

# Full Stack Development with MERN

## Topic: Online Book Store Application

### 1. Introduction

- **Project Title:** Online Book Store Application
- **Team Members:**
  - Jincy J S - Team Lead and Coordinator
  - Priyasahaana M - Frontend Developer
  - Vibiksha Bharathi P K - Backend Developer
  - Sangitaa M - Database Manager
  - Shanmugapriya T - API Integrator

### 2. Project Overview

- **Purpose:** To create a full-stack book store application using the MERN stack (MongoDB, Express, React, Node.js).
- **Goals:**
  - Build a user-friendly interface for browsing, searching, and purchasing books.
  - Implement a secure and efficient backend for managing book data, user accounts, and transactions.
  - Ensure seamless data flow between frontend and backend using RESTful APIs for a smooth user experience.
- **Features:**
  - **Book Browsing and Search:** Users can browse available books by category or use a search feature to find specific titles.
  - **User Authentication and Authorization:** Secure login and registration functionality, allowing only authorized users to make purchases.
  - **Shopping Cart and Checkout:** Users can add books to a cart and complete transactions through a checkout process.
- **Functionalities:**

This MERN bookstore app offers a comprehensive set of functionalities. Users can create accounts, log in, and securely store personal information. The application allows users to browse through a curated list of books, view details for each book, add books to their cart and proceed to checkout. Administrators have access to manage inventory, add or update book listings, and track orders. All these features are connected through RESTful APIs that ensure efficient communication between the frontend (React) and backend

(Node.js with Express). MongoDB is used as the database for managing book and user data.

### 3. Architecture

- **Frontend:** The frontend architecture of this bookstore app, built with **React**, follows a modular, component-based structure:

#### 1.Component Hierarchy:

Divided into reusable components (e.g., Header, BookList, Cart), each focusing on specific UI tasks and promoting code reuse.

#### 2.Routing with React Router:

Uses React Router for navigation, managing routes like /home, /books, and /checkout for a seamless, single-page experience.

#### 3.State Management:

Local component states manage UI data, while a context API or Redux (if used) handles the global state, ensuring consistent data across components.

#### 4.API Integration:

Communicates with the backend through centralized RESTful API calls to handle data fetching, user actions, and order processing.

#### 5.Styling and Responsiveness:

CSS modules or frameworks (e.g., Bootstrap) are used to create a responsive, mobile-friendly layout.

- **Backend:** The backend architecture of this bookstore app, built with **Node.js** and **Express.js**, is structured as follows:

#### 1.Server Setup:

Express.js serves as the core framework, for managing routes and handling HTTP requests.

#### 2.Routing:

They are organized by function, with routes for users, books, and orders (e.g., /api/books, /api/users).

#### 3.Controllers:

Controllers handle route logic, database interactions, and responses, keeping code modular and organized.

#### **4.Database Integration:**

MongoDB is accessed via Mongoose models, which define schemas for books, users, and orders.

#### **5.Middleware:**

Handles authentication (JWT), input validation, error handling, and logging.

- **Database:** The database schema of the bookstore application, built with **MongoDB** and **Mongoose**, is designed to store and manage data related to users, books, and orders. Below is a detailed description of the schema and interactions:

##### **1. User Schema**

This schema handles user-related data, including registration, authentication, and account management.

- **Fields:**
  - name: String (required)
  - email: String (unique, required)
  - password: String (required, hashed)
  - role: String (optional, default: 'user')
  - createdAt: Date (auto-generated)
- **Interactions:**
  - **Registration:** The user submits a form, and the data is saved after hashing the password using a hashing library like bcrypt.
  - **Login:** On login, the email is matched, and the password is compared with the hashed version stored in the database. If valid, a JWT token is issued for session management.

##### **2. Book Schema**

This schema stores details about the books available for purchase.

- **Fields:**
  - title: String (required)
  - author: String (required)
  - price: Number (required)
  - category: String (optional)

- description: String (optional)
- imageUrl: String (optional, URL to an image)
- createdAt: Date (auto-generated)
- **Interactions:**
  - **CRUD Operations:** Admins can create, read, update, and delete book entries via the backend.
  - **Search:** Users can search for books by title, author, or category, with queries being run through the MongoDB database.

### 3. Cart Schema

This schema stores items added to the cart by users before checkout.

- **Fields:**
  - userId: ObjectId (reference to the User model)
  - items: Array of objects (each containing a reference to a book and quantity)
  - createdAt: Date (auto-generated)
- **Interactions:**
  - **Add to Cart:** Items are added to the cart with the user ID and book ID, along with the quantity.
  - **Update Cart:** Quantity changes or item removal is handled by updating the items array.
  - **Clear Cart:** After checkout or cancellation, the cart is cleared for the user.

### 4. Order Schema

This schema stores order-related data after a user proceeds with the checkout.

- **Fields:**
  - userId: ObjectId (reference to the User model)
  - items: Array of objects (each containing book reference, quantity, and price)
  - totalAmount: Number (calculated total of the order)
  - status: String (e.g., "pending", "completed", "cancelled")
  - shippingAddress: String
  - paymentStatus: String (e.g., "paid", "unpaid")
  - createdAt: Date (auto-generated)
- **Interactions:**
  - **Create Order:** Once checkout is complete, an order is created with the selected items and total amount.
  - **Order Status:** Admins can update the status of orders (e.g., mark as completed).
  - **Order History:** Users can view their past orders via references to userId.

## MongoDB Interactions:

- **Creating Documents:** Documents are created using Mongoose's `Model.create()` or `Model.save()` methods, storing user, book, or order data in the respective collections.
- **Reading Documents:** To retrieve data, queries are made using Mongoose's `Model.find()`, `Model.findById()`, or `Model.findOne()`, often filtering by various fields such as user ID or book category.
- **Updating Documents:** Updates are handled using `Model.updateOne()` or `Model.findByIdAndUpdate()`, such as changing the cart items or updating the order status.
- **Deleting Documents:** The `Model.deleteOne()` or `Model.findByIdAndDelete()` methods are used to remove data, such as deleting a book or clearing the cart.
- **Aggregation:** For complex queries (e.g., calculating total order prices), MongoDB's aggregation framework can be used to process data directly within the database.

## Schema Relationships:

- **User-Order Relationship:** A one-to-many relationship between the User and Order schemas (one user can have multiple orders).
- **Cart-User Relationship:** A one-to-one relationship between the Cart and User, where each user has one cart at a time.
- **Order-Book Relationship:** Many-to-many relationship between Order and Book (an order can have multiple books, and a book can be part of many orders).

## 4. Setup Instructions

- **Prerequisites:**
  - **Node.js** – JavaScript runtime for building the server.
  - **Express.js** – Web framework for handling routing and middleware.
  - **MongoDB** – NoSQL database for storing user, book, cart, and order data.
  - **Mongoose** – ODM (Object Data Modeling) library for MongoDB to define schemas and interact with the database.
  - **JWT (JSON Web Token)** – For user authentication and secure session management.
  - **Bcrypt.js** – For hashing user passwords.
  - **Axios** – HTTP client for making API requests from the frontend.
  - **React** – Frontend JavaScript library for building UI components.
  - **React Router** – For handling client-side routing in React.
  - **Bootstrap** or **CSS Framework** – For responsive UI design.

- **Installation:**

1. **Clone the Repository:** Open your terminal and run the following command to clone the repository:

```
git clone https://github.com/mdalmamunit427/build-full-stack-book-store-mern-app.git
```

2. **Navigate to the Project Folder:** Move into the project directory:

```
cd build-full-stack-book-store-mern-app
```

3. **Install Backend Dependencies:** Navigate to the backend folder and install the necessary dependencies:

```
cd backend
```

```
npm install
```

4. **Install Frontend Dependencies:** Navigate to the frontend folder and install the required dependencies:

```
cd ../frontend
```

```
npm install
```

6. **Start the Backend Server**

Run the backend server:

```
cd ../backend
```

```
npm start
```

7. **Start the Frontend Server:**

Run the frontend server:

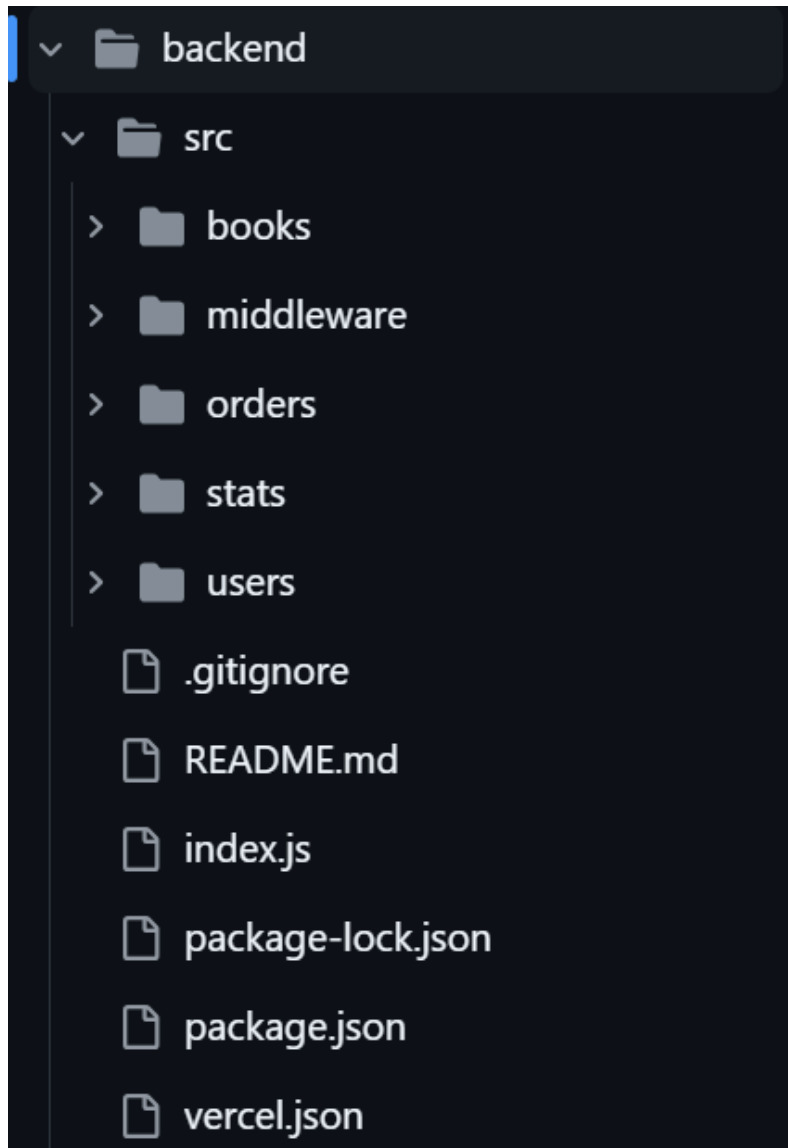
```
cd ../frontend
```

```
npm start
```

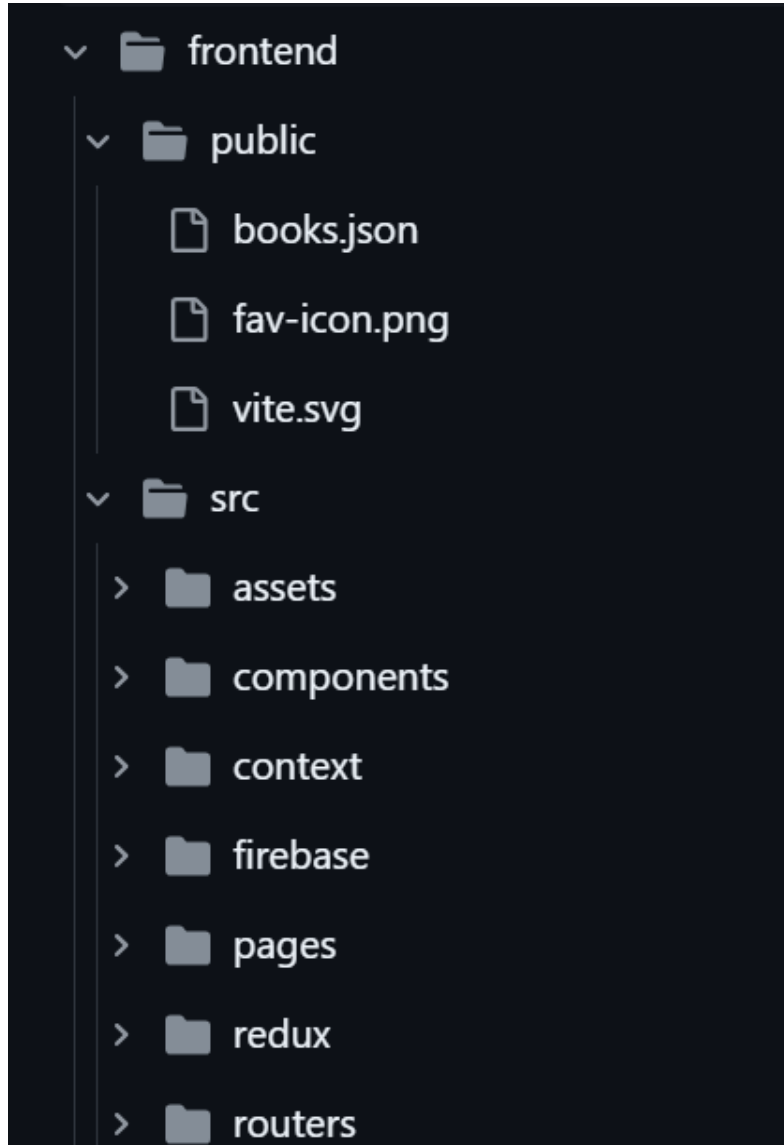
8. **Access the Application:** Open a browser and go to <http://localhost:3000> to view the application.

## 5. Folder Structure

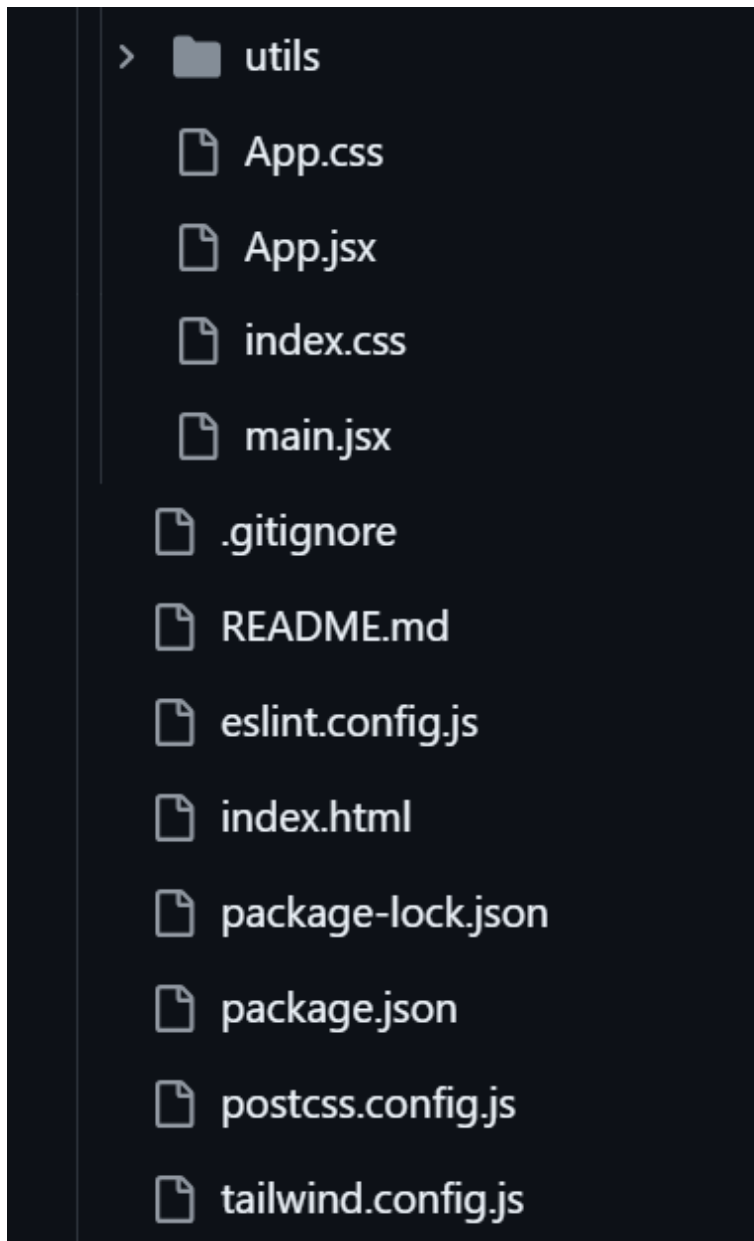
- Server:



- **Client:**







## 6. Running the Application

- **Frontend:** npm run dev in the client directory.
- **Backend:** npm start in the server directory.
- To run the entire project use npm run start:dev command.

## 7. API Documentation

### 1. Books API

**POST /books**

- **Description:** Adds a new book to the database.
- **Method:** POST

**Request Body:**

```
{  
  "title": "String",  
  "author": "String",  
  "price": "Number",  
  "trending": "Boolean"  
}
```

**Response:**

- **200 OK:**

```
{  
  "message": "Book posted successfully",  
  "book": {  
    "_id": "String",  
    "title": "String",  
    "author": "String",  
    "price": "Number",  
    "createdAt": "Date",  
    "updatedAt": "Date"  
  }  
}
```

- **500 Internal Server Error:** Failed to create the book.

**GET /books**

- **Description:** Retrieves all books from the database.
- **Method:** GET
- **Response:**
  - **200 OK:** Array of book objects, sorted by creation date.
  - **500 Internal Server Error:** Failed to fetch books.

**GET /books/**

- **Description:** Retrieves details of a single book by its ID.
- **Method:** GET
- **Path Parameters:**

- id: Book ID
- **Response:**
  - **200 OK:** Book object
  - **404 Not Found:** Book not found.
  - **500 Internal Server Error:** Failed to fetch book.

#### **PUT /books/**

- **Description:** Updates an existing book's details.
- **Method:** PUT
- **Path Parameters:**
  - id: Book ID
- **Request Body:**
  - JSON containing fields to update (e.g., title, author, price)
- **Response:**
  - **200 OK:**

```
{
  "message": "Book updated successfully",
  "book": { ...updated book object... }
}
```
  - **404 Not Found:** Book not found.
  - **500 Internal Server Error:** Failed to update a book.

#### **DELETE /books/**

- **Description:** Deletes a book by its ID.
- **Method:** DELETE
- **Path Parameters:**
  - id: Book ID
- **Response:**
  - **200 OK:**

```
{
  "message": "Book deleted successfully",
  "book": { ...deleted book object... }
}
```
  - **404 Not Found:** Book not found.
  - **500 Internal Server Error:** Failed to delete a book.

## 2. Orders API

### POST /orders

- **Description:** Creates a new order.
- **Method:** POST
- **Request Body:**

```
{  
  "email": "String",  
  "bookId": "String",  
  "quantity": "Number",  
  "totalPrice": "Number"  
}
```
- **Response:**
  - **200 OK:** Order object
  - **500 Internal Server Error:** Failed to create order.

### GET /orders/

- **Description:** Retrieves orders for a given email.
- **Method:** GET
- **Path Parameters:**
  - email: Customer email address
- **Response:**
  - **200 OK:** Array of order objects, sorted by creation date.
  - **404 Not Found:** No orders found for the given email.
  - **500 Internal Server Error:** Failed to fetch orders.

## 3. Admin Statistics API

### GET /admin/stats

- **Description:** Retrieves statistical data for admin use.
- **Method:** GET
- **Response:**
- **200 OK:**

```
{  
  "totalOrders": "Number",  
  "totalSales": "Number",  
  "trendingBooks": "Number",  
  "totalBooks": "Number",  
}
```

```

    "monthlySales": [
      {
        "_id": "YYYY-MM",
        "totalSales": "Number",
        "totalOrders": "Number"
      },
      ...
    ]
  }

```

- **500 Internal Server Error:** Failed to fetch admin stats.

## 4. User Authentication API

### POST /user/admin

- **Description:** Authenticates admin and generates JWT.
- **Method:** POST
- **Request Body:**

```

{
  "username": "String",
  "password": "String"
}

```
- **Response:**
  - **200 OK:**

```

{
  "message": "Authentication successful",
  "token": "JWT token",
  "user": {
    "username": "String",
    "role": "String"
  }
}

```
  - **404 Not Found:** Admin not found.
  - **401 Unauthorized:** Invalid password.
  - **500 Internal Server Error:** Failed to log in as admin.

## **8. Authentication**

In this project, authentication and authorization are critical to ensuring secure and personalized user experiences. Here's a breakdown of how these are implemented:

### **User Registration and Login:**

New users can register by providing their email and a secure password. During login, these credentials are validated against stored data in the MongoDB database.

### **JWT Token-Based Authentication:**

Upon a successful login, a JSON Web Token (JWT) is generated and sent to the client. This token is signed with a secret key, making it secure and tamper-resistant. The token also has a defined expiration time to ensure that sessions remain secure and need periodic re-validation.

### **Client-Side Token Storage:**

The token is stored on the client side, typically in local storage or cookies. Local storage provides easier access for client-side JavaScript, while HTTP-only cookies offer added security by preventing JavaScript access to tokens, which is useful for mitigating Cross-Site Scripting (XSS) attacks.

### **Protected Routes:**

Middleware functions are used to protect certain backend routes, allowing access only to users with valid tokens. For example, only authenticated users can add books to their cart, make purchases, or view their order history.

### **Role-Based Authorization:**

Specific actions, such as adding or removing books, may be restricted to admin users. The user's role is embedded within the JWT payload, allowing the middleware to differentiate between general users and admins, ensuring that only authorized users can perform admin-level tasks.

## 9. User Interface

The User Interface (UI) is designed to be both intuitive and visually appealing, ensuring a seamless experience for users as they navigate through the bookstore. Key UI elements include:

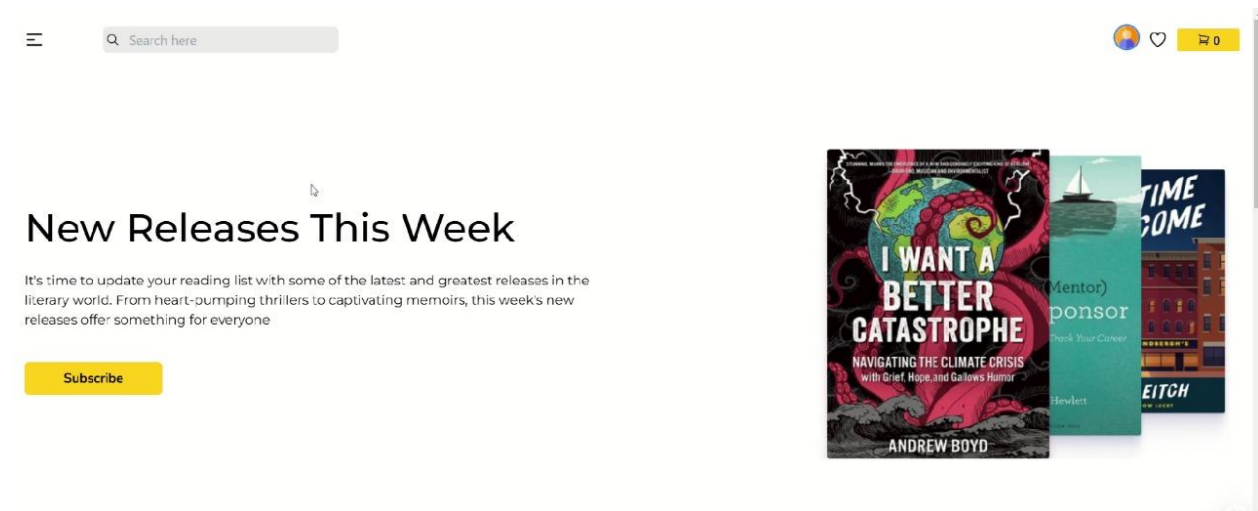
**Home Page:** Displays a catalog of available books with thumbnails, titles, authors, and prices. Users can view details of each book, add books to their cart, or initiate a purchase.

**Authentication Pages:** Simple and responsive login and signup forms make user onboarding easy.

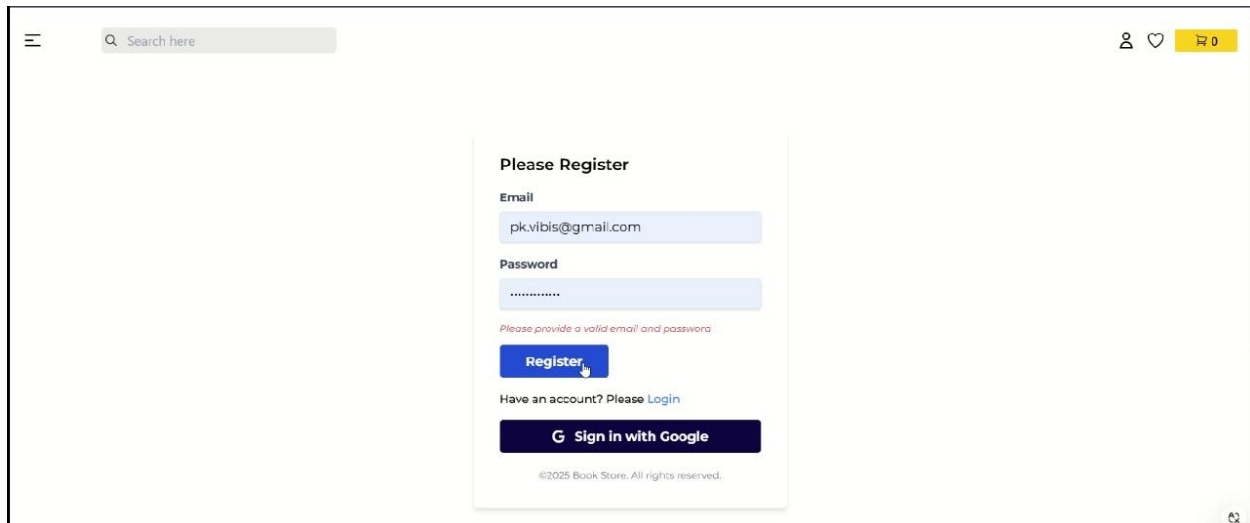
**User Dashboard:** Once logged in, users can access their personalized dashboard to manage their account, view purchase history, and check their saved items.

### Screenshots of UI:

#### The book catalog on the home page



## The login and signup forms



The image shows a web page with a registration form titled "Please Register". The form is centered on the page. At the top of the page, there is a search bar with the text "Search here" and a hamburger menu icon on the left. On the right, there are icons for a user profile, a heart, and a shopping cart with the number "0". The registration form has the following elements:

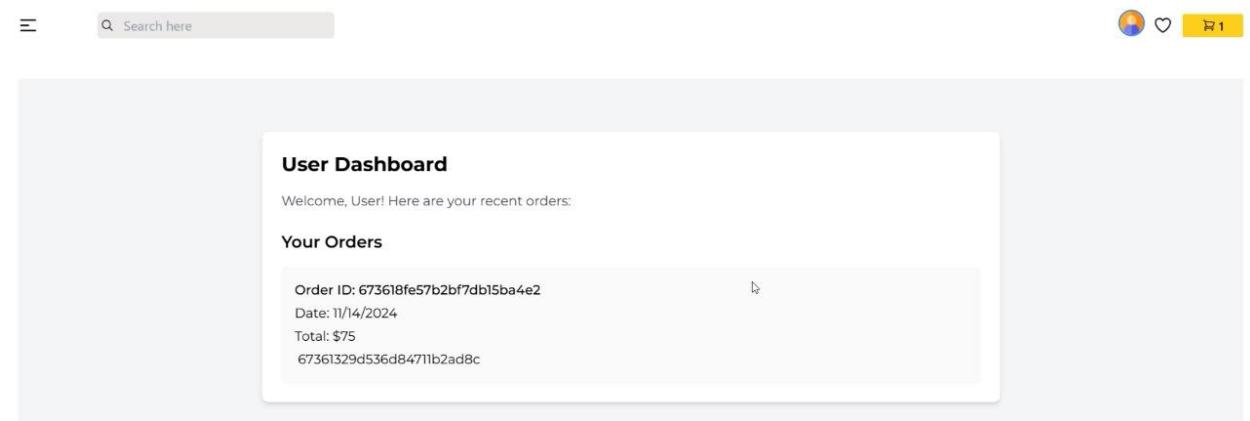
- Please Register** (Title)
- Email** (Label): A text input field containing "pk.vibis@gmail.com".
- Password** (Label): A password input field with masked characters "\*\*\*\*\*".
- Please provide a valid email and password* (Error message in red text).
- Register** (Blue button).
- [Have an account? Please Login](#) (Link).
- Sign in with Google** (Dark blue button with the Google logo).
- ©2025 Book Store. All rights reserved. (Footer text).

## User dashboard features (order history, cart management)



The image shows a user dashboard with a sidebar menu on the right. The main content area displays "Your Orders" with a list of order details. The sidebar menu includes a user profile icon, a heart icon, and a shopping cart icon with the number "1". The order details are as follows:

- Your Orders** (Section Header)
- #1** (Order Number)
- Order ID:** 673618fe57b2bf7db15ba4e2
- Name:** Vishai PK
- Email:** pk.vibis@gmail.com
- Phone:** 7639530116
- Total Price:** \$75
- Address:** paramankurichi, tamil nadu, India, 628213
- Products Id:** 67361329d536d84711b2ad8c



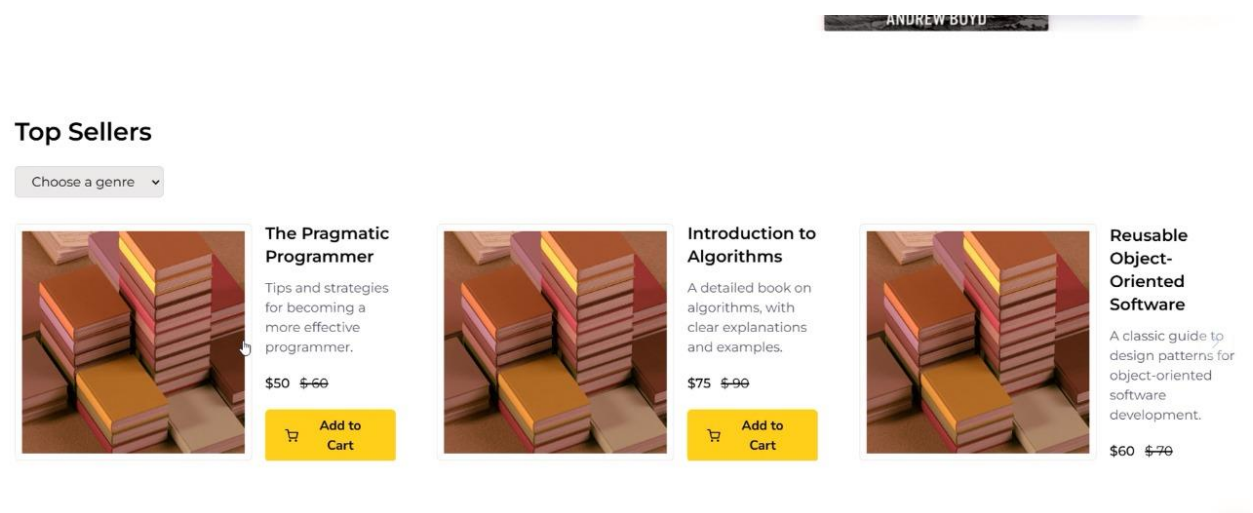
The image shows a "User Dashboard" summary card. The card has a title "User Dashboard" and a welcome message "Welcome, User! Here are your recent orders:". Below this, there is a section titled "Your Orders" which contains a table of order details.

Order ID	Date	Total
673618fe57b2bf7db15ba4e2	11/14/2024	\$75

The table also includes a "Products Id" column with the value "67361329d536d84711b2ad8c".



## Book details page with options to add to the cart or buy



## 10. Testing

Testing ensures that all functionalities work as expected and that the application remains robust under different conditions. The testing strategy includes:

**Unit Testing:** Each function and component, especially those related to user authentication, book management, and cart operations, are individually tested to ensure that they work in isolation. Jest is used for these unit tests.

**Integration Testing:** Interactions between the front-end and back-end are tested, focusing on core workflows like user registration, login, and checkout processes. API routes and database interactions are validated to ensure smooth data flow.

**End-to-End Testing:** Automated end-to-end tests simulate real user interactions from login to checkout. Cypress is used here to verify that the entire application behaves as expected when users navigate between pages and perform actions like adding items to the cart and completing purchases.

## 11. Demo

To showcase the application's features, provide a demo link. Suggested visuals include:

- **User Authentication Flow:** Login and signup screens.
- **Book Catalog:** The main catalog view displaying all available books.
- **Book Details:** A detailed view of a selected book with options to add it to the cart.

- **Cart and Checkout:** A step-by-step walkthrough of adding books to the cart and proceeding with a purchase.

**Watch the demo here:**

[https://drive.google.com/file/d/1RFICi3QfNmTsx8sJheaCIyCV-6naaJff/view?usp=drive\\_link](https://drive.google.com/file/d/1RFICi3QfNmTsx8sJheaCIyCV-6naaJff/view?usp=drive_link)

## **12.Known Issues**

Known bugs or limitations include:

- **Pagination Loading Errors:** Occasionally, page loading errors may occur if the server response is slow, requiring users to reload.
- **Token Expiry:** Users may occasionally need to re-login if the JWT expires during a session.

## **13.Future Enhancements**

Potential future improvements include:

- **Advanced Search and Filtering:** Adding search capabilities for book genre, author, and publication date would enhance the browsing experience.
- **Wishlist Feature:** Allowing users to save books for later purchase.
- **Enhanced Cart Management:** Adding features such as discount codes, estimated delivery dates, and order tracking.
- **Mobile App:** Expanding the bookstore to mobile platforms to reach a broader audience.