**Book Lending System**

**Internship Project assignment**

**Christ University**

Done by:

Santhosh Kumar P (2347148)

**Language Used:** Rust  
**Database:** SQLite  
**Frontend:** HTML + JavaScript

**Repository:** [**https://github.com/SanthoshKumar150822/Book-Lending-Web-App**](https://github.com/SanthoshKumar150822/Book-Lending-Web-App)

**Overview**

This project is a **minimalist full-stack book lending system** for internship screening task. The application enables:

* Registration and role-based management of users (Admins and Lenders)
* Addition of books with metadata and availability tracking
* Simple HTML + JS frontend for interaction
* Backend built using **pure Rust**, handling HTTP without any frameworks
* All data stored in a **SQLite** database with schema defined and auto-initialized

**Architecture**

Backend – Rust

* Built using native TCP handling with std::net::TcpListener
* HTTP parsing done manually (method + path + body extraction)
* Asynchronous processing using tokio
* Password hashing using bcrypt
* Request deserialization via serde
* Database interaction via sqlx

Frontend – HTML + JS

* Two forms:
  + User Registration
  + Book Addition
* JavaScript sends fetch POST requests with JSON payloads
* Plain, readable HTML layout (unstyled but structured)
* Compatible with local browsers (no server required)

**Dependencies and Setup**

The project uses Cargo for package management. The dependencies defined in Cargo.toml are:

- serde: for serializing and deserializing Rust data structures.

- serde\_json: for handling JSON file operations.

**Database**

Defined in sql/init.sql, loaded automatically on startup if DB is missing.

Tables:

users

* id, username, password\_hash, role
* Roles restricted to "admin" or "lender" using CHECK

books

* Core metadata: title, author, isbn, publication\_year, genre, copies
* Status: defaults to "available"

lendings

* Tracks who borrowed what, when, and returned status
* Foreign keys to both users and books

**Rest API Documentation**

To be implemented

**/register – POST**

Registers a new user

**Request:**

**json**

**{**

**"username": "john\_doe",**

**"password": "secure123",**

**"role": "admin"**

**}**

**Response:**

* 200 OK → User registered
* 400 Bad Request → Invalid payload
* 500 Internal Server Error → DB issue

**/books – POST**

Adds a new book (admin only)

**Request:**

**json**

**{**

**"title": "1984",**

**"author": "George Orwell",**

**"isbn": "0451524934",**

**"publication\_year": 1949,**

**"genre": "Dystopian",**

**"copies": 2**

**}**

**Response:**

* 200 OK → Book added
* 400 Bad Request → Invalid data
* 500 Internal Server Error → DB issue

**Project Structure**

The project is modularized into the following components:

• - main.rs: Entry point and CLI logic.

• - book.rs: Handles book-related operations.

• - member.rs: Manages members.

• - lending.rs: Implements lending and returning of books.

**Working Flow**

**User Registration**

Actors:

* Admin
* Lender (Regular User)

**Frontend Action**:

* The user fills in a registration form with:
  + username
  + password
  + role (either "admin" or "lender")

**Backend Flow**:

* Form data is serialized to JSON and sent via a POST request to /register
* The backend:
  + Parses the JSON
  + Hashes the password using **bcrypt**
  + Inserts a new record into the users table with the selected role
  + Responds with a success or failure message

**Security Notes**:

* Passwords are never stored in plaintext
* Only valid roles are allowed (CHECK(role IN ('admin', 'lender')))

**Book Addition (Admin Only)**

**Actor**:

* Admin

**Frontend Action**:

* Admin fills in a book form with:
  + title, author, isbn
  + publication\_year, genre, and copies

**Backend Flow**:

* Form data is converted into JSON and sent to POST /books
* The backend:
  + Parses and validates the input
  + Adds a new entry in the books table
  + Sets default status = 'available'

**Database Impact**:

* Book record created with associated metadata
* copies field determines how many users can borrow the book concurrently\

**Planned Features & Functional Extensions**

**User Login (POST /login)**

* Accepts username and password
* Verifies credentials against stored hash
* Creates a new session token (stored in the sessions table)
* Returns token for session-based authentication in future requests

**Book Borrowing (POST /borrow)**

* Authenticated lenders can borrow books
* System checks if copies are available
* Creates a lending record with:
  + borrowed\_at timestamp
  + due\_date based on policy (e.g., 14 days later)
* Updates copies count in books table

**Book Returning (POST /return)**

* User submits return request for a book
* Updates returned\_at in lendings
* Increases copies in books
* Calculates if the return is overdue

**Admin Dashboard (GET /admin/dashboard)**

* Displays:
  + All users with roles
  + List of all books with status and availability
  + Active lendings and their due dates
  + Overdue books with borrower info
* Helps admins manage the system efficiently

**Book Search (GET /books?query=)**

**Functionality**:

* Search by title, author, or genre
* Filter by availability, publication\_year, etc.
* Implements pagination for performance

**Security Implementation**

* Passwords hashed using **bcrypt** with default cost
* Role-based user system
* Sessions table prepared for future login/token-based auth
* Input deserialized with serde, reducing injection risks
* CORS headers implemented to allow frontend/backend communication

**Project Setup Instructions**

**Requirements:**

* Rust (with Cargo)
* SQLite installed
* sqlx-cli for schema execution (optional)

**Steps:**

**git clone https://github.com/yourname/book-lending-system**

**cd book-lending-system**

**cargo run**

The app will create book\_lending.db and start listening on localhost:3000.

Open frontend/index.html directly or use:

**npx serve frontend/**

**Code Quality & Comments**

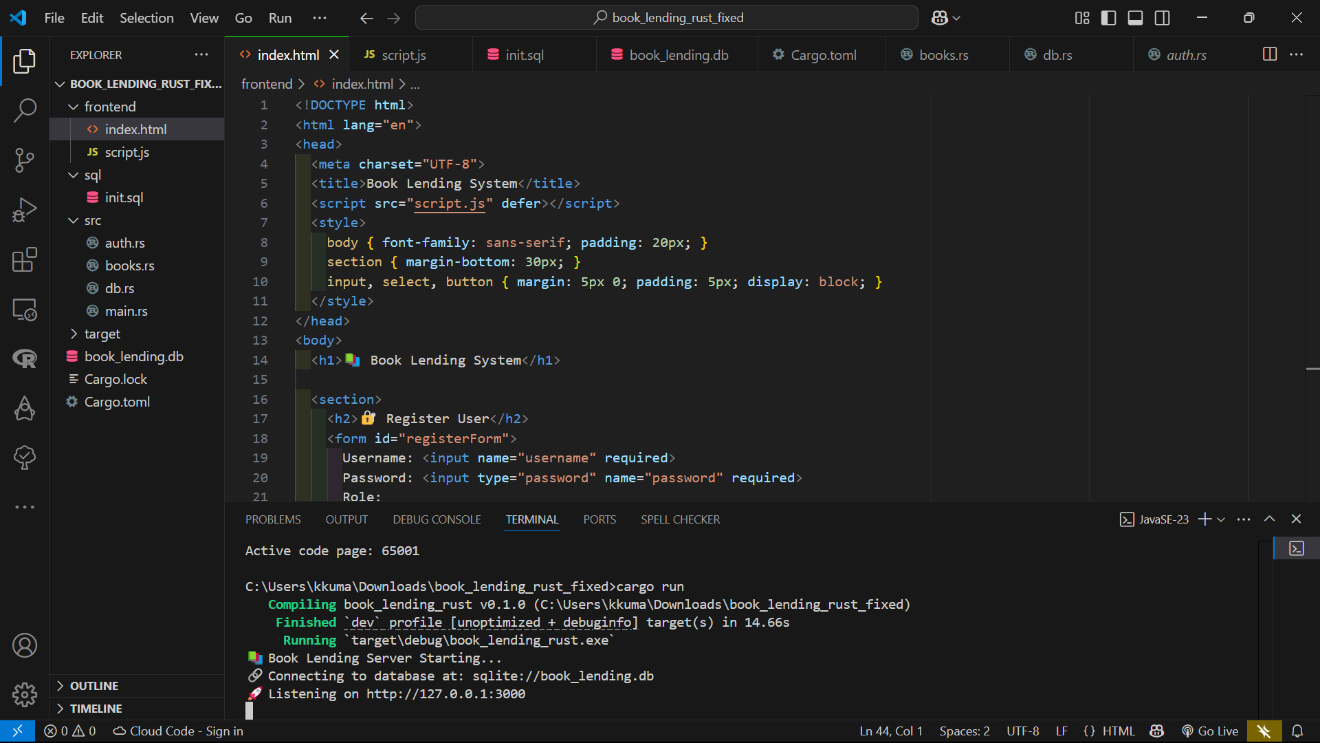
**Modularity:**

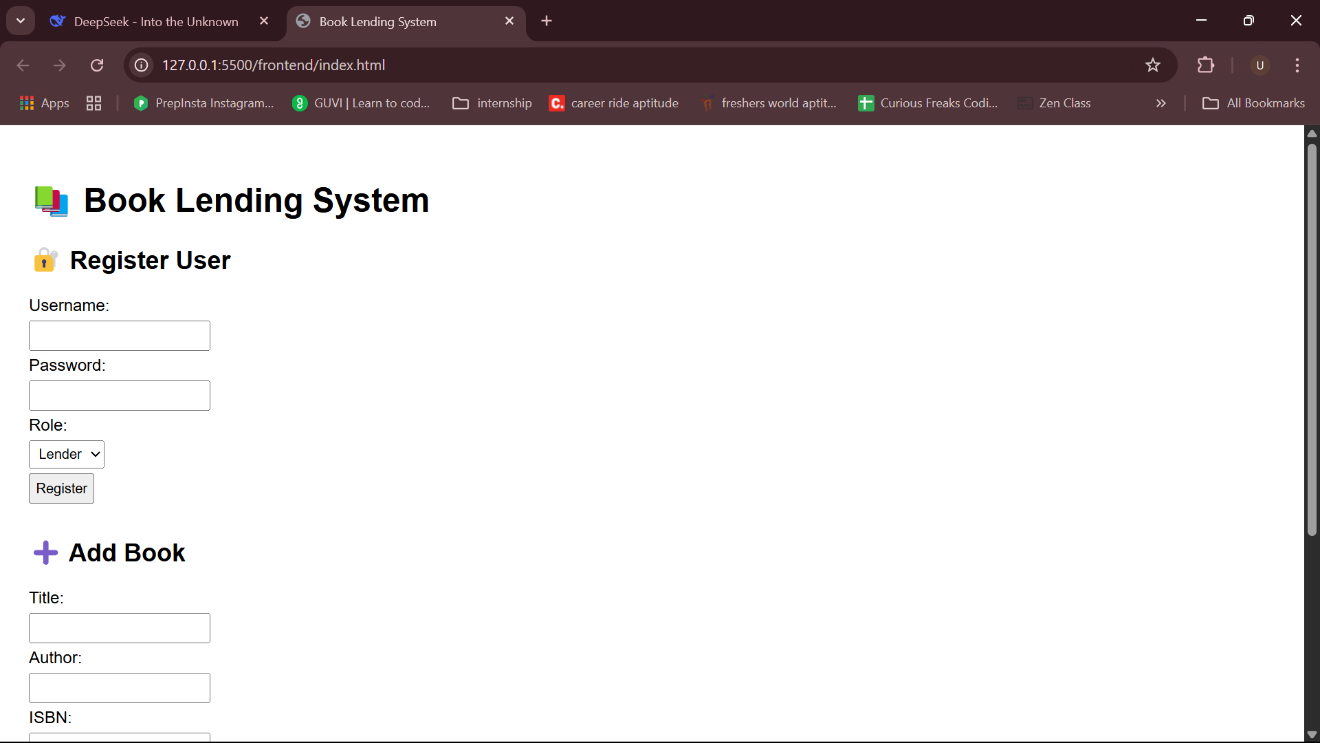
* main.rs = router + server
* auth.rs = user registration logic
* books.rs = book addition logic
* db.rs = init logic with schema runner

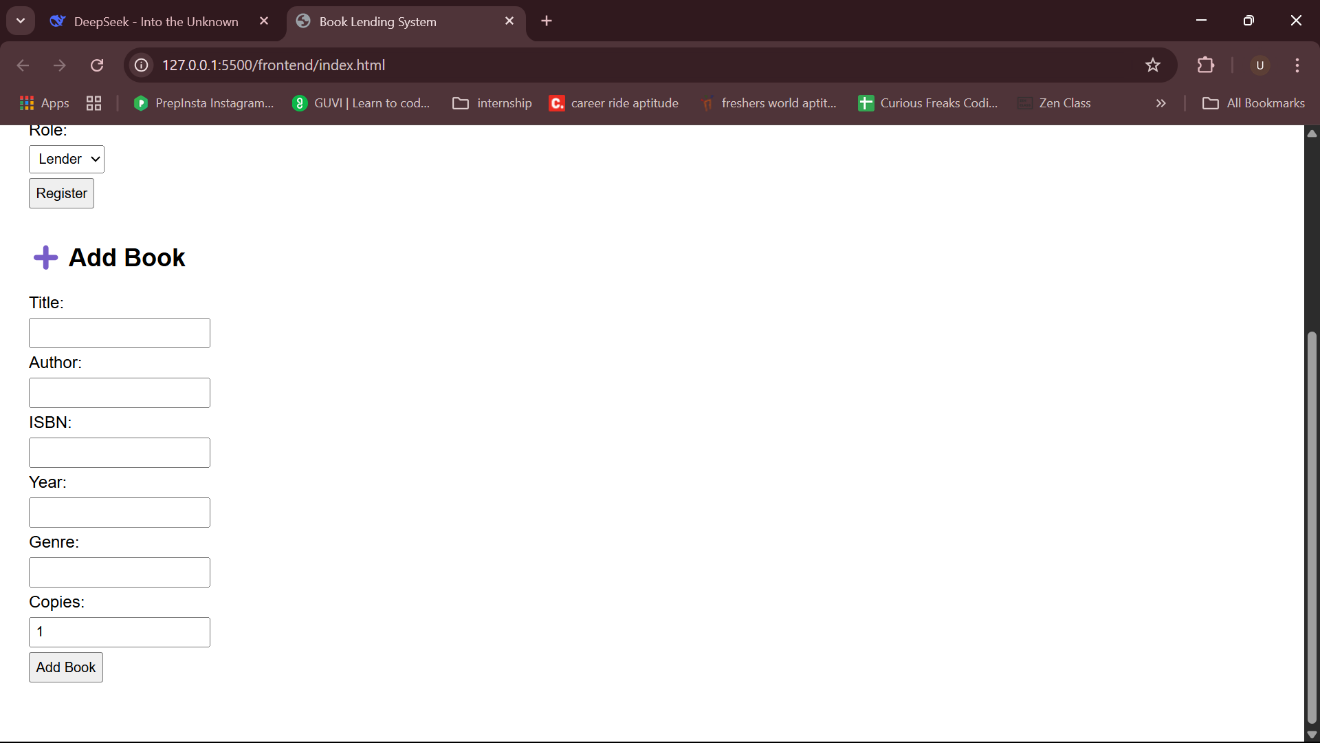
**Highlights:**

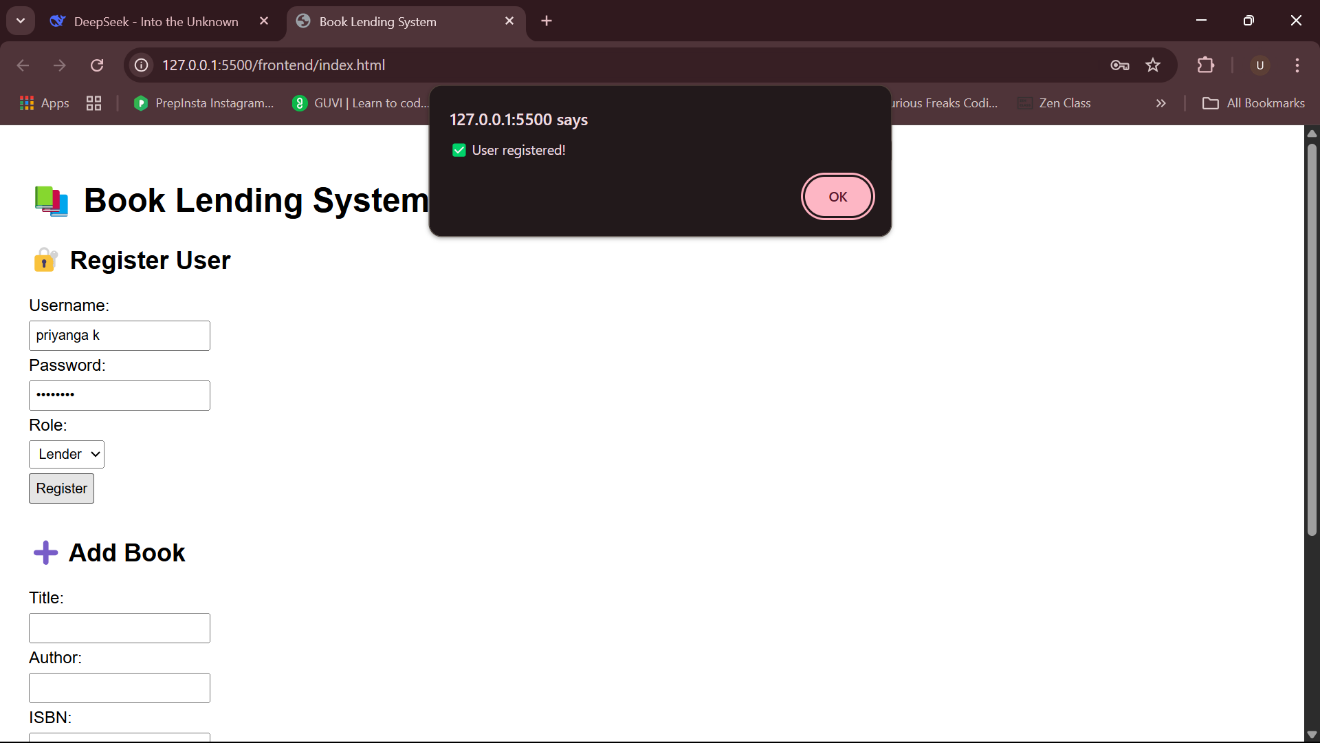
* Proper use of Rust async runtime (tokio)
* Logs at key actions (registration, book addition)
* Error handling through Result<T, E> and HTTP fallback responses
* Frontend logs form actions and handles alerts gracefully

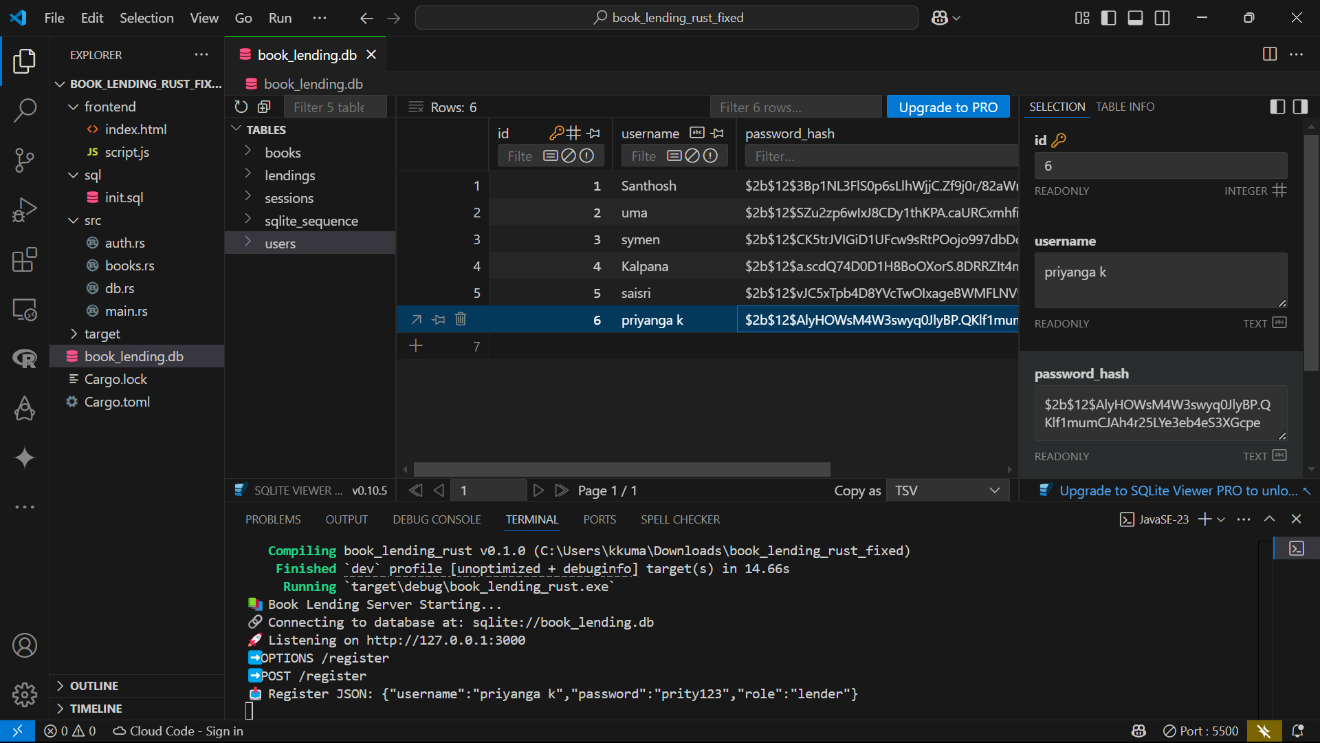
**Screenshots of implementation**

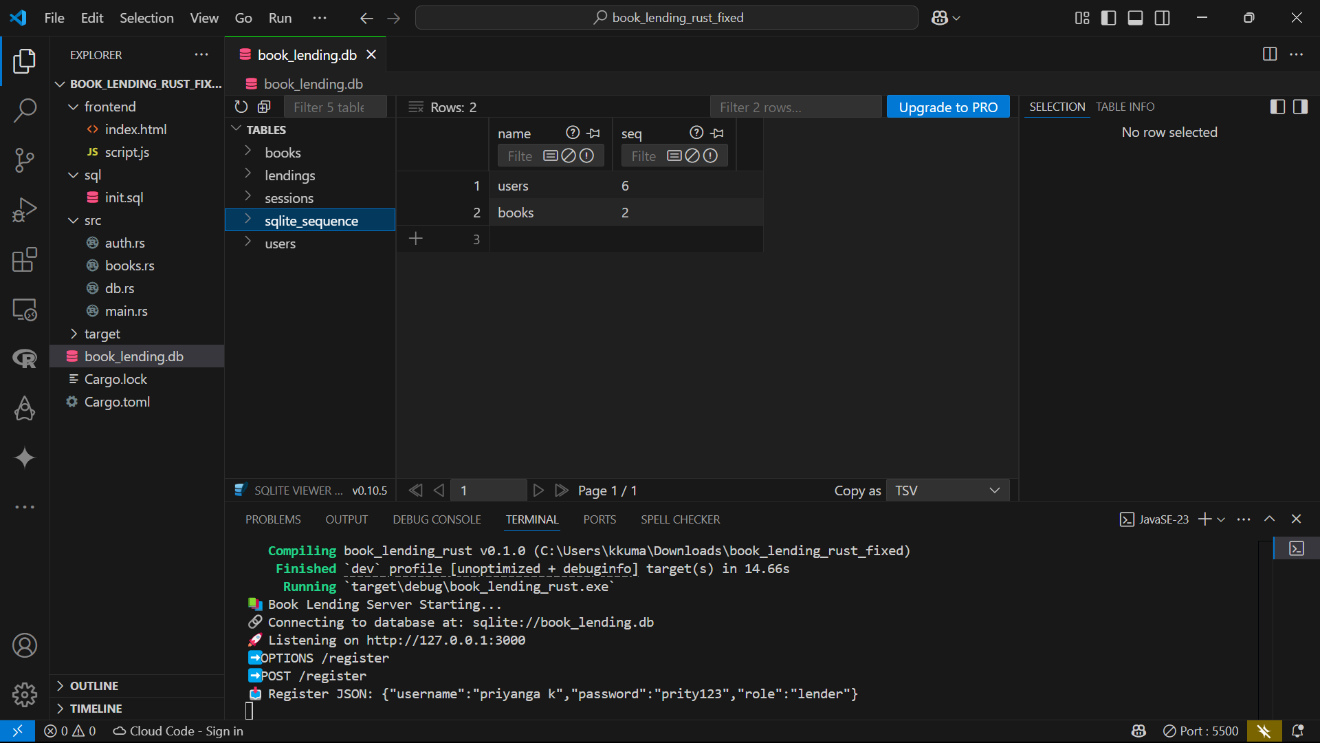


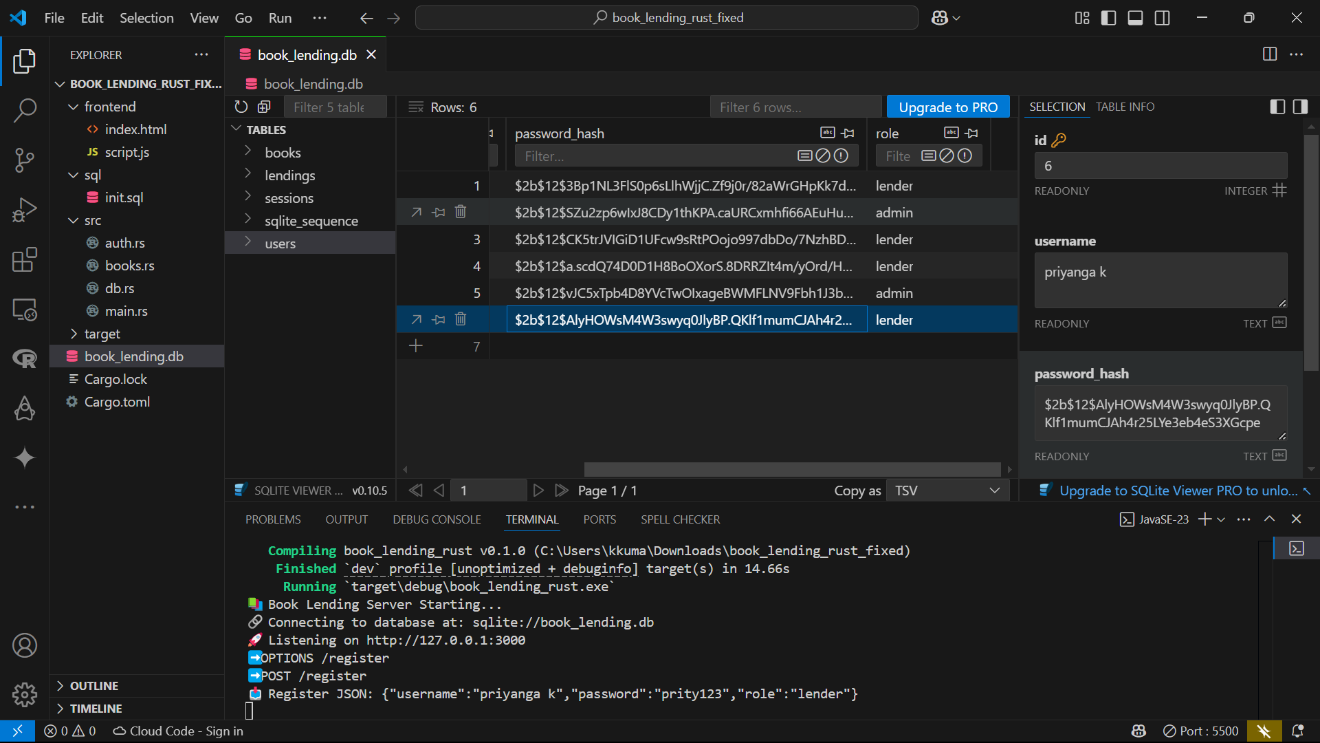


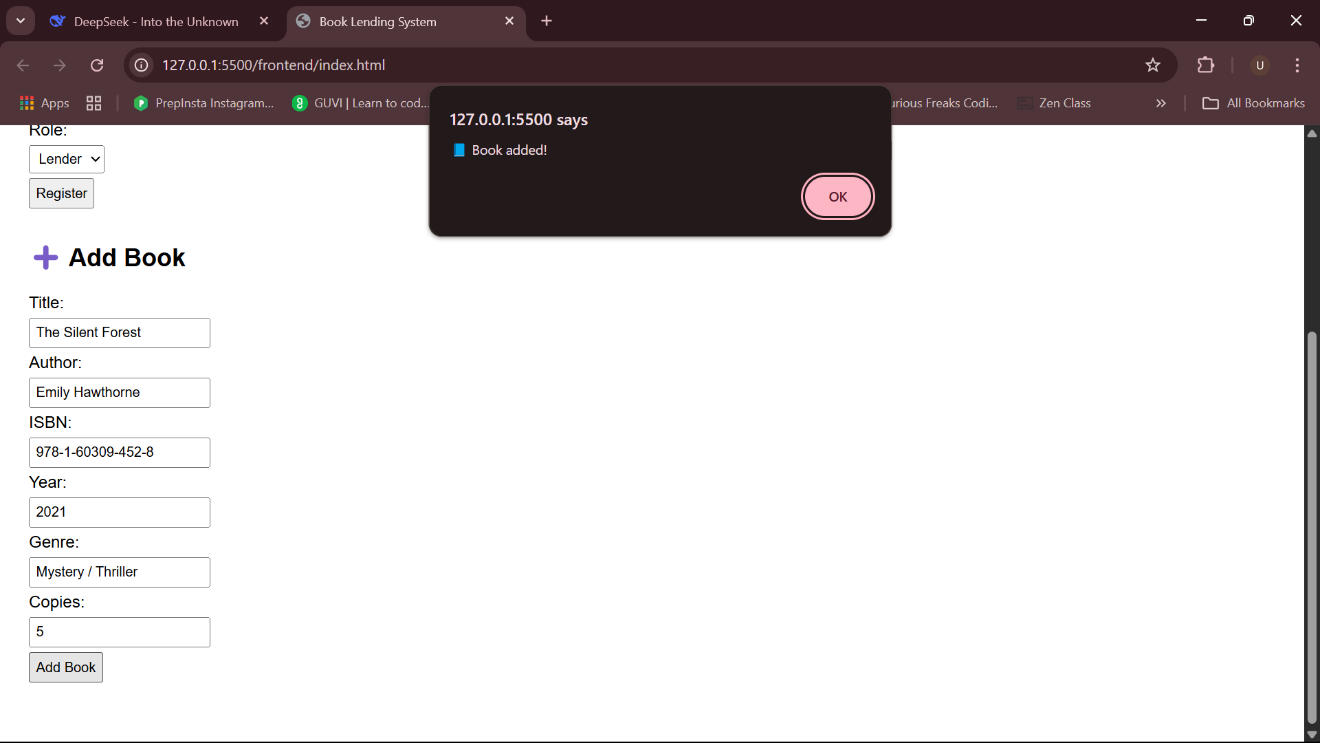


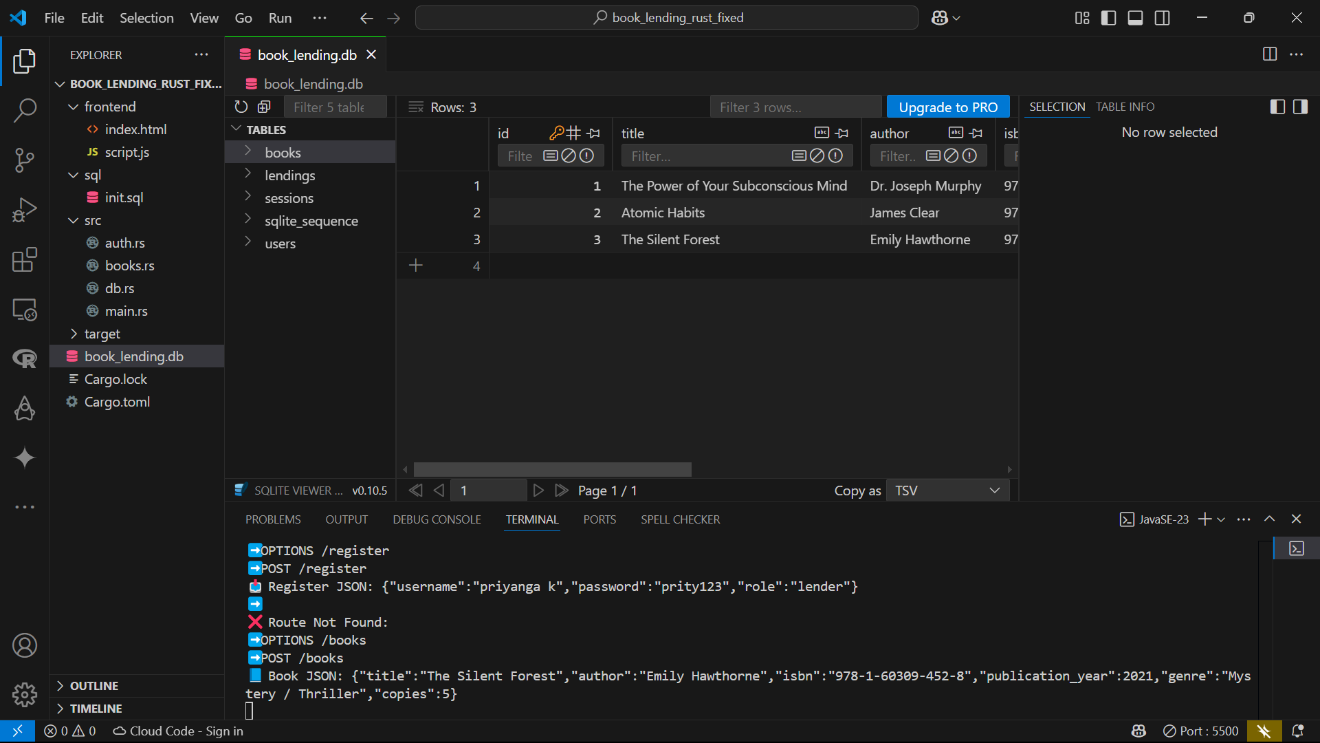


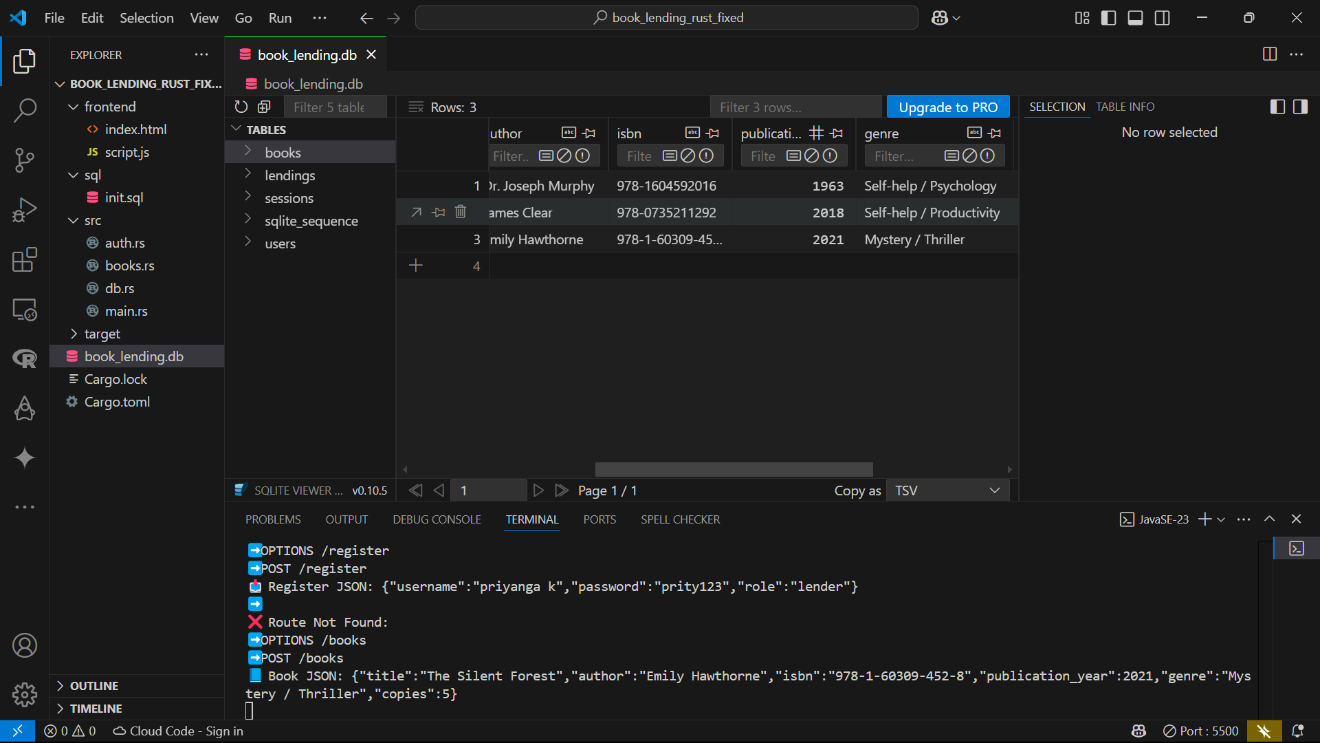


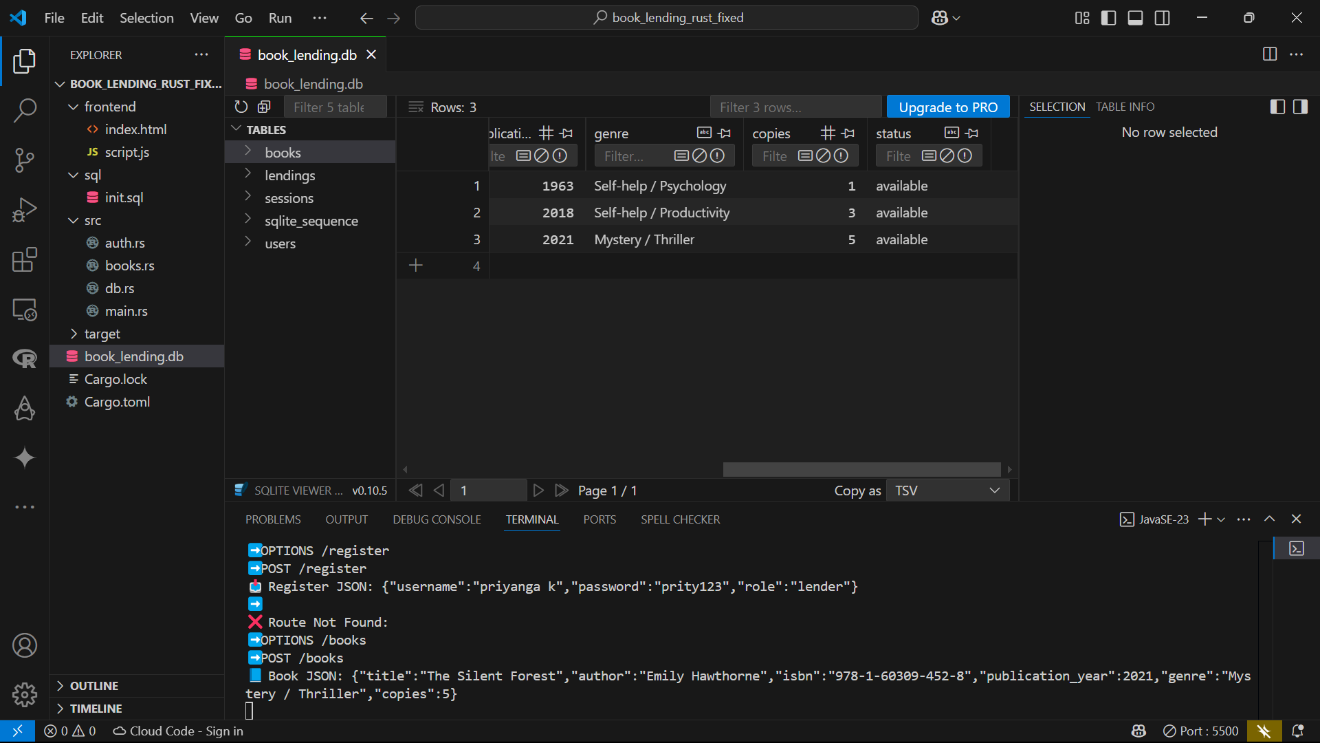








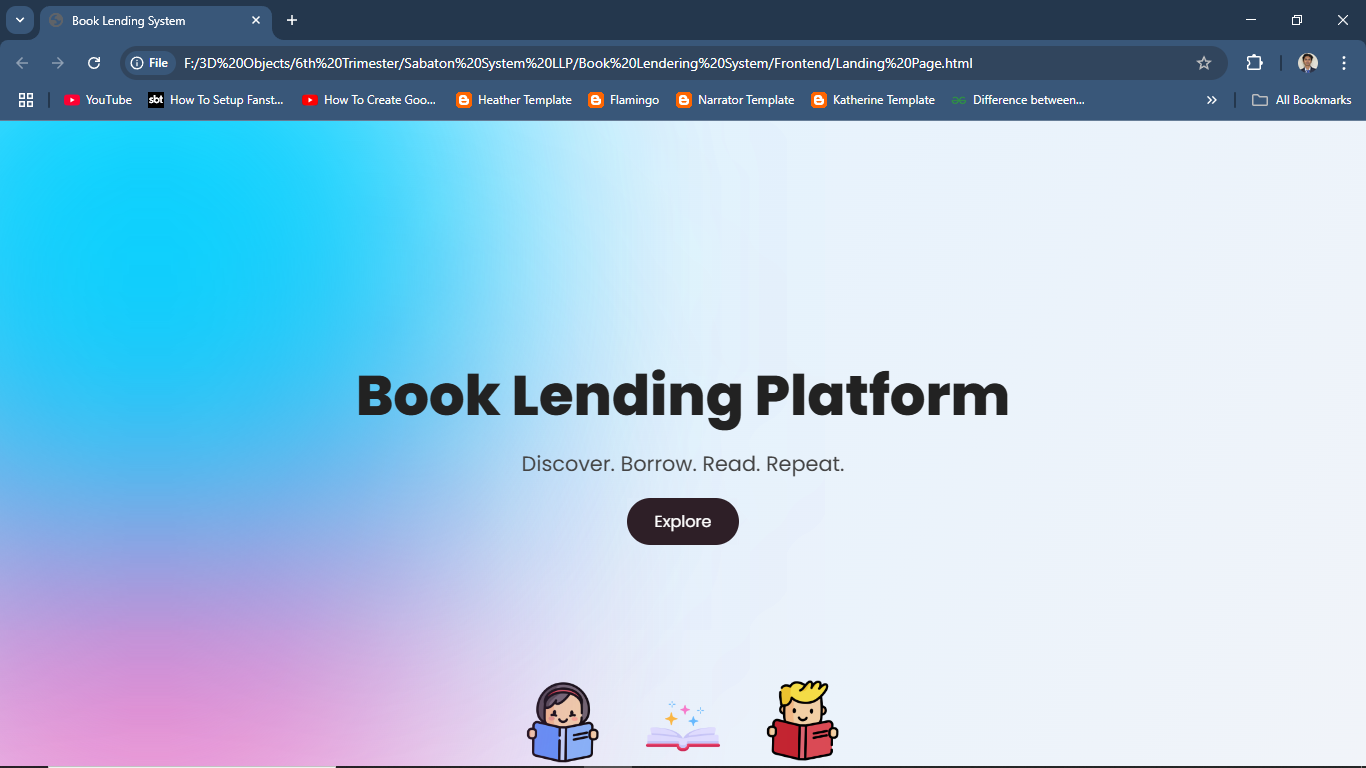


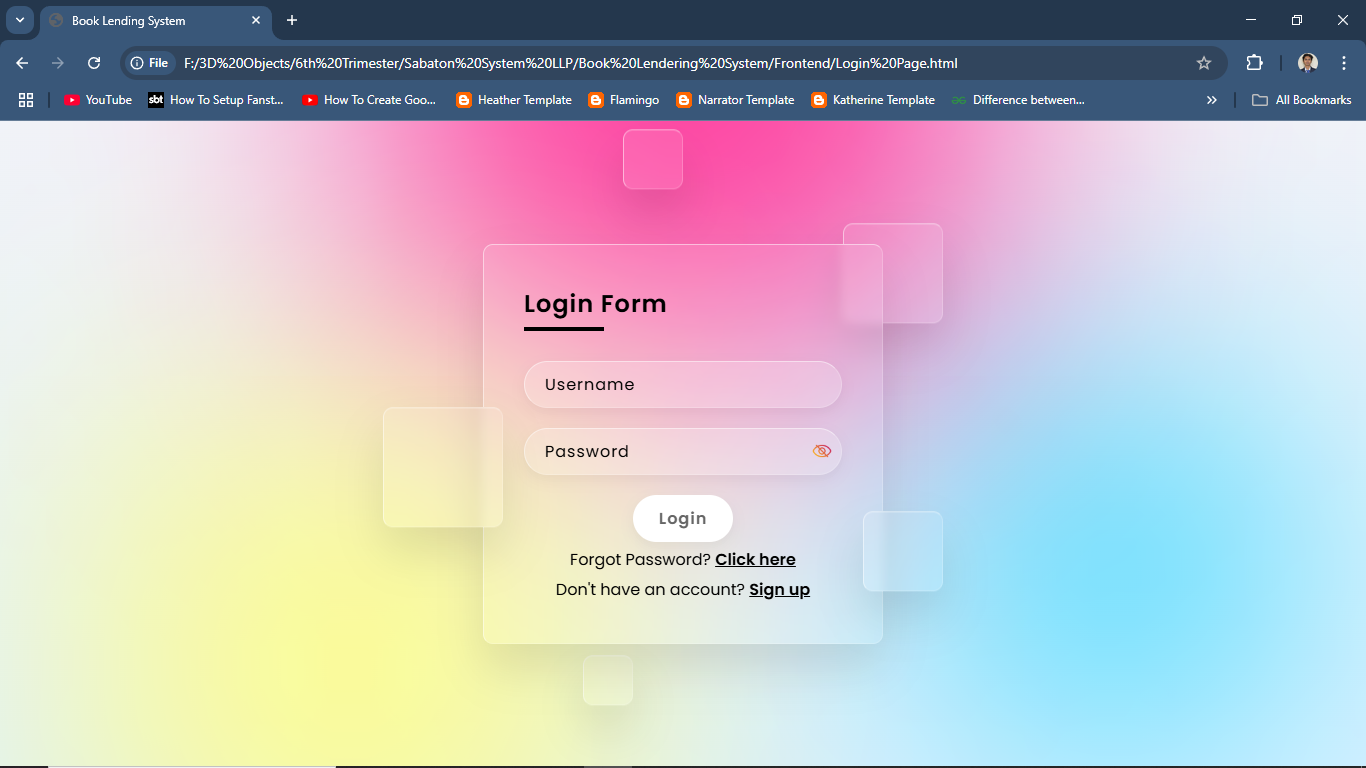


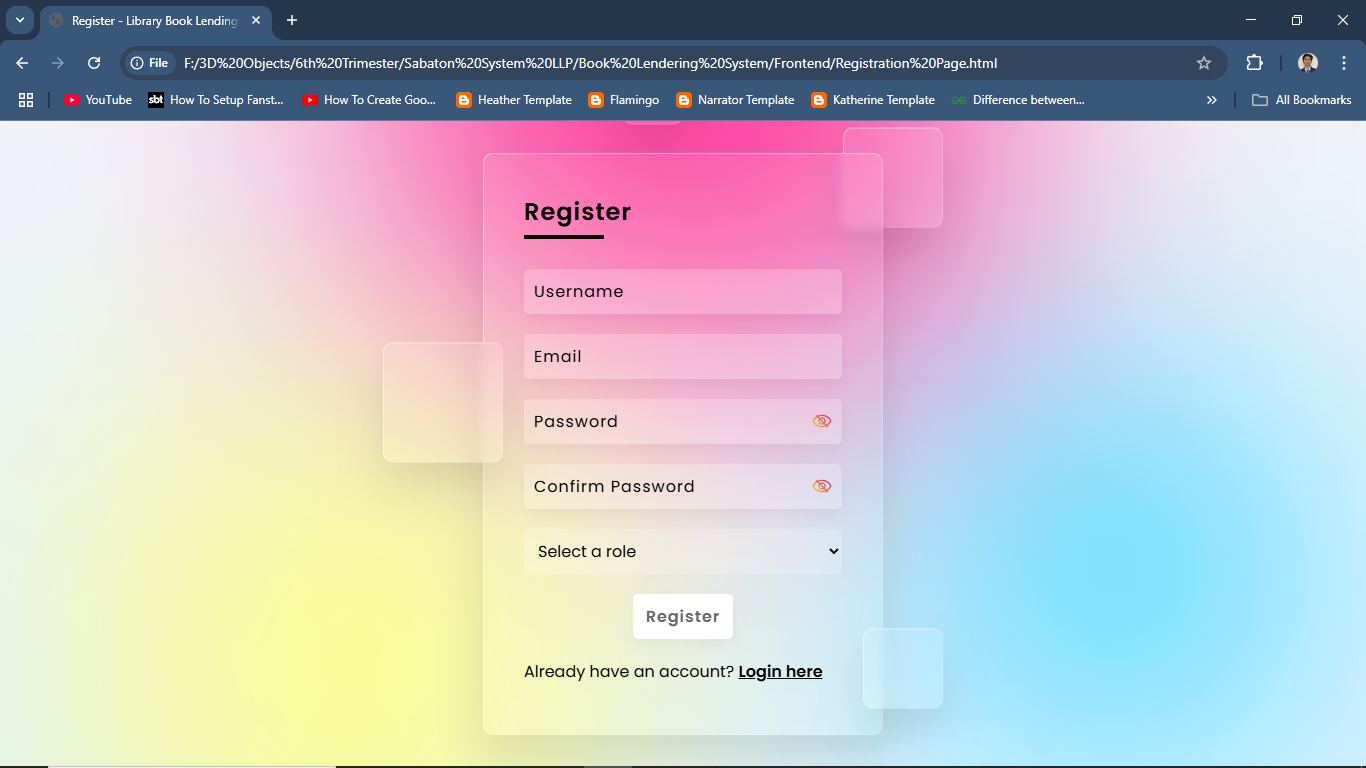
# **Implementation Video**

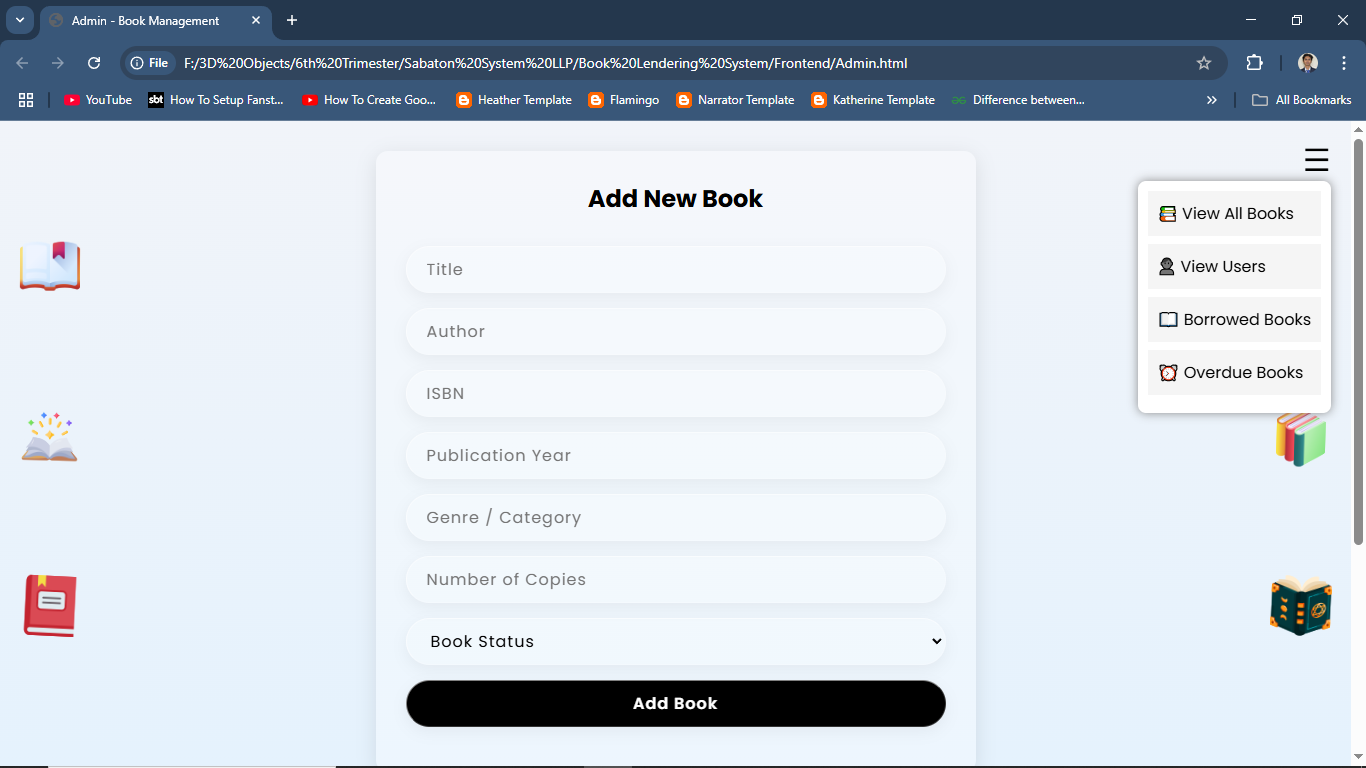
<https://drive.google.com/file/d/1TKtqNm3DSAEWUxY5U33t8-p9wMf9WD9Q/view?usp=sharing>

**Awesome GUI implemented (Backend implementation in progress).**

****

****

****

****