FULL STACK DEVELOPMENT WITH MERN

PROJECT DOCUMENTATION BOOK SELLING WEBPAGE

**CONTENT**

|  |  |
| --- | --- |
| **S.NO** | **TITLE** |
| **1** | **INTRODUCTION** |
| **2** | **PROJECT OVERVIEW** |
| **3** | **ARCHITECTURE** |
| **4** | **SETUP INSTRUCTION** |
| **5** | **FOLDER STRUCTURE** |
| **6** | **RUNNING THE APPLICATIION** |
| **7** | **API DOCUMENTATION** |
| **8** | **AUTHENTICATIION** |
| **9** | **USER INTERFACE** |
| **10** | **TESTING** |
| **11** | **SCREENSHOTS AND DEMO** |
| **12** | **KNOWN ISSUES** |
| **13** | **FUTURE ENHANCEMENTS** |
| **14** | **CONCLUSION** |

**FULL STACK DEVELOPMENT WITH MERN**

* **PROJECT TITLE: BOOK SELLING WEBPAGE**
  + **TEAM MEMBERS:**

1. **TRISHA.R**
2. **PRIYA DHARSHINI.L**
3. **PREETHA.M**
4. **SWATHI.G**
5. **MUTHU LAKSHMI.S**

**BOOK - STORE WEB APP**

# INTRODUCTION

Discover a world of stories, knowledge, and inspiration at your fingertips. Our online bookstore offers an extensive collection of books across all genres—fiction, non- fiction, academic, self-help, and more. Whether you're a bookworm, a student, or just looking for your next great read, we’ve got something for everyone.

Explore top-selling titles, rare finds, and personalized recommendations. With seamless browsing, secure payments, and fast delivery, we ensure an exceptional shopping experience.

# PROJECT OVERVIEW

1. **Purpose:**
   * The purpose of the Book Selling Webpage project is to create a user-friendly and efficient online platform for book enthusiasts to buy, sell, and explore a wide range of books. By leveraging the MERN (MongoDB, Express.js, React.js, and Node.js) stack, the project aims to streamline the process of book transactions and provide a seamless experience for buyers, sellers, and administrators.
   * The platform aspires bridge the gap between book sellers and buyers through a convenient digital interface.
   * Offer personalized services like wish lists, order tracking, and seller dashboards.
   * Enhance accessibility to books by supporting diverse genres and categories.

# Features:

The platform is designed with the following features in mind:

## Ease of Use

Create an intuitive and visually appealing user interface that ensures a smooth navigation experience for all users, including buyers, sellers, and administrators.

## Efficient Book Management

Provide tools for sellers to manage their inventories, such as adding, editing, and removing books, while also enabling buyers to search, browse, and purchase books effortlessly.

## Seamless Transactions

Facilitate secure and straightforward transactions between buyers and sellers, ensuring proper order management and tracking.

## Dynamic Features

Offer features like personalized wish lists, seller performance tracking, and real-time order updates to enhance user engagement and satisfaction.

## Scalability

Develop the platform with a modular architecture to accommodate future growth and additional features like book recommendations, reviews, and advanced search capabilities.

## Community Building

Promote a vibrant community of book lovers by connecting individuals with similar interests through a shared platform for discovering and purchasing books.

# ARCHITECTURE

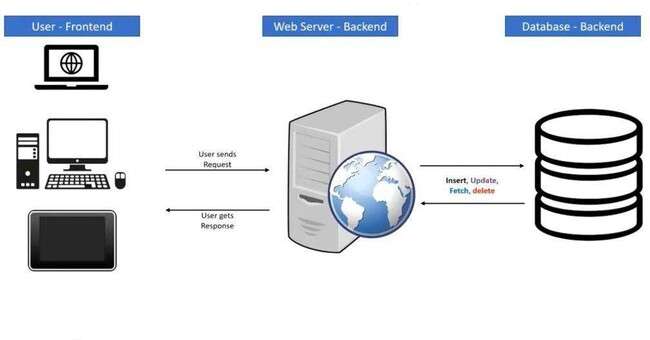


Fig. 1: Full Stack Architecture.

The frontend, also known as the client-side, is the layer of the application that users interact with directly. It is responsible for presenting data, handling user input, and providing a user-friendly interface. The frontend is built using programming languages such as HTML, CSS, and JavaScript, and frameworks like React, Angular, or Vue.js. The frontend developer's primary focus is on creating a visually appealing and functional user interface that provides a seamless user experience.

The backend, also known as the server-side, is the layer of the application that handles business logic, data storage, and retrieval. It receives requests from the frontend, processes them, and sends responses back. The backend is built using programming languages such as Java, Python, Ruby, PHP, and frameworks like Node.js, Django, or Ruby on Rails. The backend developer's primary focus is on creating a robust and scalable server-side application that can handle large volumes of data and traffic.

The database is the layer of the application that is responsible for storing and managing data. It provides a centralized repository for data storage and retrieval, allowing the backend to access and manipulate data as needed. There are two primary types of databases: relational databases, such as MySQL, PostgreSQL, or Microsoft

SQL Server, and NoSQL databases, such as MongoDB, Cassandra, or Redis. The database plays a critical role in the full stack application, as it provides the foundation for data storage and retrieval.

## Frontend:

React is used in front-end development to build user interfaces and add client- side functionality, enabling the creation of dynamic pages, interactive elements, and features for websites. React is a popular JavaScript library developed by Facebook in 2013 for building web app UIs. It enables developers to create reusable UI components and simplifies building complex interfaces. React uses a virtual DOM for efficient state management and fast updates.

## Component-based architecture:

With React, you can create reusable user interface (UI) components that can be swiftly combined to create intricate UIs.

## JSX:

By allowing you to write HTML-like code in JavaScript files, the JSX syntactic extension for JavaScript makes it easier to write and understand code.

React supports a wide range of libraries and frameworks, including Redux, React Router, and Material UI. Additionally, it is quite expandable. React is a robust and flexible framework that can be used to build user interfaces for internet applications.

## Backend:

The backend of the **Book Selling Webpage** is designed to handle requests, manage business logic, and interact with the database. It is built using **Node.js** and **Express.js** with MongoDB as the database.

## Backend Features

The backend architecture using Node.js and Express.js includes:

## Server Setup:

Node.js runs JavaScript on the server, with Express.js handling routing and middleware**.**

## Routing:

Manages API endpoints for handling requests (GET, POST, etc.).

## Middleware:

Processes requests (e.g., authentication, body parsers).

## Error Handling:

Centralized error handling for graceful failure.

## Authentication & Authorization:

Manages user sessions with JWT or OAuth.

## Environment Configuration:

Stores sensitive data in environment variables.

## Logging & Monitoring:

Tracks app activity and performance.

## Scalability:

Deployed on cloud platforms with load balancing for traffic handling.

## Database:

* + **Schema Design**

In MongoDB, data is stored in collections (similar to tables in relational databases) as documents (like rows in SQL). A schema defines the structure of these documents. In Node.js, Mongoose is commonly used to define and enforce schemas. A schema can specify fields, types, constraints (like required or unique), and defaults for data.

## MongoDB Interactions

Mongoose provides methods to perform basic CRUD (Create, Read, Update, Delete) operations with MongoDB.

**Create:** Add new records to a collection.

**Read:** Retrieve documents using queries (e.g., find users by email). **Update:** Modify existing documents (e.g., updating a user’s password). **Delete:** Remove documents from the collection (e.g. delete a user by email).

These operations are done through the Mongoose model, which acts as an interface to interact with the collection based on the schema.

## Advanced MongoDB Features

MongoDB offers several advanced features that improve performance and capabilities:

**Indexes:** Improve query speed by creating indexes on fields frequently queried (e.g., email).

**Aggregation:** Perform complex queries to manipulate or analyze data, such as counting the number of documents or grouping them by specific fields.

**Population:** Reference documents from other collections (similar to SQL joins), allowing you to fetch related data in one query.

## Schema Validation & Middleware

MongoDB and Mongoose allow for data validation and pre/post-processing using middleware:

**Validation:** Ensures data integrity, for example, ensuring an email is unique or a password is required.

**Middleware:** Functions that run before or after certain actions. For instance, hashing passwords before saving them or logging actions after saving data.

## Connecting to MongoDB

To interact with a MongoDB database, you need to establish a connection using Mongoose. Once connected, your Node.js application can perform the necessary operations on your MongoDB database.

## Error Handling & Optimizations

MongoDB provides error handling for cases like validation failures or query errors. To optimize performance, techniques like indexing, pagination, and caching are often used**.**

# SETUP INSTRUCTION

## ER Diagram

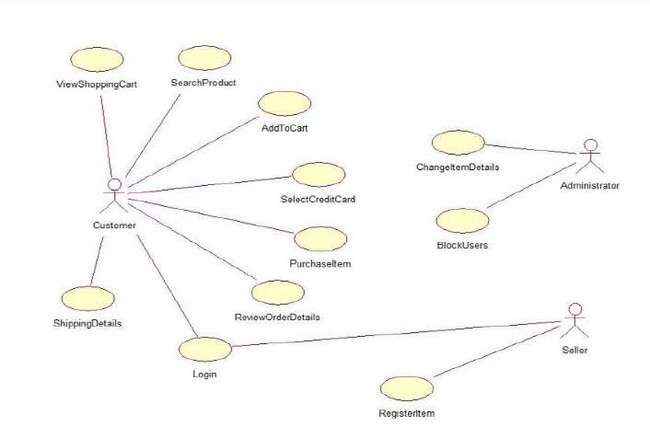


Fig. 2: ER Diagram of Book Selling Webpage

## User-Book Relationship:

Type: Many-to-Many (M:M). A single user can read or interact with many books, and a single book can be accessed by many users.

Implementation: Introduce an intermediate entity, "Interaction", with foreign keys to both User and Book tables. This table could store additional information like reading progress, reviews, or ratings.

## Book-Inventory Relationship:

Type: One-to-Many (1:M). Each book can have multiple copies in inventory, but each copy belongs to one book.

Implementation: Maintain a separate Inventory table with fields like BookID (foreign key), quantity, location, and condition.

## User-Order Relationship:

Type: One-to-Many (1:M). A single user can place multiple orders, but each order belongs to one user.

Implementation: Keep the UserID foreign key in the Order table to track user purchase history.

## Additional Relationships:

Book-Author Relationship: Many-to-Many (M:M). A book can have multiple authors, and an author can write multiple books. (Similar to User-Book, use an intermediate "WrittenBy" table)

Book-Genre Relationship: Many-to-Many (M:M). A book can belong to multiple genres, and a genre can have many books. (Similar to User-Book, use an intermediate "CategorizedAs" table)

Review-User Relationship: Many-to-One (M:1). A review is written by one user, but a user can write many reviews. (Keep UserID as a foreign key in the Review table)

## Key Features:

* **User Registration and Authentication:** Allow users to register accounts securely, log in, and authenticate their identity to access the book store platform.
* **Book Listings:** Display a comprehensive list of available books with details such as title, author, genre, description, price, and availability status.
* **Book Selection:** Provide users with options to select their preferred books based on factors like genre, author, ratings, and popularity.
* **Purchase Process:** Allow users to add books to their cart, specify quantities, and complete purchases securely. Upon successful completion, an order is generated, and the inventory is updated accordingly.
* **Order Confirmation:** Provide users with a confirmation page or notification containing details of their order, including book information, total price, and order ID.
* **Order History:** Allow users to view their past and current orders, providing options to track shipments, review purchased books, and rate their shopping experience.
* **Organizer Dashboard:** Offer administrators an interface to manage book listings, inventory levels, user accounts, orders, and other platform-related activities.
* **Create Item:** Organizer can create items and add new items and he can get the items and he can update items.
* **Admin Dashboard:** Offer administrators an interface to manage book listings, inventory levels, user accounts, orders, and other platform-related activities. Manage the users and organizers.
* **Reporting and Analytics:** Generate reports and analytics on book sales, popular genres, user demographics, and other relevant metrics to gain insights into platform usage and performance.
* **Integration with External APIs:** Integrate with third-party APIs for services like payment processing, shipping logistics, and book recommendations to enhance the functionality and user experience of the book store platform.

## Prerequisites

To develop a full-stack Book Store App using React js, Node.js,Express js and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

**Node.js and npm:** Install Node.js, which includes npm (Node Package Manager), on your development machine. Node.js is required to run JavaScript on the server side.

* + Download: <https://nodejs.org/en/download/>
  + Installation instructions:<https://nodejs.org/en/download/package-manager/>

**MongoDB:** Set up a MongoDB database to store hotel and booking information.

Install MongoDB locally or use a cloud-based MongoDB service.

* + Download:<https://www.mongodb.com/try/download/community>
  + Installation instructions: <https://docs.mongodb.com/manual/installation/>

**Express.js:** Express.js is a web application framework for Node.js. Install Express.js to handle server-side routing, middleware, and API development.

* + Installation: Open your command prompt or terminal and run the following command: **npm install express**

**React js:** React is a JavaScript library for building client-side applications. And Creating Single Page Web-Application

**Getting Started:** Create React App is an officially supported way to create single-page React applications. It offers a modern build setup with no configuration.

## Quik Start

* + npm create vite@latest
  + cd my-app
  + npm install
  + npm run dev

If you've previously installed create-react-app globally via npm install -g create- react-app, we recommend you uninstall the package using npm uninstall -g create-react- app or yarn global remove create-react-app to ensure that npx always uses the latest version.

## Create a new React project:

* + Choose or create a directory where you want to set up your React project.
  + Open your terminal or command prompt.
  + Navigate to the selected directory using the cd command.
  + Create a new React project by running the following command: npx create- react-app your-app-name. Wait for the project to be created:
  + This command will generate the basic project structure and install the necessary dependencies

## Navigate into the project directory:

* + After the project creation is complete, navigate into the project directory by running the following command**: cd your-app-name**

## Start the development server:

* + To launch the development server and see your React app in the browser, run the following command: **npm run dev**
  + The npm start will compile your app and start the development server.
  + Open your web browser and navigate to https://localhost:5173 to see your React app.

You have successfully set up React on your machine and created a new React project. You can now start building your app by modifying the generated project files in the src directory.

Please note that these instructions provide a basic setup for React. You can explore more ad- vanced configurations and features by referring to the official React documentation: <https://react.dev/>

**HTML, CSS, and JavaScript:** Basic knowledge of HTML for creating the structure of your app, CSS for styling, and JavaScript for client-side interactivity is essential.

**Database Connectivity:** Use a MongoDB driver or an Object-Document Mapping (ODM) library like Mongoose to connect your Node.js server with the MongoDB database and perform CRUD (Create, Read, Update, Delete) operations.

**Front-end Library:** Utilize React to build the user-facing part of the application, including products listings, booking forms, and user interfaces for the admin dashboard.

**Version Control**: Use Git for version control, enabling collaboration and tracking changes throughout the development process. Platforms like GitHub or Bitbucket can host your repository.

* Git: Download and installation instructions can be found at: [https://git-](https://git-scm.com/downloads) [scm.com/downloads](https://git-scm.com/downloads)

**Development Environment:** Choose a code editor or Integrated Development Environment (IDE) that suits your preferences, such as Visual Studio Code, Sublime Text, or WebStorm.

* Visual Studio Code: Download from <https://code.visualstudio.com/download>
* Sublime Text: Download from <https://www.sublimetext.com/download>
* WebStorm: Download from <https://www.jetbrains.com/webstorm/download>

1. **Installation**

**Step-by-Step Guide for Installation:**

* + **Clone the Repository:**

Open a terminal and navigate to the folder where you want to clone the project.

## Run the following command to clone the repository:

* + - **git clone:** https: //github.com/priya-dharshini143/NM- Book-Store.git

## Navigate to Project Directory:

* + - After cloning, navigate into the project folder:
      * cd backend
      * cd frontend

## Install Node.js Dependencies:

* + - Ensure Node.js and npm are installed. You can check by running:
      * node -v
      * npm -v

If Node.js is not installed, download and install it from Node.js official website.

* + - Install the project dependencies listed in package.json:
      * npm install

## Set Up Environment Variables:

* + - Add necessary environment variables such as database URI, port, and API keys.
    - Example:**DB\_URI=**mongodb+srv://pd4008208:RIYA143@cl uster0.pqg7g.mongodb.net/?retryWrites=true&w=majority&a ppName=Cluster0

## Start the Application:

* + - Once dependencies are installed and environment variables are set, start the application:
      * npm start - This will run the server,
      * npm run dev - This will run the client, and you should see the application running at http://localhost:5173.

## Verify Setup:

Open your browser or use Postman to test the API and ensure everything is working properly.

That's it! Your Node.js and MongoDB environment should now be set up and ready for development.

# FOLDER STRUCTURE

## CLIENT:

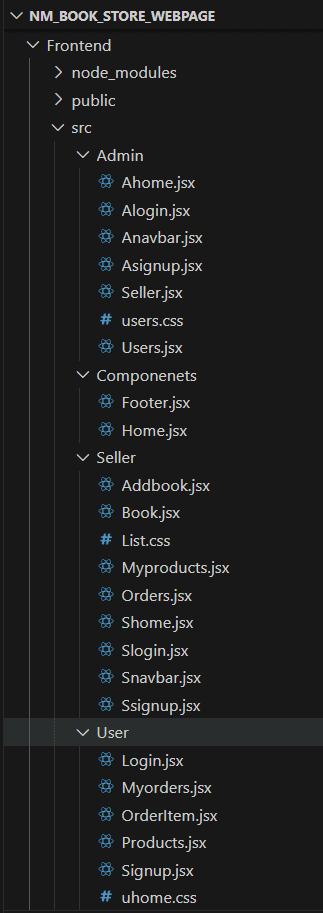


Fig. 3. 1: Frontend Project Structure

## Main Project Folder

1. **Frontend**:
   * This seems to be the root folder for the client-side (React) code

## Subfolders

1. **node\_modules**:
   * Contains dependencies installed via npm or yarn.

## public:

* + Likely contains static assets like images or fonts and the base index.html file for rendering the React app.

1. **src**:
   * Main folder for all source code.

## Structure Within src Admin Folder

* Components specific to admin functionalities:
  + **Ahome.jsx**: Admin dashboard/home.
  + **Alogin.jsx**: Login page for the admin.
  + **Anavbar.jsx**: Admin-specific navigation bar.
  + **Asignup.jsx**: Signup page for admin accounts.
  + **Seller.jsx**: Likely the main component for managing sellers.
  + **users.css**: CSS styles for admin user-related pages.
  + **Users.jsx**: Page or component for managing users.

## Components Folder

* Shared components used across multiple parts of the app:
  + **Footer.jsx**: Footer of the webpage.
  + **Home.jsx**: Homepage component, possibly common for all users.

## Seller Folder

* Components for seller functionalities:
  + **Addbook.jsx**: Form or page to add books for sale.
  + **Book.jsx**: Likely a reusable book display component.
  + **List.css**: CSS styles for seller-related lists.
  + **Myproducts.jsx**: Displays products/books added by the seller.
  + **Orders.jsx**: Displays or manages orders for sellers.
  + **Shome.jsx**: Seller-specific homepage.
  + **Slogin.jsx**: Login page for sellers.
  + **Snavbar.jsx**: Navigation bar for seller pages.
  + **Signup.jsx**: Signup page for sellers.

## User Folder

* Components for customer functionalities:
  + **Login.jsx**: Login page for users.
  + **Myorders.jsx**: Page showing user's past orders.
  + **OrderItem.jsx**: Individual order item component.
  + **Products.jsx**: Product display page for customers.
  + **Signup.jsx**: Signup page for users.
  + **Uhome.jsx**: User-specific homepage.
  + **Uitem.jsx**: Represents an individual item (product/book).
  + **Unavbar.jsx**: Navigation bar for user pages.
  + **Wishlist.jsx**: Wishlist page for users.
  + **uhome.css**: CSS styles for user-specific pages.

## Other Files in src

* **App.css** & **App.jsx**: Likely the main component and styles that handle the layout or routing for the entire app.
* **index.css**: Global styles for the app.
* **main.jsx**: Main entry point for the React application.

## Project Configuration Files

* **.eslintrc.cjs**: ESLint configuration file to enforce code standards and styles.
* **.gitignore**: Specifies which files/folders Git should ignore (e.g., node\_modules, logs).
* **index.html**: Base HTML template for the React app.

## package-lock.json and package.json:

* + Define the project's dependencies and scripts.
* **vite.config.js**: Configuration file for Vite, the development build tool.

## Key Observations

1. **Role Separation**:
   * The project has clear separation for **admin**, **seller**, and **user**

functionalities.

## Reusability:

* + Shared components (e.g., Navbar, Footer, Book) are neatly organized, enhancing modularity.

## Styling:

* + Dedicated CSS files for different roles and components.

## Build Tool:

* + The project uses **Vite**, a modern and fast build tool for React applications.

## SERVER:



Fig. 3. 2: Backend Project Structure

## Backend

* + This folder contains all server-side code and configurations.

## Subfolders db

* + Likely stands for "database" or "data access" and contains modules for different entities.

## Admin

* + **Admin.js**:
    - Handles backend operations related to admin functionalities, such as managing users, books, or orders.

## Seller

* + **Additem.js**:
    - Manages adding new items (e.g., books or products) by sellers.

## Sellers.js:

* + - Handles backend operations related to seller accounts or functionalities.

## Users

* + **myorders.js**:
    - Manages backend logic for retrieving and handling user orders.

## userschema.js:

* + - Defines the schema/model for user data, likely using a database ORM (e.g., Mongoose for MongoDB).

## Wishlist.js:

* + - Backend functionality for managing a user's wishlist.

## Other files in db:

* + **config.js**:
    - Likely contains configurations for the database connection, like credentials, URLs, or environment variables.

## node\_modules

* + Contains all dependencies installed via npm or yarn for the backend.

## uploads

* + A folder for storing uploaded files, such as book images or user profile pictures.

**Root Files**

**package.json and package-lock.json**

* + **package.json**:
    - Lists project dependencies, scripts, and metadata.

## package-lock.json:

* + - A lock file that ensures consistency in dependency versions.

## server.js

* + The main entry point for the backend server.
    - Sets up the server (e.g., Express.js).
    - Defines routes and middleware.
    - Likely initializes the database connection.

## Key Observations

1. **Modularity**:
   * Backend code is split into logical modules for **Admin**, **Seller**, and **Users**, ensuring organized and maintainable code.

## Database Interaction:

* + The presence of userschema.js suggests a schema-based approach, possibly with MongoDB (Mongoose) or similar.

## Uploads Handling:

* + The uploads folder hints at file management for storing user-generated content (e.g., images).

## Scalable Architecture:

* + Each role (Admin, Seller, User) has separate modules, making it easy to add or modify features independently.

## Server Setup:

* + The server.js is crucial for initializing the backend, connecting to the database, and routing API calls.

# RUNNING THE APPLICATION

Provide commands to start the frontend and backend servers locally.

* + - Frontend: **npm run dev** in the client directory.
    - Backend: **npm start** in the server directory.

# API DOCUMENTATION

**Backend > db > Admin > Admin.js**

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({ name:String,

email: String, password: String, userId:{

type:mongoose.Schema.Types.ObjectId, ref:"admin",

}

})

module.exports =mongoose.model('admin',UserSchema)

**Backend > db > Seller > Additem.js**

const mongoose = require('mongoose');

const bookSchema = new mongoose.Schema({ title: {

type: String, required: true,

},

author: { type: String,

required: true,

},

genre: { type: String,

required: true,

},

// itemtype:String, itemImage:String, description:String, price:String,

userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }, userName:String,

})

module.exports =mongoose.model('books',bookSchema)

**Backend > db > Seller > Sellers.js**

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({ name:String,

email: String, password: String, userId:{

type:mongoose.Schema.Types.ObjectId, ref:"vendor",

}

})

module.exports =mongoose.model('Seller',UserSchema)

**Backend > Users > myorders.js**

const mongoose = require('mongoose');

const bookschema = new mongoose.Schema({ flatno:String,

pincode:String, city:String, state:String, totalamount:String,

seller:String, sellerId:String, booktitle:String, bookauthor:String, bookgenre:String, itemImage:String, description:String,

userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }, userName:String,

BookingDate: {

type: String, // Store dates as strings

default: () => new Date().toLocaleDateString('hi-IN') // Set the default value to the current date in "MM/DD/YYYY" format

},

Delivery: {

type: String, // Store dates as strings default: () => {

// Set the default value to the current date + 7 days in "MM/DD/YYYY" format const currentDate = new Date();

currentDate.setDate(currentDate.getDate() + 7); // Add 7 days const day = currentDate.getDate();

const month = currentDate.getMonth() + 1; // Month is zero-based, so add 1 const year = currentDate.getFullYear();

// Format the date in "MM/DD/YYYY" format const formattedDate = `${month}/${day}/${year}`;

return formattedDate;

}

}

})

module.exports =mongoose.model('myorders',bookschema)

**Backend > db > Users > userschema.js**

const mongoose = require('mongoose');

const UserSchema = new mongoose.Schema({ name:String,

email: String, password: String, userId:{

type:mongoose.Schema.Types.ObjectId, ref:"user",

}

})

module.exports =mongoose.model('users',UserSchema)

**Backend > db > Users > Wishlist.js**

const mongoose = require('mongoose');

const wishlistItemSchema = new mongoose.Schema({

itemId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }, userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User' }, userName:String,

itemImage:String, title:String,

});

module.exports = mongoose.model('WishlistItem', wishlistItemSchema);

**Backend > db > config.js**

const mongoose = require('mongoose');

mongoose.connect('mongodb+srv://pd4008208:RIYA- [143@cluster0.pqg7g.mongodb.net/?retryWrites=true&w=majority&appName=Cluste](mailto:143@cluster0.pqg7g.mongodb.net) r0', {

});

**Backend > db > server.js**

const express = require('express') const PORT = 4000

const cors = require('cors') require('./db/config')

const multer = require('multer'); // Import multer const Admin = require('./db/Admin/Admin') const users = require('./db/Users/userschema')

const seller = require('./db/Seller/Sellers') const items = require('./db/Seller/Additem')

const myorders = require('./db/Users/myorders') const WishlistItem = require('./db/Users/Wishlist')

const app = express()

app.use(express.json())

app.use(cors({

origin: ["http://localhost:5173"],

methods: ["POST", "GET", "DELETE", "PUT"],

credentials: true

}));

const storage = multer.diskStorage({

destination: 'uploads', // The directory where uploaded files will be stored filename: function (req, file, callback) {

callback(null, Date.now() + '-' + file.originalname); // Set the file name

},

});

const upload = multer({ storage }); app.use('/uploads', express.static('uploads'));

// Admin //

// Login

app.post('/alogin', (req, resp) => {

const { email, password } = req.body; Admin.findOne({ email: email })

.then(user => { if (user) {

if (user.password === password) {

return resp.json({ Status: "Success", user: { id:user.id,name: user.name, email: user.email } })

} else {

resp.json("login fail")

}

} else {

resp.json("no user")

}

})

})

// Register Api

app.post('/asignup', (req, resp) => {

const { name, email, password } = req.body; Admin.findOne({ email: email })

.then(use => {

if (use) {

resp.json("Already have an account")

} else {

Admin.create({ email: email, name: name, password: password })

.then(result => resp.json(" Account Created"))

.catch(err => resp.json(err))

}

}).catch(err => resp.json("failed "))

})

app.get('/users',(req,res)=>{ users.find()

.then((user)=>{ res.status(200).json(user)

})

.catch(() => { res.sendStatus(500)

})

})

app.delete('/userdelete/:id',(req,res)=>{ const { id }=req.params users.findByIdAndDelete(id)

.then(() => { res.sendStatus(200);

})

.catch((error) => {

res.status(500).json({ error: 'Internal server error' });

});

})

app.delete('/userorderdelete/:id', async (req, res) => { const { id } = req.params;

try {

await myorders.findByIdAndDelete(id); res.sendStatus(200);

} catch (error) {

res.status(500).json({ error: 'Internal server error' });

}

});

app.delete('/useritemdelete/:id', async (req, res) => { const { id } = req.params;

try {

await items.findByIdAndDelete(id); res.sendStatus(200);

} catch (error) {

res.status(500).json({ error: 'Internal server error' });

}

});

app.get('/sellers',(req,res)=>{ seller.find()

.then((seller)=>{ res.status(200).json(seller)

})

.catch(() => { res.sendStatus(500)

})

})

app.delete('/sellerdelete/:id',(req,res)=>{ const { id }=req.params seller.findByIdAndDelete(id)

.then(() => { res.sendStatus(200);

})

.catch((error) => {

res.status(500).json({ error: 'Internal server error' });

});

})

app.get('/orders', (req, res) => { myorders.find()

.then((orders) => { res.status(200).json(orders)

})

.catch(() => { res.sendStatus(500)

})

});

// Seller //

// login api

app.post('/slogin', (req, resp) => {

const { email, password } = req.body; seller.findOne({ email: email })

.then(user => { if (user) {

if (user.password === password) {

return resp.json({ Status: "Success", user: { id: user.id, name: user.name, email: user.email } })

} else {

resp.json("login fail")

}

} else {

resp.json("no user")

}

})

})

// Register Api

app.post('/ssignup', (req, resp) => {

const { name, email, password } = req.body; seller.findOne({ email: email })

.then(use => { if (use) {

resp.json("Already have an account")

} else {

seller.create({ email: email, name: name, password: password })

.then(result => resp.json(" Account Created"))

.catch(err => resp.json(err))

}

}).catch(err => resp.json("failed "))

})

// addBook

app.post('/items', upload.single('itemImage'), async (req, res) => {

const { title, author, genre, description, price, userId, userName } = req.body; const itemImage = req.file.path; // The path to the uploaded image

try {

const item = new items({ itemImage, title, author, genre, description, price, userId, userName });

await item.save(); res.status(201).json(item);

} catch (err) {

res.status(400).json({ error: 'Failed to create item' });

}

});

//getbooks

app.get('/getitem/:userId', async (req, res) => { const userId = req.params.userId;

try {

const tasks = await items.find({ userId }).sort('position'); res.json(tasks);

} catch (err) {

res.status(500).json({ error: 'Failed to fetch tasks' });

}

});

//delete book

app.delete('/itemdelete/:id', (req, res) => { const { id } = req.params; items.findByIdAndDelete(id)

.then(() => { res.sendStatus(200);

})

.catch((error) => {

res.status(500).json({ error: 'Internal server error' });

});

})

//getorders

app.get('/getsellerorders/:userId', async (req, res) => { const sellerId = req.params.userId;

try {

const tasks = await myorders.find({ sellerId }).sort('position'); res.json(tasks);

} catch (err) {

res.status(500).json({ error: 'Failed to fetch tasks' });

}

});

// users //

// login

app.post('/login', (req, res) => {

const { email, password } = req.body; users.findOne({ email: email })

.then(user => { if (user) {

if (user.password === password) {

return res.json({ Status: "Success", user: { id: user.id, name: user.name, email: user.email } })

}

else {

res.json("Invalid Password")

}

}

else {

res.json("User not found")

}

})

})

app.post('/signup', (req, resp) => {

const { name, email, password } = req.body; users.findOne({ email: email })

.then(use => { if (use) {

resp.json("Already have an account")

} else {

users.create({ email: email, name: name, password: password })

.then(result => resp.json(" Account Created"))

.catch(err => resp.json(err))

}

}).catch(err => resp.json("failed "))

})

app.get('/item', async (req, res) => { try {

const images = await items.find(); res.json(images);

} catch (error) { console.error(error);

res.status(500).send('Server Error');

}

});

// Single item

app.get('/item/:id', async (req, res) => { const id = req.params.id;

try {

const item = await items.findById({ \_id: id }); res.json(item);

} catch (err) {

res.status(500).json({ error: err.message });

}

});

app.post('/userorder', async (req, res) => {

const { flatno, city, state, pincode, totalamount, seller, sellerId, BookingDate, description, Delivery, userId, userName: String, booktitle, bookauthor, bookgenre, itemImage } = req.body;

try {

const order = new myorders({ flatno, city, state, pincode, totalamount, seller, sellerId, BookingDate, description, userId, Delivery, userName: String, booktitle, bookauthor, bookgenre, itemImage });

await order.save(); res.status(201).json(order);

} catch (err) {

res.status(400).json({ error: 'Failed to create policy' });

}

});

app.get('/getorders/:userId', async (req, res) => { const userId = req.params.userId;

try {

const tasks = await myorders.find({ userId }).sort('position'); res.json(tasks);

} catch (err) {

res.status(500).json({ error: 'Failed to fetch tasks' });

}

});

app.get('/wishlist', async (req, res) => { try {

const wishlistItems = await WishlistItem.find(); res.json(wishlistItems);

} catch (error) { console.error(error); res.status(500).send('Server Error');

}

});

app.get('/wishlist/:userId', async (req, res) => { const userId = req.params.userId;

try {

const tasks = await WishlistItem.find({ userId }).sort('position'); res.json(tasks);

} catch (err) {

res.status(500).json({ error: 'Failed to fetch tasks' });

}

});

app.post('/wishlist/add', async (req, res) => {

const { itemId, title,itemImage,userId,userName } = req.body;

try {

// Check if the item is already in the wishlist

const existingItem = await WishlistItem.findOne({ itemId }); if (existingItem) {

return res.status(400).json({ msg: 'Item already in wishlist' });

}

// Create a new wishlist item

const newItem = new WishlistItem({ itemId,title,itemImage,userId,userName}); await newItem.save();

res.json(newItem);

} catch (error) { console.error(error);

res.status(500).send('Server Error');

}

});

app.post('/wishlist/remove', async (req, res) => { const { itemId } = req.body;

try {

// Find and remove the item from the wishlist await WishlistItem.findOneAndDelete({ itemId }); res.json({ msg: 'Item removed from wishlist' });

} catch (error) { console.error(error); res.status(500).send('Server Error');

}

});

app.listen(PORT, () => {

console.log(`server is running on ${PORT}`)

})

# AUTHENTICATION

Authentication is the process of verifying the identity of a user. It ensures that the person or system trying to access the application is who they claim to be. Here's how it's typically handled:

1. **Login Credentials**: Users provide a username and password. The system checks these credentials against a database. If the credentials match, the user is authenticated.
2. **Token-based Authentication:** After successful login, the system may issue an authentication token (e.g., JSON Web Token - JWT) that the user includes in subsequent requests. This token proves the user's identity without needing to re-enter credentials.
3. **Session Management**: In some systems, a session is created after authentication, allowing the user to access resources until they log out or the session expires.

## AUTHORIZATION:

Once the user is authenticated, authorization determines what resources they are allowed to access and what actions they can perform. It ensures that users can only access information or perform actions based on their roles and permissions.

1. **Role-based Access Control (RBAC):** Users are assigned roles (e.g., admin, user, manager) based on their function within the system. Each role has specific permissions that grant or restrict access to certain resources or actions.
2. **Permissions:** Permissions are the rules that define what users can do. For example, an admin may have permissions to view, edit, and delete resources, while a regular user may only be able to view resources.

By separating authentication and authorization, the system can maintain strict control over who can log in and what they can do once logged in, ensuring security and minimizing the risk of unauthorized access.

## TOKENS:

* **Definition:** Tokens are pieces of data used to verify a user's identity or session. They are commonly used for authentication and authorization purposes, especially in web applications and APIs.

## How Tokens Work:

* + **Authentication:** A user logs in, and the server issues a token (e.g., JWT). The user sends this token in subsequent requests to authenticate themselves.
  + **Expiration:** Tokens can have an expiration time after which they are no longer valid. Some systems use refresh tokens to generate new tokens when expired.

## SESSIONS:

* **Definition:** A session is a server-side storage mechanism used to keep track of a user's state during their interaction with an application. It allows the system to remember user information, preferences, and the user's current actions across multiple requests.

## How Sessions Work:

* + **Session ID:** When a user logs in, a unique session ID is created and stored on the server. This ID is sent to the user’s browser as a cookie.
  + **Session Management:** On each subsequent request, the browser sends the session ID in the cookie, allowing the server to retrieve the associated session data (like user information or authentication status).
  + **Expiration:** Sessions typically expire after a certain period of inactivity or when a user logs out.

# USER INTERFACE

## Admin:

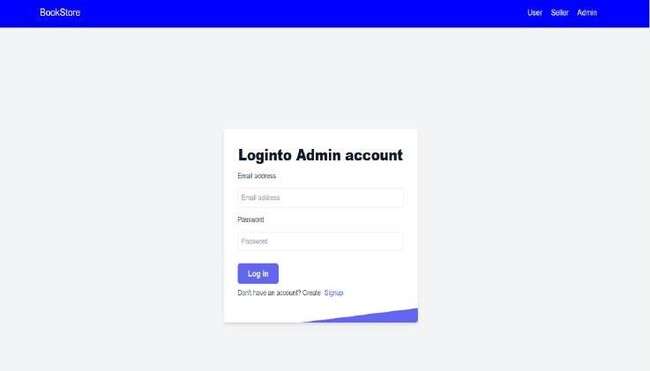


Fig. 4. 1: Admin Login Page



Fig. 4. 2: Admin Dashboard

The administrative interface of the platform commences with the Admin Login Page, as depicted in Figure 4.1. This page features a straightforward and secure login mechanism, comprising an email address text box, a password text box, and a login button. Upon successful authentication, the login button redirects the administrator to the Admin Dashboard page. Additionally, the login page provides a convenient "Don't have an account? Sign up" option, facilitating easy registration for new administrators.

Upon logging in, administrators are directed to the Admin Dashboard, illustrated in Figure 4.2. This dashboard presents a comprehensive overview of key platform metrics, including the total number of users, vendors, books or items, and orders. The dashboard also features a bar graph, providing a visual representation of these metrics. This intuitive and informative dashboard enables administrators to efficiently monitor and manage platform activity, ensuring seamless operations and informed decision- making.



Fig. 4. 3: Admin’s User count Page



Fig. 4. 4: Admin’s Vendors count Page

The administrative interface of the platform provides two comprehensive pages for user and vendor management, as depicted in Figures 4.3 and 4.4, respectively.

Figure 4.3 illustrates the Admin's User Count Page, which presents a detailed overview of the platform's registered users. This page provides key metrics, including the total user count, user details, and search and filter options. Administrators can efficiently manage user accounts, track user activity, and make informed decisions about platform operations.

Figure 4.4 showcases the Admin's Vendors Count Page, which offers a centralized view of all active vendors on the platform. This page displays essential vendor metrics, including the total vendor count, vendor details, and search and filter options. Administrators can effectively manage vendor accounts, monitor vendor activity, and optimize platform operations to ensure seamless interactions between vendors and users.

## Seller:

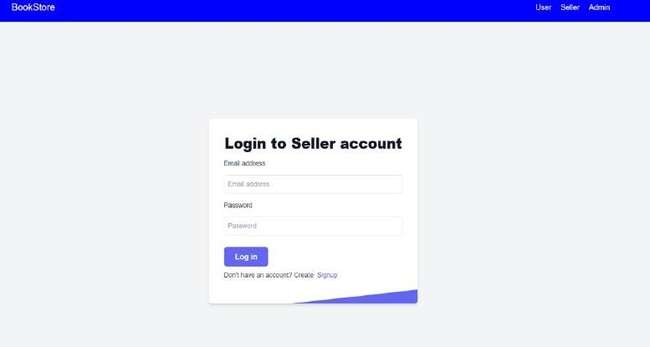


Fig. 4. 5: Seller Login Page



Fig. 4. 6: Seller Dashboard

The platform provides a dedicated interface for sellers, as illustrated in Figures

4.5 and 4.6.

Figure 4.5 displays the Seller Login Page, which shares similarities with the Admin Login Page. This page features a secure login mechanism, allowing sellers to access their accounts and manage their activities on the platform.

Upon successful login, sellers are directed to the Seller Dashboard, as shown in Figure 4.6. This dashboard presents key performance indicators (KPIs) specific to the seller's activities, including the count of items or books posted and the count of orders received. This personalized dashboard enables sellers to track their progress, monitor their sales, and make informed decisions about their inventory and marketing strategies.

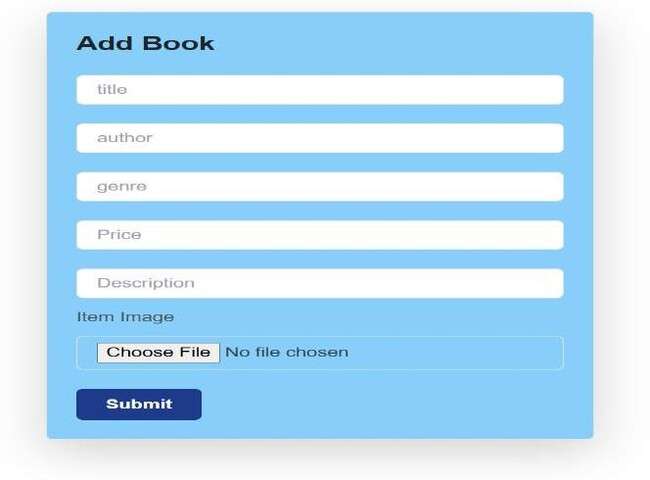


Fig. 4. 7: Add Book Page

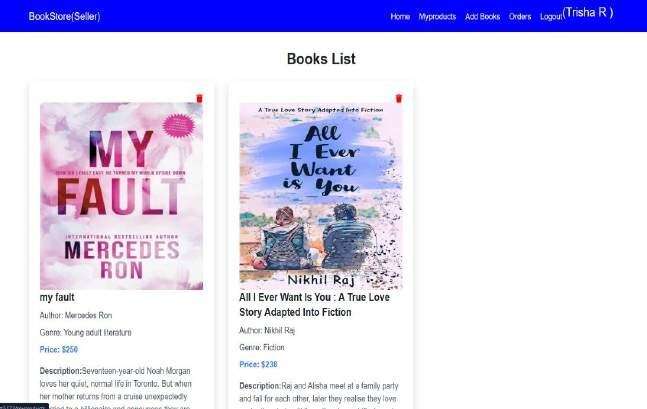


Fig. 4. 8: List of Books uploaded by the Seller



Fig. 4. 9: Orders Page

The platform provides a comprehensive interface for sellers to manage their book inventory, as illustrated in Figures 4.7, 4.8, and 4.9.

Figure 4.7 displays the Add Books Page, which enables sellers to add new books to their inventory. This page features a user-friendly form that requires sellers to input essential book details, including title, author, genre, price, and description. Additionally, sellers can upload the cover page of the book, allowing for a visually appealing representation of their inventory.

The Books List Page, shown in Figure 4.8, presents a comprehensive list of books uploaded by the seller. This page provides a centralized view of the seller's inventory, facilitating easy management and organization of their books.

Figure 4.9 illustrates the Orders Page, where sellers can view and manage orders received for their books. This page provides a clear and concise overview of order details, enabling sellers to efficiently fulfil orders and maintain a high level of customer satisfaction.

## User:

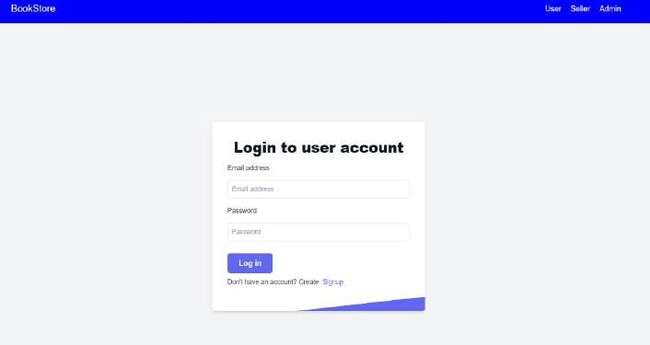


Fig. 4. 10: User Login Page

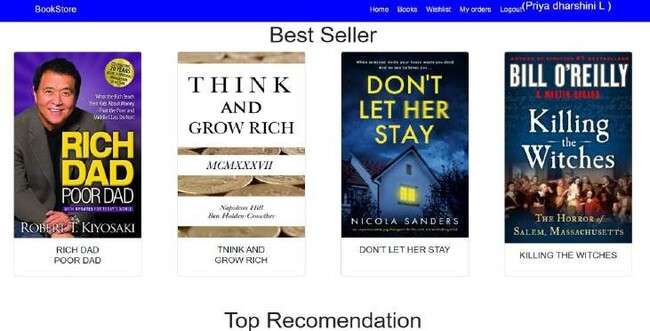


Fig. 4. 11(a) User Dashboard



Fig. 4.11(b) User Dashboard

The platform provides a dedicated interface for users, as illustrated in Figures 4.10, 4.11(a), and 4.11(b).

Figure 4.10 displays the User Login Page, which serves as the entry point for users to access their accounts. This page features a secure login mechanism, allowing users to authenticate and access the platform's features.

Upon successful login, users are directed to the User Home Page, as shown in Figures 4.11(a) and 4.11(b). This page presents a personalized and engaging experience,

showcasing the best seller books and top recommendations. The page is designed to facilitate discovery and exploration, enabling users to easily find and access their preferred books.

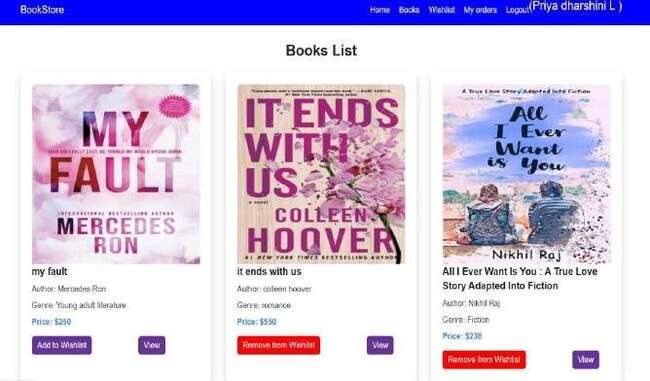


Fig. 4. 12: Books List Page

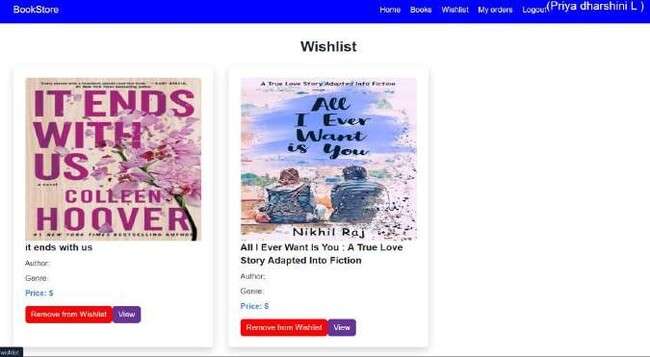


Fig. 4. 13: Whishlist Page

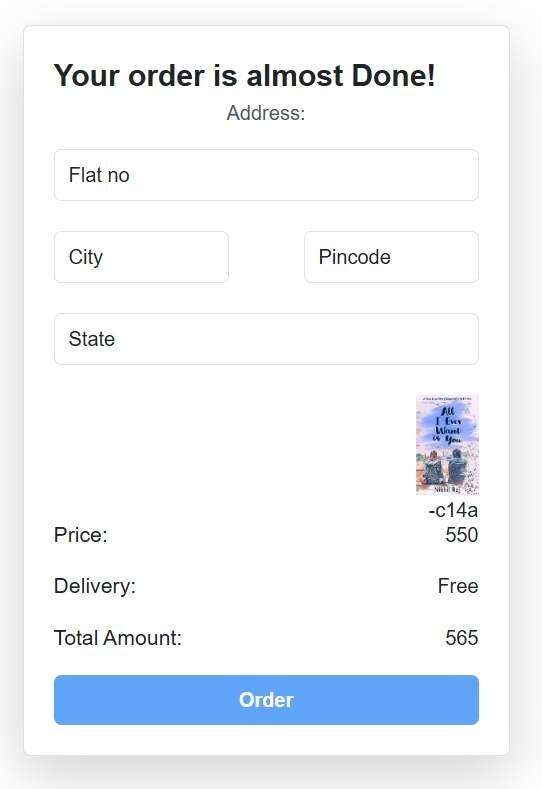


Fig. 4. 14: Order Placement Page



Fig. 4. 15: List of Orders placed by the User

The platform provides a seamless and intuitive experience for users to browse, select, and purchase books, as illustrated in Figures 4.12, 4.13, 4.14, and 4.15.

Figure 4.12 displays the Book List Page, which presents a comprehensive and organized list of books available on the platform. This page enables users to browse and discover new books, facilitating a engaging and interactive experience.

Figure 4.13 shows the Wishlist Page, where users can view and manage the books, they have added to their wish list. This feature allows users to save books for future reference, making it easy to keep track of desired titles.

Figure 4.14 illustrates the Order Placement Page, where users can fill in their details to place an order. This page provides a secure and user-friendly interface for users to enter their shipping and payment information, ensuring a smooth and efficient checkout process.

Figure 4.15 displays the Order List Page, which presents a comprehensive list of orders placed by the user. This page enables users to track the status of their orders, view order details, and access order history, providing a convenient and transparent experience.

# TESTING:

In the development of the book-selling webpage project using the MERN stack, testing played a crucial role in ensuring the reliability, functionality, and user experience of the application. Below is an overview of the testing strategy and tools utilized throughout the project:

## Testing Strategy

1. **Unit Testing**
   * **Objective:** Validate individual components, functions, and modules in isolation.
   * **Scope:** Focused on React components, API endpoints, and utility functions.
   * **Outcome:** Ensured that each module behaved as expected with different inputs.

## Integration Testing

* + **Objective:** Verify that different modules and components work together seamlessly.
  + **Scope:** Tested interactions between the front end (React) and the back end (Express and MongoDB).
  + **Outcome:** Ensured data consistency and communication between the client and server.

## End-to-End (E2E) Testing

* + **Objective:** Test the entire user journey, from logging in to purchasing a book.
  + **Scope:** Covered core functionalities like user registration, book searching, adding to the wishlist, and completing a purchase.
  + **Outcome:** Ensured a smooth and error-free user experience.

## Performance Testing

* + **Objective:** Evaluate the application’s response time and scalability under varying loads.
  + **Scope:** Tested with different numbers of concurrent users to simulate real- world usage.
  + **Outcome:** Identified and optimized performance bottlenecks.

## Manual Testing

* + **Objective:** Catch UI/UX inconsistencies and edge cases that automated tests might miss.
  + **Scope:** Performed on all major features, focusing on responsiveness and design.
  + **Outcome:** Delivered a polished and user-friendly interface.

## Tools Used

1. **Jest**
   * Used for writing unit tests for React components and utility functions.
   * Allowed mocking of APIs and other dependencies for isolated testing.

## React Testing Library

* + Employed for testing the behavior and rendering of React components.
  + Provided tools for simulating user interactions, such as clicks and form submissions.

## Postman

* + Used for manual testing of API endpoints.
  + Validated proper request-response cycles, ensuring accurate data handling.

## Lighthouse

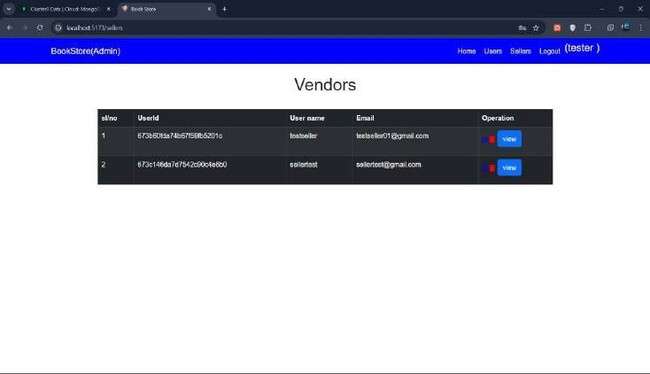
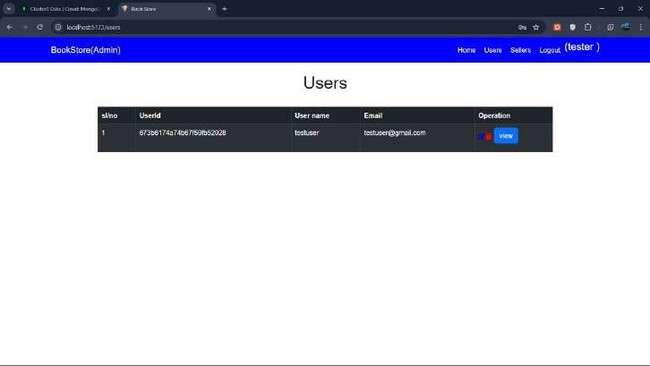
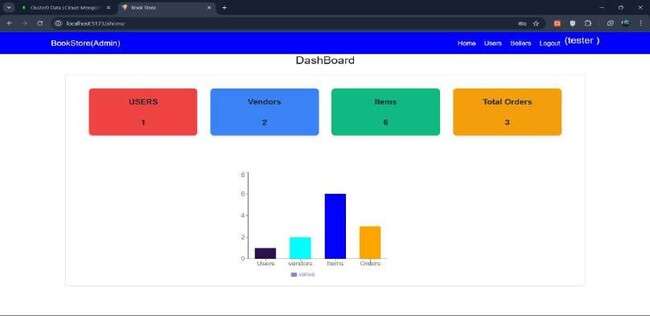
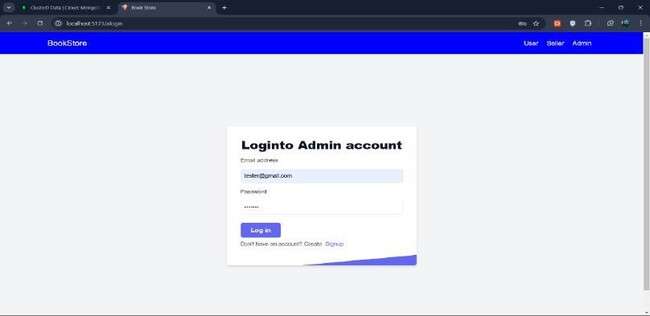
* + Used for performance testing and auditing the webpage for speed, accessibility, and SEO.
  + Helped optimize images, scripts, and loading strategies.

## MongoDB Compass

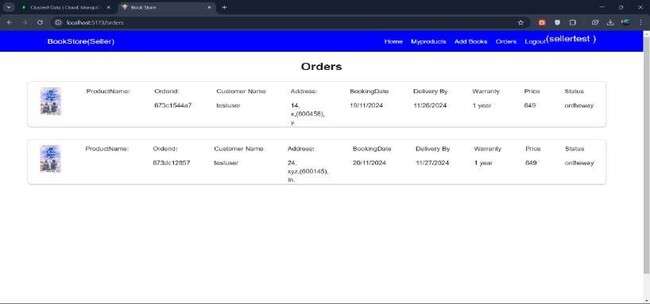
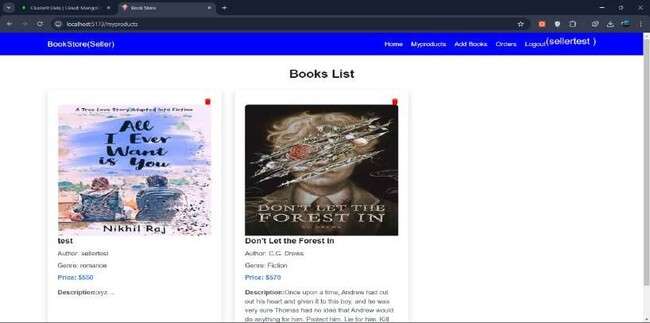
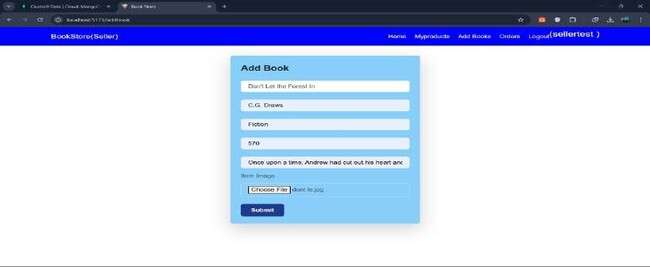
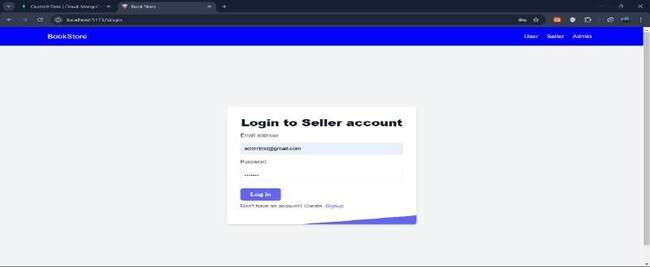
* + Utilized to verify database operations during testing.
  + Ensured data integrity for CRUD operations in the application.

# SCREENSHOTS

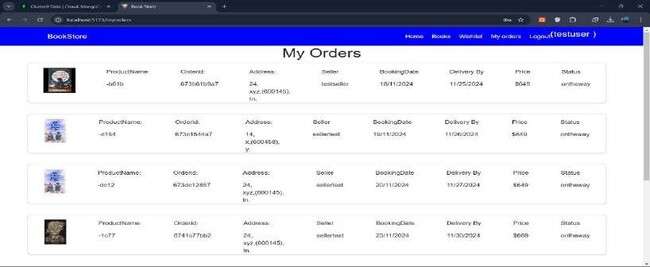
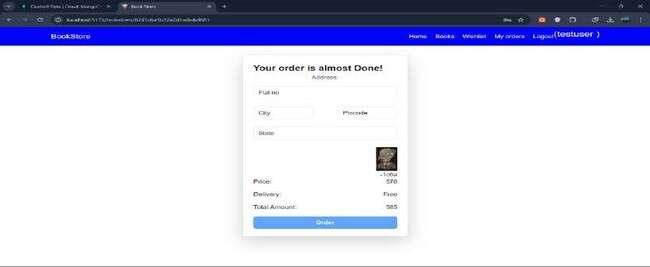
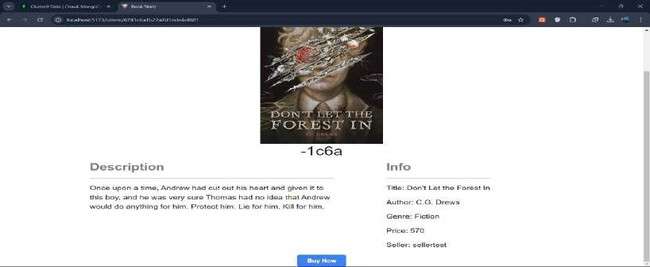
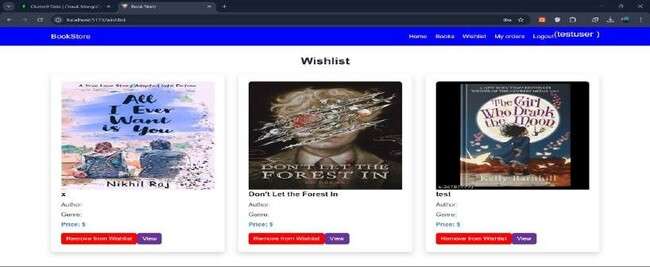
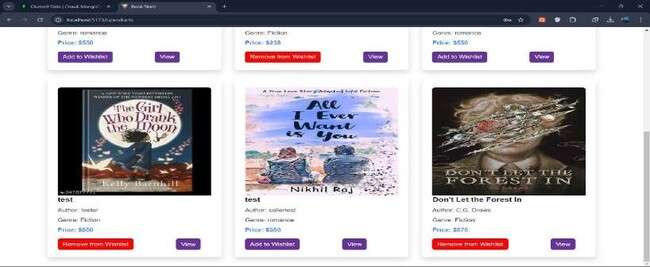
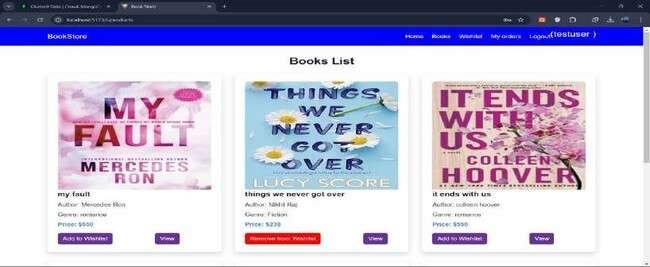
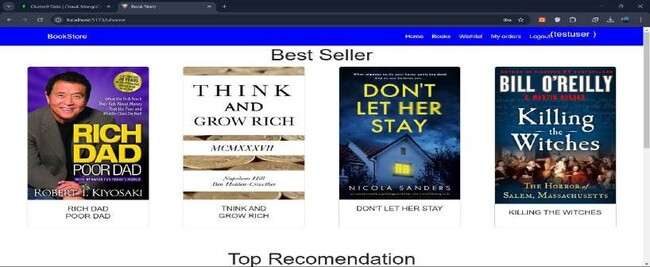
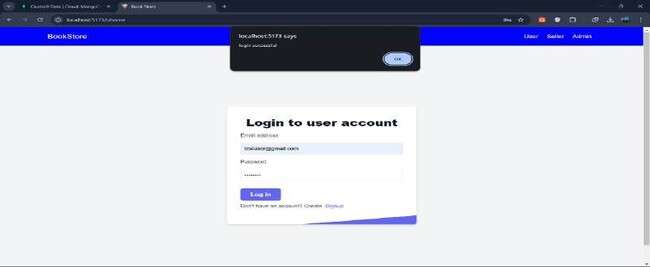
**Admin Pages:**



**Seller Pages:**



**User Pages:**



# KNOWN ISSUES:

As with any project in development, our book-selling webpage has some known issues and bugs that developers and users should be aware of. These issues are being tracked and will be addressed in upcoming updates. Documenting them ensures transparency and helps both users and contributors understand the current state of the application.

* Image Loading Delays: Book covers take time to load, especially in high traffic.
* Authentication and Session Issues: Users face login problems, including session timeouts and authentication errors.
* Wishlist Synchronization Problem: Wishlist updates are not reflected immediately for some users.
* Pagination Inefficiencies: The books page loads slowly when displaying a large inventory.
* Mobile Responsiveness Limitations: Some UI elements don't adapt well to smaller screens.

These issues are being actively tracked in our project's issue tracker, and we welcome feedback or bug reports from users and contributors. Addressing these challenges is an ongoing process, and regular updates will be rolled out to enhance the overall user experience and application stability.

# FUTURE ENHANCEMENTS

To ensure continuous improvement and provide an optimal user experience, we have identified five key future enhancements for our book-selling webpage. These planned features and improvements aim to enhance functionality, performance, and usability:

## Advanced Search and Filter Options:

Currently, users can only perform basic searches. Introducing advanced search capabilities with filters for genre, author, price range, publication year, and user

ratings will make it easier for customers to find books that meet their specific preferences.

## Personalized Recommendations:

Implementing a recommendation engine powered by machine learning will allow us to provide users with tailored book suggestions based on their browsing history, wishlist, and previous purchases. This will enhance user engagement and encourage repeat visits.

## Integrated Payment Gateway:

To streamline the checkout process, we plan to integrate multiple payment gateways, including credit/debit cards, digital wallets, and UPI. This will provide users with more secure and convenient payment options.

## Enhanced Seller Dashboard:

Expanding the seller dashboard with features such as sales analytics, order tracking, and inventory management will empower sellers to better manage their accounts and optimize their book listings.

## Mobile Application Development:

To cater to the growing number of mobile users, we plan to develop a dedicated mobile application for Android and iOS platforms. The app will offer a more seamless experience with optimized navigation, faster loading times, and mobile-specific features such as push notifications.

These enhancements are aimed at making our platform more user-friendly, efficient, and engaging, ensuring it remains competitive and valuable to both buyers and sellers.

# CONCLUSION:

In conclusion, this project successfully demonstrates the development of a book- selling webpage using the MERN stack. Key features such as user authentication, book listings, and wishlists have been implemented, providing a solid foundation for future enhancements.