        return new response(ok, null, ok ? "Book added successfully." : "Failed to add book.");

                    } catch (Exception e) {

                        return new response(false, null, "Invalid book data: " + e.getMessage());

                    }

                }

                case "search\_books": {

                    String keyword = (String) *clientRequest*.getData().get("keyword");

                    Book\_DAO book\_DAO = new Book\_DAO();

                    List<book> books = book\_DAO.searchBooks(keyword);

                    return new response(true, books, "Search completed successfully.");

                }

                case "search\_user": {

                    String keyword = (String) *clientRequest*.getData().get("keyword");

                    User\_DAO user\_dao1 = new User\_DAO();

                    List<User> users = user\_dao1.searchUser(keyword);

                    return new response(true, users, "Search completed successfully.");

                }

                case "delete\_book": {

*int* id = (*int*) *clientRequest*.getData().get("bookId");

                    Book\_DAO book\_DAO = new Book\_DAO();

*boolean* success = book\_DAO.deleteBook(id);

                    return new response(success, null, success ? "Book deleted successfully." : "Failed to delete book.");

                }

                case "update\_book": {

                    Map<String, Object> data = *clientRequest*.getData();

*int* id = (*int*) data.get("id");

                    String title = (String) data.get("title");

                    String author = (String) data.get("author");

                    String category = (String) data.get("category");

*int* total = (*int*) data.get("total");

*int* available = (*int*) data.get("available");

                    Book\_DAO book\_DAO = new Book\_DAO();

*boolean* updated = book\_DAO.updateBook(id, title, author, category, total, available);

                    return new response(updated, null, updated ? "Book updated successfully." : "Failed to update book.");

                }

                case "get\_borrow\_stats": {

                    BorrowRecordDAO borrowRecordDAO = new BorrowRecordDAO();

                    List<String> stats = borrowRecordDAO.getBorrowStats();

                    return new response(true, stats,"Stats loaded" );

                }

                case "check\_overdue\_for\_user": {

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    BorrowRecordDAO borrowRecordDAO = new BorrowRecordDAO();

*boolean* hasOverdue = borrowRecordDAO.hasOverdueBooks(userId);

                    return new response(true, hasOverdue,"Checked");

                }

                case "get\_all\_borrow\_records": {

                List<Map<String, Object>> records = BorrowRecordDAO.getAllBorrowRecords();

                return new response(true, records, "All borrow records loaded.");

                }

                case "search\_borrow\_records": {

                    String keyword = (String) *clientRequest*.getData().get("keyword");

                    List<Map<String, Object>> results = BorrowRecordDAO.searchBorrowRecords(keyword);

                    return new response(true, results, "Search complete");

                }

                case "get\_overdue\_book\_ids":{

*int* userId = (*int*) *clientRequest*.getData().get("userId");

                    BorrowRecordDAO borrowRecordDAO = new BorrowRecordDAO();

                    List<Integer> overdueIds = borrowRecordDAO.getOverdueBookIds(userId);

                    return new response(true, overdueIds,"Fetched overdue IDs");

                }

                    default:

                        return new response(false, null, "Unsupported action: " + *clientRequest*.getAction());

                }

            } catch (Exception e) {

                System.err.println("Error handling request: " + *clientRequest*.getAction() + " - " + e.getMessage());

                return new response(false, null, "Server error: " + e.getMessage());

            }

    }

}

The ServerMain class serves as the entry point for launching the library management system's server infrastructure, implementing a concurrent TCP socket server that listens on port 9090 and employs a fixed-size thread pool with 20 worker threads to efficiently manage client connections. Upon initialization, it creates a ServerSocket bound to the designated port and enters an infinite loop where it continuously accepts incoming client connections via serv.accept(), immediately delegating each connected client's Socket to a new ClientHandler instance that executes asynchronously within the thread pool via threadPool.execute(). This architecture ensures robust handling of high-volume concurrent requests while preventing resource exhaustion, with the thread pool gracefully shutting down upon server termination through the finally block's threadPool.shutdown() call to ensure clean system exit and socket release under all operational conditions.

package server;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import server.handlers.ClientHandler;

public class ServerMain {

    public static *void* main(String[] *args*) {

        final *int* port = 9090;

        ExecutorService threadPool = Executors.newFixedThreadPool(20);

        try(ServerSocket serv = new ServerSocket(port)){

            System.out.println("Server started on port " + port);

            while (true) {

                Socket clienSocket = serv.accept();

                System.out.println("Client connected: "+ clienSocket.getInetAddress());

                threadPool.execute(new ClientHandler(clienSocket));

            }

        } catch (Exception e) {

            System.out.println("Server failed to start on port " + port);

            e.printStackTrace();

        } finally {

            threadPool.shutdown();

        }

    }

}

#### Request and Response

The request class is a serializable network protocol object that serves as the standardized envelope for client-server communications in the library management system, encapsulating both the operational intent through its mandatory action string parameter and associated payload data within a generic HashMap<String, Object> collection, enabling structured transmission of multi-typed parameters via the putData() method while maintaining consistent deserialization handling through its fixed serialVersionUID implementation and Java's built-in serialization mechanics, thereby allowing network endpoints to precisely interpret and process commands with their corresponding arguments.

package network;

import java.io.Serializable;

import java.util.HashMap;

public class request implements Serializable {

    private static final *long* serialVersionUID = 1L;

    private String action;

    private HashMap<String, Object> data = new HashMap<>();

    public request(String *action*) {

        this.action = *action*;

    }

    public String getAction() {

        return action;

    }

    public HashMap<String,Object> getData() {

        return data;

    }

    public *void* putData(String *key*, Object *value*) {

        data.put(*key*, *value*);

    }

}

The response class functions as the standardized serializable container for all server-to-client communications in the library management system, designed to encapsulate operation outcomes through three core fields: a boolean success flag indicating operation completion status, a human-readable message providing contextual execution details, and a polymorphic data Object payload that dynamically carries result sets (like collections of books/users), single entities, or primitive values. Implementing Java's Serializable interface with explicit version control (serialVersionUID), it enables efficient network transmission of structured responses containing both operational metadata and domain-specific return values, following a consistent pattern of read-only access via isSuccess(), getMessage(), and getData() methods to maintain data integrity during client-side processing of server responses.

package network;

import java.io.Serializable;

public class response implements Serializable{

    private static final *long* serialVersionUID = 1L;

    private *boolean* success;

    private String message;

    private Object data;

    public response(*boolean* *success*, Object *data*, String *message*) {

        this.success = *success*;

        this.data = *data*;

        this.message = *message*;

    }

    public *boolean* isSuccess() {

        return success;

    }

    public String getMessage() {

        return message;

    }

    public Object getData() {

        return data;

    }

}

#### Login Module

The LoginController class serves as the authentication gateway and navigation hub within a JavaFX-based client application for a library management system, orchestrating user authentication through synchronous network calls to a backend server while maintaining responsive UI interactions via asynchronous threading. It captures user credentials from usernameField and passwordField inputs, constructs serialized request objects containing authentication data, and establishes socket connections to localhost:9090 to transmit login attempts, subsequently processing server response objects that either contain authenticated User entities with role-based permissions or error messages. Upon successful validation, it dynamically instantiates role-specific interfaces (AdminView or UserView) by loading corresponding FXML resources, injecting the authenticated user context into subsequent controllers, and applying CSS styling themes while seamlessly transitioning views within the existing Stage. Concurrently, it implements singleton view management for registration screen access through lazy-loaded FXML instantiation and cached controller references, with comprehensive error handling workflows that generate user-aligned JavaFX alerts for network failures, invalid inputs, and navigation exceptions, all while leveraging Task<V> concurrency patterns to separate blocking network I/O from the JavaFX application thread, thereby ensuring UI responsiveness during server communication operations.

package client.controllers;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import database.User\_DAO;

import javafx.concurrent.Task;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.control.Alert;

import javafx.scene.control.PasswordField;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

import models.User;

import network.request;

import network.response;

public class LoginController {

    @FXML TextField usernameField;

    @FXML PasswordField passwordField;

    User\_DAO userDao = new User\_DAO();

    private static Parent registerRoot;

    private static RegisterController registerController;

    @FXML

    private *void* handleLogin() {

        String login\_username = usernameField.getText();

        String login\_password = passwordField.getText();

        if (login\_username.isEmpty() || login\_password.isEmpty()) {

            showAlert("Login Error", "Username and password cannot be empty.");

            return;

        }

        Task<response> loginTask = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (

                    Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream())

                ) {

                    request loginRequest = new request("login");

                    loginRequest.putData("username", login\_username);

                    loginRequest.putData("password", login\_password);

                    outputStream.writeObject(loginRequest);

                    outputStream.flush();

                    return (response) inputStream.readObject();

                }

            }

        };

        loginTask.setOnSucceeded(*evt* *->* {

            response loginResponse = loginTask.getValue();

            if (loginResponse.isSuccess()) {

                User user = (User) loginResponse.getData();

                try {

                    FXMLLoader loader;

                    Parent root;

                    Stage stage = (Stage) usernameField.getScene().getWindow();

                    if ("admin".equals(user.getRole())) {

                        loader = new FXMLLoader(getClass().getResource("/client/views/AdminView.fxml"));

                        root = loader.load();

                        AdminController controller = loader.getController();

                        controller.setCurrentUser(user);

                        stage.setTitle("Admin Dashboard");

                        Scene scene = new Scene(root);

                        scene.getStylesheets().add(getClass().getResource("/css/admin.css").toExternalForm());

                        stage.setScene(scene);

                    } else {

                        loader = new FXMLLoader(getClass().getResource("/client/views/UserView.fxml"));

                        root = loader.load();

                        UserController controller = loader.getController();

                        controller.setCurrentUser(user);

                        stage.setTitle("User Dashboard");

                        Scene scene = new Scene(root);

                        scene.getStylesheets().add(getClass().getResource("/css/user.css").toExternalForm());

                        stage.setScene(scene);

                    }

                } catch (Exception e) {

                    e.printStackTrace();

                    showAlert("UI Error", "Failed to load the dashboard: " + e.getMessage());

                }

            } else {

                showAlert("Login Failed", loginResponse.getMessage());

            }

        });

        loginTask.setOnFailed(*evt* *->* {

            Throwable ex = loginTask.getException();

            ex.printStackTrace();

            showAlert("Connection Error", "Failed to connect to server: " + ex.getMessage());

        });

        new Thread(loginTask).start();

    }

    private *void* showAlert(String *title*, String *content*) {

        Alert alert = new Alert(Alert.AlertType.ERROR);

        alert.setTitle(*title*);

        alert.setHeaderText(null);

        alert.setContentText(*content*);

        alert.showAndWait();

    }

    @FXML

    private *void* goToRegister() {

        try {

            if (registerRoot == null) {

                FXMLLoader loader = new FXMLLoader();

                loader.setLocation(getClass().getResource("/client/views/RegisterView.fxml"));

                registerRoot = loader.load();

                registerController = loader.getController();

                 registerRoot.getStylesheets().add(getClass().getResource("/css/register.css").toExternalForm());

            }

            Stage stage = (Stage) usernameField.getScene().getWindow();

            stage.getScene().setRoot(registerRoot);

            registerController.resetForm();

        } catch (Exception e) {

            e.printStackTrace();

            Alert alert = new Alert(Alert.AlertType.ERROR);

            alert.setTitle("Navigation Error");

            alert.setHeaderText(null);

            alert.setContentText("Failed to navigate to the registration page: " + e.getMessage());

            alert.showAndWait();

        }

    }

}

#### Registration Module

The RegisterController class is a JavaFX form controller that manages user account registration functionality within a client-side library management application, implementing role-based account creation with dynamic UI adaptation, comprehensive input validation, and background network communication with the backend server. It handles user interactions through TextField, PasswordField, and ChoiceBox components that capture username, password, password confirmation, and role selection ("user" or "admin"), dynamically controlling the visibility of an activation code input field based on role selection through its updateActivationFieldVisibility() method that displays the activation field exclusively when "admin" is chosen. Upon registration attempts, it performs sequential validation checks including field completeness, password-confirmation matching, and password complexity rules (6+ characters with letters/digits/symbols), then constructs a serializable request("register") object and transmits it via socket connection to localhost:9090 using a background Task<V> thread to prevent UI blocking, processing server response through event handlers that either display error alerts for failures or trigger a 5-second delayed navigation to the login screen via PauseTransition with auto-closing success notification on successful account creation, while maintaining form integrity through resetForm() for recurring registration sessions and ensuring thread-safe UI updates via Platform.runLater() within its showAlert() utility method.

package client.controllers;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import database.User\_DAO;

import javafx.animation.PauseTransition;

import javafx.application.Platform;

import javafx.collections.FXCollections;

import javafx.concurrent.Task;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.control.Alert;

import javafx.scene.control.ChoiceBox;

import javafx.scene.control.PasswordField;

import javafx.scene.control.TextField;

import javafx.stage.Stage;

import javafx.util.Duration;

import network.request;

import network.response;

public class RegisterController {

    @FXML private TextField usernameField;

    @FXML private PasswordField passwordField;

    @FXML private PasswordField confirmPasswordField;

    @FXML private PasswordField activationcodeField;

    @FXML private ChoiceBox<String> role;

    User\_DAO userDao = new User\_DAO();

    private *boolean* initialized = false;

    @FXML

    private *void* initialize() {

        if (initialized) return;

        initialized = true;

        role.setItems(FXCollections.observableArrayList("user", "admin"));

        role.valueProperty().addListener((*obs*, *oldVal*, *newVal*)*->* {

            updateActivationFieldVisibility(*newVal*);

        });

        resetForm();

    }

    public *void* resetForm() {

        if (usernameField != null) usernameField.clear();

        if (passwordField != null) passwordField.clear();

        if (confirmPasswordField != null) confirmPasswordField.clear();

        if (activationcodeField != null) {

            activationcodeField.clear();

            activationcodeField.setVisible(false);

        }

        if (role != null) {

            role.setValue("user");

            updateActivationFieldVisibility("user");

        }

    }

    private *void* updateActivationFieldVisibility(String *selectedRole*) {

*boolean* isAdmin = "admin".equals(*selectedRole*);

        if (activationcodeField != null) {

            activationcodeField.setVisible(isAdmin);

        }

    }

    @FXML

    private *void* handleRegister() {

        String username = usernameField.getText();

        String password = passwordField.getText();

        String confirmPassword = confirmPasswordField.getText();

        String selectedRole = role.getValue();

        String activationCode = activationcodeField.getText();

        if (selectedRole.equals("admin")) {

            if (username.isEmpty() || password.isEmpty() || confirmPassword.isEmpty() || activationCode.isEmpty()) {

                showAlert(Alert.AlertType.ERROR, "Registration Error", null, "All fields must be completed for admin registration.");

                return;

            }

        } else {

            if (username.isEmpty() || password.isEmpty() || confirmPassword.isEmpty()) {

                showAlert(Alert.AlertType.ERROR, "Registration Error", null, "Username, password, and confirmation password are required.");

                return;

            }

        }

        if (!samePassword(password, confirmPassword)) {

            showAlert(Alert.AlertType.ERROR, "Registration Error", null, "The passwords do not match.");

            return;

        }

        if (!isValidPassword(password)) {

            showAlert(Alert.AlertType.ERROR, "Registration Error", null, "Password must be at least 6 characters and contain letters, numbers, or symbols.");

            return;

        }

        Task<response> registerTask = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090); ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream()); ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream())) {

                    request registerRequest = new request("register");

                    registerRequest.putData("username", username);

                    registerRequest.putData("password", password);

                    registerRequest.putData("role", selectedRole);

                    registerRequest.putData("activationCode", activationCode);

                    outputStream.writeObject(registerRequest);

                    outputStream.flush();

                    return (response) inputStream.readObject();

                }

            }

        };

        registerTask.setOnSucceeded(*evt* *->* {

            response res = registerTask.getValue();

            if (res.isSuccess()) {

                Alert alert = new Alert(Alert.AlertType.INFORMATION);

                alert.setTitle("Registration Success");

                alert.setHeaderText(null);

                alert.setContentText("Registration successful.\nRedirecting to login page in 5 seconds...");

                alert.show();

                PauseTransition delay = new PauseTransition(Duration.seconds(5));

                delay.setOnFinished(*e* *->* {

                    alert.close();

                    try {

                        FXMLLoader loader = new FXMLLoader(getClass().getResource("/client/views/LoginView.fxml"));

                        Parent loginRoot = loader.load();

                        Stage stage = (Stage) usernameField.getScene().getWindow();

                        Scene scene = new Scene(loginRoot, 800, 600);

                        stage.setScene(scene);

                        stage.setTitle("Login");

                        stage.centerOnScreen();

                    } catch (Exception ex) {

                        ex.printStackTrace();

                        showAlert(Alert.AlertType.ERROR, "Navigation Error", null, "Failed to load login page.");

                    }

                });

                delay.play();

            } else {

                showAlert(Alert.AlertType.ERROR, "Registration Failed", null, res.getMessage());

            }

        });

        registerTask.setOnFailed(*evt* *->* {

            Throwable ex = registerTask.getException();

            showAlert(Alert.AlertType.ERROR, "System Error", null, "Registration failed: " + ex.getMessage());

        });

        new Thread(registerTask).start();

    }

    private *boolean* samePassword(String *password1*, String *password2*) {

        return *password1* != null && *password1*.equals(*password2*);

    }

    private *boolean* isValidPassword(String *password*) {

        if (*password* == null || *password*.length() < 6) {

            return false;

        }

*boolean* hasLetter = *password*.matches(".\*[a-zA-Z].\*");

*boolean* hasDigit = *password*.matches(".\*\\d.\*");

        return hasLetter || hasDigit;

    }

    private *void* showAlert(Alert.AlertType *type*, String *title*, String *header*, String *content*) {

        Platform.runLater(() *->* {

            Alert alert = new Alert(*type*);

            alert.setTitle(*title*);

            alert.setHeaderText(*header*);

            alert.setContentText(*content*);

            alert.showAndWait();

        });

    }

}

#### User Interface Module

The UserController class manages the user dashboard interface within a JavaFX-based library management client, establishing socket-based communication with the server at localhost:9090 to facilitate user-specific library operations while maintaining responsive UI interactions through asynchronous JavaFX Task implementations for networked requests. Upon authentication with a User object passed via setCurrentUser(), it displays personalized greetings, initiates overdue book checks through checkOverdueBooks() that sends request("get\_overdue\_book\_ids") to the server and triggers alerts for any late returns, and dynamically renders borrowed books in a TilePane layout populated with BookCardController instances generated from /client/views/BookCard.fxml templates, applying /css/bookcard.css styling while visually highlighting overdue items using setTitleRed(). This controller implements multi-step book management workflows including keyword-based searches through borrowed items via handleSearchBorrowedBooks() that dispatches request("search\_borrowed\_books") with text queries, batch returning of selected books through iterative request("return") operations in handleReturn() after selection validation, refreshing borrowed items lists with loadBorrowedBooks() using request("get\_borrowed\_books"), and launching a styled recommendation interface (RecommendView) through handleRecommend() applying /css/recommand.css in a new window - all while performing UI updates thread-safely with Platform.runLater() and maintaining selection states through traversal of card component userData properties to capture chosen books with getSelectedBookIds(), consistently separating network I/O operations from JavaFX thread execution to ensure interface responsiveness during server interactions.

package client.controllers;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.util.ArrayList;

import java.util.List;

import javafx.application.Platform;

import javafx.concurrent.Task;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.Parent;

import javafx.scene.Scene;

import javafx.scene.control.Alert;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.TilePane;

import javafx.scene.layout.VBox;

import models.User;

import models.book;

import network.request;

import network.response;

public class UserController {

    @FXML private TilePane bookTilePane;

    @FXML private TextField searchField;

    @FXML private Label welcomeLabel;

    @FXML private Button returnButton;

    @FXML private Button recommendButton;

    private User currentUser;

    private List<Integer> overdueBookIds = new ArrayList<>();

    public *void* setCurrentUser(User *user*){

        this.currentUser = *user*;

        welcomeLabel.setText("Welcome, " + currentUser.getUsername());

        checkOverdueBooks();

    }

    private List<Integer> getSelectedBookIds() {

        List<Integer> ids = new ArrayList<>();

        for (*var* node : bookTilePane.getChildren()) {

            Object userData = node.getUserData();

            if (userData instanceof BookCardController) {

                BookCardController controller = (BookCardController) userData;

                if (controller.isSelected()) {

                    ids.add(controller.getBook().getId());

                }

            }

        }

        return ids;

    }

   @FXML

    private *void* handleSearchBorrowedBooks() {

        String keyword = searchField.getText().trim();

        if (keyword.isEmpty()) return;

        Task<response> searchTask = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream())) {

                    request searchRequest = new request("search\_borrowed\_books");

                    searchRequest.putData("userId", currentUser.getId());

                    searchRequest.putData("keyword", keyword);

                    outputStream.writeObject(searchRequest);

                    outputStream.flush();

                    return (response) inputStream.readObject();

                }

            }

        };

        searchTask.setOnSucceeded(*evt* *->* {

            response res = searchTask.getValue();

            if (res.isSuccess()) {

                List<book> bookList = (List<book>) res.getData();

                loadBookCards(bookList, true);

            } else {

                showAlert("Search Failed", res.getMessage());

            }

        });

        searchTask.setOnFailed(*evt* *->* {

            Throwable e = searchTask.getException();

            e.printStackTrace();

            showAlert("Error", "Search failed: " + e.getMessage());

        });

        new Thread(searchTask).start();

    }

    public *void* loadBorrowedBooks() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("get\_borrowed\_books");

                    req.putData("userId", currentUser.getId());

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                List<book> borrowedBooks = (List<book>) res.getData();

                loadBookCards(borrowedBooks, true);

            } else {

                showAlert("Error", "Failed to load borrowed books.");

            }

        });

        task.setOnFailed(*e* *->* showAlert("Error", "Failed to load borrowed books."));

        new Thread(task).start();

    }

    private *void* checkOverdueBooks() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("get\_overdue\_book\_ids");

                    req.putData("userId", currentUser.getId());

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                overdueBookIds = (List<Integer>) res.getData();

                if (!overdueBookIds.isEmpty()) {

                    showAlert("Overdue Reminder", "You have overdue books. Please return them as soon as possible!");

                }

            }

        });

        new Thread(task).start();

    }

    private *void* loadBookCards(List<book> *bookList*, *boolean* *showBorrowCheckbox*) {

        bookTilePane.getChildren().clear();

        for (book currentbook : *bookList*) {

            try {

                FXMLLoader loader = new FXMLLoader(getClass().getResource("/client/views/BookCard.fxml"));

                VBox card = loader.load();

                BookCardController controller = loader.getController();

                if (overdueBookIds.contains(currentbook.getId())) {

                    controller.setTitleRed();

                }

                controller.setBook(currentbook);

                controller.setShowBorrowOption(*showBorrowCheckbox*);

                card.getStylesheets().add(getClass().getResource("/css/bookcard.css").toExternalForm());

                card.setUserData(controller);

                bookTilePane.getChildren().add(card);

            } catch (IOException e) {

                e.printStackTrace();

            }

        }

    }

   @FXML

    private *void* handleRecommend() {

        try {

            FXMLLoader loader = new FXMLLoader(getClass().getResource("/client/views/RecommandView.fxml"));

            Parent recommendPage = loader.load();

            RecommendController controller = loader.getController();

            controller.setCurrentUser(currentUser);

*javafx*.*stage*.Stage stage = new javafx.stage.Stage();

            stage.setTitle("Book Recommand");

            Scene scene = new Scene(recommendPage, 800, 600);

            scene.getStylesheets().add(getClass().getResource("/css/recommand.css").toExternalForm());

            stage.setScene(scene);

            stage.show();

        } catch (IOException e) {

            e.printStackTrace();

            showAlert("Fail load", "Fail to open:" + e.getMessage());

        }

    }

    @FXML

    private *void* handleReturn() {

        List<Integer> selectedBookIds = getSelectedBookIds();

        if (selectedBookIds == null || selectedBookIds.isEmpty()) {

            showAlert("No Selection", "Please select at least one book to return.");

            return;

        }

        for (Integer bookId : selectedBookIds) {

            Task<response> task = new Task<>() {

                @Override

                protected response call() throws Exception {

                    try (Socket socket = new Socket("localhost", 9090);

                        ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                        ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                        request req = new request("return");

                        req.putData("userId", currentUser.getId());

                        req.putData("bookId", bookId);

                        out.writeObject(req);

                        out.flush();

                        return (response) in.readObject();

                    }

                }

            };

            task.setOnSucceeded(*evt* *->* {

                response res = task.getValue();

                showAlert("Return Result", res.getMessage());

            });

            task.setOnFailed(*evt* *->* {

                showAlert("Return Failed", "Failed to return book ID: " + bookId);

            });

            new Thread(task).start();

        }

    }

    @FXML

    private *void* handleShowBorrowedBooks() {

        loadBorrowedBooks();

    }

    private *void* showAlert(String *title*, String *content*) {

        Platform.runLater(() *->* {

            Alert alert = new Alert(Alert.AlertType.INFORMATION);

            alert.setTitle(*title*);

            alert.setHeaderText(null);

            alert.setContentText(*content*);

            alert.showAndWait();

        });

    }

}

#### Admin Interface Module

The AdminController class serves as the comprehensive management hub for administrators within the library system's JavaFX client, orchestrating all core administrative operations through dynamic table interfaces bound to observable data collections (bookList, userList, borrowRecordList) that synchronize with backend server responses via asynchronous socket communications to localhost:9090. Upon initialization, it configures JavaFX TableView components with cell value factories to render book properties (ID/title/author/category/availability), user attributes (username/role/status), and detailed borrowing records (ID/username/title/borrow\_date/due\_date/returned status), while implementing visual cues like automatic row highlighting for overdue returns through custom TableRow rendering logic. The controller executes diverse administrative workflows including full inventory management with handleAddBook(), handleUpdateBook(), and handleDeleteBook() methods that construct corresponding request objects; user administration via handleToggleFreeze() for account control; comprehensive search operations across books, users, and borrow records with handleSearchBook(), handleSearchUser(), and handleSearchBorrowRecord(); and statistical oversight through handleRefresh() which concurrently retrieves real-time data via get\_all\_books, get\_users, and get\_borrow\_stats server requests.

All network operations follow a non-blocking pattern using JavaFX Task threads that establish socket connections with 5-second timeouts, serialize operation-specific request payloads, deserialize server response objects, and execute thread-safe UI updates via Platform.runLater(), maintaining interface responsiveness during data transfers. The controller implements unified error handling through showAlert() dialogs that surface server messages or network exceptions, while its stateless design relies on on-demand server data refreshes rather than local caching, ensuring administrative views consistently reflect current system state across inventory levels, user statuses, and borrowing activities within the library ecosystem.

package client.controllers;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.util.Date;

import java.util.List;

import java.util.Map;

import javafx.application.Platform;

import javafx.beans.property.SimpleIntegerProperty;

import javafx.beans.property.SimpleStringProperty;

import javafx.collections.FXCollections;

import javafx.collections.ObservableList;

import javafx.concurrent.Task;

import javafx.fxml.FXML;

import javafx.scene.control.Alert;

import javafx.scene.control.Button;

import javafx.scene.control.ListView;

import javafx.scene.control.TableColumn;

import javafx.scene.control.TableRow;

import javafx.scene.control.TableView;

import javafx.scene.control.TextArea;

import javafx.scene.control.TextField;

import javafx.scene.control.cell.PropertyValueFactory;

import models.User;

import models.book;

import network.request;

import network.response;

public class AdminController {

    private final ObservableList<book> bookList = FXCollections.observableArrayList();

    private final ObservableList<User> userList = FXCollections.observableArrayList();

    @FXML private TableView<book> bookTable;

    @FXML private TableColumn<book, String> colTitle, colAuthor, colCategory;

    @FXML private TableColumn<book, Integer> colId, colAvailable;

    @FXML private TextField titleField, authorField, categoryField, copiesField;

    @FXML private TableView<User> userTable;

    @FXML private TableColumn<User, String> colUsername, colRole;

    @FXML private TableColumn<User, Boolean> colFrozen;

    @FXML private TextField searchField, searchField1;

    @FXML private TextField bookIdField;

    @FXML private TextField updateTitleField, updateAuthorField, updateCategoryField;

    @FXML private TextField updateTotalField, updateAvailableField;

    @FXML private TextField borrowSearchField;

    @FXML private Button deleteButton, updateButton;

    @FXML private ListView<String> borrowStatsList;

    @FXML private TextArea summaryStatsArea;

    @FXML private TableView<Map<String, Object>> borrowRecordTable;

    @FXML private TableColumn<Map<String, Object>, Integer> colBorrowId;

    @FXML private TableColumn<Map<String, Object>, String> colBorrowUsername;

    @FXML private TableColumn<Map<String, Object>, String> colBorrowTitle;

    @FXML private TableColumn<Map<String, Object>, String> colBorrowDate;

    @FXML private TableColumn<Map<String, Object>, String> colDueDate;

    @FXML private TableColumn<Map<String, Object>, String> colReturned;

    private final ObservableList<Map<String, Object>> borrowRecordList = FXCollections.observableArrayList();

    @FXML

    public *void* initialize() {

        colId.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleIntegerProperty(*data*.getValue().getId()).asObject());

        colTitle.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleStringProperty(*data*.getValue().getTitle()));

        colAuthor.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleStringProperty(*data*.getValue().getAuthor()));

        colCategory.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleStringProperty(*data*.getValue().getCategory()));

        colAvailable.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleIntegerProperty(*data*.getValue().getAvailable\_copies()).asObject());

        colUsername.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleStringProperty(*data*.getValue().getUsername()));

        colRole.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleStringProperty(*data*.getValue().getRole()));

        colFrozen.setCellValueFactory(*data* *->*

            new javafx.beans.property.SimpleBooleanProperty(*data*.getValue().isFrozen()).asObject());

        colBorrowId.setCellValueFactory(*data* *->* new SimpleIntegerProperty((Integer)*data*.getValue().get("id")).asObject());

        colBorrowUsername.setCellValueFactory(*data* *->* new SimpleStringProperty((String)*data*.getValue().get("username")));

        colBorrowTitle.setCellValueFactory(*data* *->* new SimpleStringProperty((String)*data*.getValue().get("title")));

        colBorrowDate.setCellValueFactory(*data* *->* new SimpleStringProperty(String.valueOf(*data*.getValue().get("borrow\_date"))));

        colDueDate.setCellValueFactory(*data* *->* new SimpleStringProperty(String.valueOf(*data*.getValue().get("due\_date"))));

        colReturned.setCellValueFactory(*data* *->* new SimpleStringProperty((Boolean.TRUE.equals(*data*.getValue().get("returned")) ? "Yes" : "No")));

        borrowRecordTable.setItems(borrowRecordList);

        colUsername.setCellValueFactory(new PropertyValueFactory<>("username"));

        colRole.setCellValueFactory(new PropertyValueFactory<>("role"));

        colFrozen.setCellValueFactory(new PropertyValueFactory<>("frozen"));

        bookTable.setItems(bookList);

        userTable.setItems(userList);

        borrowRecordTable.setRowFactory(*tv* *->* new TableRow<Map<String, Object>>() {

            @Override

            protected *void* updateItem(Map<String, Object> *item*, *boolean* *empty*) {

                super.updateItem(*item*, *empty*);

                if (*item* == null || *empty*) {

                    setStyle("");

                } else {

*boolean* returned = Boolean.TRUE.equals(*item*.get("returned"));

                    Object dueObj = *item*.get("due\_date");

                    Date due = null;

                    if (dueObj instanceof Date) {

                        due = (Date) dueObj;

                    } else if (dueObj instanceof java.sql.Timestamp) {

                        due = new Date(((java.sql.Timestamp) dueObj).getTime());

                    }

*boolean* overdue = !returned && due != null && due.before(new Date());

                    if (overdue) {

                        setStyle("-fx-background-color: #ffcccc;");

                    } else {

                        setStyle("");

                    }

                }

            }

        });

        handleRefresh();

    }

    private User currentUser;

    public *void* setCurrentUser(User *user*) {

        this.currentUser = *user*;

        System.out.println("Admin user set: " + *user*.getUsername());

    }

    @FXML

    private *void* handleRefresh() {

        loadBooks();

        loadUsers();

        loadStats();

    }

    private *void* loadBooks() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000); // 5秒超时

                    request req = new request("get\_all\_books");

                    req.putData("keyword", "");

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                List<book> books = (List<book>) res.getData();

                bookList.setAll(books);

            } else {

                showAlert("Failed to load books: " + res.getMessage());

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Error loading books: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    private *void* loadUsers() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000); // 5秒超时

                    request req = new request("get\_users");

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                List<User> users = (List<User>) res.getData();

                System.out.println("成功获取用户数: " + users.size());

                for (User u : users) {

                    System.out.println(u.getUsername() + " | " + u.getRole());

                }

                userList.setAll(users);

            } else {

                showAlert("Failed to load users: " + res.getMessage());

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Error loading users: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    private *void* loadStats() {

        Task<response> task = new Task<>() {

        @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000);

                    request req = new request("get\_borrow\_stats");

                    out.writeObject(req);

                    out.flush();

                    response statsRes = (response) in.readObject();

                    request req2 = new request("get\_all\_borrow\_records");

                    out.writeObject(req2);

                    out.flush();

                    response recordsRes = (response) in.readObject();

                    Map<String, Object> result = new *java*.*util*.HashMap<>();

                    result.put("stats", statsRes);

                    result.put("records", recordsRes);

                    return new response(true, result, "All stats loaded");

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                Map<String, Object> result = (Map<String, Object>) res.getData();

                response statsRes = (response) result.get("stats");

                response recordsRes = (response) result.get("records");

                if (statsRes != null && statsRes.isSuccess()) {

*java*.*util*.List<String> stats = (java.util.List<String>) statsRes.getData();

                    summaryStatsArea.setText(String.join("\n", stats));

                }

                if (recordsRes != null && recordsRes.isSuccess()) {

*java*.*util*.List<Map<String, Object>> records = (java.util.List<Map<String, Object>>) recordsRes.getData();

                    borrowRecordList.setAll(records);

                }

            } else {

                showAlert("Failed to load stats: " + res.getMessage());

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Error loading stats: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleLoadAllBooks() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000); // 5秒超时

                    request req = new request("get\_all\_books");

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            showAlert(res.getMessage());

            if (res.isSuccess()) {

                List<book> books = (List<book>) res.getData();

                bookTable.getItems().setAll(books);

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Failed to load books: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleAddBook() {

        String title = titleField.getText();

        String author = authorField.getText();

        String category = categoryField.getText();

*int* copies;

        try {

            copies = Integer.parseInt(copiesField.getText());

        } catch (NumberFormatException e) {

            showAlert("Invalid copies number.");

            return;

        }

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000); // 5秒超时

                    request req = new request("add\_book");

                    req.putData("title", title);

                    req.putData("author", author);

                    req.putData("category", category);

                    req.putData("copies", copies);

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            showAlert(res.getMessage());

            if (res.isSuccess()) {

                handleRefresh();

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Failed to add book: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleToggleFreeze() {

        User selected = userTable.getSelectionModel().getSelectedItem();

        if (selected == null) {

            showAlert("Please select a user first.");

            return;

        }

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    socket.setSoTimeout(5000); // 5秒超时

                    request req = new request("toggle\_freeze");

                    req.putData("userId", selected.getId());

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            showAlert(res.getMessage());

            if (res.isSuccess()) {

                handleRefresh();

            }

        });

        task.setOnFailed(*e* *->* {

            Throwable ex = task.getException();

            showAlert("Failed to toggle freeze: " + (ex != null ? ex.getMessage() : "unknown error"));

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleSearchBook() {

        String keyword = searchField.getText().trim();

        if (keyword.isEmpty()) return;

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("search\_books");

                    req.putData("keyword", keyword);

                    oos.writeObject(req);

                    return (response) ois.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            List<book> books = (List<book>) task.getValue().getData();

            bookTable.getItems().setAll(books);

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleSearchUser() {

        String keyword = searchField1.getText().trim();

        if (keyword.isEmpty()) return;

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("search\_user");

                    req.putData("keyword", keyword);

                    oos.writeObject(req);

                    return (response) ois.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            List<User> users = (List<User>) task.getValue().getData();

            userTable.getItems().setAll(users);

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleDeleteBook() {

*int* id = Integer.parseInt(bookIdField.getText().trim());

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("delete\_book");

                    req.putData("bookId", id);

                    oos.writeObject(req);

                    return (response) ois.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            if ((Boolean) task.getValue().getData()) handleRefresh();

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleUpdateBook() {

*int* id = Integer.parseInt(bookIdField.getText().trim());

        String title = updateTitleField.getText();

        String author = updateAuthorField.getText();

        String category = updateCategoryField.getText();

*int* total = Integer.parseInt(updateTotalField.getText());

*int* available = Integer.parseInt(updateAvailableField.getText());

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream ois = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("update\_book");

                    req.putData("id", id);

                    req.putData("title", title);

                    req.putData("author", author);

                    req.putData("category", category);

                    req.putData("total", total);

                    req.putData("available", available);

                    oos.writeObject(req);

                    return (response) ois.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            if ((Boolean) task.getValue().getData()) handleRefresh();

        });

        new Thread(task).start();

    }

    @FXML

    private *void* handleSearchBorrowRecord() {

        String keyword = borrowSearchField.getText().trim();

        if (keyword.isEmpty()) return;

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("search\_borrow\_records");

                    req.putData("keyword", keyword);

                    out.writeObject(req);

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*e* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                List<String> records = (List<String>) res.getData();

                borrowStatsList.getItems().setAll(records);

            } else {

                showAlert("Search failed: " + res.getMessage());

            }

        });

        new Thread(task).start();

    }

    private *void* showAlert(String *msg*) {

        Platform.runLater(() *->* {

            Alert alert = new Alert(Alert.AlertType.INFORMATION);

            alert.setHeaderText(null);

            alert.setContentText(*msg*);

            alert.showAndWait();

        });

    }

}

#### Bookcard view

The BookCardController class is a specialized JavaFX controller component for rendering individual book display cards within the library management system's user interface, dynamically presenting book metadata through FXML-bound UI elements including an ImageView for cover art display, Label elements for title, author, and brief introduction text, and a conditional CheckBox for borrow/return selection operations. It manages book-specific data binding via the setBook() method which populates visual elements by extracting properties from book model objects—loading cover images from img\_path filesystem paths and configuring text labels for title, author, and introduction—while providing interactive functionality through the conditionally visible borrowCheckBox controlled by setShowBorrowOption() to enable/disable borrowing selections based on context (e.g., hidden when viewing recommendations). Additional visual signaling is implemented through setTitleRed() which applies red bold styling to highlight critical states like overdue books, and maintains internal state management with currentBook reference storage alongside selection status validation via isSelected(), serving as a reusable presentation component across book lists, search results, and borrowing interfaces within the client application.

package client.controllers;

import javafx.fxml.FXML;

import javafx.scene.control.CheckBox;

import javafx.scene.control.Label;

import javafx.scene.image.Image;

import javafx.scene.image.ImageView;

import models.book;

public class BookCardController {

    @FXML private ImageView cover;

    @FXML private Label title;

    @FXML private Label author;

    @FXML private Label intro;

    @FXML private CheckBox borrowCheckBox;

    private book currentBook;

    private *boolean* showBorrowOption = true;

    public *void* setBook(book *currentBook*) {

        this.currentBook = *currentBook*;

        title.setText(*currentBook*.getTitle());

        author.setText("By " + *currentBook*.getAuthor());

        intro.setText(*currentBook*.getBrief\_introduction());

        Image image = new Image("file:" + *currentBook*.getImg\_path(), true);

        cover.setImage(image);

    }

    public *boolean* isSelected() {

        return borrowCheckBox.isSelected();

    }

    public book getBook() {

        return this.currentBook;

    }

    public CheckBox getBorrowCheckBox() {

        return borrowCheckBox;

    }

    public *void* setTitleRed() {

        title.setStyle("-fx-text-fill: red; -fx-font-weight: bold;");

    }

    public *void* setShowBorrowOption(*boolean* *show*) {

        this.showBorrowOption = *show*;

        borrowCheckBox.setVisible(*show*);

        if (!*show*) {

            borrowCheckBox.setSelected(false);

        }

    }

}

#### Recommendation Algorithm Module

The recommender class implements a content-based book recommendation system for the library management application, utilizing cosine similarity metric analysis to generate personalized suggestions based on user borrowing history and book metadata characteristics. Its core functionality resides in the contentBasedRecommend() method which constructs a user preference profile as a normalized average vector (userProfile) by aggregating TF-IDF feature vectors (bookFeatures) from historically borrowed books (userHistory), then calculates similarity scores against all non-borrowed books in the catalog via cosineSimilarity() computations that measure angular similarity between vectors in high-dimensional space. The recommendation process involves filtering out previously borrowed titles, ranking candidate books by descending cosine similarity values (higher values indicating stronger content alignment with established preferences), and selecting the top 10 highest-scoring entries through Java Streams operations that sort the scoreMap entries and limit results. This algorithm effectively transforms textual metadata (categories, authors, descriptions) into quantitative preference models, enabling personalized discovery of novel titles that match established user interests within the library collection.

package recommender;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

public class recommender {

    public static *double* cosineSimilarity(*double*[] *vecA*, *double*[] *vecB*) {

*double* dot = 0.0, normA = 0.0, normB = 0.0;

        for (*int* i = 0; i < *vecA*.length; i++) {

            dot += *vecA*[i] \* *vecB*[i];

            normA += *vecA*[i] \* *vecA*[i];

            normB += *vecB*[i] \* *vecB*[i];

        }

        return dot / (Math.sqrt(normA) \* Math.sqrt(normB) + 1e-10);

    }

    public List<Integer> contentBasedRecommend(*int* *userId*, Map<Integer, *double*[]> *bookFeatures*, List<Integer> *userHistory*) {

        List<Integer> recommendations = new ArrayList<>();

        if (*userHistory*.isEmpty()) return recommendations;

        // 计算用户历史图书的平均特征向量（表示用户偏好）

*int* dim = *bookFeatures*.values().iterator().next().length;

*double*[] userProfile = new *double*[dim];

        for (*int* id : *userHistory*) {

*double*[] vec = *bookFeatures*.get(id);

            if (vec != null) {

                for (*int* i = 0; i < dim; i++) {

                    userProfile[i] += vec[i];

                }

            }

        }

        for (*int* i = 0; i < dim; i++) {

            userProfile[i] /= *userHistory*.size();

        }

        // 计算所有图书与用户偏好的相似度

        Map<Integer, Double> scoreMap = new HashMap<>();

        for (Map.Entry<Integer, *double*[]> entry : *bookFeatures*.entrySet()) {

*int* id = entry.getKey();

            if (*userHistory*.contains(id)) continue; // 不推荐已看过的书

*double* sim = cosineSimilarity(userProfile, entry.getValue());

            scoreMap.put(id, sim);

        }

        // 返回相似度最高的前 N 个图书ID

        scoreMap.entrySet().stream()

            .sorted(Map.Entry.<Integer, Double>comparingByValue().reversed())

            .limit(10)

            .forEach(*e* *->* recommendations.add(*e*.getKey()));

        return recommendations;

    }

}

The TFIDFProcessor class implements a comprehensive text feature extraction pipeline for the library recommendation system, transforming raw book metadata into numerical TF-IDF vectors that quantify term significance across the document corpus by first tokenizing each book's combined textual data (category, author, and introduction) into normalized tokens through lowercase conversion, punctuation removal, and whitespace splitting while simultaneously constructing a global document frequency (DF) map that tracks token occurrence across all books, then processing each book's term frequencies (TF) by counting token occurrences normalized against total token count, and finally calculating term-specific inverse document frequency (IDF) weights as logarithmic ratios of total documents to document frequency, multiplying TF and IDF values to produce discriminative feature vectors that emphasize distinctive rare terms over common ones which are stored within each book's tfidfMap attribute for subsequent similarity analysis and content-based recommendation generation.

package recommender;

import java.util.\*;

public class TFIDFProcessor {

    // 生成所有图书的 TF-IDF 特征并设置回 book 对象

    public static *void* computeTFIDF(List<*models*.book> *books*) {

        List<String[]> tokenizedDocs = new ArrayList<>();

        Map<String, Integer> dfMap = new HashMap<>();

        // 分词 + 统计 document frequency

        for (models.book b : *books*) {

            String[] tokens = tokenize(b.getCombinedText());

            tokenizedDocs.add(tokens);

            Set<String> seen = new HashSet<>();

            for (String token : tokens) {

                if (seen.add(token)) {

                    dfMap.put(token, dfMap.getOrDefault(token, 0) + 1);

                }

            }

        }

*int* totalDocs = *books*.size();

        // 为每本书计算 tf-idf 向量

        for (*int* i = 0; i < *books*.size(); i++) {

            models.book b = *books*.get(i);

            String[] tokens = tokenizedDocs.get(i);

            Map<String, Double> tfMap = new HashMap<>();

            // tf

            for (String token : tokens) {

                tfMap.put(token, tfMap.getOrDefault(token, 0.0) + 1.0);

            }

            for (String token : tfMap.keySet()) {

                tfMap.put(token, tfMap.get(token) / tokens.length);

            }

            // tf-idf

            Map<String, Double> tfidf = new HashMap<>();

            for (String token : tfMap.keySet()) {

*int* df = dfMap.getOrDefault(token, 1);

*double* idf = Math.log((*double*) totalDocs / df);

                tfidf.put(token, tfMap.get(token) \* idf);

            }

            b.setTfidfMap(tfidf); // 设置回 book

        }

    }

    // 简易分词器（按空格、标点分割）

    private static String[] tokenize(String *text*) {

        return *text*.toLowerCase().replaceAll("[^a-zA-Z0-9 ]", " ").split("\\s+");

    }

}

The Vectorizer class serves as a feature engineering utility in the library recommendation system that transforms categorical and textual book attributes into unified numerical feature vectors for machine learning applications, initializing with catalog-wide distinct category lists, author lists, and vocabulary mappings to establish consistent vectorization indices. It implements domain-specific encoding methods including one-hot encoding for book categories through vectorizeCategory(), author-based encoding via vectorizeAuthor(), and description vectorization using TF-IDF weights from vectorizeDescription() that maps term importance scores from precomputed tfidfMap objects to corresponding vocabulary indices. The class culminates with mergeVectors() which applies configurable weighting (category weight=1.0, author weight=0.8, description weight=1.5) to each feature component before concatenating them into a single high-dimensional vector representation, enabling unified mathematical comparison of books based on metadata characteristics for downstream similarity analysis.

package recommender;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

public class Vectorizer {

    List<String> allCategories;

    List<String> allAuthors;

    Map<String, Integer> wordToIndex;

    public Vectorizer(List<String> *categories*, List<String> *authors*, Set<String> *allWords*) {

        this.allCategories = *categories*;

        this.allAuthors = *authors*;

        this.wordToIndex = new HashMap<>();

*int* idx = 0;

        for (String word : *allWords*) {

            wordToIndex.put(word, idx++);

        }

    }

    public *double*[] vectorizeCategory(String *category*) {

*double*[] vec = new *double*[allCategories.size()];

        for (*int* i = 0; i < allCategories.size(); i++) {

            vec[i] = allCategories.get(i).equals(*category*) ? 1.0 : 0.0;

        }

        return vec;

    }

    public *double*[] vectorizeAuthor(String *author*) {

*double*[] vec = new *double*[allAuthors.size()];

        for (*int* i = 0; i < allAuthors.size(); i++) {

            vec[i] = allAuthors.get(i).equals(*author*) ? 1.0 : 0.0;

        }

        return vec;

    }

    public *double*[] vectorizeDescription(Map<String, Double> *tfidfMap*) {

*double*[] vec = new *double*[wordToIndex.size()];

        for (Map.Entry<String, Double> entry : *tfidfMap*.entrySet()) {

            String word = entry.getKey();

            if (wordToIndex.containsKey(word)) {

                vec[wordToIndex.get(word)] = entry.getValue();

            }

        }

        return vec;

    }

    public *double*[] mergeVectors(*double*[] *catVec*, *double*[] *authorVec*, *double*[] *descVec*) {

*double* w1 = 1.0, w2 = 0.8, w3 = 1.5; // 可调节权重

        for (*int* i = 0; i < *catVec*.length; i++) *catVec*[i] \*= w1;

        for (*int* i = 0; i < *authorVec*.length; i++) *authorVec*[i] \*= w2;

        for (*int* i = 0; i < *descVec*.length; i++) *descVec*[i] \*= w3;

*double*[] merged = new *double*[*catVec*.length + *authorVec*.length + *descVec*.length];

        System.arraycopy(*catVec*, 0, merged, 0, *catVec*.length);

        System.arraycopy(*authorVec*, 0, merged, *catVec*.length, *authorVec*.length);

        System.arraycopy(*descVec*, 0, merged, *catVec*.length + *authorVec*.length, *descVec*.length);

        return merged;

    }

}

#### Recommendation Module

package client.controllers;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.net.Socket;

import java.sql.Connection;

import java.sql.PreparedStatement;

import java.sql.ResultSet;

import java.sql.SQLException;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.HashSet;

import java.util.List;

import java.util.Map;

import java.util.Set;

import database.Book\_DAO;

import database.DbUtil;

import javafx.application.Platform;

import javafx.concurrent.Task;

import javafx.fxml.FXML;

import javafx.fxml.FXMLLoader;

import javafx.scene.control.Alert;

import javafx.scene.control.Button;

import javafx.scene.control.TextField;

import javafx.scene.layout.TilePane;

import javafx.scene.layout.VBox;

import models.User;

import models.book;

import network.request;

import network.response;

import recommender.TFIDFProcessor;

import recommender.Vectorizer;

import recommender.recommender;

public class RecommendController {

    @FXML private TilePane recommendTilePane;

    @FXML private Button borrowSelectedButton;

    @FXML private TextField searchField;

    private User currentUser;

    private Book\_DAO bookDao = new Book\_DAO();

    public *void* setCurrentUser(User *user*) {

        this.currentUser = *user*;

        Task<List<book>> localRecommendTask = createLocalRecommendTask();

        localRecommendTask.setOnSucceeded(*evt* *->* {

            List<book> localBooks = localRecommendTask.getValue();

            if (localBooks != null && !localBooks.isEmpty()) {

                loadBookCards(localBooks);

            } else {

                loadRecommendations();

            }

        });

        localRecommendTask.setOnFailed(*evt* *->* {

            loadRecommendations();

        });

        new Thread(localRecommendTask).start();

    }

    @FXML

    private *void* handleSearchRecommendedBooks() {

        String keyword = searchField.getText().trim();

        if (keyword.isEmpty()) return;

        Task<response> searchTask = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                    ObjectOutputStream outputStream = new ObjectOutputStream(socket.getOutputStream());

                    ObjectInputStream inputStream = new ObjectInputStream(socket.getInputStream())) {

                    request searchRequest = new request("search\_recommended\_books");

                    searchRequest.putData("userId", currentUser.getId());

                    searchRequest.putData("keyword", keyword);

                    outputStream.writeObject(searchRequest);

                    outputStream.flush();

                    return (response) inputStream.readObject();

                }

            }

        };

        searchTask.setOnSucceeded(*evt* *->* {

            response res = searchTask.getValue();

            if (res.isSuccess()) {

                List<book> bookList = (List<book>) res.getData();

                loadBookCards(bookList);

            } else {

                showAlert("Search Failed", res.getMessage());

            }

        });

        searchTask.setOnFailed(*evt* *->* {

            Throwable e = searchTask.getException();

            e.printStackTrace();

            showAlert("Error", "Search failed: " + e.getMessage());

        });

        new Thread(searchTask).start();

    }

    private *void* loadRecommendations() {

        Task<response> task = new Task<>() {

            @Override

            protected response call() throws Exception {

                try (Socket socket = new Socket("localhost", 9090);

                     ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                     ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                    request req = new request("get\_recommendations");

                    req.putData("userId", currentUser.getId());

                    out.writeObject(req);

                    out.flush();

                    return (response) in.readObject();

                }

            }

        };

        task.setOnSucceeded(*evt* *->* {

            response res = task.getValue();

            if (res.isSuccess()) {

                List<book> books = (List<book>) res.getData();

                loadBookCards(books);

            } else {

                showAlert("Fail gain recommand", res.getMessage());

            }

        });

        task.setOnFailed(*evt* *->* showAlert("Internet Error", "Fail gain recommand"));

        new Thread(task).start();

    }

    private Task<List<book>> createLocalRecommendTask() {

        return new Task<>() {

            @Override

            protected List<book> call() throws Exception {

                List<book> allBooks = getAllBooksInLibrary();

                TFIDFProcessor.computeTFIDF(allBooks);

                List<String> allCategories = getAllCategoriesLocal();

                List<String> allAuthors = getAllAuthorsLocal();

                Set<String> allWords = getAllWordsLocal();

                List<Integer> userHistory = getUserBorrowedBookIds(currentUser);

                Vectorizer vectorizer = new Vectorizer(allCategories, allAuthors, allWords);

                Map<Integer, *double*[]> bookFeatureMap = new HashMap<>();

                for (book b : allBooks) {

*double*[] catVec = vectorizer.vectorizeCategory(b.getCategory());

*double*[] authorVec = vectorizer.vectorizeAuthor(b.getAuthor());

*double*[] descVec = vectorizer.vectorizeDescription(b.getTfidfMap());

*double*[] feature = vectorizer.mergeVectors(catVec, authorVec, descVec);

                    bookFeatureMap.put(b.getId(), feature);

                }

                List<Integer> recommendIds = new recommender()

                    .contentBasedRecommend(currentUser.getId(), bookFeatureMap, userHistory);

                List<book> recommendBooks = new ArrayList<>();

                for (Integer id : recommendIds) {

                    book bookDetails = bookDao.getBookById(id);

                    if (bookDetails != null) {

                        recommendBooks.add(bookDetails);

                    }

                }

                return recommendBooks;

            }

        };

    }

    private List<book> getAllBooksInLibrary() {

        return bookDao.getAllBooks();

    }

    private List<String> getAllCategoriesLocal() {

        List<String> categories = new ArrayList<>();

        String sql = "SELECT DISTINCT category FROM books";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql);

             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {

                categories.add(rs.getString("category"));

            }

        } catch (SQLException e) {

            System.err.println("Error fetching categories: " + e.getMessage());

        }

        return categories;

    }

    private List<String> getAllAuthorsLocal() {

        List<String> authors = new ArrayList<>();

        String sql = "SELECT DISTINCT author FROM books";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql);

             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {

                authors.add(rs.getString("author"));

            }

        } catch (SQLException e) {

            System.err.println("Error fetching authors: " + e.getMessage());

        }

        return authors;

    }

    private Set<String> getAllWordsLocal() {

        Set<String> words = new HashSet<>();

        String sql = "SELECT title, author, category, brief\_introduction FROM books";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql);

             ResultSet rs = ps.executeQuery()) {

            while (rs.next()) {

                addWordsFromString(words, rs.getString("title"));

                addWordsFromString(words, rs.getString("author"));

                addWordsFromString(words, rs.getString("category"));

                addWordsFromString(words, rs.getString("brief\_introduction"));

            }

        } catch (SQLException e) {

            System.err.println("Error fetching words: " + e.getMessage());

        }

        return words;

    }

    // 辅助方法：从字符串提取单词

    private *void* addWordsFromString(Set<String> *words*, String *text*) {

        if (*text* == null || *text*.trim().isEmpty()) return;

        String processed = *text*

            .toLowerCase() // 转换为小写

            .replaceAll("[^a-zA-Z0-9\\s]", "") // 移除非字母数字字符

            .replaceAll("\\s+", " "); // 清理多余空格

        for (String word : processed.split(" ")) {

            if (!word.isEmpty()) {

*words*.add(word);

            }

        }

    }

    private List<Integer> getUserBorrowedBookIds(User *user*) {

        List<Integer> bookIds = new ArrayList<>();

        if (*user* == null) return bookIds;

        String sql = "SELECT DISTINCT book\_id FROM borrow\_records WHERE user\_id = ?";

        try (Connection conn = DbUtil.getConnection();

             PreparedStatement ps = conn.prepareStatement(sql)) {

            ps.setInt(1, *user*.getId());

            ResultSet rs = ps.executeQuery();

            while (rs.next()) {

                bookIds.add(rs.getInt("book\_id"));

            }

        } catch (SQLException e) {

            System.err.println("Error fetching borrowed book IDs: " + e.getMessage());

        }

        return bookIds;

    }

    @FXML

    private *void* handleBackToMain() {

        Platform.runLater(() *->* {

            borrowSelectedButton.getScene().getWindow().hide();

        });

    }

    private *void* loadBookCards(List<book> *books*) {

        recommendTilePane.getChildren().clear();

        for (book b : *books*) {

            try {

                FXMLLoader loader = new FXMLLoader(getClass().getResource("/client/views/BookCard.fxml"));

                VBox card = loader.load();

                BookCardController controller = loader.getController();

                controller.setBook(b);

                controller.setShowBorrowOption(true);

                card.getStylesheets().add(getClass().getResource("/css/bookcard.css").toExternalForm());

                card.setUserData(controller);

                recommendTilePane.getChildren().add(card);

            } catch (Exception e) {

                e.printStackTrace();

            }

        }

    }

    private List<Integer> getSelectedBookIds() {

        List<Integer> ids = new ArrayList<>();

        for (*var* node : recommendTilePane.getChildren()) {

            Object userData = node.getUserData();

            if (userData instanceof BookCardController) {

                BookCardController controller = (BookCardController) userData;

                if (controller.isSelected()) {

                    ids.add(controller.getBook().getId());

                }

            }

        }

        return ids;

    }

    @FXML

    private *void* handleBorrowSelected() {

        List<Integer> selectedIds = getSelectedBookIds();

        for (Integer id : selectedIds) {

            Task<response> task = new Task<>() {

                @Override

                protected response call() throws Exception {

                    try (Socket socket = new Socket("localhost", 9090);

                         ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

                         ObjectInputStream in = new ObjectInputStream(socket.getInputStream())) {

                        request req = new request("borrow");

                        req.putData("userId", currentUser.getId());

                        req.putData("bookId", id);

                        out.writeObject(req);

                        out.flush();

                        return (response) in.readObject();

                    }

                }

            };

            task.setOnSucceeded(*evt* *->* showAlert("Result", task.getValue().getMessage()));

            task.setOnFailed(*evt* *->* showAlert("Error", "Fail to borrow: " + id));

            new Thread(task).start();

        }

    }

    private *void* showAlert(String *title*, String *content*) {

        Platform.runLater(() *->* {

            Alert alert = new Alert(Alert.AlertType.INFORMATION);

            alert.setTitle(*title*);

            alert.setHeaderText(null);

            alert.setContentText(*content*);

            alert.showAndWait();

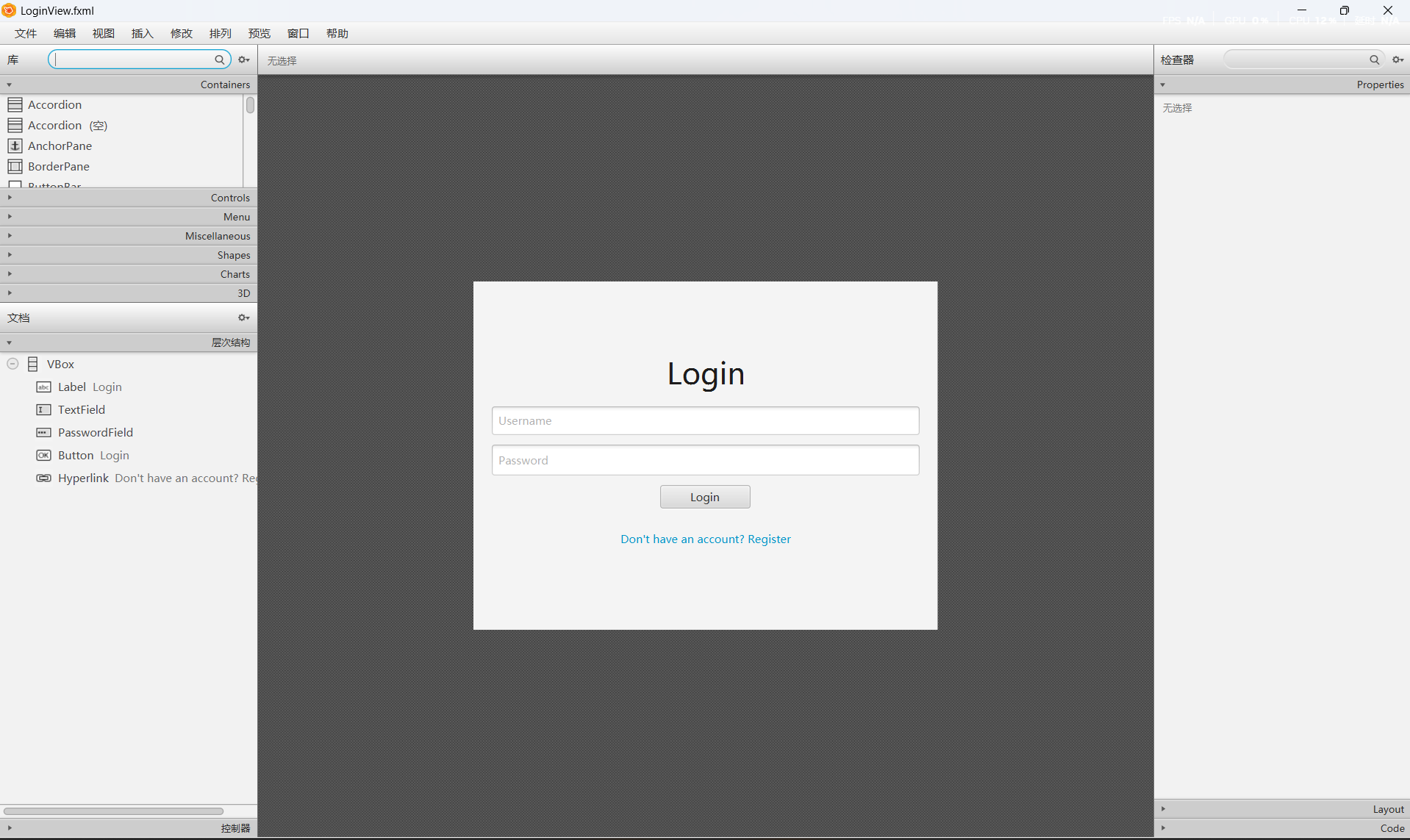
        });

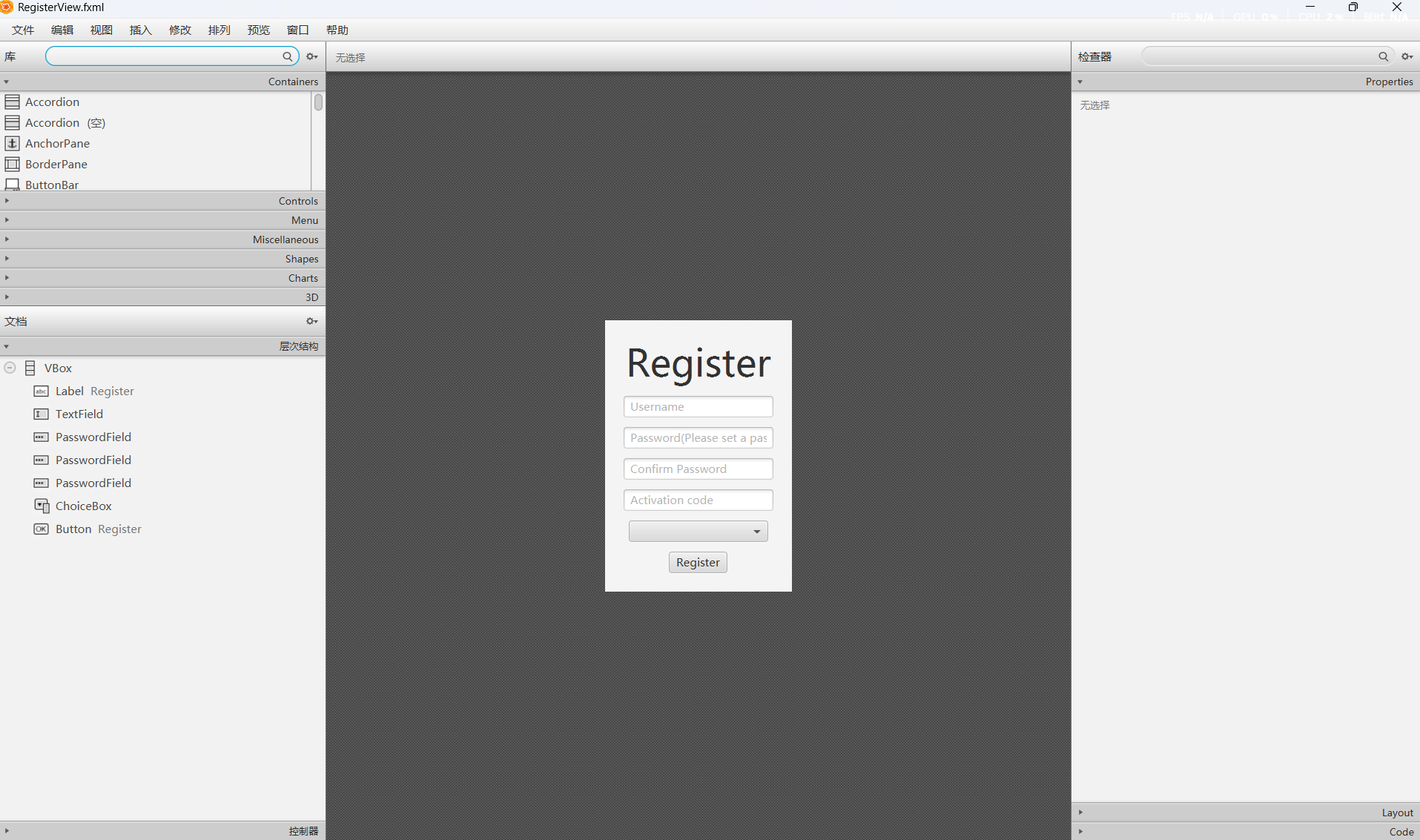
    }

}

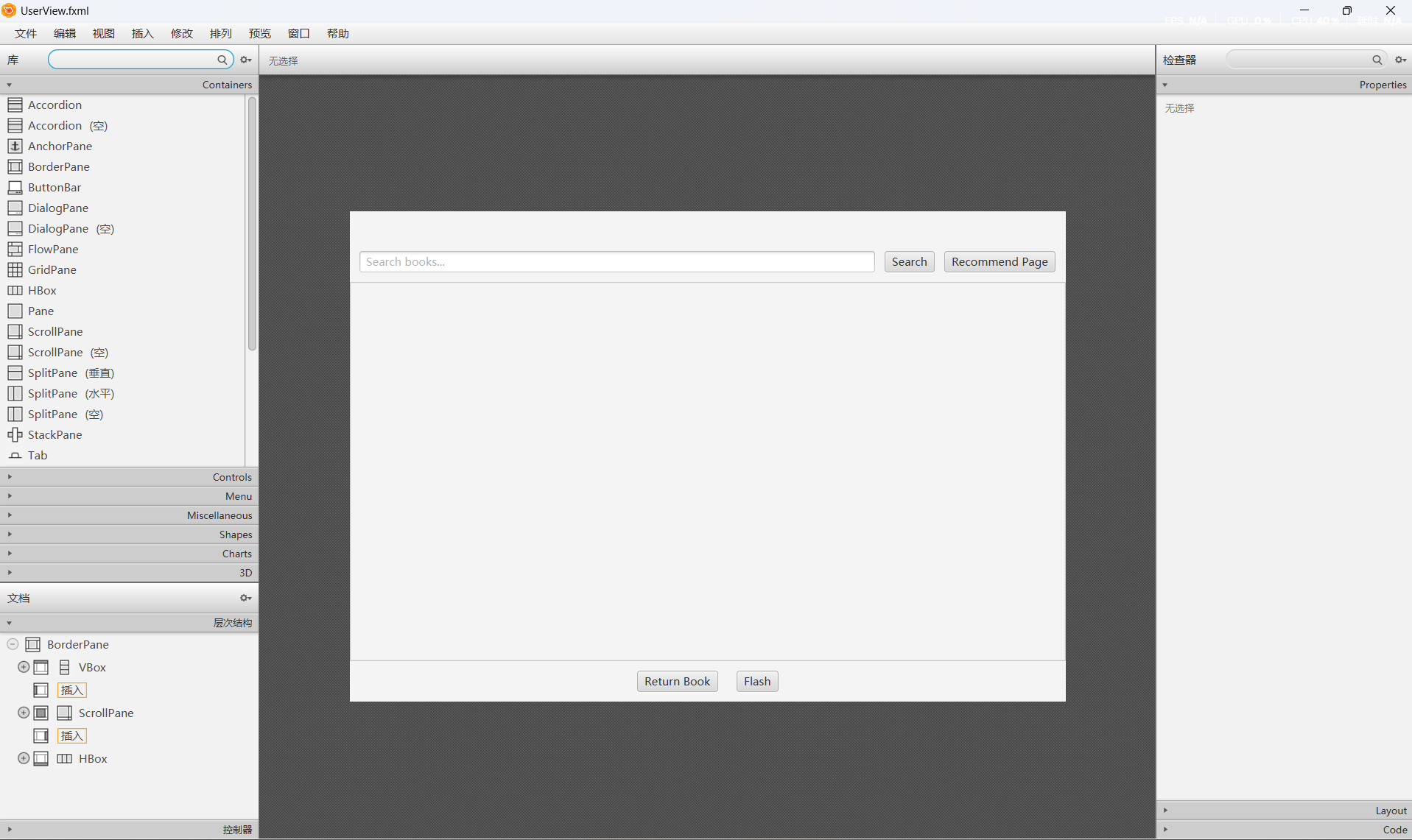
## GUI Design (JavaFX)

### Login and Registration FX

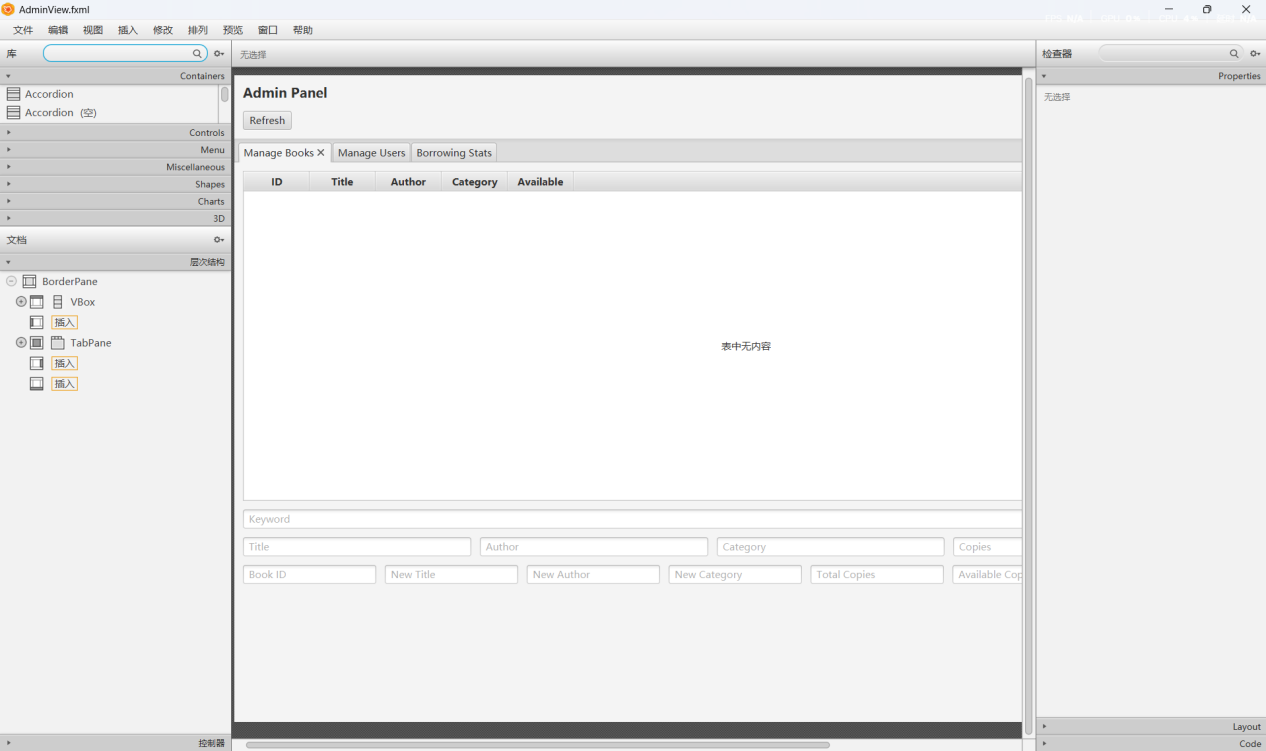




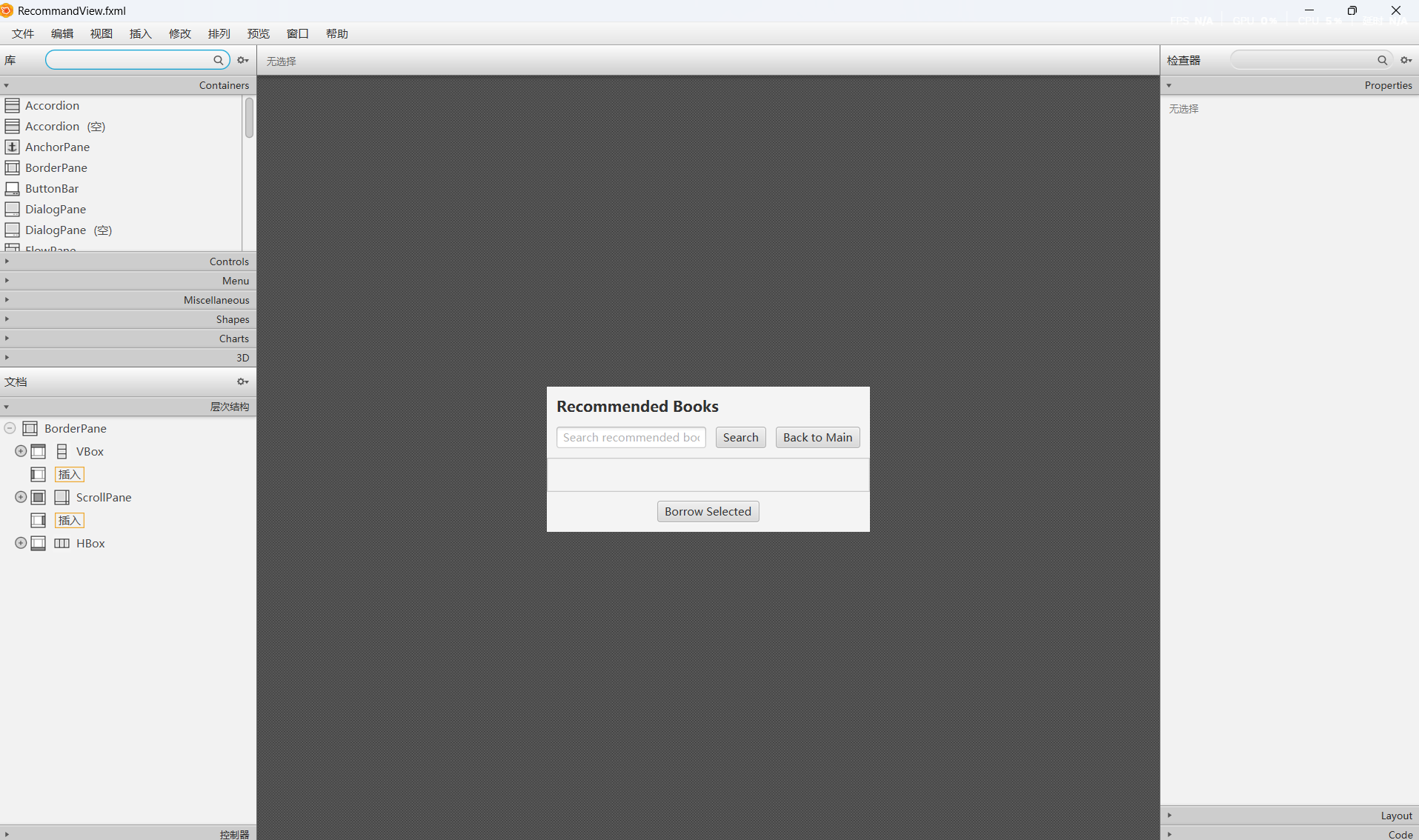
### User Interface



### Admin Interface



### Recommend Interface.



### Bookcard Panel and Css File

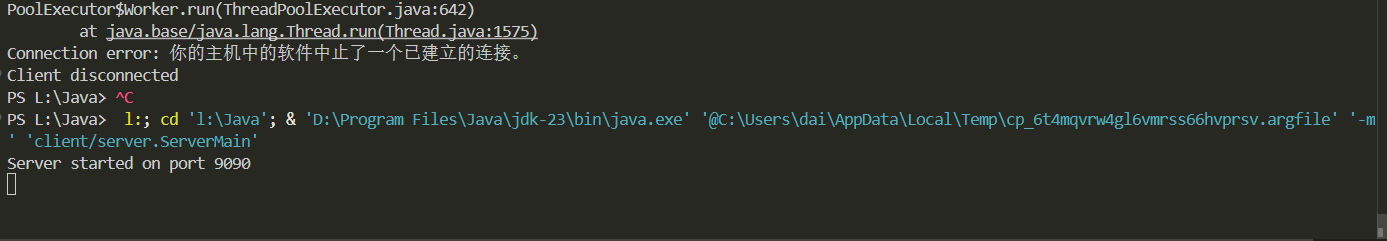


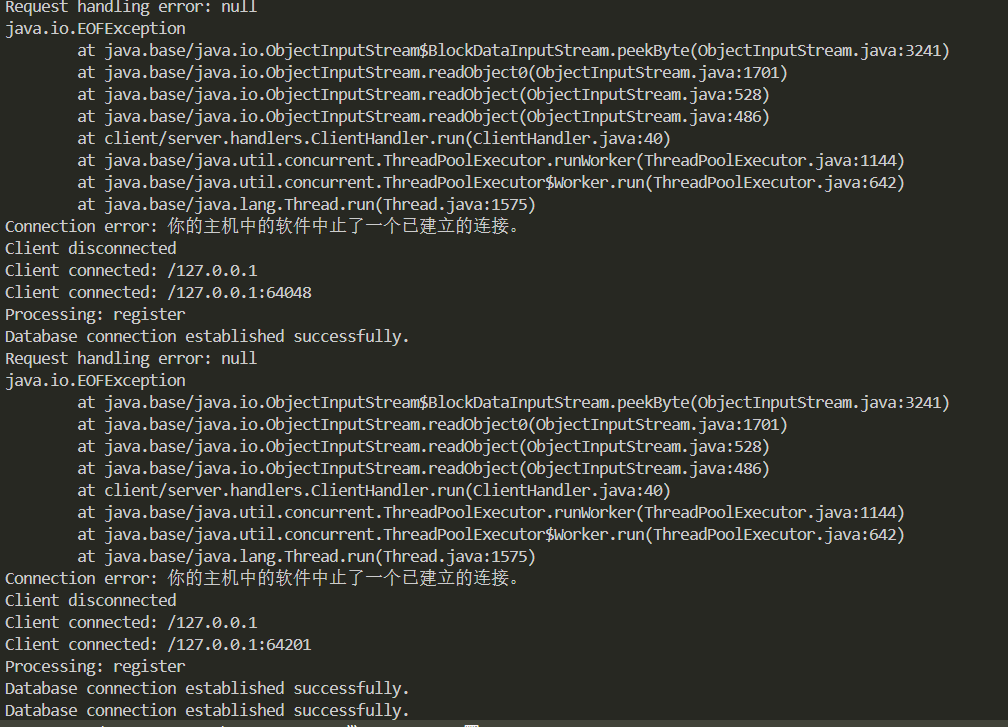


## Testing & Evaluation

### Server Test

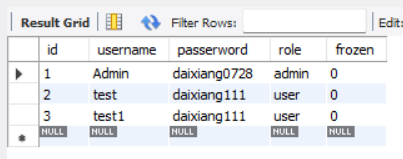
Here the port taken by the server is localhost 9090, and now the server has been successfully started and there will be many logs Feedback output statements.



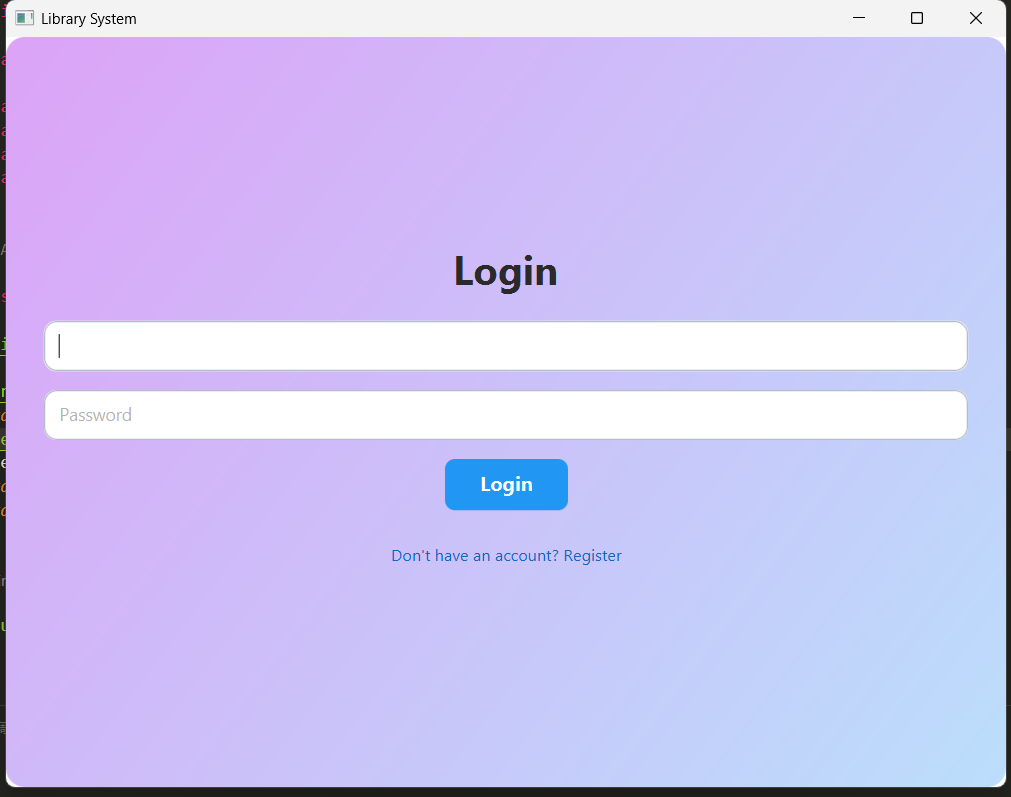


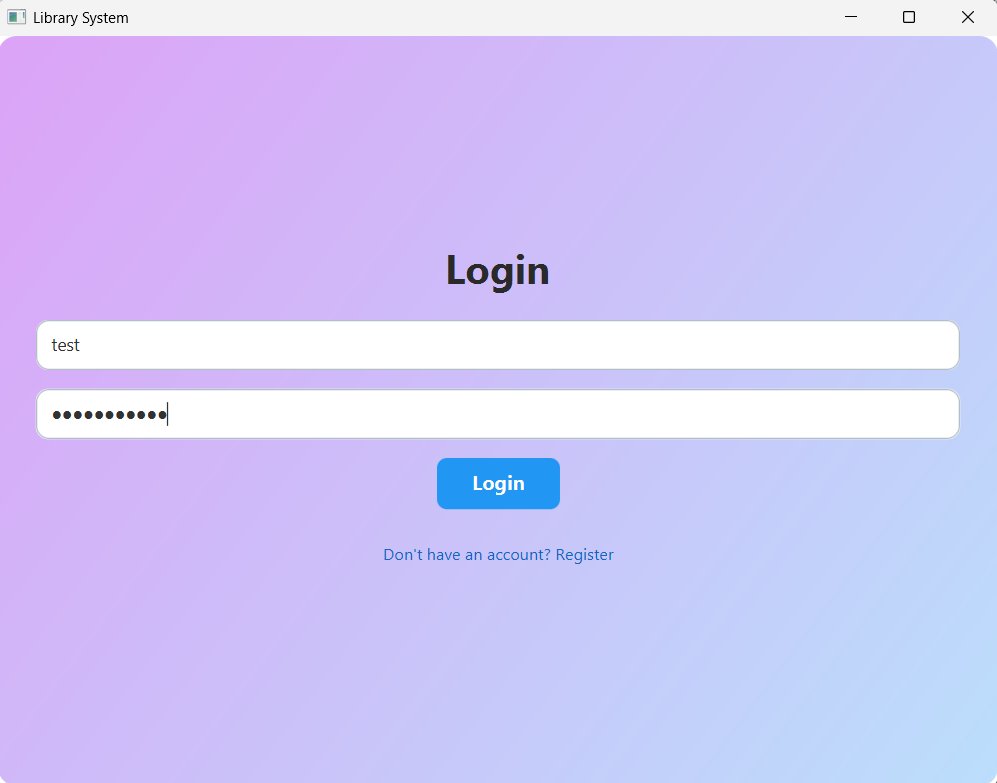
### Login Function

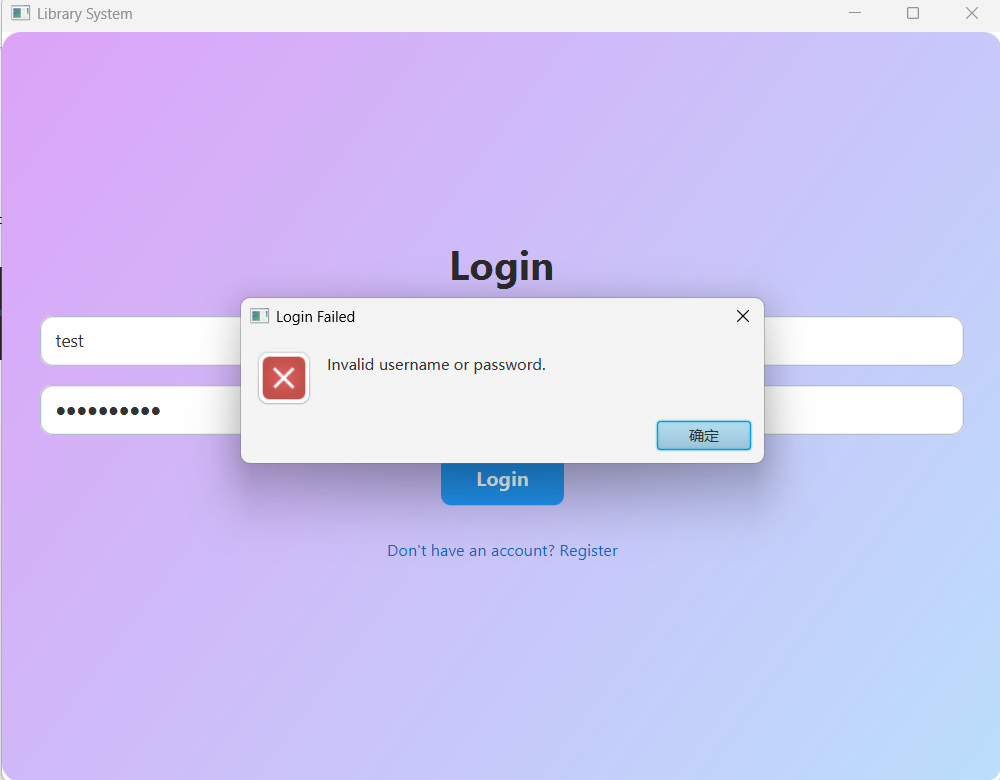
Shown here are tests stored in the database.

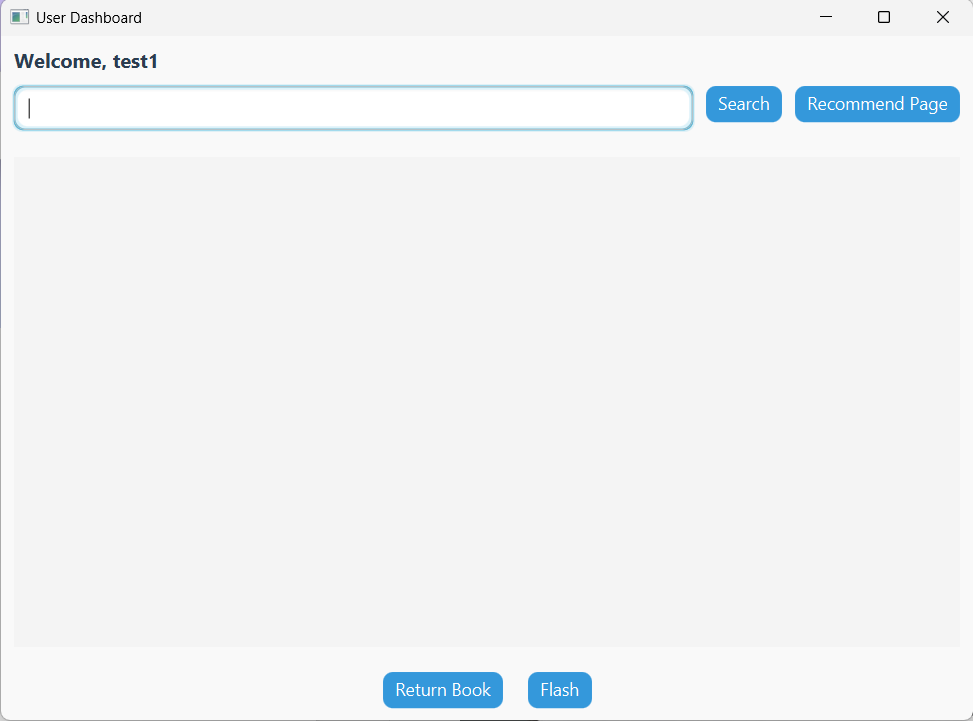


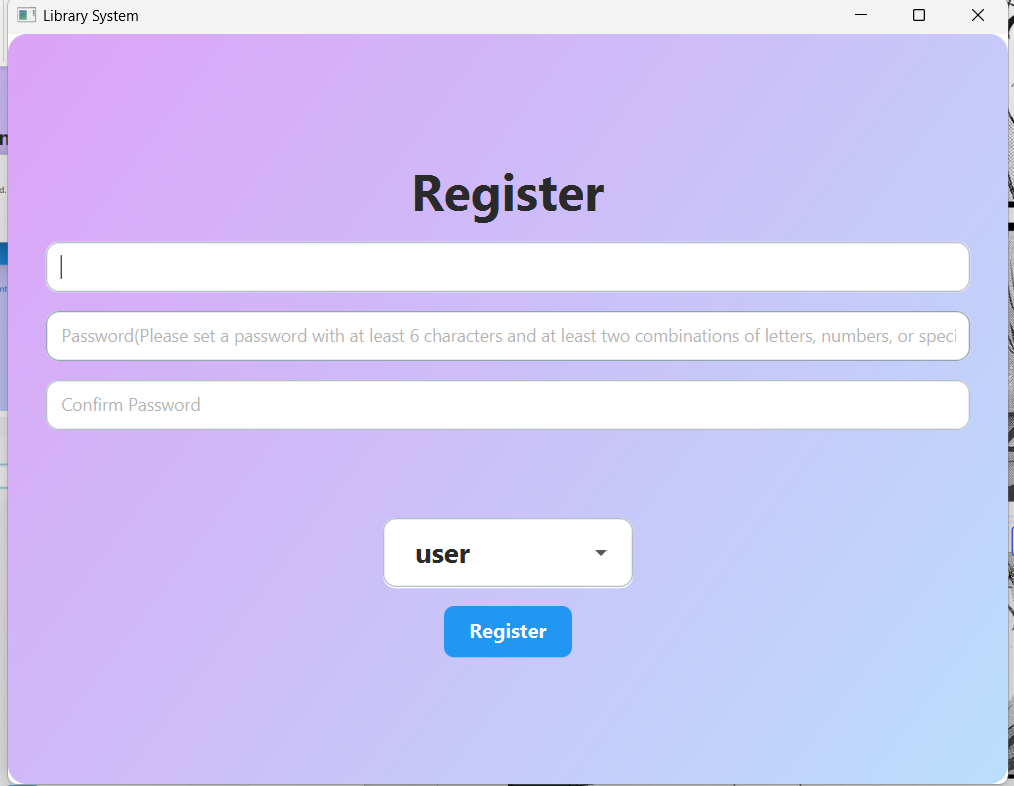
Here is an overview of the login interface, entering the password will have a circular mask, if you enter the wrong password, the aleart window will pop up; Otherwise, go to the page corresponding to the role of the account.Click on the link control below to enter the register screen.





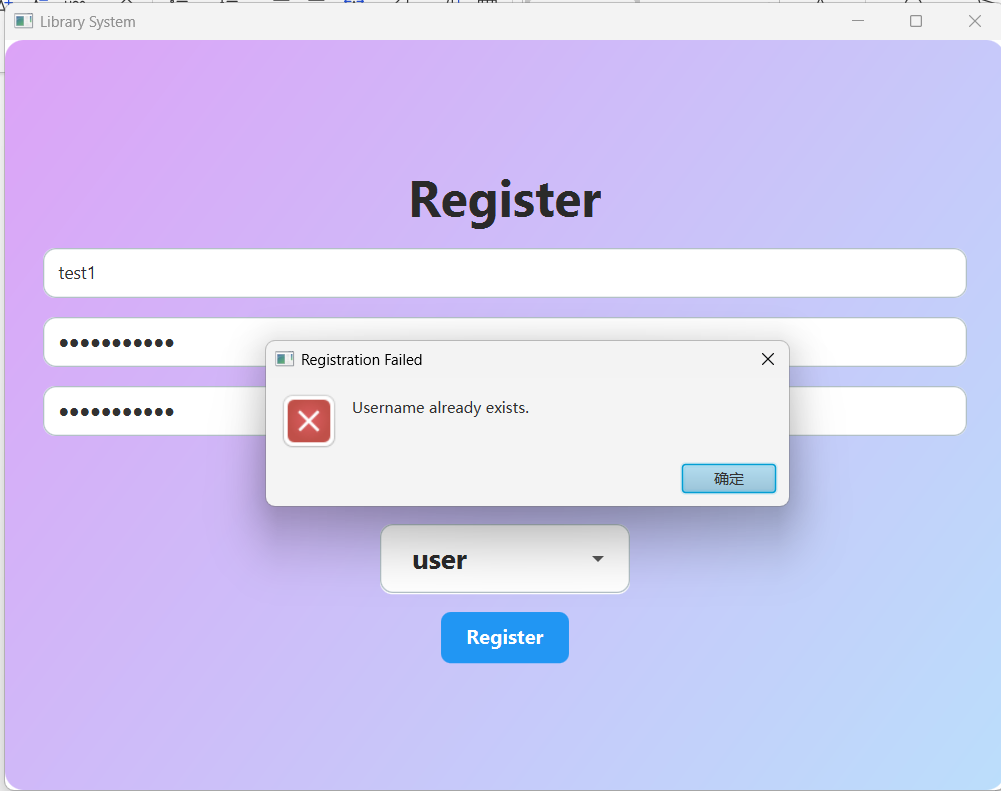


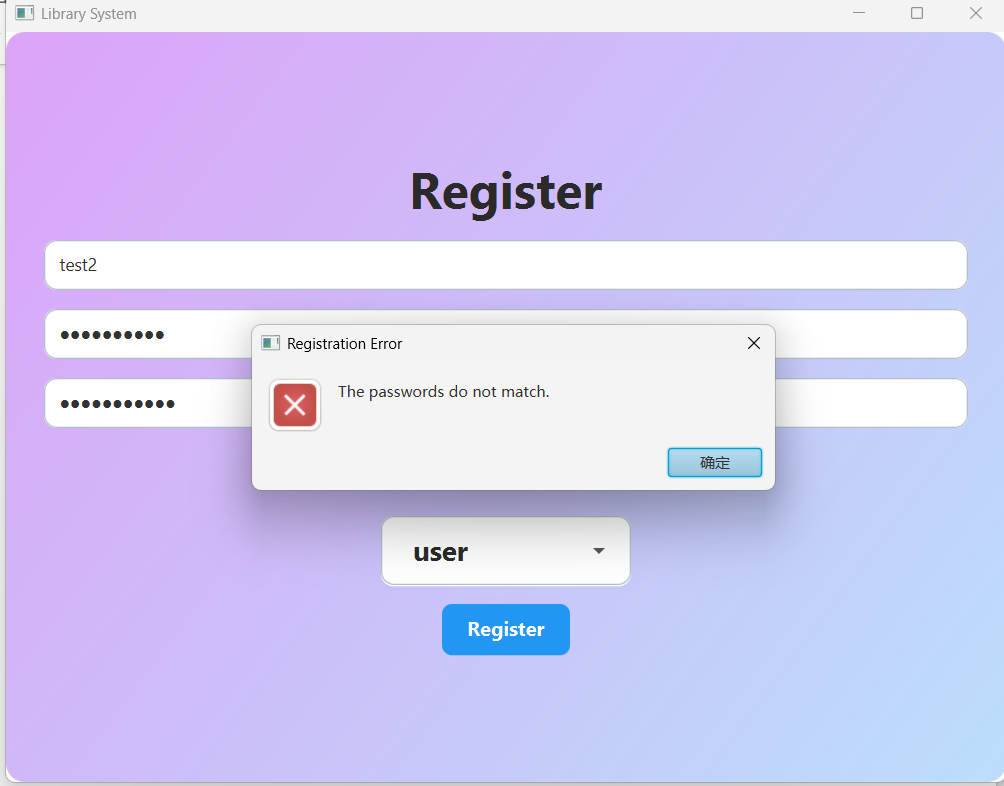


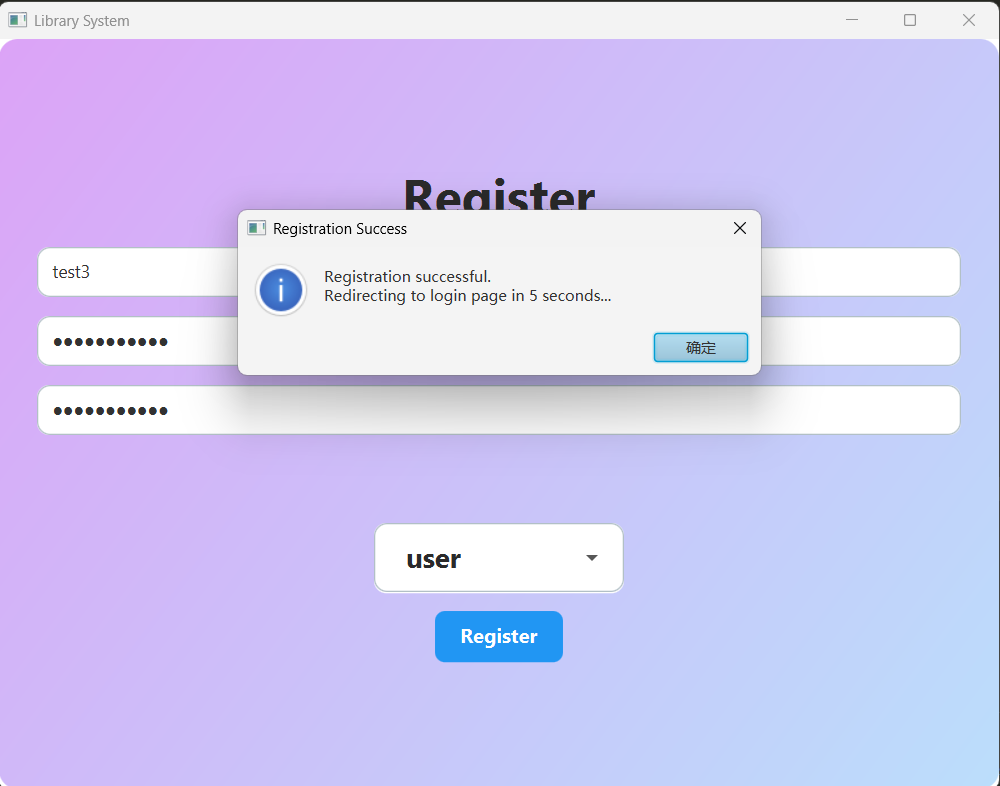


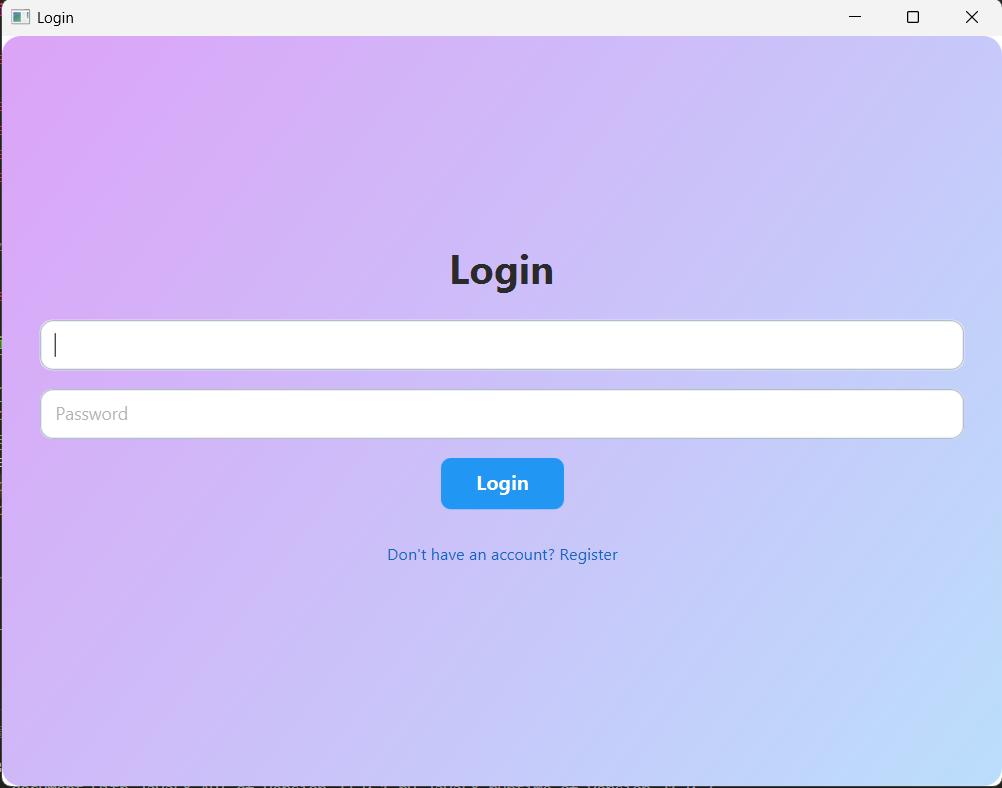
### Register Function

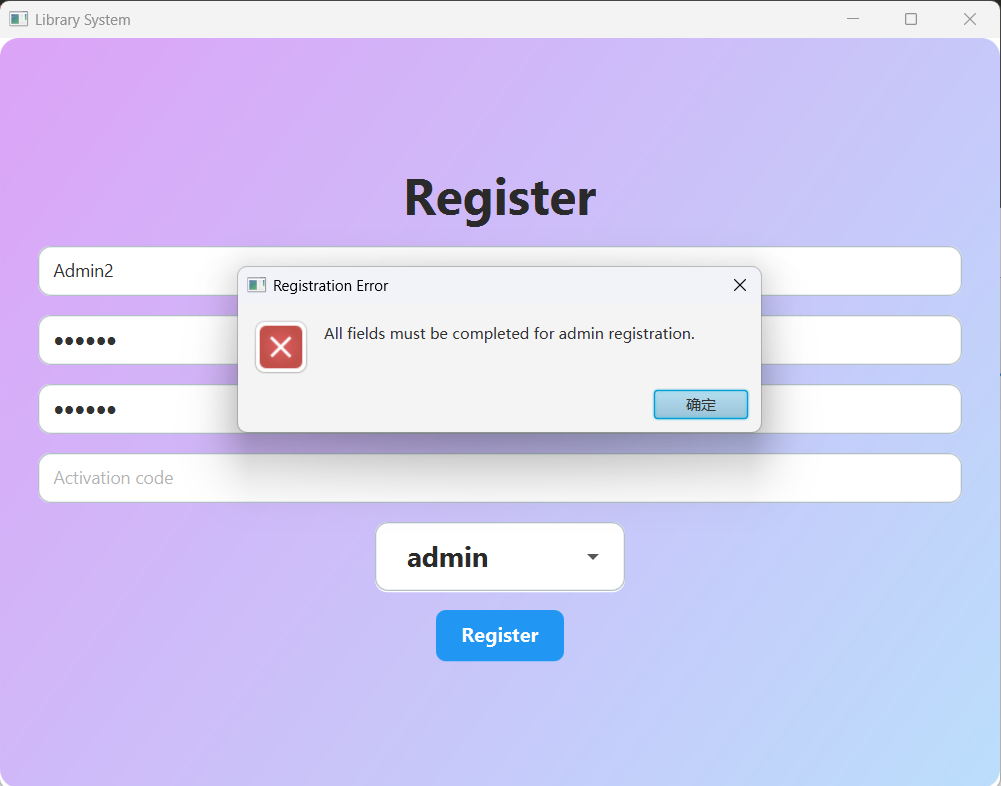
Here is the registration interface, there are only two registered roles here, one is admin and one is user, you don't need to enter the activation code when you select user, but you need an activation code when you choose admin, the activation code is my fixed value, and you can add dynamic in the future. Duplication is not allowed when setting the account name, the password has a strength requirement, and the password must be kept twice, and there are many detection systems.

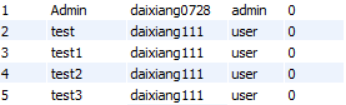






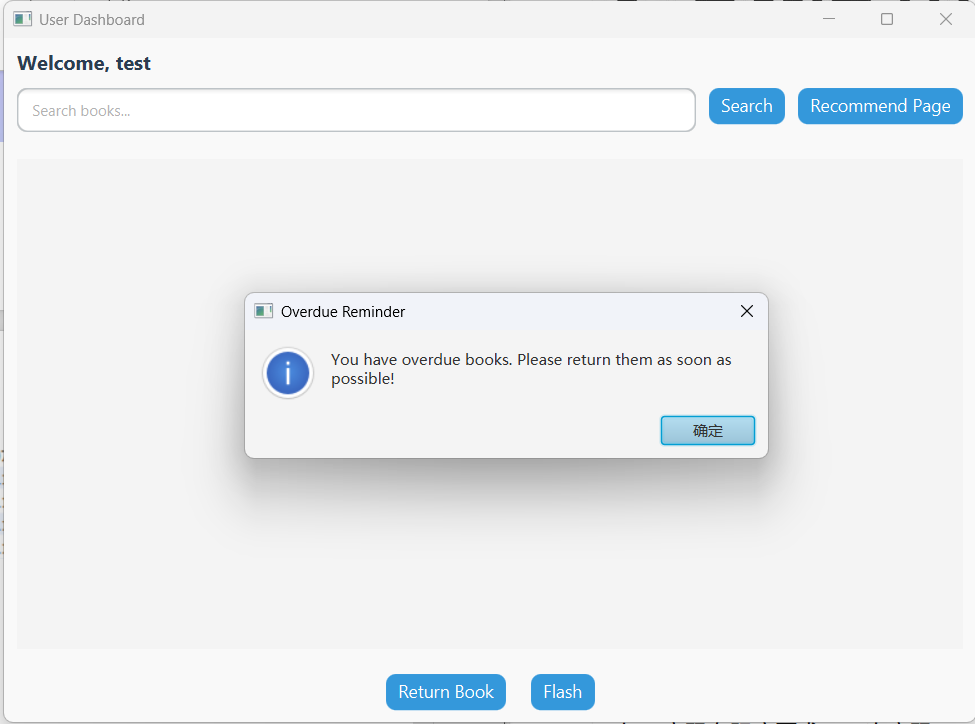


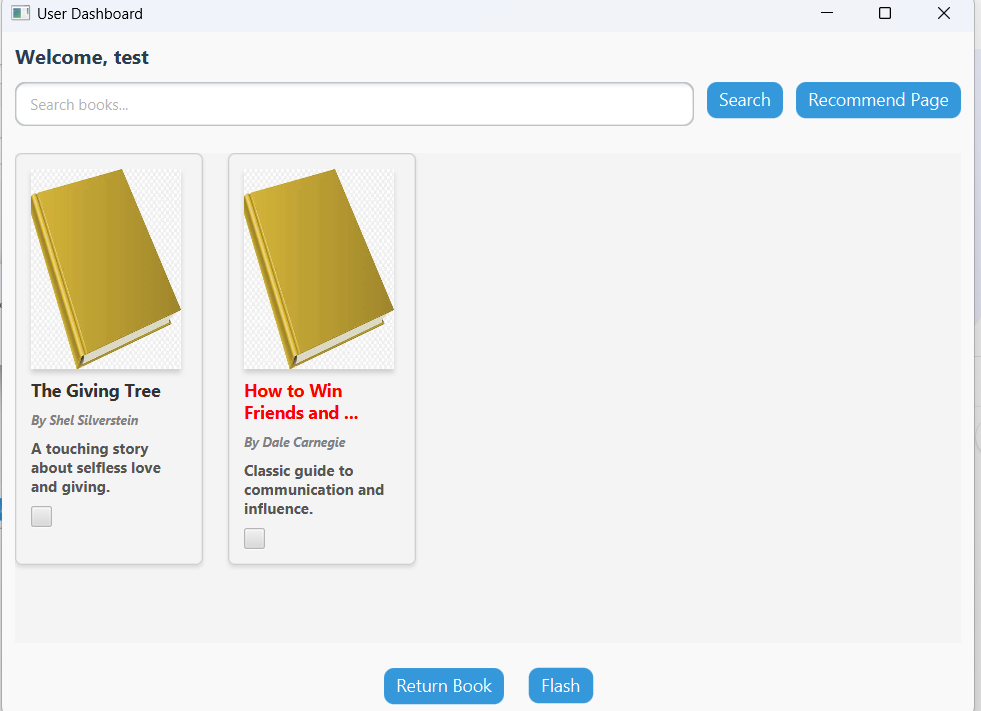


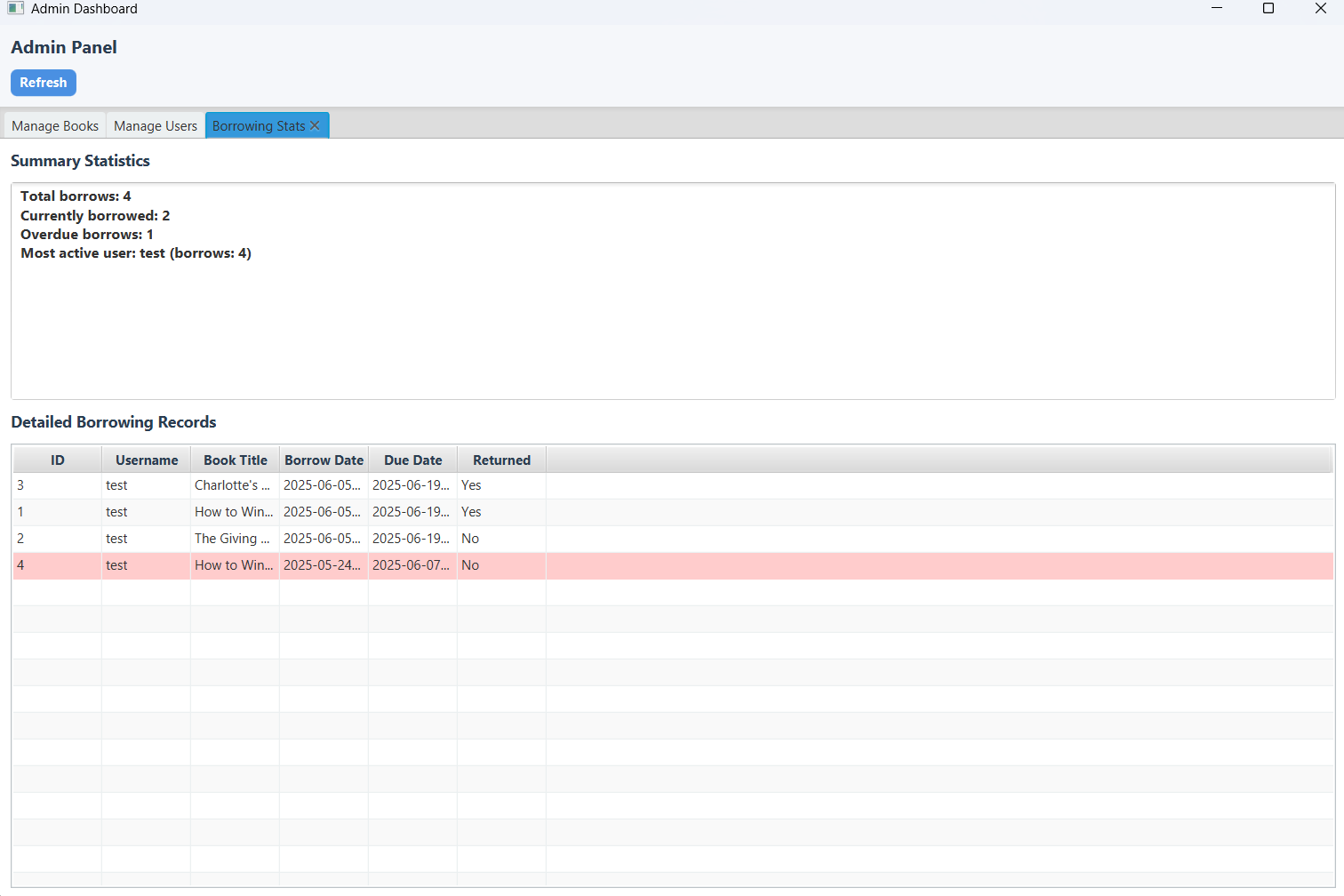


### Overdue Function

This is the overdue function test, when the user goes online, the server will send a Reminder to the user client to remind the user that he has overdue books, and the books that have not been returned will also be marked in red in the personal library, and similarly, the overdue data on the administrator page is also marked in red.

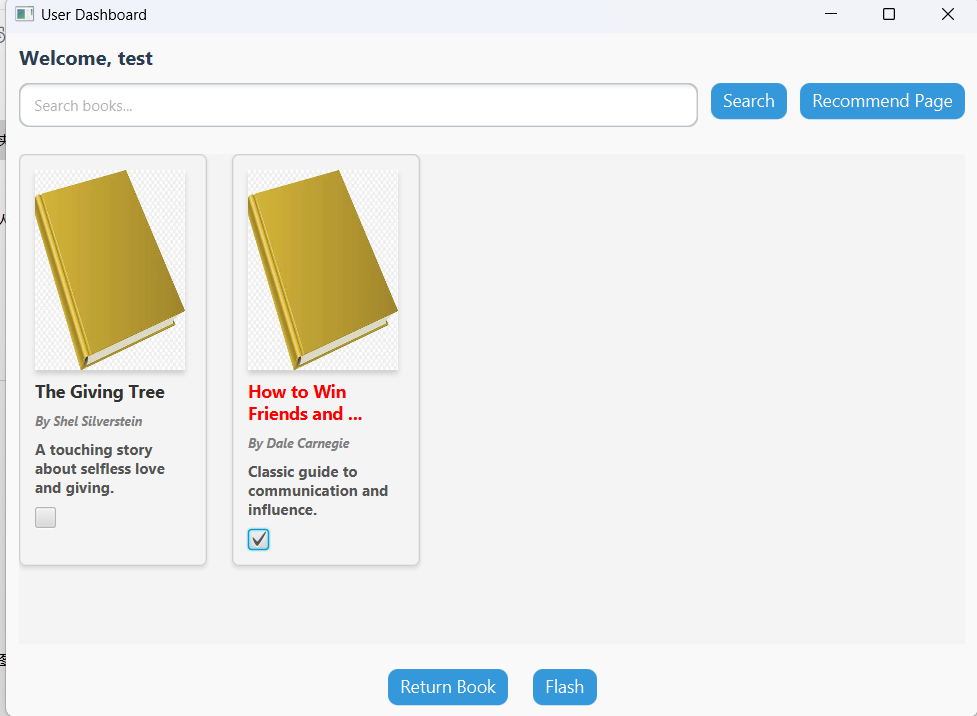




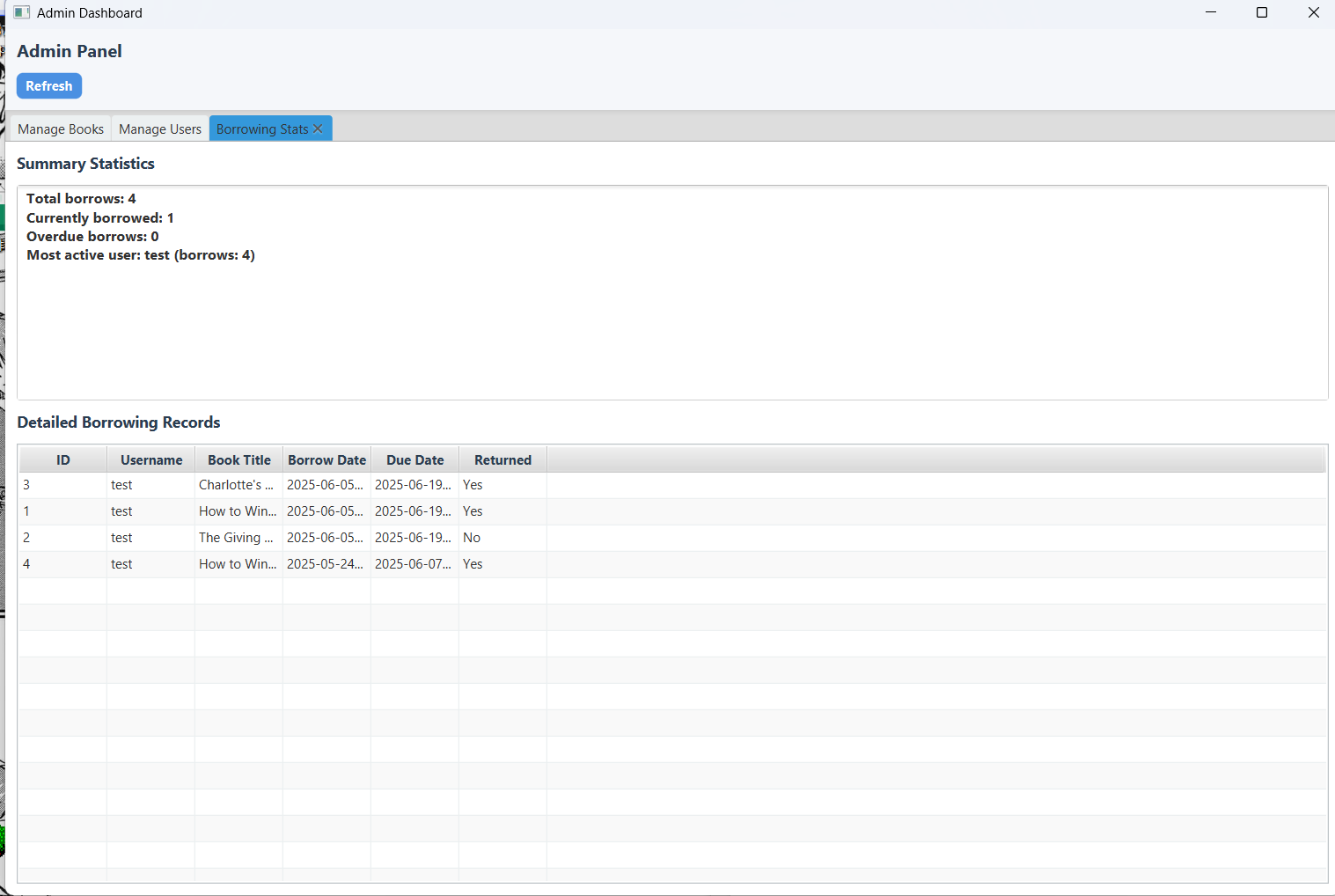
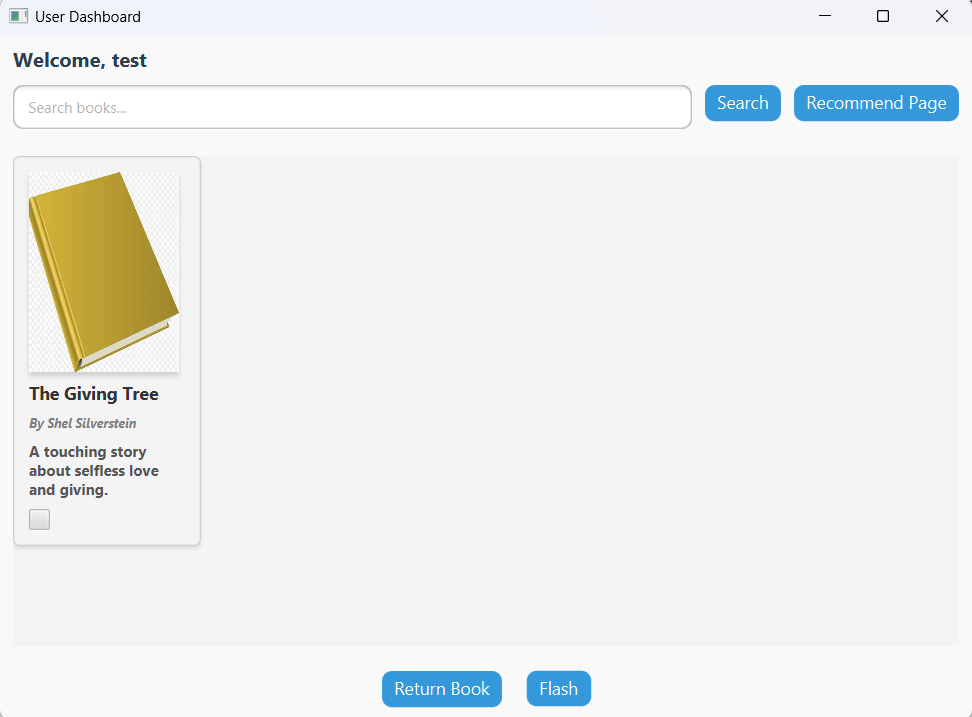


### Return book function

For the return function, we only need to come to the customer interface, check the checkbox and click return book, we can return the book, and after returning the book, the admin interface and the user interface will be changed accordingly.

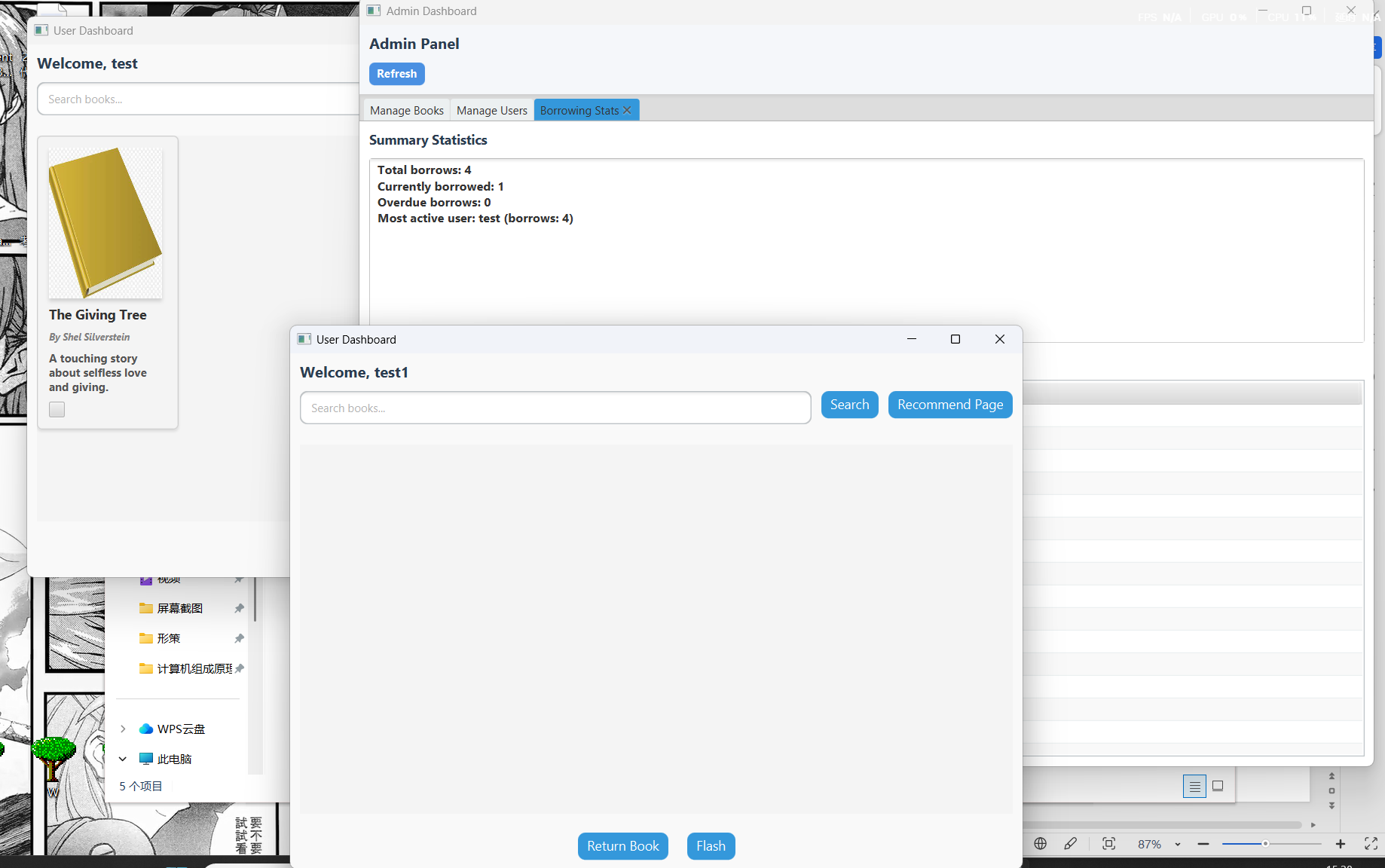


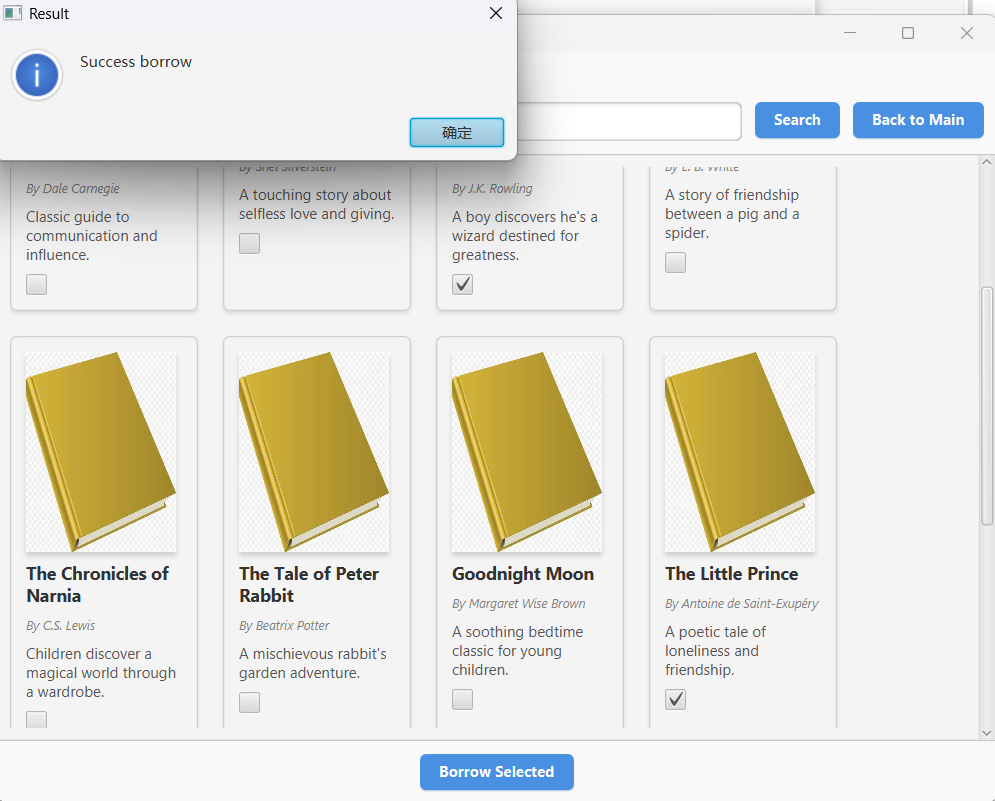


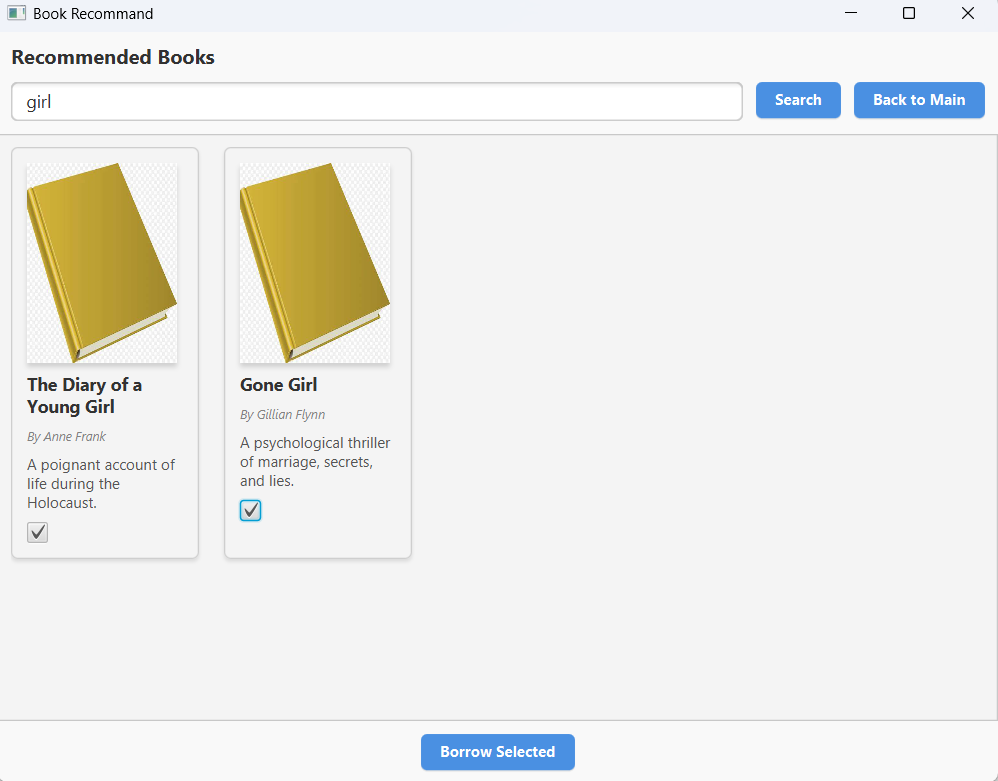
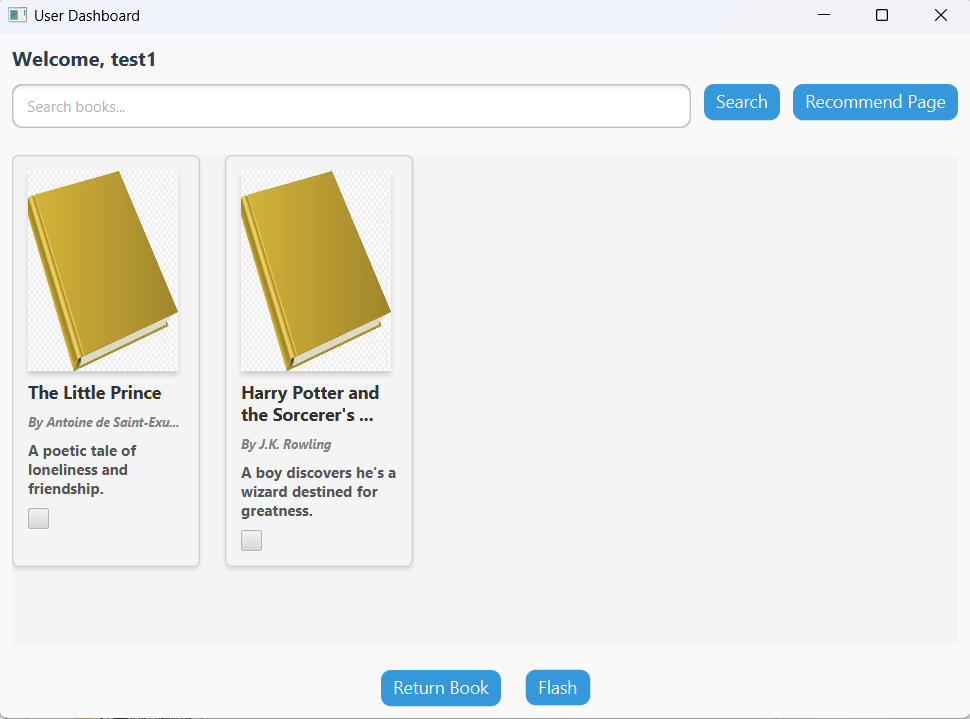


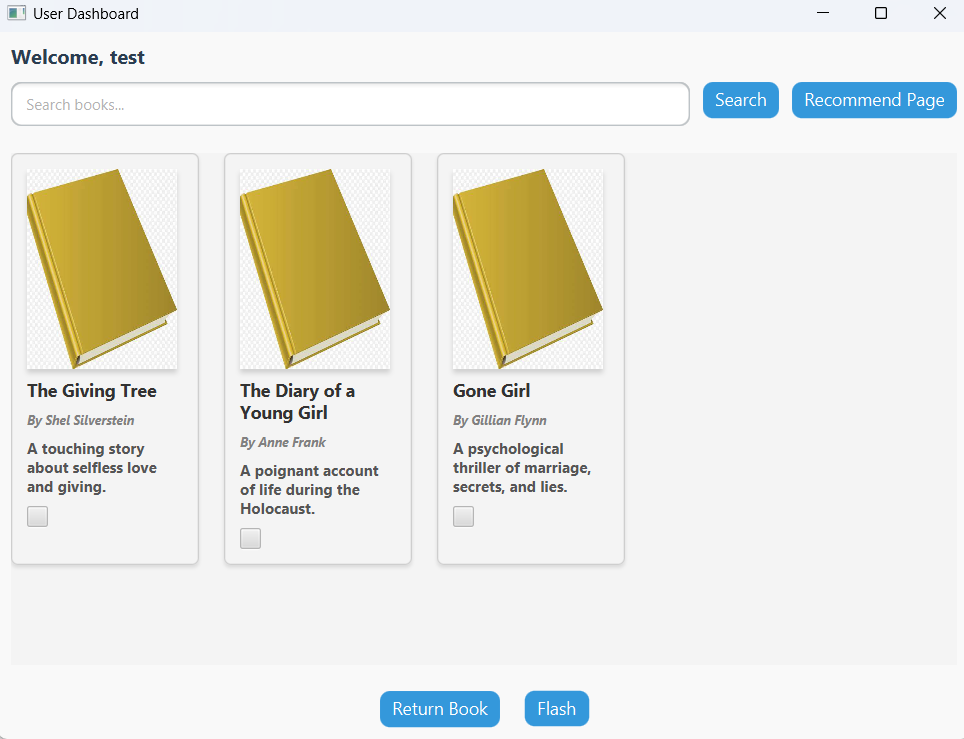
### Multiple Thread Test & Borrow Function

T his shows a multi-threaded implementation, which can be used to enable two users and one administrator at the same time, and can borrow books at the same time, and the administrator interface will be updated accordingly, and the search box in it will also work normally



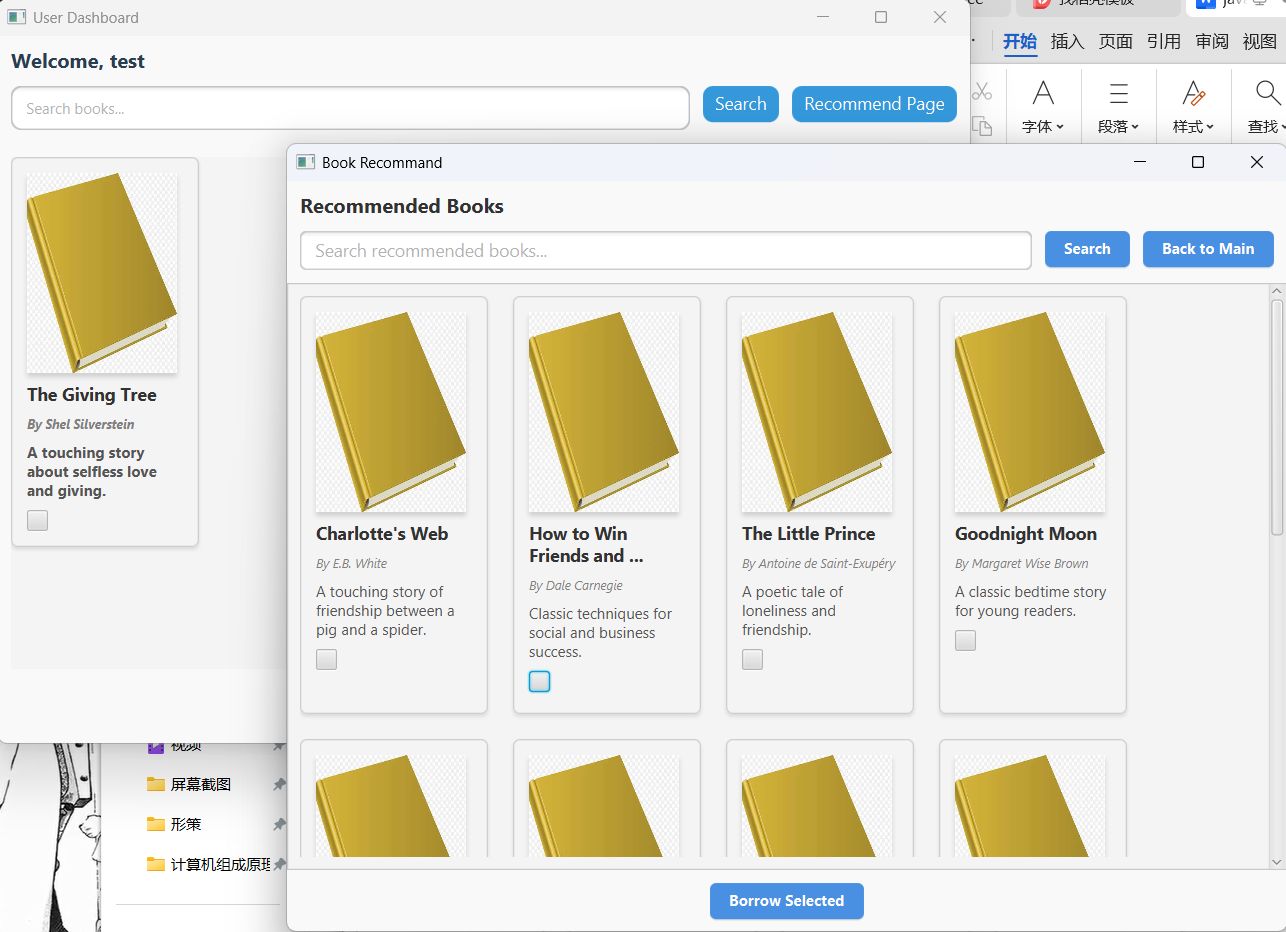


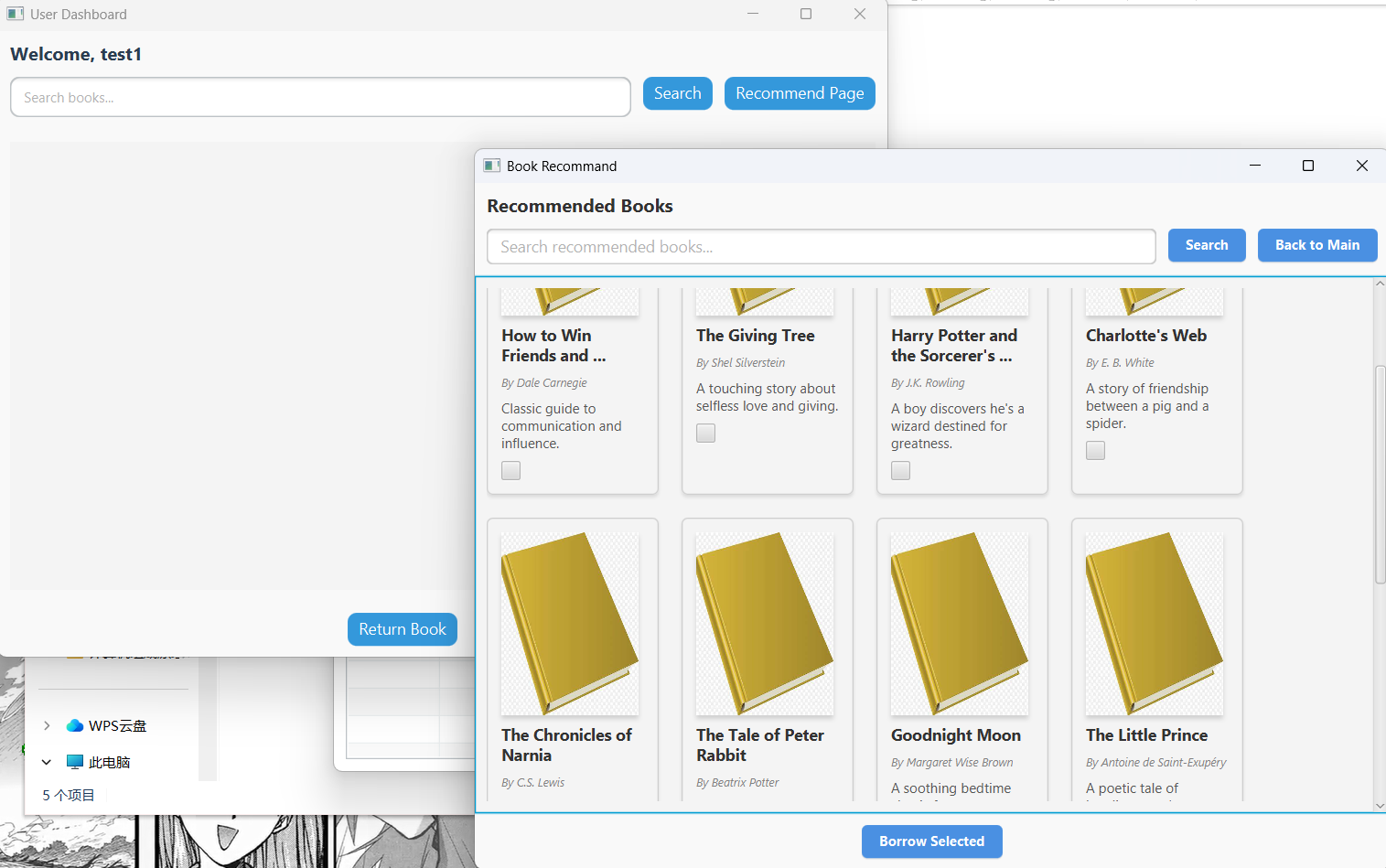


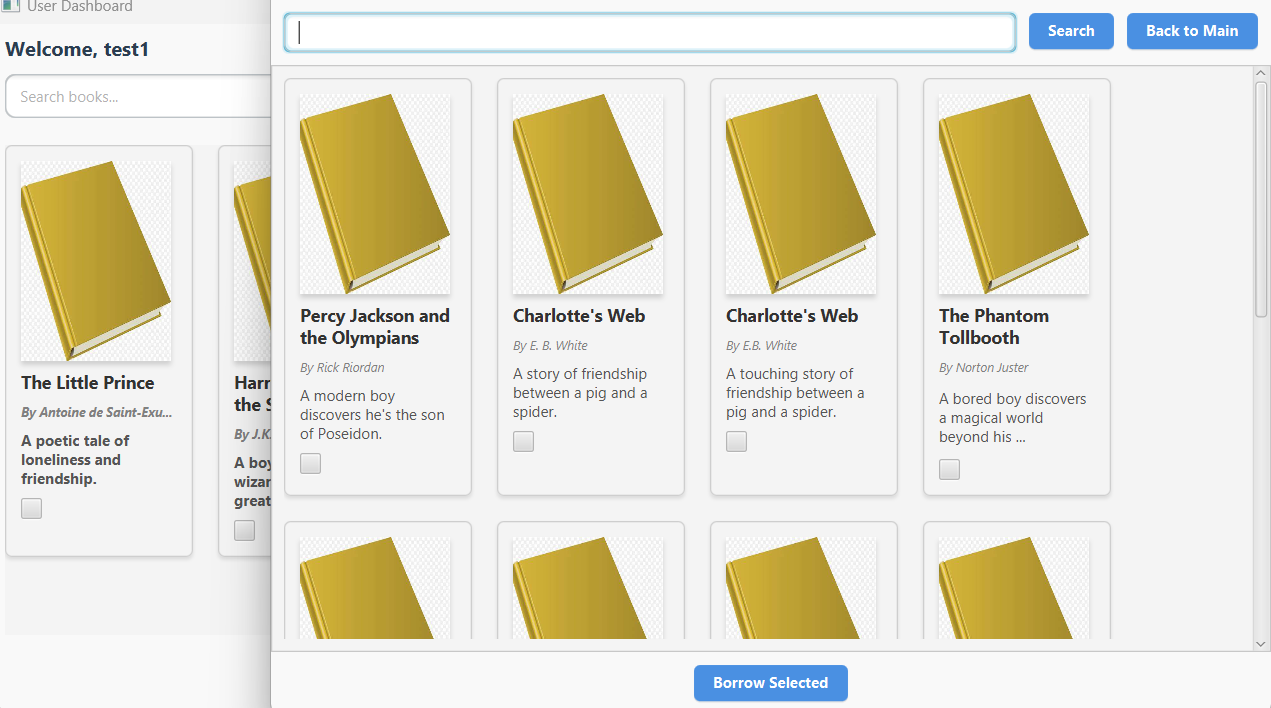


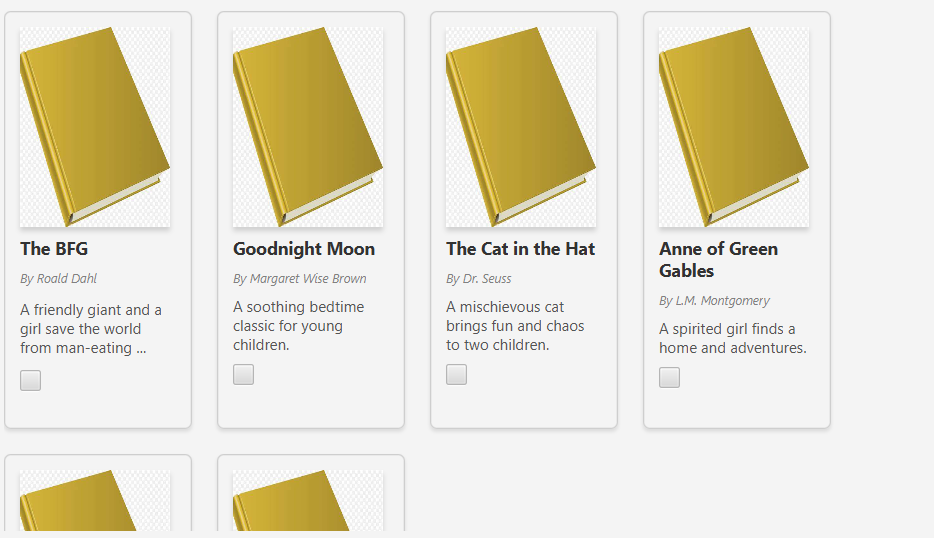
### Recommend Function

Here is a show of the recommended book function, those who have borrowed books will use local recommendations, some books here have a high repetition rate, which may be the reason why the amount of data is too small, and those who have not borrowed books use the default recommendation of the network; In the graph, we can see that the recommended books after borrowing them are significantly different.



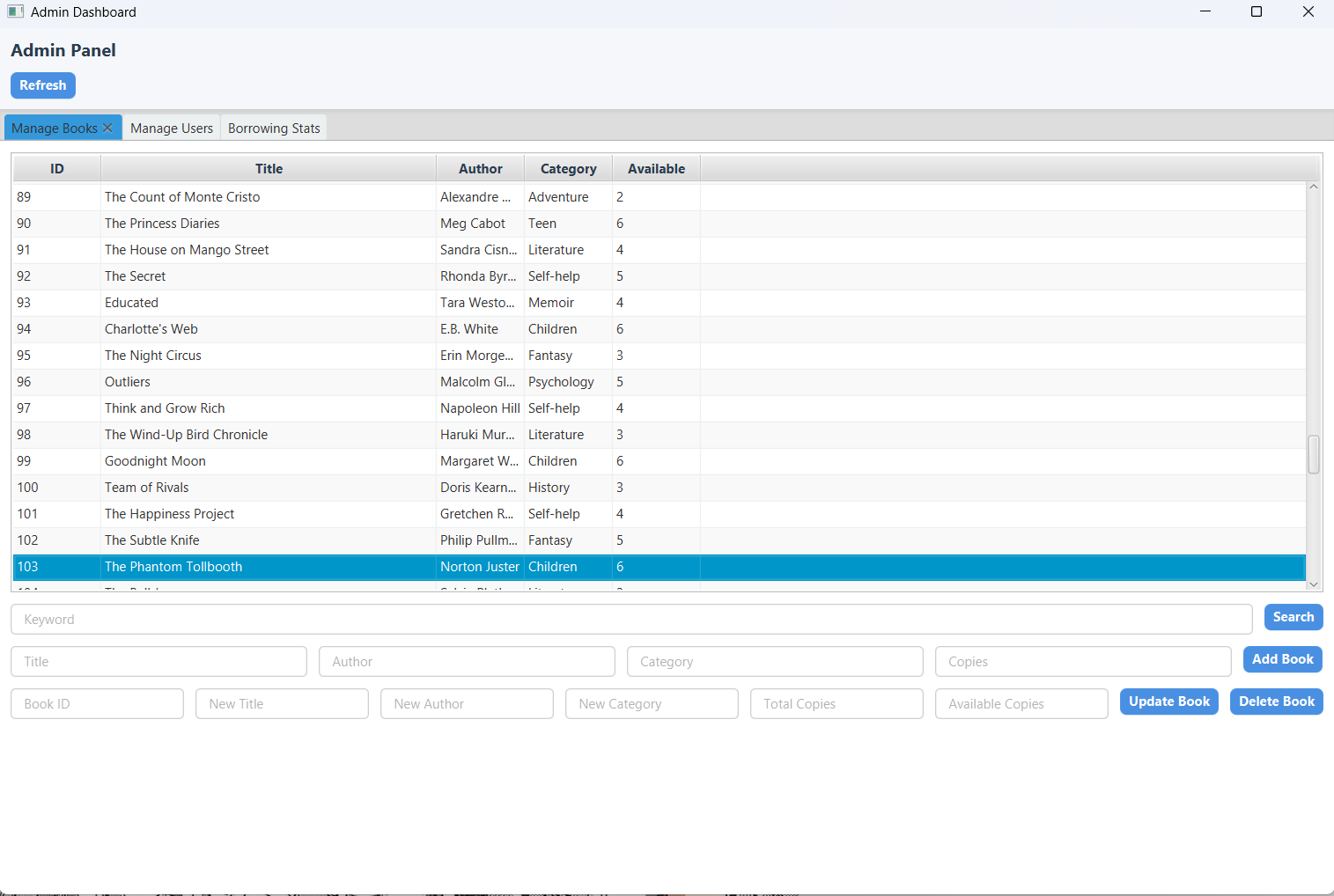


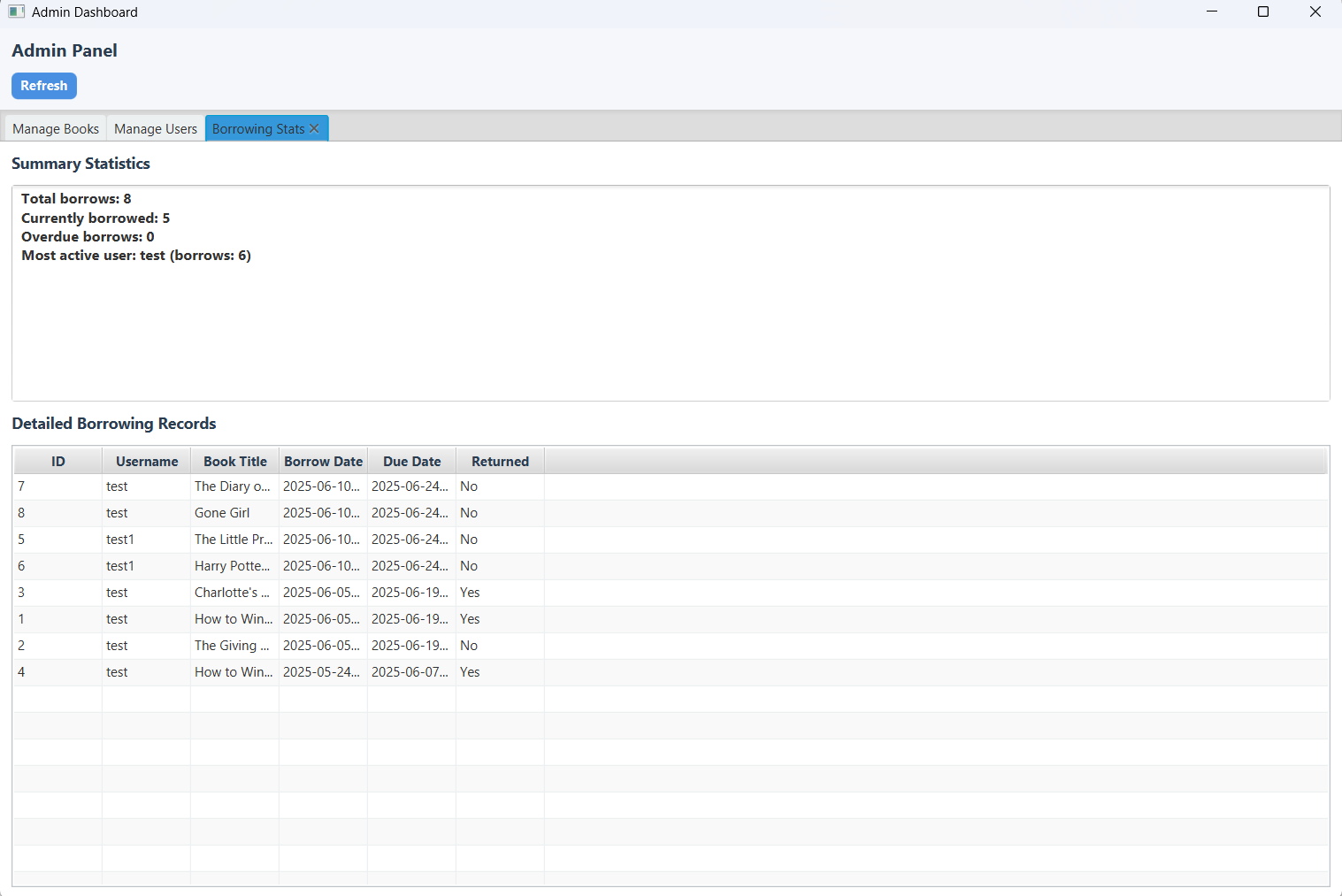
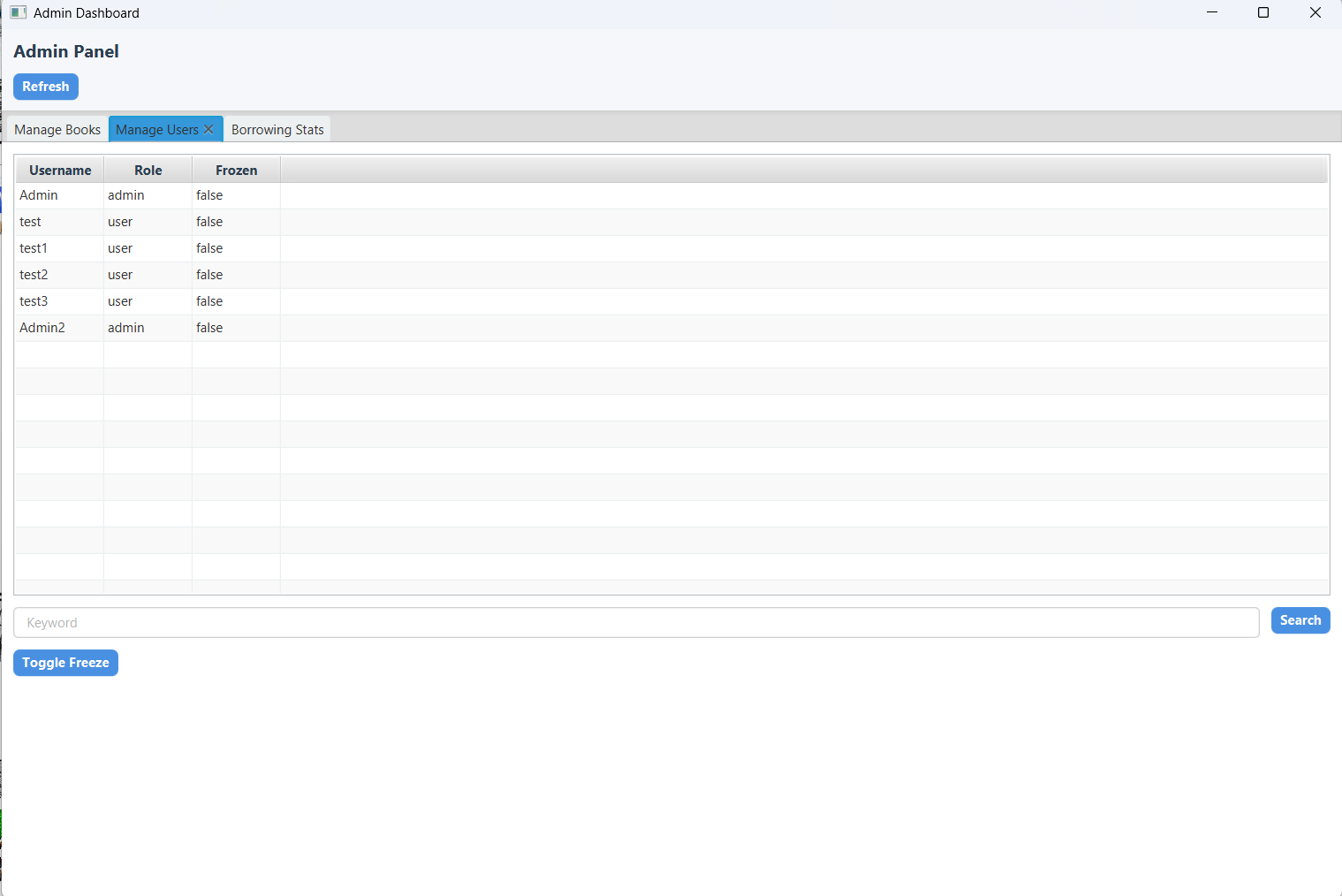




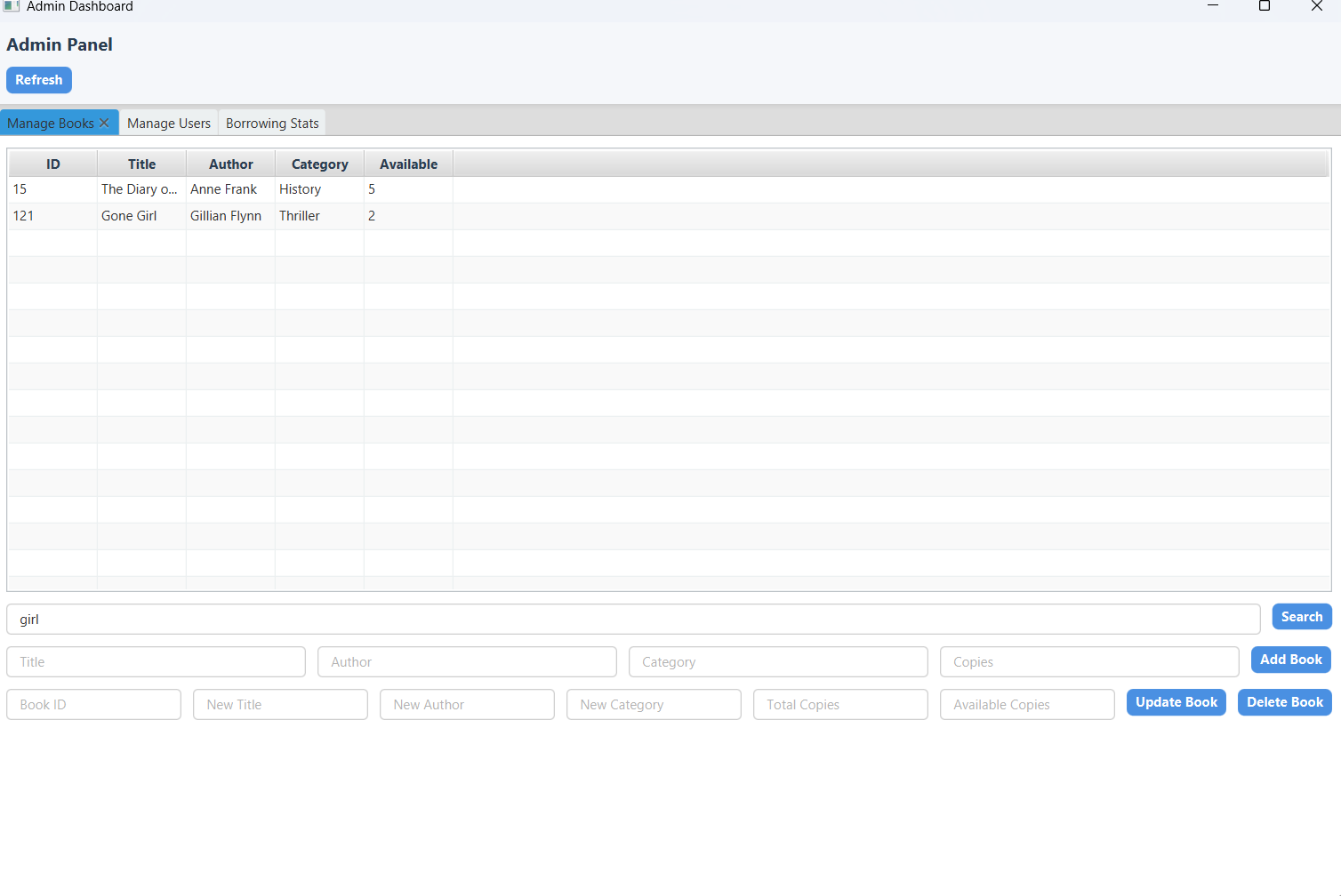
### Admin View Function

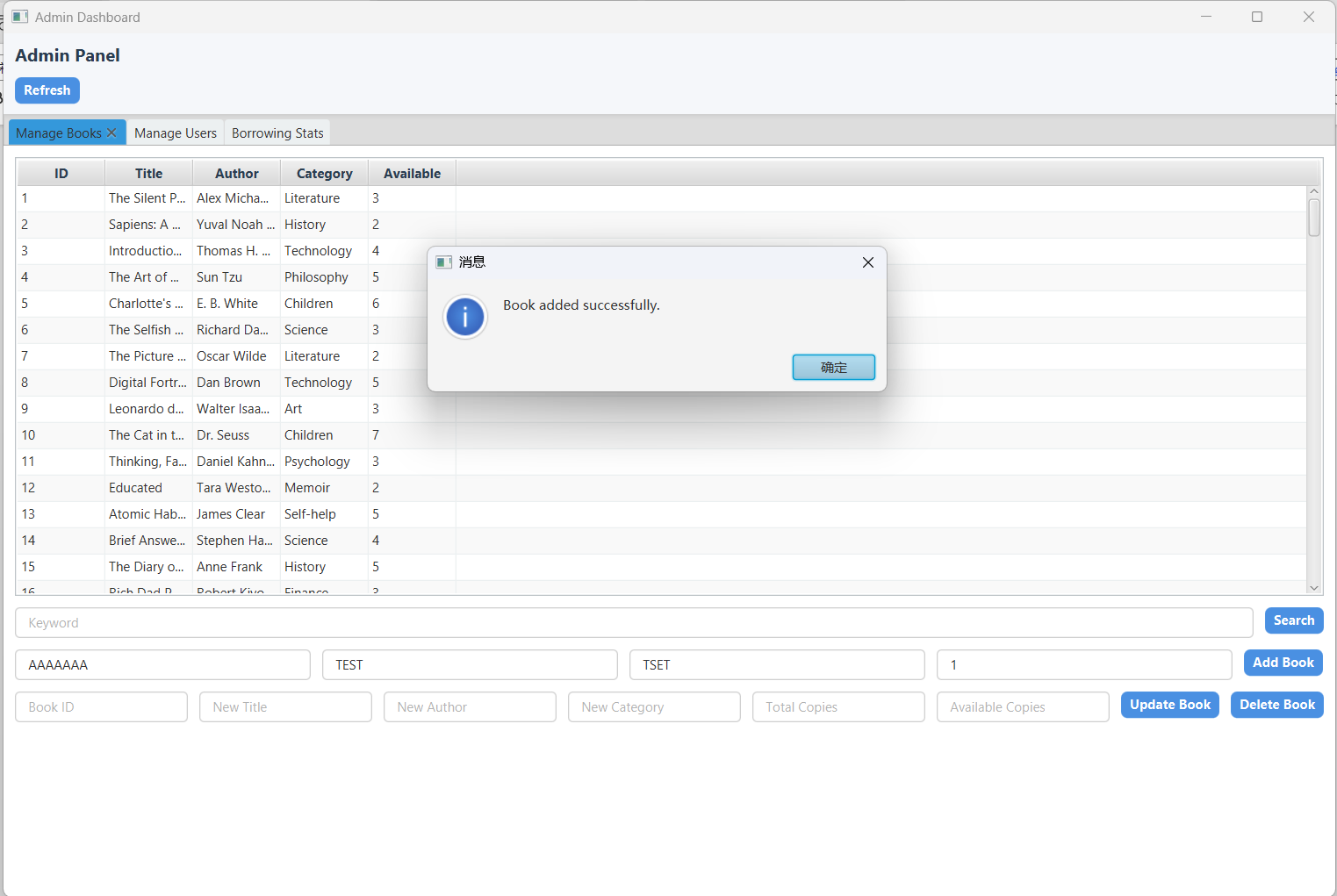
Here the administrator can view all the information about the database

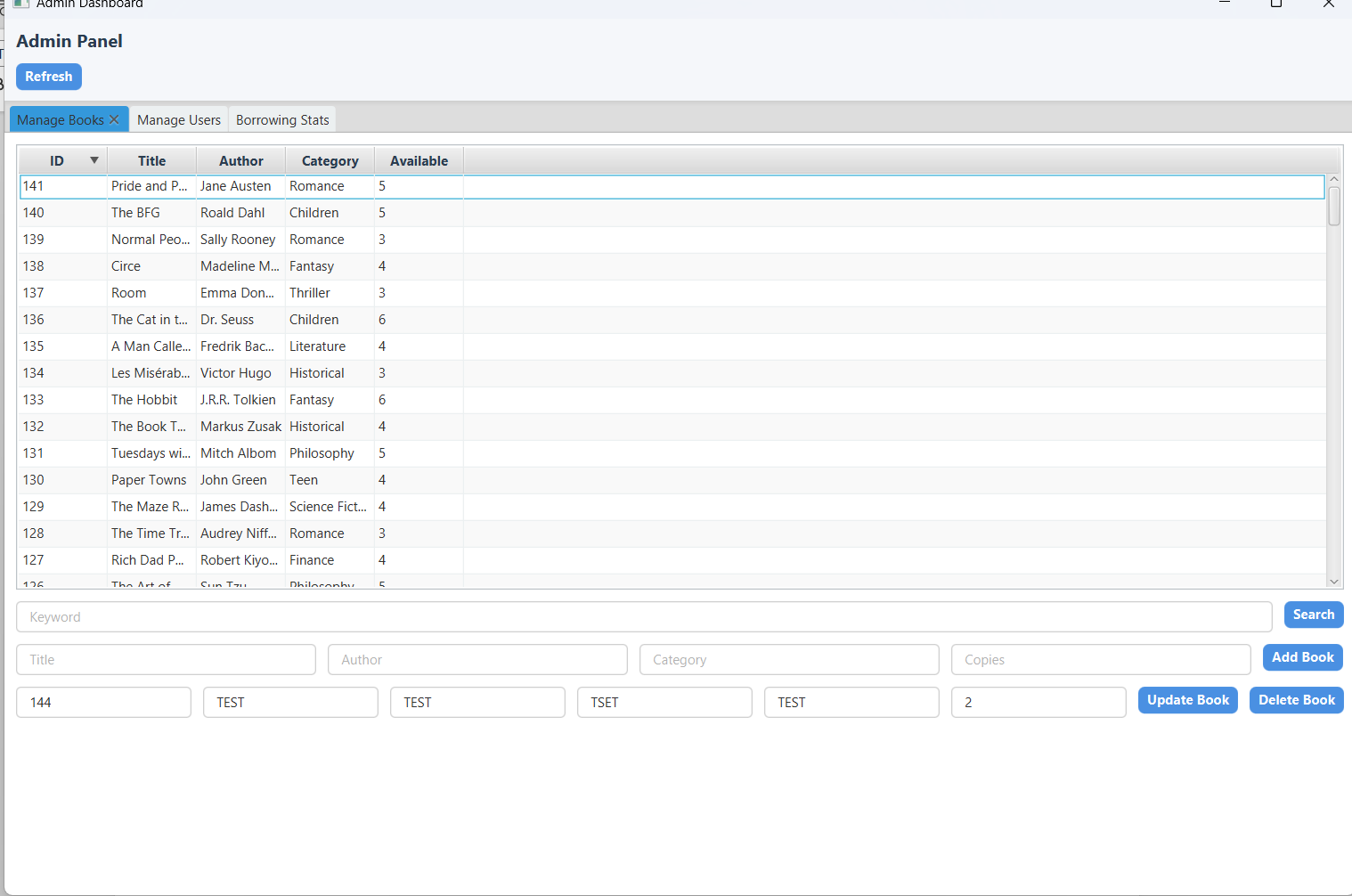




### Add,Delete,Change Function

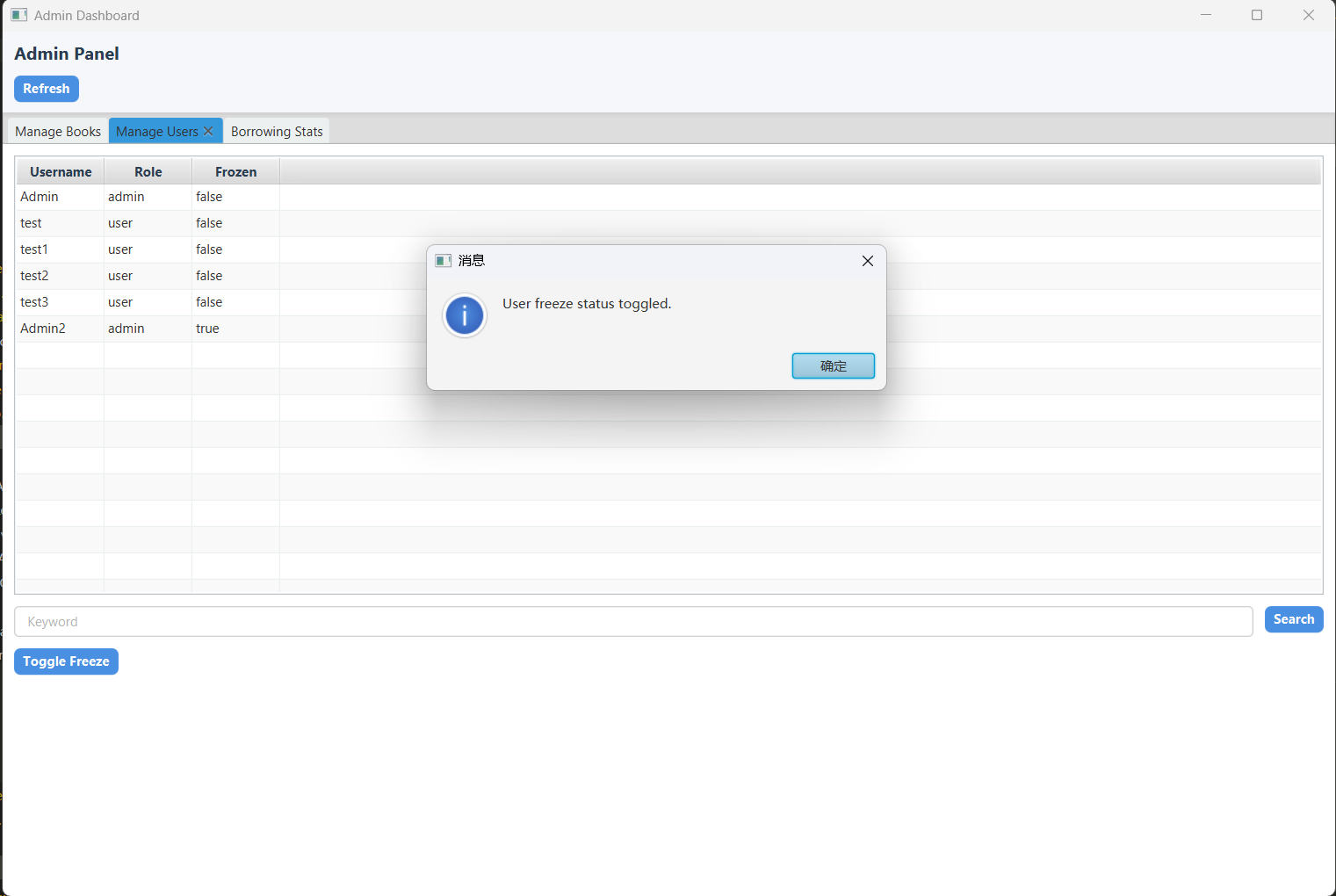


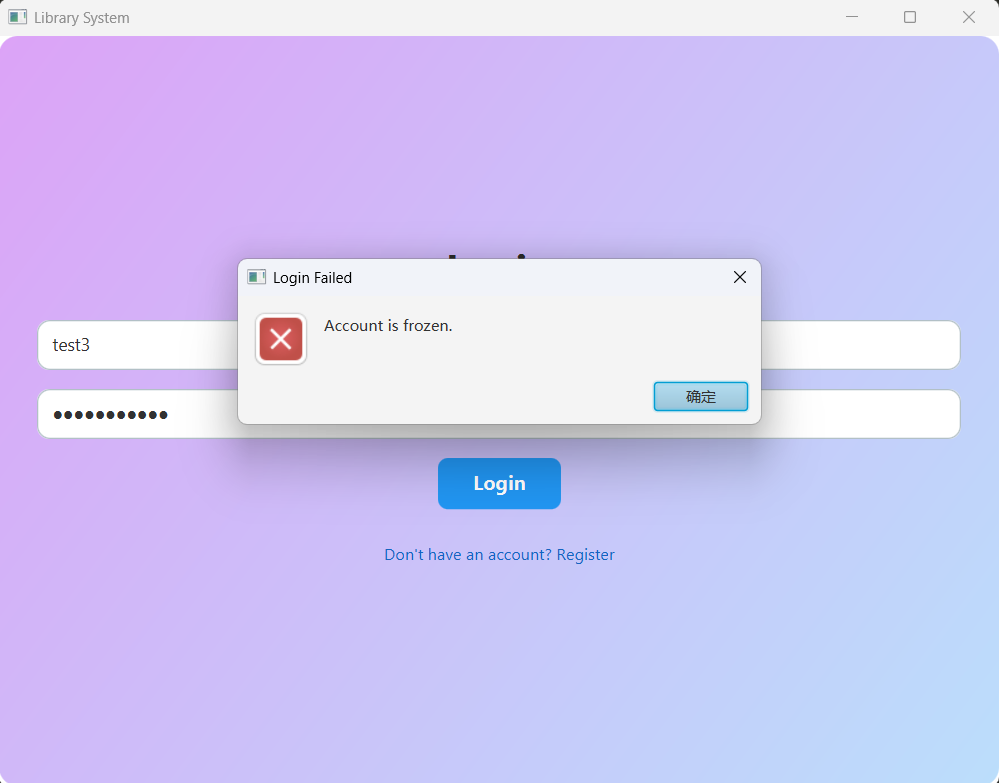
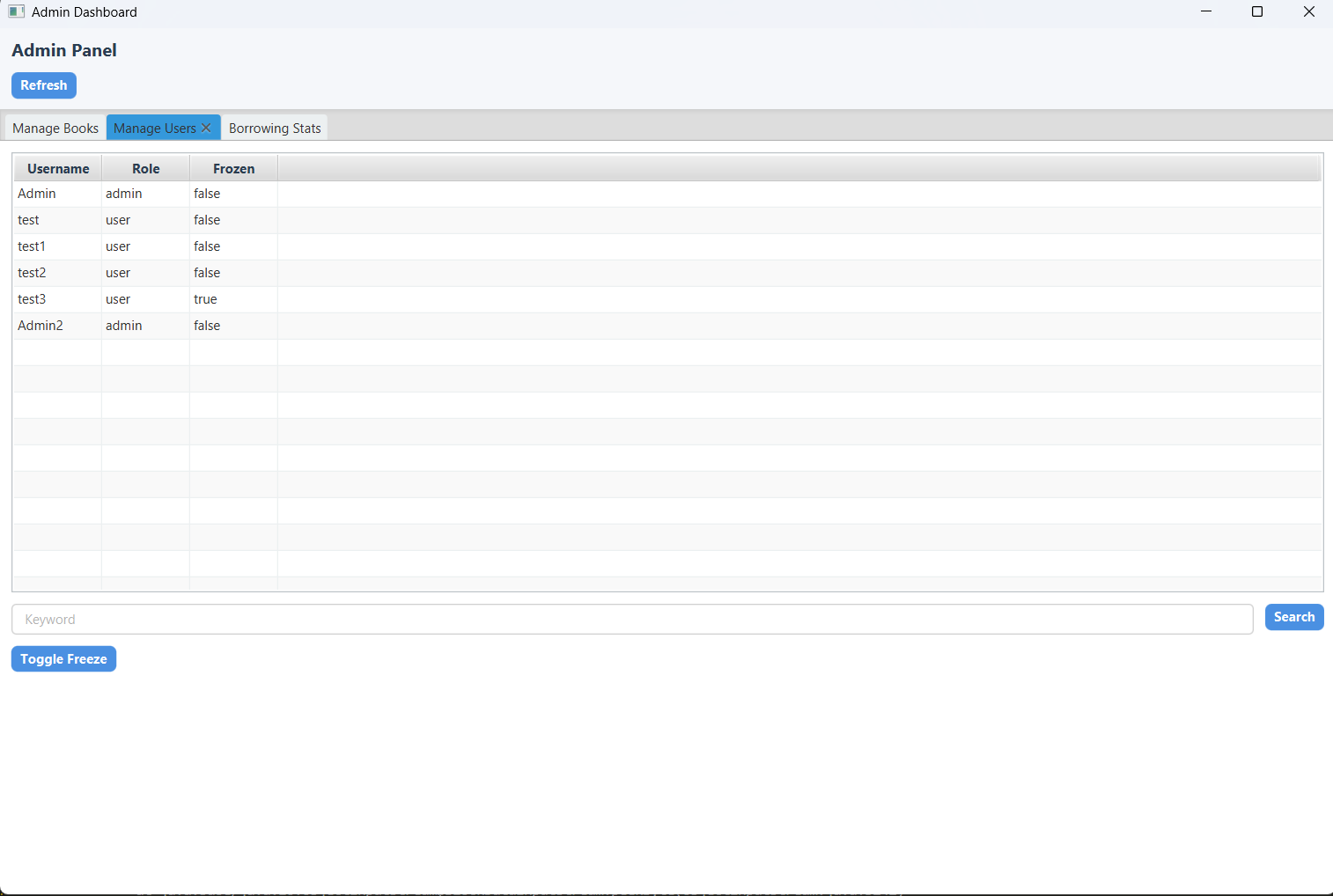




### Frozen Function

The administrator can manage whether to suspend the current user through the toggole button, and the following aleart will be displayed when the user is logged in





## Summary

This project has been a comprehensive and practical exercise in building a complete multi-user library management system using Java. Throughout the development process, we applied a wide range of Java technologies, including object-oriented programming, JavaFX for GUI design, multithreading for server concurrency, JDBC for database interaction, and socket-based client-server communication. The system supports both administrator and normal user roles, each with distinct functionalities and interfaces, which allowed us to practice inheritance, polymorphism, and dynamic UI rendering.

One of the most challenging aspects of the project was ensuring concurrent access and thread safety on the server side. To address this, we implemented a thread pool architecture that allows the server to handle multiple client requests simultaneously without data conflicts or crashes. Another complex but rewarding task was integrating a basic content-based recommendation algorithm. By vectorizing book attributes such as category, author, and description (via TF-IDF), and computing cosine similarity, the system can now generate personalized book suggestions based on the user's reading history.

In addition to functionality, user experience (UX) was also a priority. Using JavaFX, we designed responsive and intuitive interfaces for both login/registration and the main operation panels. Features such as popup alerts, search filters, checkboxes for batch operations, and real-time feedback (e.g., overdue reminders) greatly enhanced usability.

From a backend perspective, we designed a relational database schema consisting of users, books, and borrowing records. The schema ensures data normalization and supports essential operations like borrowing tracking, book availability updates, and user management. Admins are empowered to manage books, freeze/unfreeze users, and view borrowing statistics, while normal users can browse, borrow, return books, and receive timely notifications.

Overall, this project greatly deepened our understanding of full-stack Java development. It combined theory and practice by integrating multiple paradigms and technologies into a cohesive application. The final system is functional, extensible, and user-friendly, meeting the project requirements.

Looking forward, the system could be enhanced with features such as book rating and review modules, RESTful API support, cloud-based database deployment, and a hybrid recommendation system that combines collaborative and content-based filtering for better accuracy. Additionally, security improvements like password hashing and session control would be beneficial for real-world use.

In conclusion, this project not only demonstrated the practical application of Java but also strengthened our problem-solving and system integration skills. It has laid a solid foundation for future development in software engineering and intelligent systems.