Full Stack Development with MERN

1. INTRODUCTION:

PROJECT TITLE: Bookie Nest – Online Book Store Management System

The Bookie Nest application transforms the traditional book shopping experience by offering a seamless and user-friendly online platform. Users can easily browse a wide range of books, view details, add items to their cart, and make secure purchases from the comfort of their homes. Leveraging a robust client-server architecture built with **Express.js**, **React**, and **MongoDB**, the application ensures efficient data storage, dynamic content rendering, and smooth interaction.

With features such as **user authentication**, **book management**, **shopping cart**, and **order tracking**, Bookie Nest delivers a personalized and efficient shopping experience for both users and administrators. This documentation serves as a comprehensive guide for understanding, developing, and deploying the application.

TEAM MEMBERS:

1. **HARSHAL**
   i. Led the project by assigning tasks and overseeing team collaboration.
   ii. Developed backend APIs using Node.js and Express for core functionalities.
   iii. Managed MongoDB database design and ensured secure, efficient data operations.
   iv. Took responsibility for writing significant parts of the technical documentation.
   v. Resolved key issues related to backend integration and system logic.
2. **ANURAG**
   i. Contributed to organizing and refining the overall project documentation.
   ii. Provided valuable suggestions for feature enhancements and user experience.
   iii. Reviewed the team’s code contributions and ensured consistency across modules.
3. **BAIBHAV**
   i. Built and implemented the primary user interface using React.
   ii. Integrated RESTful APIs with frontend components for real-time updates.
   iii. Created reusable UI components to streamline user interaction.
   iv. Helped with frontend-related technical writing.
4. **AKHAND**
   i. Styled the application using modern CSS libraries for a clean and responsive design.
   ii. Maintained UI/UX consistency across all web pages.
   iii. Assisted Harshal in integrating styling with functional components.
   iv. Contributed to the planning documentation and testing phase.

2. PROJECT OVERVIEW:

• PURPOSE:

The **Book Store Management System** is a comprehensive platform designed to streamline the process of managing and interacting with book-related data. It allows administrators to manage books, categories, and inventory, while providing users with a seamless experience to browse, purchase, and review books. The system is built to be scalable, secure, and user-friendly, catering to both administrators and end-users.
Built with **Express.js** and **MongoDB**, the application leverages a robust client-server architecture to ensure efficient data handling and real-time updates. Key features include user authentication, order management, and personalized book recommendations, enhancing convenience and user engagement.

• GOALS:

1. Provide a centralized platform for managing book inventory and user orders.
2. Enable users to browse, purchase, and review books with ease.
3. Ensure secure and efficient handling of user, admin, and product data.

• FEATURES:

1. **User Authentication and Authorization:**
   a. Secure login and registration for users and admins with role-based access control.
2. **Admin Features:**
   a. Add, edit, and delete books and manage categories and inventory.
   b. View and manage user orders and reviews.
3. **User Features:**
   a. Browse books by genre, author, or popularity.
   b. Add books to cart and place orders.
   c. Mark books as favorites and leave reviews and ratings.
4. **Responsive Design:**
   a. Optimized for both desktop and mobile devices to ensure a smooth browsing and shopping experience.
5. **Dynamic Content Rendering:**
   a. Real-time updates for book listings, cart contents, orders, and reviews.

3. ARCHITECTURE:

• FRONTEND:

The frontend is built using **React**, a widely-used JavaScript library for building dynamic and responsive user interfaces. The architecture follows a component-based structure, promoting code reusability and ease of maintenance.

1. State Management:

- Managed using React's `useState` and `useContext` hooks to handle both local and global states effectively.

2. Routing:

- Implemented using `react-router-dom` to allow smooth navigation across different pages like Home, Book Details, Cart, Profile, and Admin Panel.

3. API Integration:

- **Axios** is used to communicate with backend APIs for tasks such as fetching books, submitting reviews, handling login/logout, and managing orders.

4. Styling:

- CSS Modules, **Tailwind CSS**, or **Bootstrap** are used to provide a clean, responsive, and consistent design across devices.

• BACKEND:

The backend is developed using **Node.js** and **Express.js**, offering a powerful and scalable server-side foundation for the application.

1. RESTful APIs:

- CRUD operations are provided for users, books, orders, and reviews.

2. Middleware:

- Custom middleware handles **authentication** (JWT-based), **authorization**, **error handling**, and **input validation**.

3. Security:

- Passwords are securely hashed using **bcrypt**.
- **JWT tokens** are used for managing secure sessions and user roles.

4. Scalability:

- The backend follows a modular structure with separate **routes**, **controllers**, and **middleware**, ensuring scalability and ease of maintenance.

• DATABASE:

The database uses **MongoDB**, a NoSQL database well-suited for storing flexible and hierarchical data models.

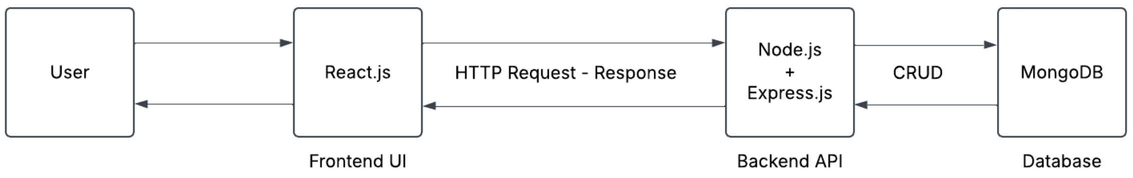
1. Schemas:

- **User Schema:** Stores user data, including credentials and roles (admin/user).
- **Book Schema:** Stores book details like title, author, description, genre, price, stock, and image URL.
- **Order Schema:** Tracks user orders, including books purchased, total amount, and order status.
- **Review Schema:** Stores reviews and ratings provided by users on books.
- **Cart Schema (optional):** Maintains the current items in a user's cart before checkout.

2. Interactions:

- **Mongoose** is used as the ODM for schema definition, validation, and querying.
- Schema relationships (e.g., books in an order or user reviews) are handled using **ObjectId references**.

TECHNICAL ARCHITECTURE:



4. SETUP INSTRUCTIONS

PREREQUISITES:

Before setting up the project, ensure the following software is installed on your system:

- **Node.js (v14 or higher):**
Required to run the backend server and frontend development environment.

- **MongoDB:**
Either a local MongoDB setup or a cloud instance (e.g., MongoDB Atlas) is required to store and manage application data.

- **npm (Node Package Manager):**
Comes bundled with Node.js. It is used to install all necessary dependencies for both backend and frontend projects.

INSTALLATION

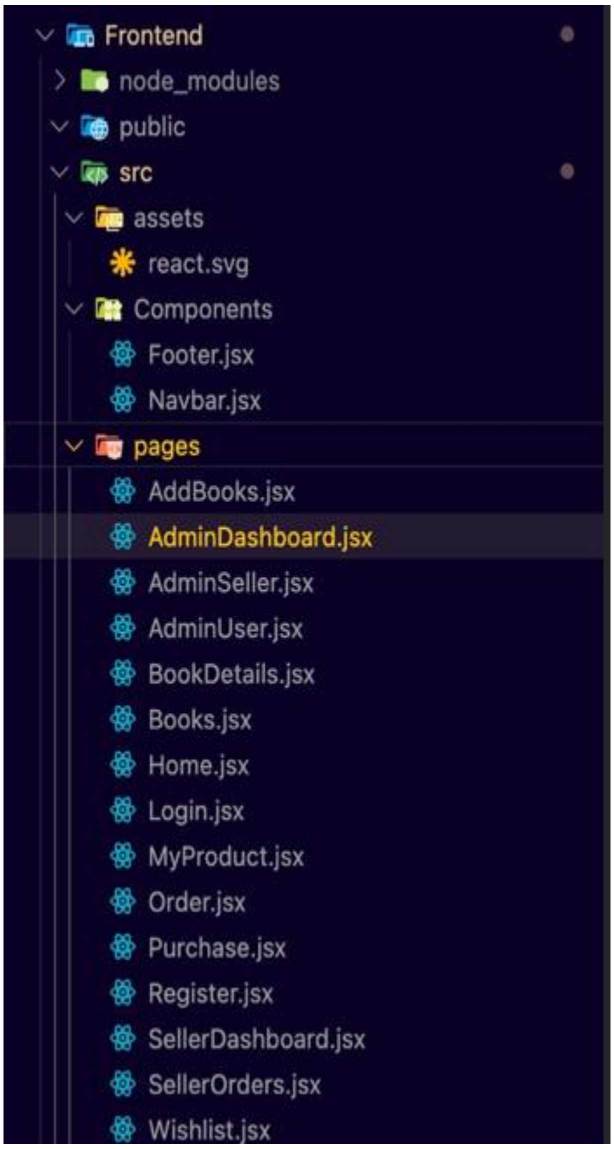
1. **Clone the Repository:**
   Clone the project repository to your local machine:
   - `git clone <your-repository-url>`
   - `cd your-project-folder`
2. **Set Up the Backend:**
   Navigate to the backend folder and install the required dependencies:
   - `cd backend`
   - `npm install`
3. **Create Environment Variables:**
   Inside the backend folder, create a `.env` file and add the following configuration:
   - `MONGODB_CONNECTION_LINK=<your-mongodb-uri>`
   - `JWT_SECRET=<your-jwt-secret>`
   - `PORT=5000`
4. **Set Up the Frontend:**
   Navigate to the frontend folder and install its dependencies:
   - `cd ../frontend`
   - `npm install`
5. **Start the Backend Server:**
   Run the backend server:
   - `cd ../backend`
   - `npm start`
6. **Start the Frontend Development Server:**
   Run the frontend server:
   - `cd ../frontend`
   - `npm run dev`
7. **Access the Application:**
   Open your browser and go to:
   http://localhost:3000 to view and use the Book Store Website.

5. Folder Structure

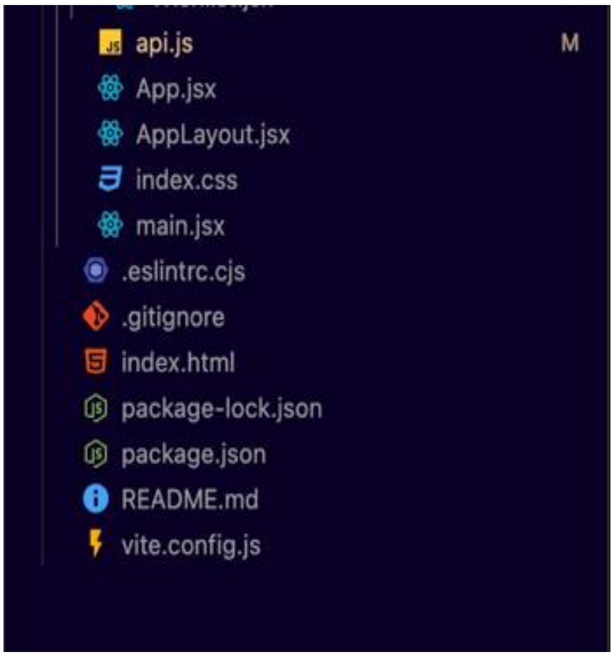
This folder structure ensures a clear separation of concerns, making the project easy to navigate and maintain. This folder structure ensures a clear separation of concerns, making the project easy to navigate and maintain.

Client (Frontend):

The React frontend is organized as follows:



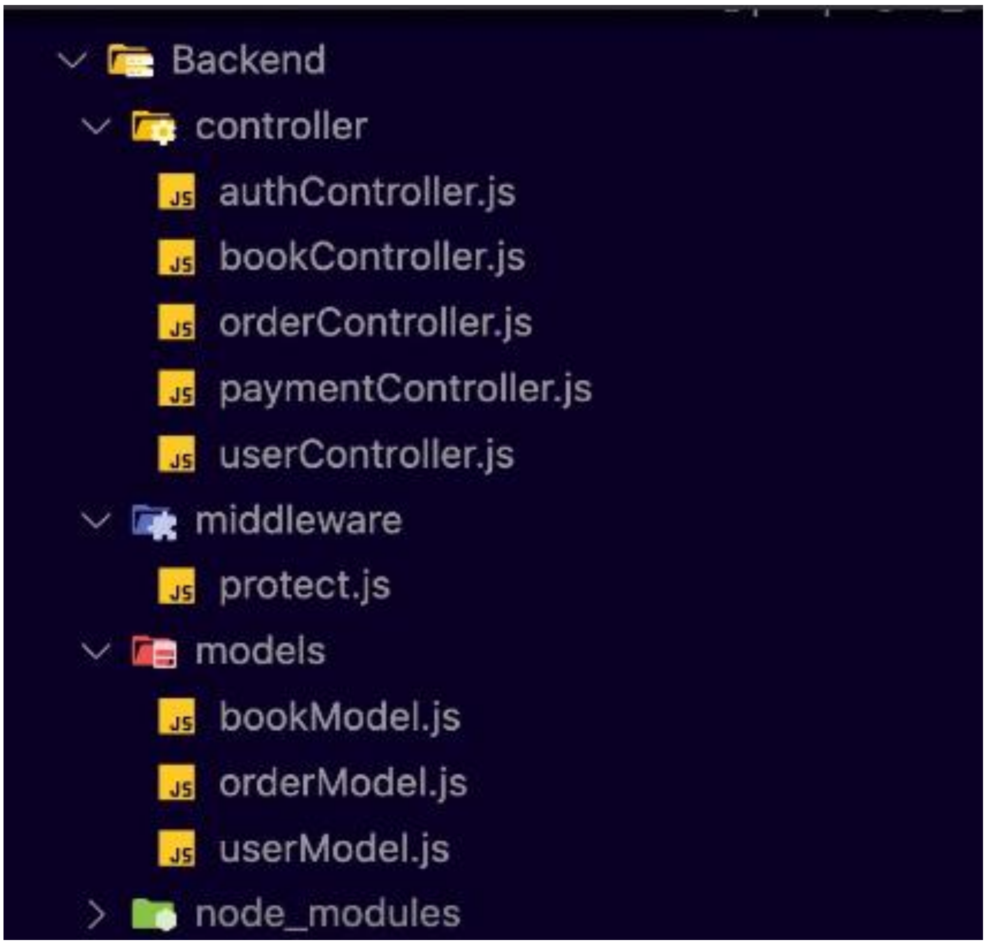
```
Frontend
  node_modules
  public
  src
    assets
      react.svg
    Components
      Footer.jsx
      Navbar.jsx
    pages
      AddBooks.jsx
      AdminDashboard.jsx
      AdminSeller.jsx
      AdminUser.jsx
      BookDetails.jsx
      Books.jsx
      Home.jsx
      Login.jsx
      MyProduct.jsx
      Order.jsx
      Purchase.jsx
      Register.jsx
      SellerDashboard.jsx
      SellerOrders.jsx
      Wishlist.jsx
```



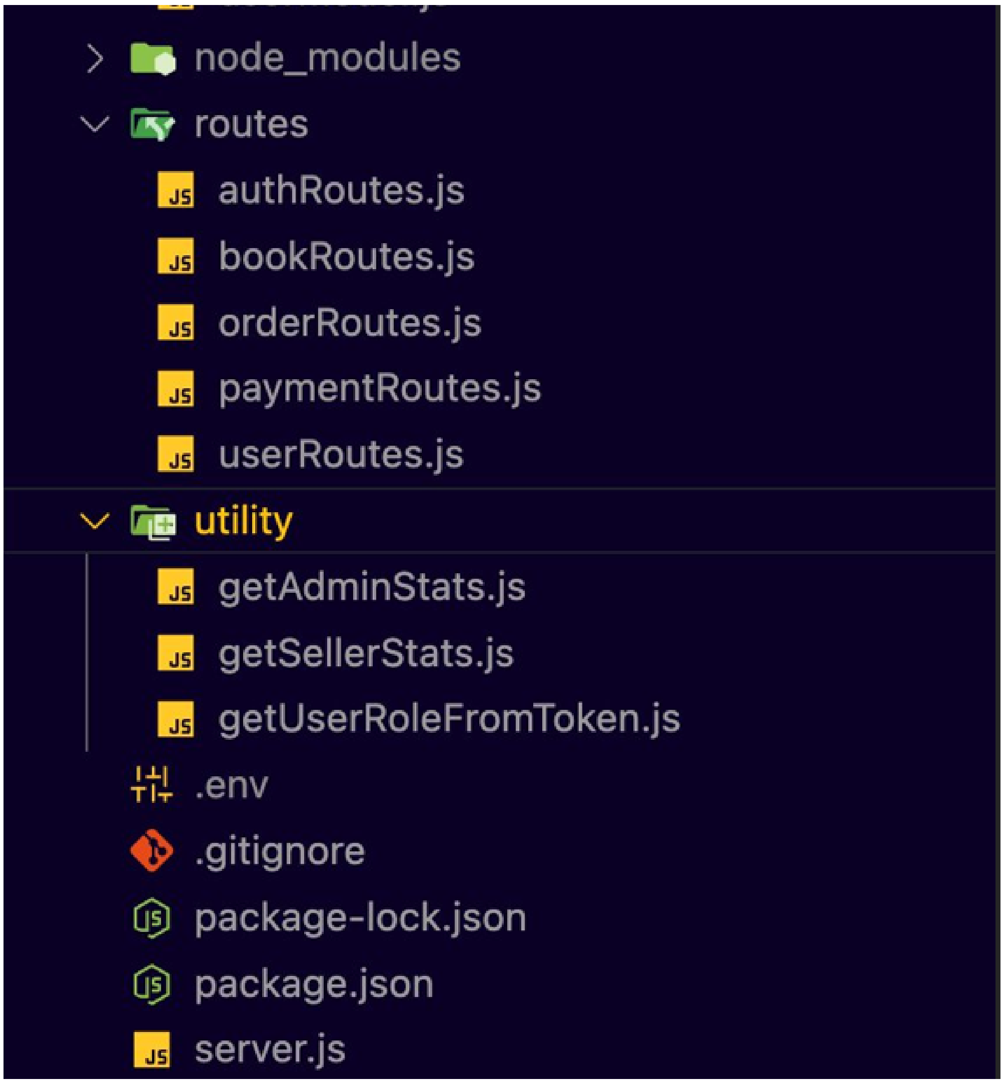
```
api.js                M
App.jsx
AppLayout.jsx
index.css
main.jsx
.eslintrc.cjs
.gitignore
index.html
package-lock.json
package.json
README.md
vite.config.js
```

Server (Backend):

The Node.js backend is structured as follows:



```
Backend
  controller
    authController.js
    bookController.js
    orderController.js
    paymentController.js
    userController.js
  middleware
    protect.js
  models
    bookModel.js
    orderModel.js
    userModel.js
  node_modules
```



```
> node_modules
v routes
    authRoutes.js
    bookRoutes.js
    orderRoutes.js
    paymentRoutes.js
    userRoutes.js
v utility
    getAdminStats.js
    getSellerStats.js
    getUserRoleFromToken.js
  .env
  .gitignore
  package-lock.json
  package.json
  server.js
```

6. RUNNING THE APPLICATION:

To run the Book Store Website locally on your machine, you need to start both the frontend and backend servers. Make sure you have Node.js and MongoDB installed before proceeding.

1. Clone the Repository:
   a. `git clone <your-repository-url>`
   b. `cd your-project-folder`
2. Backend Setup (Server)
   a. Navigate to the backend directory: `cd backend`
   b. Install dependencies: `npm install`
   c. Start the backend server: `npm start`
      i. The backend will typically run on http://localhost:5000
3. Frontend Setup (Client)
   a. Open a new terminal and navigate to the frontend directory: `cd frontend`
   b. Install frontend dependencies: `npm install`

c. Start the frontend server: `npm run dev`
   i. The frontend will typically run on http://localhost:3000

7. API Documentation

Below is the list of API endpoints exposed by the Bookie Nest backend.

Method	Endpoint	Description	Middleware	Request Body	Response Example
POST	/register	Register an admin	-	{ "email": "admin@example.com", "password": "123456" }	{ "success": true, "token": "..." }
POST	/login	Admin login	-	{ "email": "admin@example.com", "password": "123456" }	{ "success": true, "token": "..." }
POST	/register	User registration	-	{ "name": "John", "email": "john@example.com", "password": "123456" }	{ "success": true, "token": "..." }
POST	/login	User login	-	{ "email": "john@example.com", "password": "123456" }	{ "success": true, "token": "..." }
PUT	/editprofile	Edit user profile	-	{ "name": "John Updated", "email": "new@example.com" }	{ "success": true, "user": { ... } }
POST	/addpurchase	Add a new purchase	fetchUser	{ "showId": "...", "seats": ["A1", "A2"] }	{ "success": true, "purchase": { ... } }
POST	/savebook	Save book to favorites	fetchUser	{ "bookId": "..." }	{ "success": true }
DELETE	/unsavebook/:bookId	Remove saved book	fetchUser	-	{ "success": true }
GET	/getsavedbooks	Get saved books	fetchUser	-	{ "books": [...] }
POST	/addbook	Add a new book	fetchAdmin	{ "title": "Book", "genre": "...", ... }	{ "success": true, "book": { ... } }
GET	/getbooks	Get all books	—	—	{ "books": [...] }
GET	/getbookdetails/:bookId	Get book details	—	—	{ "book": { ... } }

Method	Endpoint	Description	Middleware	Request Body	Response Example
PUT	/editbook/:bookId	Edit book details	—	{ "title": "New Title", ... }	{ "success": true }
POST	/addreview	Add review	fetchUser	{ "bookId": "...", "rating": 4, "comment": "Nice!" }	{ "success": true }
PUT	/editreview	Edit review	—	{ "reviewId": "...", "rating": 5 }	{ "success": true }
DELETE	/deletereview/:reviewId	Delete review	—	—	{ "success": true }
GET	/getreviews/:bookId	Get reviews by book	—	—	{ "reviews": [...] }

8. AUTHENTICATION:

Authentication and authorization are handled using **JSON Web Tokens (JWT)** to ensure secure access to protected routes and resources.

1. User Authentication Flow

a. **User Registration** (`/auth/register`)
  i. Users provide basic details (e.g., name, email, password).
  ii. Passwords are hashed using **bcrypt** before being stored in the database.

b. **User Login** (`/auth/login`)
  i. The user submits email and password.
  ii. If valid, a **JWT token** is generated and returned in the response.

2. JWT Token Structure

a. The token contains the user's ID and role (if implemented).
b. It is signed using a secret key stored in the backend (usually in `.env` as `JWT_SECRET`).
c. The token has an expiration time (e.g., 1h or 24h) to enhance security.

Example Login Response:

```
json
Copy code
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6...",
  "user": {
    "id": "6612f6a5b23c7a87e05aef91",
    "name": "Harshal",
    "email": "harshal2022@vitbhopal.ac.in"
  }
}
```

3. Authorization

a. Protected routes (like adding, updating, or deleting a book) require a valid JWT token in the request headers.
b. **How to send the token:**
In the frontend, include the token in the request headers:

```
makefile
Copy code
Authorization: Bearer <token>
```

4. Middleware for Protection

Custom middleware (e.g., `fetchUser`) is used to:
   i. Check the token's validity.
   ii. Decode the user info from it.
   iii. Attach the user to `req.user` for use in route controllers.

User Authentication Middleware:

```
js
Copy code
const jwt = require("jsonwebtoken");
const secretKey = "SSC";

const fetchUser = (req, res, next) => {
  // get user from jwt token and add id to req object
  const token = req.header("auth-token");
  if (!token) {
    return res.json({
      status: false,
      msg: "Login to access features :)",
    });
  }
  try {
    // extract payload data from the jwt by verifying jwt with the help of
secret key
    const data = jwt.verify(token, secretKey);
    req.user = data;
    next();
  } catch (error) {
    return res.json({
      status: false,
      msg: "Server issue :(",
    });
  }
};

module.exports = fetchUser;
```

9. USER INTERFACE:

The **Book Store Website** features a clean, responsive UI built with **React.js** and styled using **CSS**. Below are screenshots showcasing different parts of the application:

1. Home Page
Displays a list of books fetched from the backend. Users can browse all available books with basic details like **title**, **author**, **genre**, and **rating**.

2. Book Details Page
When a user clicks on a book, they are taken to a detailed view with more info like **synopsis**, **author bio**, **publication year**, and **price**.

3. Login Page
Secure login form for users and admins. On successful login, a **JWT token** is issued and securely stored for authenticated access.

4. Responsive Design
Fully responsive layout across **desktop**, **tablet**, and **mobile devices**, ensuring a seamless user experience on all screen sizes.

10. TESTING:

Testing Objectives

- Ensure functionality of book listings, user login/signup, and order system.
- Validate responsiveness and UI/UX across various devices.
- Guarantee backend stability and database consistency.
- Confirm security measures like authentication and data protection.

2. Types of Testing

A. Unit Testing

- Test individual components and backend functions.
- Examples:
    - React components like **Book Card**, **Search Bar**, etc.
    - Node.js API routes: `/api/books`, `/api/orders`, `/api/users`.

B. Integration Testing

- Ensure the frontend and backend communicate correctly.
- Test end-to-end flows like:
    - User logs in → selects a book → places an order → receives confirmation.

C. Functional Testing

- Verify user actions behave as expected.
- Scenarios:

- Browse books.
- Place an order.
- Filter books by genre, price, or rating.

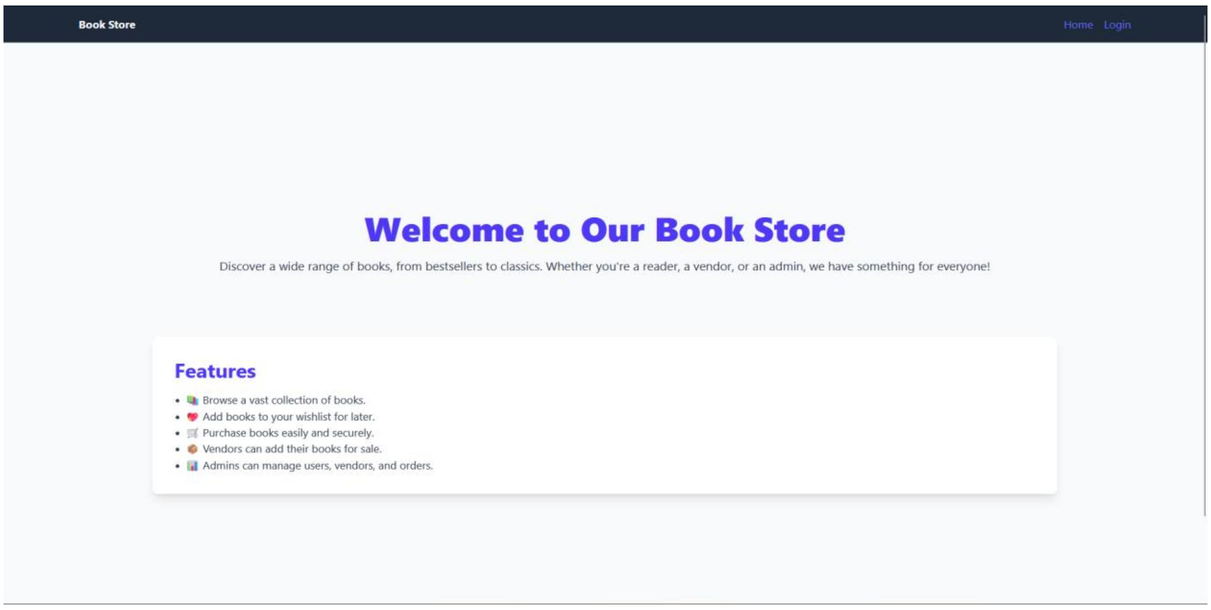
D. Performance Testing

- Test API response time under load using tools like **Postman** or **Apache JMeter**.
- Ensure the system handles high traffic during peak hours.

E. Security Testing

- Verify **JWT token validation**, role-based access, and input validation.
- Use tools like **OWASP ZAP** or **Postman** for basic security checks.

11. SCREENSHOTS:

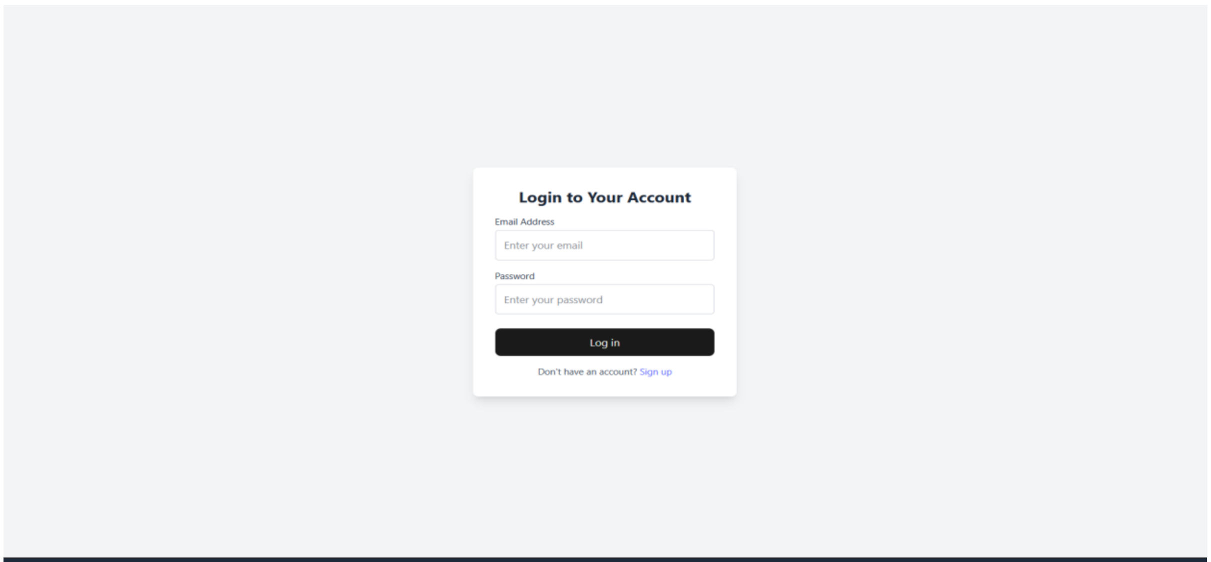


Welcome to Our Book Store

Discover a wide range of books, from bestsellers to classics. Whether you're a reader, a vendor, or an admin, we have something for everyone!

Features

- 📚 Browse a vast collection of books.
- ❤️ Add books to your wishlist for later.
- 🛒 Purchase books easily and securely.
- 📦 Vendors can add their books for sale.
- 📊 Admins can manage users, vendors, and orders.



Login to Your Account

Email Address

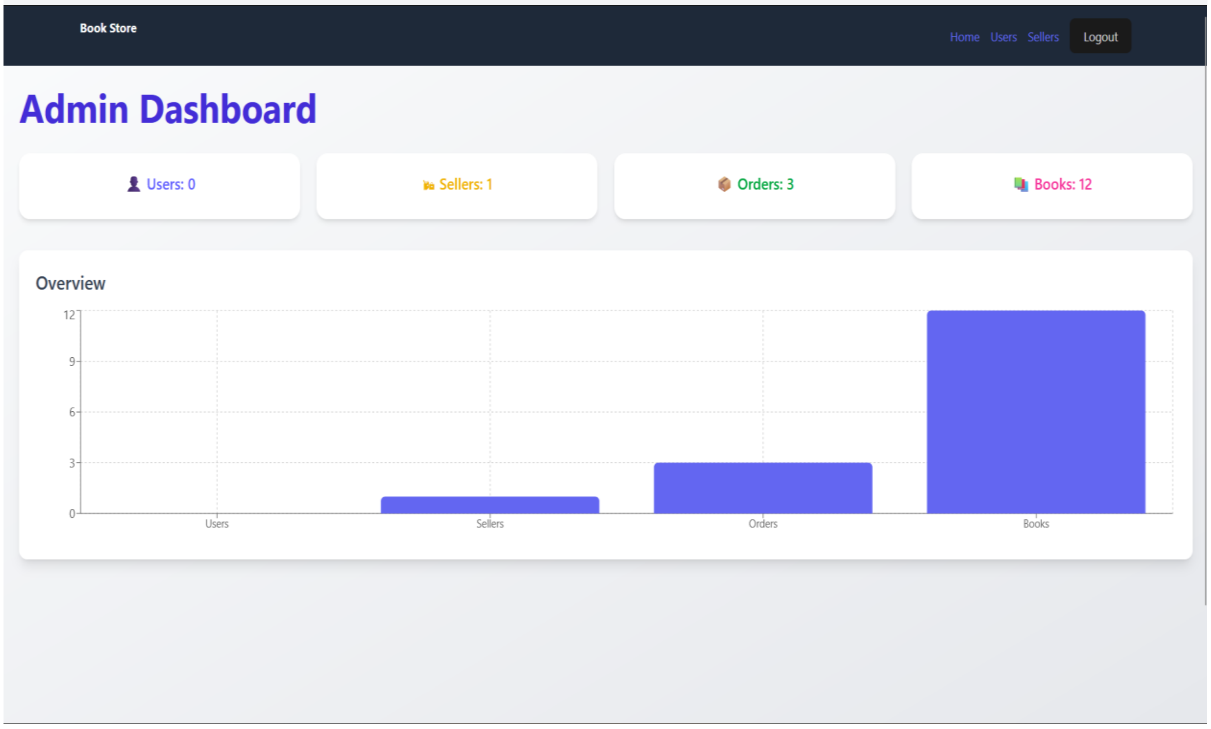
Enter your email

Password

Enter your password

Log in

Don't have an account? Sign up



Book Store

Home Users Sellers Logout

Admin Dashboard

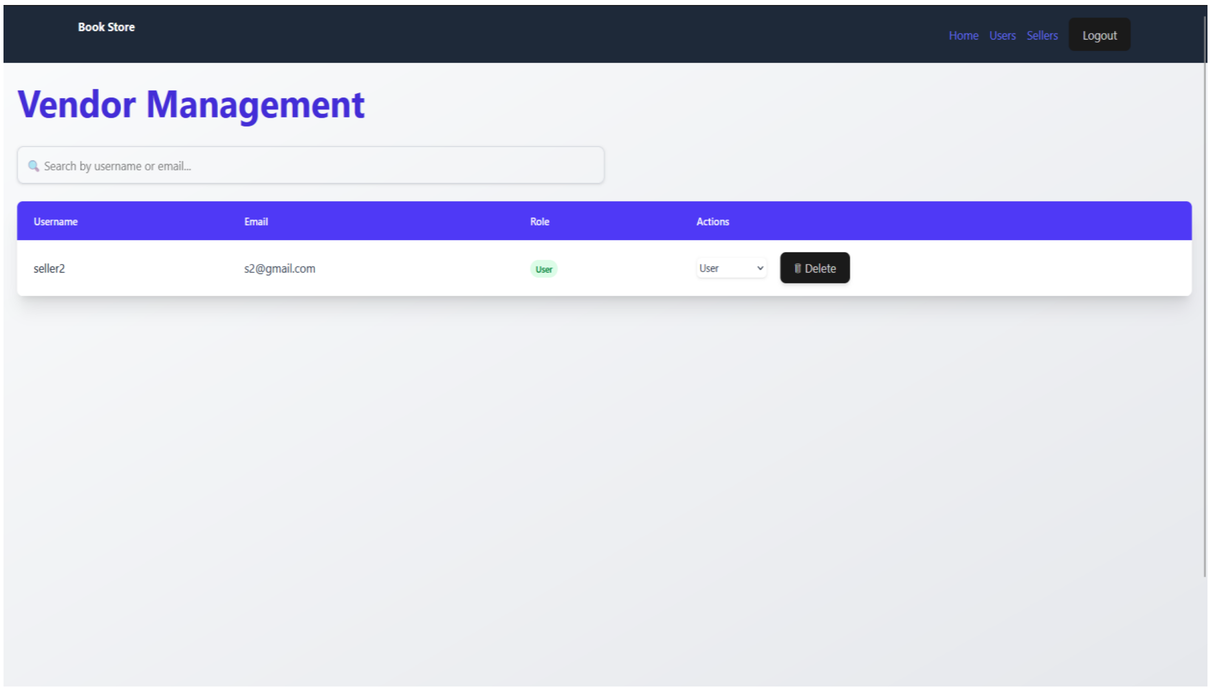
Users: 0

Sellers: 1

Orders: 3

Books: 12

Overview

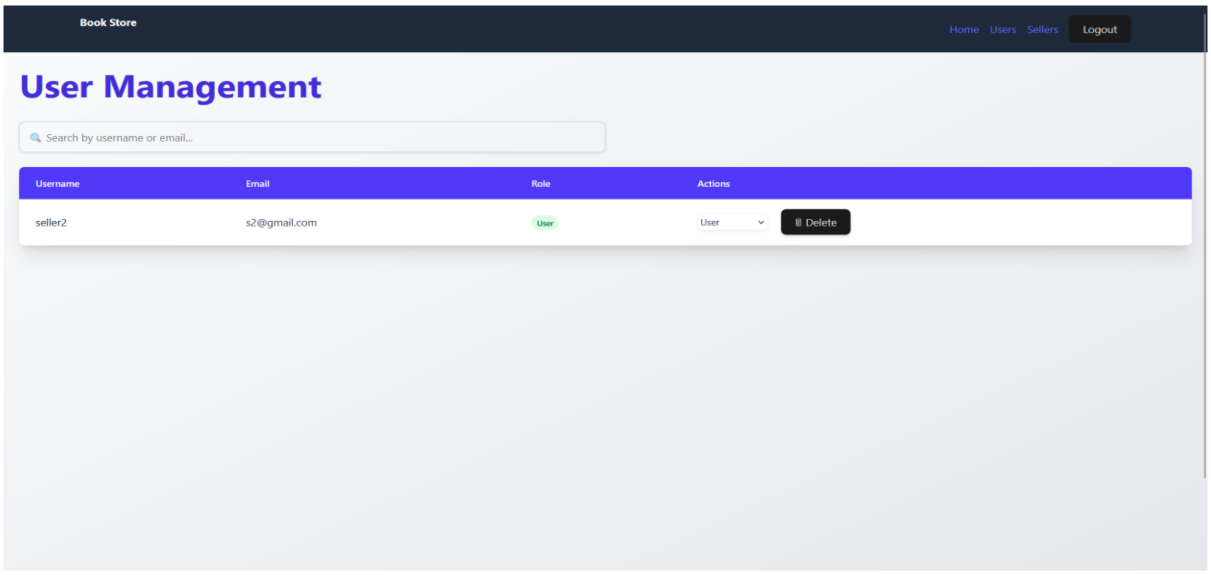

Book Store

Home Users Sellers Logout

Vendor Management

Search by username or email...

Username	Email	Role	Actions
seller2	s2@gmail.com	User	User ⌄ Delete


Book Store

Home Users Sellers Logout

User Management

Search by username or email...

Username	Email	Role	Actions
seller2	s2@gmail.com	User	User ⌄ Delete

12. KNOWN ISSUES:

This section outlines any existing bugs, limitations, or issues that users or developers should be aware of when using or contributing to the Book Store application.

1. No Pagination on Book List

- All books are fetched and displayed at once.
- May cause slow performance as the number of books increases.

2. Lack of Image Upload Functionality

- Currently, book covers must be added manually via external URLs.
- There is no feature to upload and store images locally or on a cloud service.

3. Basic Error Handling

- API error messages are not consistently user-friendly.
- Backend errors may not be clearly displayed on the frontend.

4. No Role-Based Access Control (RBAC)

- There is no differentiation between regular users and admin users.
- All authenticated users can potentially access restricted features like adding/editing books.

5. Missing Password Reset Functionality

- Users cannot reset their password via email or OTP.
- This limits the account recovery process.

6. Minimal Form Validation

- Some input fields (like price or year) lack thorough validation.
- This may allow incorrect or malformed data into the database.

7. Mobile Responsiveness Bugs

- Some UI elements misalign or overlap on smaller screens.
- Optimization needed for devices below 360px width.

FUTURE ENHANCEMENTS:

1. Implement Pagination and Infinite Scroll

- Improve performance and user experience on the book list page by limiting the number of books displayed at once.
- Add infinite scroll or numbered pagination.

2. Role-Based Access Control (RBAC)

- Differentiate between regular users and admins.
- Allow only admins to add/edit/delete books, while regular users can only view content.

3. Book Image Upload

- Integrate a file upload feature for book covers using services like **Cloudinary**, **Firebase Storage**, or local server storage.

4. Search, Filter & Sort

- Add search functionality by title, author, or genre.
- Filter books by genre, price, or rating.
- Sort books based on release date, popularity, or rating.

5. User Profile & Favorites

- Enable users to create profiles and save favorite books to a wishlist.
- Show personalized recommendations based on user activity.

6. Password Reset & Email Verification

- Add "Forgot Password" functionality using email OTP/token.
- Require email verification after registration for improved security.

7. Improved Mobile UI

- Further optimize the layout for all screen sizes.
- Ensure smooth responsiveness for small devices (≤360px).

8. Admin Dashboard

- Create a dedicated admin panel to manage books, users, and site analytics.