BOOKSTORE PLATFORM USING MERN

TEAM MEMBERS AND NM ID

1.KARTHIKEYAN S - DFE5E40976F31CCBDC44C87525818039
2.KARTHIK M - C92C69E7D1A32C4AD0E46F1923B84D54
3.HARIS P - 8DFDB0A2672A942811B46C2919C62A30
4.KABILAN B - 5296D06FAF5E45E82E04B94054FD3923

ABSTRACT

The bookstore-Management-System is a web-based application designed to simplify the process of browsing, purchasing, and managing books through an efficient and user-friendly platform. Built using the MERN stack (MongoDB, Express.js, React.js, Node.js), the system enables customers to easily search for books, add them to their cart, and complete purchases, while store administrators can manage inventory, track orders, and update book details effectively. The platform also supports user authentication and role-based access control to ensure security and privacy.

The system is designed to cater to both customers and bookstore administrators, offering a seamless interface for browsing books, managing purchases, and keeping track of inventory. The project emphasizes scalability, performance, and security, incorporating technologies like JWT for authentication and bcrypt for password encryption. It also features a responsive and mobile-friendly design to ensure accessibility across all devices.

This report explores the system's functional and non-functional requirements, system architecture, database design, and API design, along with detailed steps on implementation and deployment. Additionally, it outlines the security features integrated into the application, such as JWT authentication and role-based access control. Furthermore, the report discusses testing methodologies including unit and integration tests, and explores challenges faced during development, along with their solutions. The final section looks at future enhancements and the potential for expanding the system's features, such as introducing advanced recommendation systems or integrating with third-party sellers.

The bookstore-Management-System offers a robust solution for improving the efficiency of book management, contributing to better service delivery in the bookstore industry, and providing an enhanced user experience for both customers and administrators.

TABLE OF CONTENT

1.INTRODUCTION

Project Overview:

The bookstore-Management-System is a comprehensive web application designed to streamline the process of browsing, purchasing, and managing books. This system provides a platform for customers to search for books based on various criteria, such as genre, author, or price range, and place orders. Additionally, it allows bookstore administrators to manage the inventory, update book details, and track customer orders.

Purpose and Motivation:

With the increasing number of online book purchases and the demand for seamless shopping experiences, traditional methods of managing bookstore operations can often lead to inefficiencies, such as inventory mismanagement and long order processing times. This project is motivated by the need for a digital solution that enhances the customer shopping experience while improving the efficiency of managing a bookstore's inventory and order processing.

Objectives:

The key objectives include creating an easy-to-use interface for customers to search and purchase books, secure access for both customers and administrators, and tools for administrators to manage the bookstore’s inventory. The system places a strong emphasis on scalability, usability, and security to provide a robust and efficient platform for both customers and bookstore staff.

Target Audience:

The primary users of the bookstore-Management-System are:

Customers: People searching for books and making purchases.Bookstore

Administrators: Individuals managing the inventory,order processing, and customer interactions.Administrative Staff: Those overseeing the system's operation and ensuring it runs smoothly.

2.PROJECT REQUIREMENT:

Functional Requirements:

Customers:

Register/login, manage personal profiles, browse and search for books by genre, author, or price, add books to the cart, place orders, and view order history.

Bookstore Administrators:

Register, update personal profiles, manage inventory (add, remove, or update book details), view customer orders, and update order statuses

(e.g., shipped, out of stock).

Admin:

Oversee user management (customers and administrators), monitor system performance, verify orders, and ensure inventory is up to date.

Non-Functional Requirements:

Performance:

Handle high user traffic, including multiple customers browsing and purchasing books at once, and execute searches efficiently.

Reliability:

Ensure system availability and minimize downtime to provide a seamless shopping experience for customers.

Security:

Implement JWT for secure customer and admin authentication, encrypt passwords, and follow best practices for securing user data and transactions (e.g., payment details).

Scalability:

Design the system to easily expand with an increasing number of books, customers, and features like new payment gateways or recommendation systems.

Responsiveness:

Provide a UI that adapts smoothly across different devices and screen sizes, ensuring a consistent shopping experience on desktops, tablets, and smartphones.

3.SYSTEM ARCHITECTURE:

The system architecture of the bookstore-Management-System is designed to ensure smooth interaction between the frontend, backend, and the database. The architecture follows a client-server model, using the MERN stack (MongoDB, Express.js, React.js, Node.js). Below, I will elaborate on the architecture and include details on the various components involved.

Architecture Overview:

The architecture can be broken down into three major components:

- Frontend (Client-side): React.js
- Backend (Server-side): Node.js with Express.js
- Database: MongoDB

These components work together to facilitate communication between the user and the system, making it easy to browse, purchase books, and manage inventory.

1. Frontend (React.js):

Role: The frontend handles the user interface and ensures smooth communication with the backend via API calls.

Components:

- **React Components:**These include UI elements such as buttons, forms, product cards, search filters, and the shopping cart used by customers and administrators. Components are reusable and state-driven to provide a dynamic experience.

- **State Management:**The application uses React's state management for storing the user's session, cart data, order history, and dynamic UI updates (e.g., displaying available books, price updates, etc.).

- **API Integration:**Axios or the Fetch API is used to make asynchronous requests to the backend for data retrieval, such as searching for books, updating inventory, placing orders, and managing customer profiles.

- **Routing:**React Router manages navigation within the app, allowing users to seamlessly move between pages like the homepage, book search, product details, checkout page, and user dashboard.

User Interaction:

- **Customer Side:**Customers can log in, browse books by genre, author, or price range, add books to their cart, place orders, and view their order history.

- **Admin Side:**Administrators can manage the bookstore’s inventory (add, remove, or update book details), view customer orders, update order statuses (e.g., processed, shipped), and manage system settings.

- **Admin Panel:**The admin panel allows bookstore staff to monitor inventory, oversee customer orders, and manage other system functionalities like user roles or store settings.

2. Backend (Node.js + Express.js):

- **Role:**The backend provides the core logic of the application, handles requests from the frontend, and interacts with the database to manage book listings, customer orders, and user profiles.

Components:

- **Node.js:**Node.js is a JavaScript runtime that allows the server to run JavaScript code. As a non-blocking, event-driven runtime, it is highly efficient in handling multiple customer requests simultaneously, such as browsing books, placing orders, and managing inventory.

- **Express.js:**Express.js is a web framework for Node.js that simplifies the development of APIs. It provides routing, middleware, and error handling. Express.js enables the system to expose RESTful endpoints to handle tasks such as user registration, book browsing, order placements, inventory updates, and more.

Key Features:

- **Authentication:**The backend handles user authentication using JWT (JSON Web Tokens). Once a user (customer or administrator) logs in, the server generates a JWT, which is used to authenticate subsequent requests. This ensures secure access to protected routes, such as placing an order or updating inventory.

- **Book Management:**The backend manages all book data, including adding, updating, and deleting books in the bookstore inventory. Administrators can modify book details like title, author, price, and stock availability.

- **Order Management:**The backend handles the creation, updating, and viewing of customer orders. It keeps track of order statuses (e.g., processing, shipped, delivered) and customer information.

- **Role-based Access Control:**The backend restricts access to specific API endpoints based on the user's role (e.g., customer, administrator). For example, only administrators can access routes to manage inventory or view all customer orders.

- **Error Handling:**Error middleware catches and returns appropriate responses when something goes wrong, such as invalid input, unauthorized access, or system errors.

3. Database (MongoDB):

Role: MongoDB stores and manages application data, including user information (customers and administrators), books in the inventory, customer orders, and other system-related data.

Features:

- **NoSQL:**MongoDB is a NoSQL database, which provides flexibility and scalability for the system. It uses collections and documents rather than traditional tables and rows, making it easier to store and manage complex, hierarchical data like books, orders, and customer profiles.

Collections:

The database consists of several collections, such as:

- **Users:** Stores customer and administrator details, including login credentials and profiles.

- **Books:** Stores information about each book, such as title, author, price, genre, and stock levels.

- **Orders:** Stores details of customer orders, including order items, quantities, total price, and order status.

Mongoose ORM:

MongoDB's native query language can be low-level and cumbersome, so Mongoose is used as an Object Data Modeling (ODM) library to interact with MongoDB. Mongoose provides an abstraction layer to simplify database operations (CRUD) and offers features like validation, schema modeling, and query building.

4.PROJECT OBJECTIVES

1. **Develop a full-stack bookstore application** using the MERN stack, emphasizing scalability and responsiveness.
2. **Create an intuitive front-end** that enables easy navigation, book searching, and purchasing for users.
3. **Implement a robust back-end** capable of handling multiple API requests while ensuring data security and fast response times.
4. **Integrate a NoSQL database (MongoDB)** to store book details, user profiles, order history, and reviews.
5. **Optimize cross-platform functionality** to ensure the application is compatible with desktop and mobile devices.
6. **Conduct thorough testing** to validate the application's functionality, security, and reliability.

5.TECHNOLOGY STACK

- **MongoDB**: Used for storing book details, user data, and transaction history. MongoDB's flexibility is ideal for managing dynamic data structures.
- **Express.js**: A back-end framework for building the API endpoints that support book searches, user account management, and transactions.
- **React.js**: A front-end library used to create a responsive, component-based interface, allowing users to interact with the bookstore efficiently.
- **Node.js**: Used for server-side scripting, handling API requests, and managing interactions between the front end and database.

6.IMPLEMENTATION

1. **MongoDB Setup**: Set up a MongoDB database on MongoDB Atlas or a local server for data storage.

2. **Node.js and Express.js Setup**: Initialize the back-end with necessary modules (`Express`, `Mongoose`) to handle server operations.

3. **React.js Setup**: Scaffold the front-end with Create React App, setting up components, routes, and integrating Redux for state management.

7.DEVELOPMENT PROCESS

- **Requirements Analysis**: Define features such as book listing, cart management, and user authentication.

- **Frontend Development**: Develop reusable components for books, users, and order handling, and implement responsive design.
- **Backend Development**: Create RESTful APIs for data handling, integrate MongoDB, and set up authentication.
- **Testing and Debugging**: Conduct unit and integration testing to validate the reliability of both front-end components and back-end APIs.

8.TESTING:

Types of Testing:

Unit Testing:

- Unit testing focuses on testing individual components or units of code to ensure they function correctly in isolation. Each unit (such as a function or a module) is tested by providing specific inputs and checking if the output is correct. This type of testing helps detect bugs early and ensures that every small part of the system is working as expected.

Example: For the BookList component in React, unit tests can be written to check if the component renders correctly, handles events like adding books to the cart, and manages state updates (such as the book count).
Integration Testing:

Integration testing:

- Integration testing ensures that different modules or components of the application work together as expected. While unit tests focus on individual components, integration tests validate the interactions between them. For example, when a user places an order, integration testing would check if the interaction between the frontend (React), backend (Express), and database (MongoDB) functions correctly.

Example: In the case of placing an order, an integration test can ensure that when an order is placed, the data flows from the frontend to the backend, the stock is updated in the database, and an order confirmation is generated.

End-to-End Testing:

- End-to-end testing is the process of testing the complete workflow of the application, from start to finish, to ensure that all components and interactions work as expected in a real-world scenario. This involves simulating user actions such as logging in, browsing books, adding to the cart, placing an order, and confirming the order.

Example: Testing the entire flow of placing an order, including user registration, book search, adding to the cart, proceeding to checkout, and order confirmation.

Testing Tools:

- **Jest:** Used for unit tests of React components.
- **Mocha:** Used for backend testing with Express.js routes.
- **Supertest:** Used for testing API routes in Express.
- **Postman:** Used for manual API testing and testing complex API requests.
- **Cypress:** Used for end-to-end testing to simulate user interactions and verify the overall workflow.

Unit Tests for Components:

Tests using Jest for front-end components, such as:

- Checking if the BookList component renders books correctly.
- Verifying if the Cart component updates when items are added or removed.
- Ensuring OrderForm validates input and triggers the correct order submission.

Integration Tests:

Tests using Supertest to verify:

- That the book creation route correctly adds new books to the database.
- That the order route properly processes the order, interacts with the database, and sends the correct response.
- That user authentication routes correctly issue and validate JWT tokens.

9.PROJECT STRUCTURE

```
book_store_using_MERN_stack
├── backend/
│         ├── db/
│         │    ├── admin/
│         │    ├── seller/
│         │    ├── user/
│         ├── uploads/
│         ├── package-lock.json
│         ├── package.json
│         ├── server.js
├── frontend/
│         ├── public/
│         ├── src/
│         │    ├── admin/
│         │    ├── componenets/
│         │    ├── seller/
│         │    ├── user/
│         │    ├── app.css
│         │    ├── app.jsx
│         │    ├── index.css
│         │    ├── main.jsx
│         ├── components/
│         ├── insex.html
│         ├── package-lock.json
│         ├── package.json
│         ├── vite.config.js
└── README.md
```

10.BACKEND

```
Backend/db/Admin/Admin.jsBackend/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)
// const express = require('express')
// const cors = require('cors')
// require('./db/config')
// const items = require('./db/Vendor/Additem')
// const multer = require('multer');

// // Set up Multer for file upload
// const storage = multer.diskStorage({
//   destination: 'uploads',
//   filename: function (req, file, callback) {
//     callback(null, Date.now() + '-' +
file.originalname);
//   },
// });


// const upload = multer({ storage });

// const app = express();
// app.use(express.json())
// app.use(cors(
//   {
//       origin: ["http://localhost:3000"],
//        methods: ["POST", "GET", "DELETE", "PUT"],
//        credentials: true
//    }
// ))
// app.use('/uploads', express.static('uploads'));


// //  ITEMS //

// app.post('/items', upload.single('itemImage'), async
(req, res) => {
//      const { description, itmetype,price } = req.body;
//      const itemImage = req.file.path; // The path to
the uploaded image

//      try {
//        const item = new items({ description,
itmetype,price,itemImage  });
//        await item.save();
//        res.status(201).json(item);
//      } catch (err) {
//        res.status(400).json({ error: 'Failed to create
item' });
//      }
//    });


// app.listen(8000, () => {
//    console.log("listening at 8000")
// })
```

/seller.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/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);
```

/myorder.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,
        default: () => new
Date().toLocaleDateString('hi-IN')
    },
    Delivery: {
      type: String,
      default: () => {
        const currentDate = new Date();
        currentDate.setDate(currentDate.getDate() + 7);
        const day = currentDate.getDate();
        const month = currentDate.getMonth() + 1;
        const year = currentDate.getFullYear();
        const formattedDate = `${month}/${day}/${year}`;
        return formattedDate;
      }
    }
})

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

```
/userschems.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/config.js

```
// const mongoose = require("mongoose");
// // Middleware
// const MONGO_URI =
'mongodb+srv://elf:elf123@myprojects.inzgx1q.mongodb.net/
BookStore?retryWrites=true&w=majority'
// // Connect to MongoDB using the connection string
// mongoose.connect(MONGO_URI, {
// //   useNewUrlParser: true,
// //   useUnifiedTopology: true,
// }).then(() => {
//   console.log(`Connection successful`);
// }).catch((e) => {
//   console.log(`No connection: ${e}`);
// });

// // mongodb://localhost:27017



const mongoose = require('mongoose');

mongoose.connect('mongodb://127.0.0.1:27017/BookStore', {
//   useNewUrlParser: true,
//   useUnifiedTopology: true,
});
```

Package.json

```
{
  "name": "backend",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit
1",
    "start": "nodemon server.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.18.2",
    "mongoose": "^8.0.1",
    "multer": "^1.4.5-lts.1",
    "nodemon": "^3.0.1"
  }
}
```

11.FRONTEND

Frontend/index.html

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <link rel="icon" type="image/svg+xml"
href="/vite.svg" />
    <meta name="viewport" content="width=device-width,
initial-scale=1.0" />
    <title>Book Store</title>
    <link
href="https://cdn.jsdelivr.net/npm/tailwindcss@2.2.16/dis
t/tailwind.min.css" rel="stylesheet">
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.jsx"></script>
  </body>
```
</html>

Frontend/src/index.css

```css
:root {
  font-family: Inter, system-ui, Avenir, Helvetica,
Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}

a {
  font-weight: 500;
  color: #646cff;
  text-decoration: inherit;
}
a:hover {
  color: #535bf2;
}

body {
  margin: 0;
  display: flex;
  place-items: center;
  min-width: 320px;
  min-height: 100vh;
```

```css
}

h1 {
  font-size: 3.2em;
  line-height: 1.1;
}

button {
  border-radius: 8px;
  border: 1px solid transparent;
  padding: 0.6em 1.2em;
  font-size: 1em;
  font-weight: 500;
  font-family: inherit;
  background-color: #1a1a1a;
  cursor: pointer;
  transition: border-color 0.25s;
}
button:hover {
  border-color: #646cff;
}
button:focus,
button:focus-visible {
  outline: 4px auto -webkit-focus-ring-color;
}

@media (prefers-color-scheme: light) {
  :root {
    color: #213547;
    background-color: #ffffff;
  }
  a:hover {
    color: #747bff;
  }
  button {
    background-color: #f9f9f9;
  }
}
```

Frontend/src/App.css

```
#root {
  max-width: 1280px;
  margin: 0 auto;
  padding: 2rem;
  text-align: center;
}
.logo {
  height: 6em;
  padding: 1.5em;
  will-change: filter;
  transition: filter 300ms;
}
.logo:hover {
  filter: drop-shadow(0 0 2em #646cffaa);
}
.logo.react:hover {
  filter: drop-shadow(0 0 2em #61dafbaa);
}
@keyframes logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

```css
@media (prefers-reduced-motion: no-preference) {
  a:nth-of-type(2) .logo {
    animation: logo-spin infinite 20s linear;
  }
}
.card {
  padding: 2em;
}
.read-the-docs {
  color: #888;
}
```

Frontend/src/App.jsx

```jsx
import 'bootstrap/dist/css/bootstrap.min.css';
import Login from './User/Login'
import { BrowserRouter, Route, Routes } from
'react-router-dom';
import Signup from './User/Signup';
import Unavbar from './User/Unavbar';
import Addbook from './Seller/Addbook';
import Item from './Seller/Book';
import Myproducts from './Seller/Myproducts';
import Slogin from './Seller/Slogin';
import Book from './Seller/Book';
import Orders from './Seller/Orders';
import Products from './User/Products';
import Uitem from './User/Uitem';
import Myorders from './User/Myorders';
import Uhome from './User/Uhome';
import OrderItem from './User/OrderItem';
import Shome from './Seller/Shome';
import Ssignup from './Seller/Ssignup';
import Home from './Componenets/Home';
import Alogin from './Admin/Alogin';
import Asignup from './Admin/Asignup';
import Ahome from './Admin/Ahome';
import Users from './Admin/Users';
import Vendors from './Admin/Seller';
import Seller from './Admin/Seller';
import Wishlist from './User/Wishlist';
// import './App.css'

function App() {
  return (
      <div>
       <BrowserRouter>
       <Routes>
       <Route path='/' element={<Home/>} />

      {/* Admin */}
      <Route path='/alogin' element={<Alogin/>} />
      <Route path='/asignup' element={<Asignup/>} />
      <Route path='/ahome' element={<Ahome/>} />
      <Route path='/users' element={<Users/>} />
      <Route path='/sellers' element={<Seller/>} />

           {/* seller */}
        <Route path='/slogin' element={<Slogin/>} />
      <Route path='/ssignup' element={<Ssignup/>} />
       <Route path='/shome' element={<Shome/>} />
       <Route path='/myproducts' element={<Myproducts/>}
/>
       <Route path='/addbook' element={<Addbook/>} />
       <Route path='/book/:id' element={<Book/>} />
       <Route path='/orders' element={<Orders/>} />
      {/* user */}
      <Route path='/login' element={<Login/>}/>
      <Route path='/signup' element={<Signup/>} />
        <Route path='/nav' element={<Unavbar/>}/>
         <Route path='/uhome' element={<Uhome/>} />
         <Route path='/uproducts' element={<Products/>}
/>
       <Route path='/uitem/:id' element={<Uitem/>} />
       <Route path="/orderitem/:id"
element={<OrderItem/>} />
       <Route path="/myorders" element={<Myorders />} />
       <Route path="/wishlist" element={<Wishlist />} />

       </Routes>
       </BrowserRouter>
      </div>
  )
}

export default App
```

Frontend/src/main.jsx

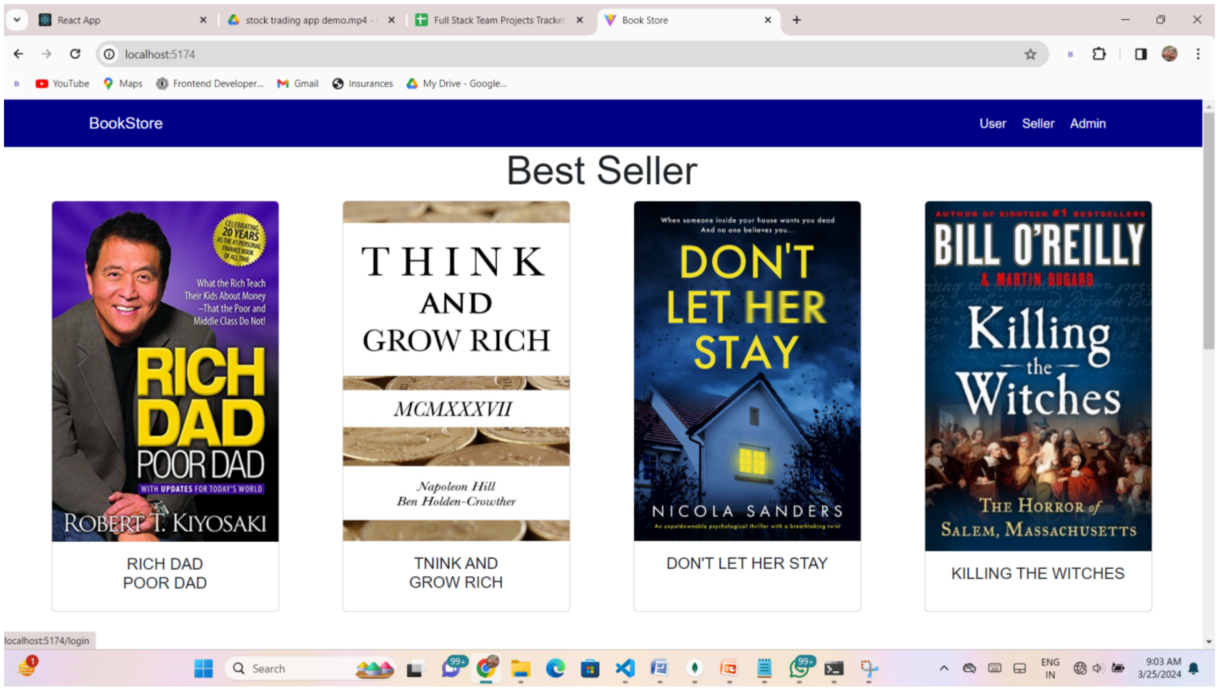
```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
ReactDOM.createRoot(document.getElementById('root')).rend
er(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Frontend/package.json

```
{
  "name": "frontend",
  "private": true,
  "version": "0.0.0",
  "type": "module",
  "scripts": {
    "dev": "vite",
    "build": "vite build",
    "lint": "eslint . --ext js,jsx
--report-unused-disable-directives --max-warnings 0",
    "preview": "vite preview"
  },
  "dependencies": {
    "axios": "^1.6.2",
    "bootstrap": "^5.3.2",
    "react": "^18.2.0",
    "react-bootstrap": "^2.9.1",
    "react-dom": "^18.2.0",
    "react-icons": "^4.12.0",
    "react-router-dom": "^6.19.0",
    "recharts": "^2.10.1",
    "tailwindcss": "^3.3.5"
  },
  "devDependencies": {
    "@types/react": "^18.2.37",
    "@types/react-dom": "^18.2.15",
    "@vitejs/plugin-react": "^4.2.0",
    "eslint": "^8.53.0",
    "eslint-plugin-react": "^7.33.2",
    "eslint-plugin-react-hooks": "^4.6.0",
    "eslint-plugin-react-refresh": "^0.4.4",
    "vite": "^5.0.0"
  }}
```

12.OUTPUT

Landing page:-



Best Seller

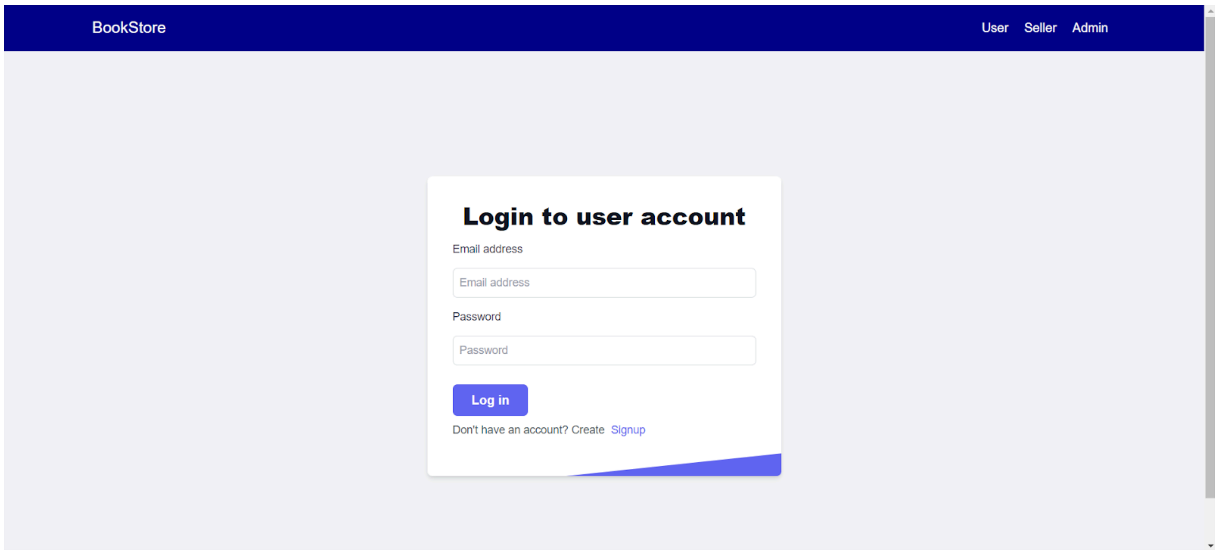
RICH DAD POOR DAD

TNINK AND GROW RICH

DON'T LET HER STAY

KILLING THE WITCHES

Login Page:-



Login to user account

Email address

Email address

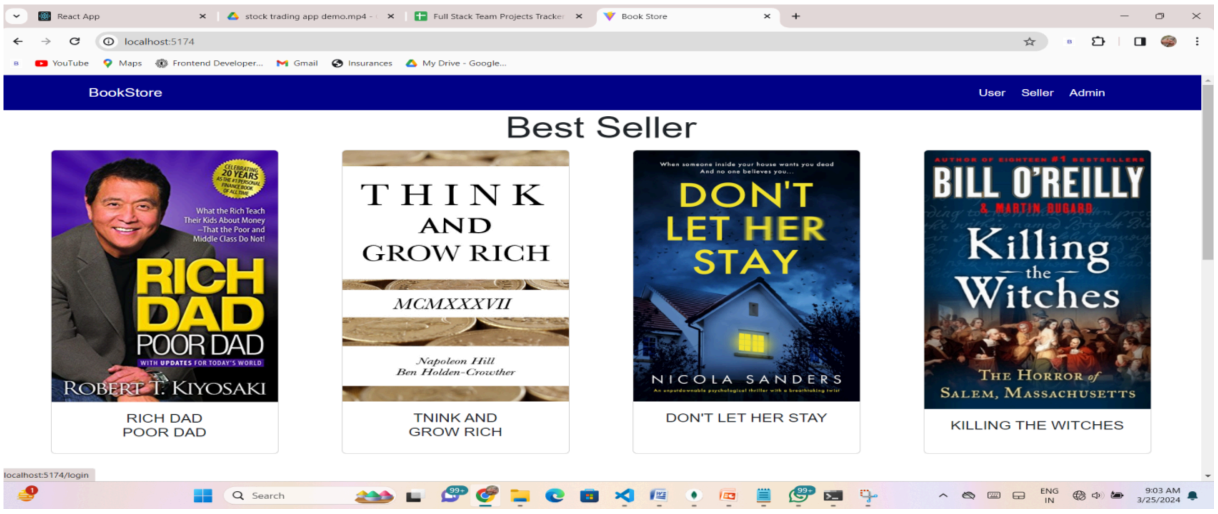
Password

Password

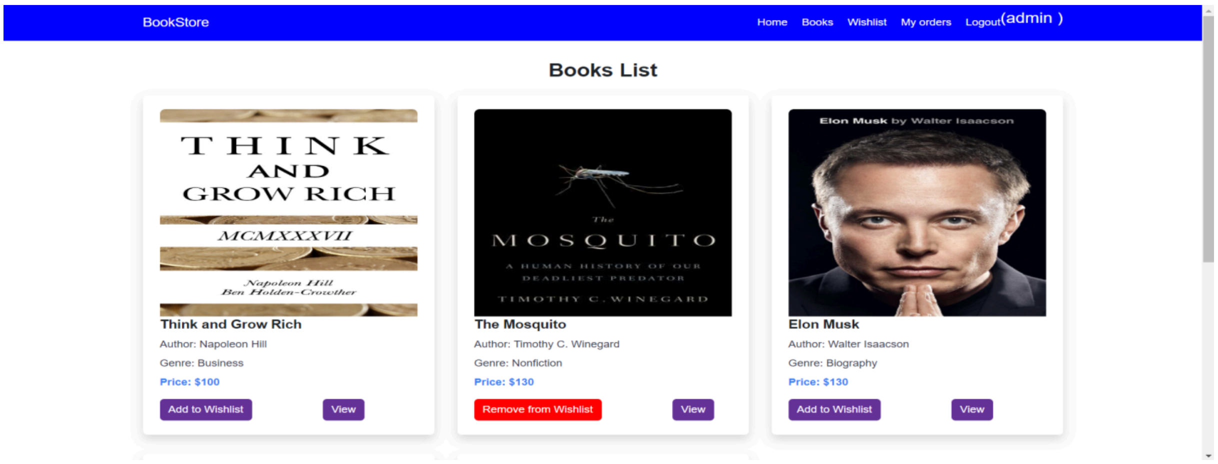
Log in

Don't have an account? Create Signup

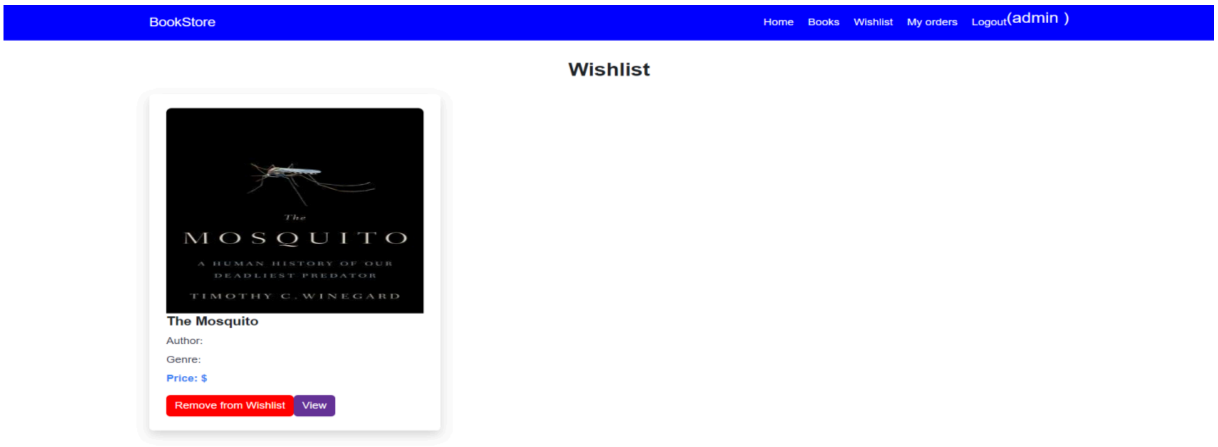
Home Page:-



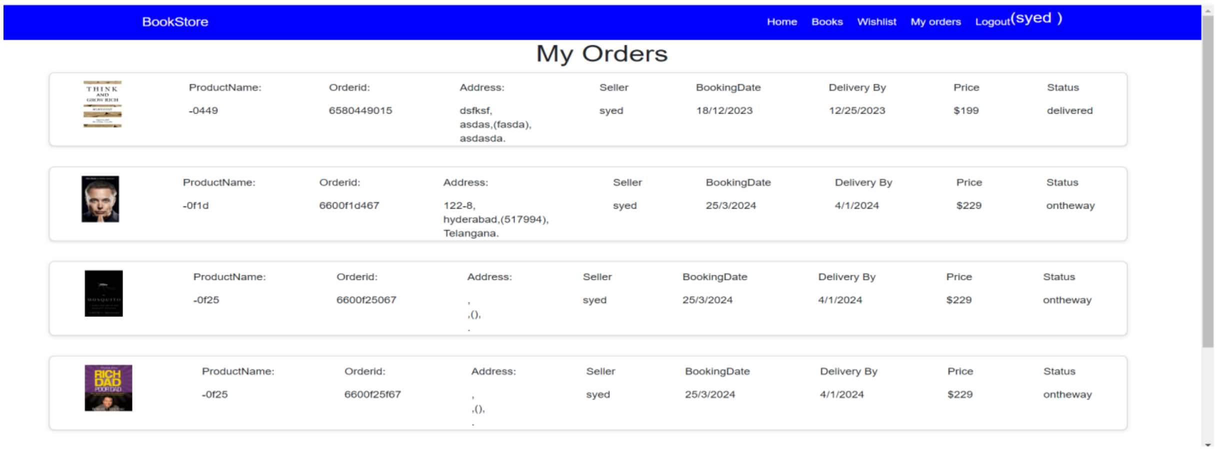
Books Page:-



Wishlist Page:-



My Bookings Page :-

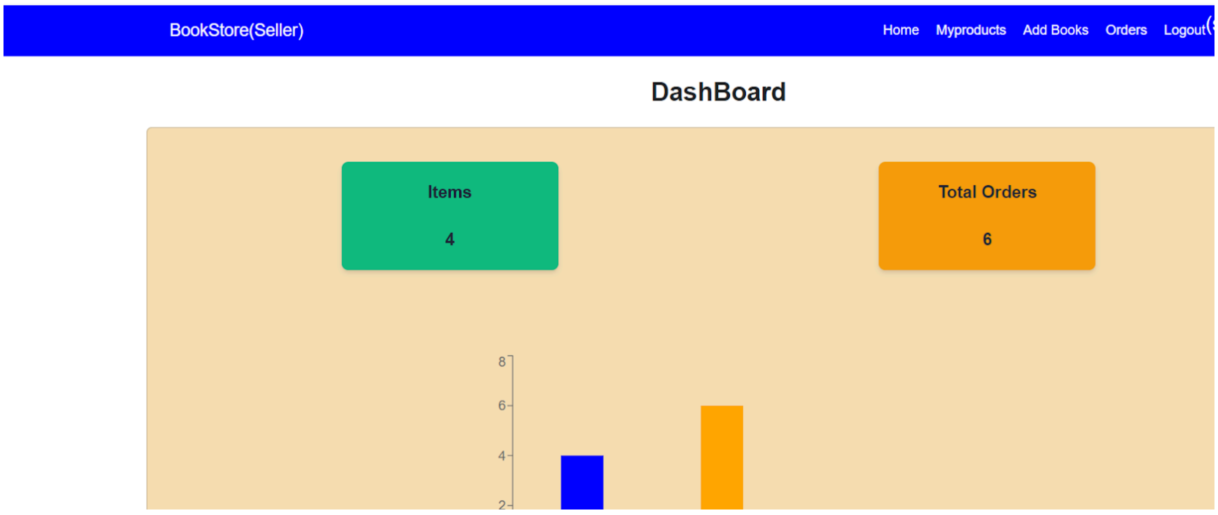


BookStore — Home · Books · Wishlist · My orders · Logout(syed)

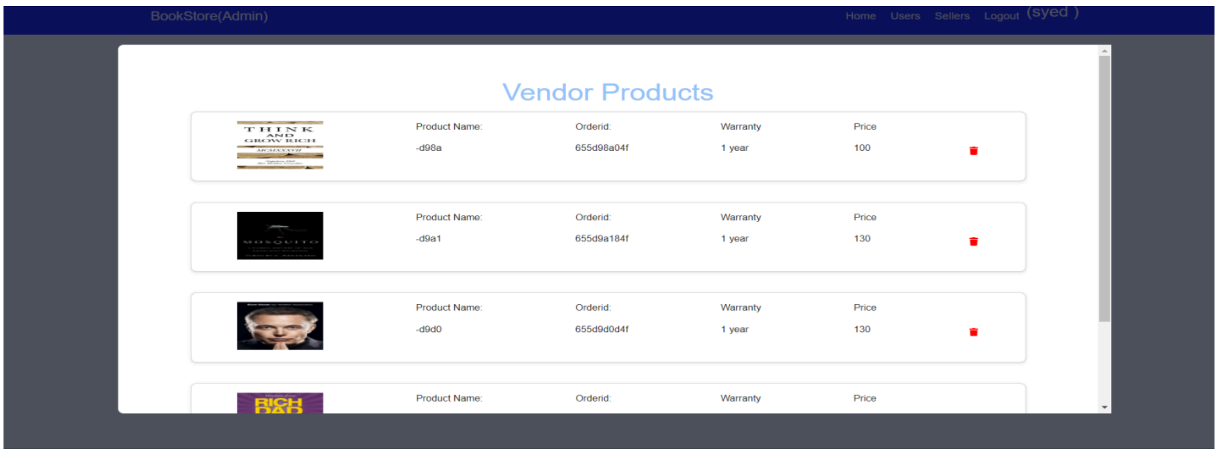
My Orders

	ProductName:	Orderid:	Address:	Seller	BookingDate	Delivery By	Price	Status
	-0449	6580449015	dsfksf, asdas,(fasda), asdasda.	syed	18/12/2023	12/25/2023	$199	delivered
	-0f1d	6600f1d467	122-8, hyderabad,(517994), Telangana.	syed	25/3/2024	4/1/2024	$229	ontheway
	-0f25	6600f25067	, ,(), .	syed	25/3/2024	4/1/2024	$229	ontheway
	-0f25	6600f25f67	, ,(), .	syed	25/3/2024	4/1/2024	$229	ontheway

Seller Dashboard:-



Seller Items:

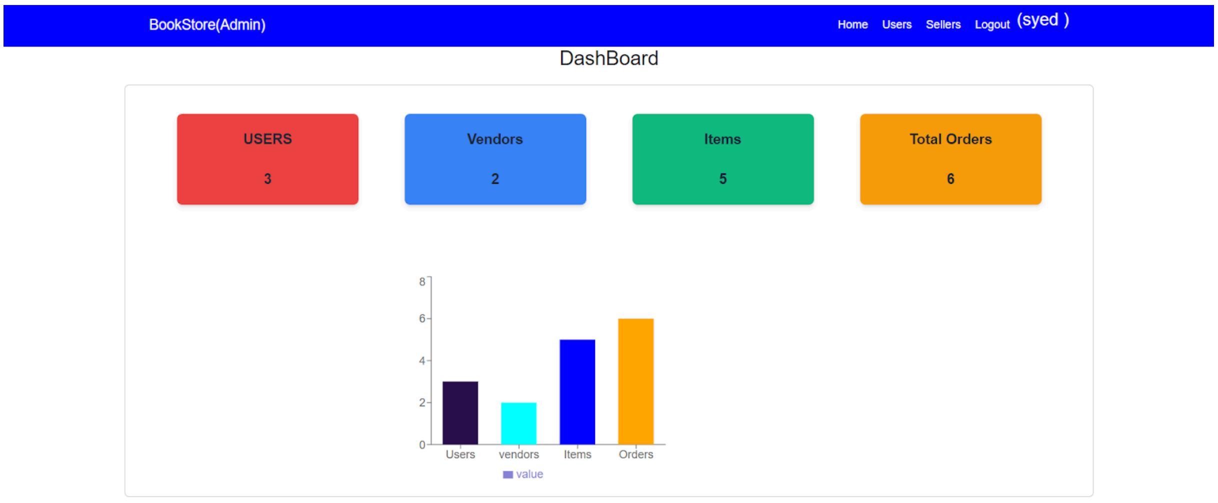


BookStore(Admin) — Home · Users · Sellers · Logout (syed)

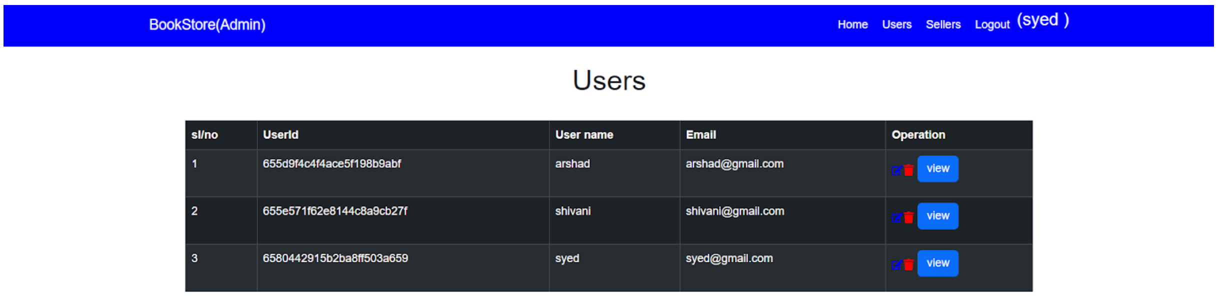
Vendor Products

	Product Name:	Orderid:	Warranty	Price
	-d98a	655d98a04f	1 year	100
	-d9a1	655d9a184f	1 year	130
	-d9d0	655d9d0d4f	1 year	130

Admin Dashboard:-



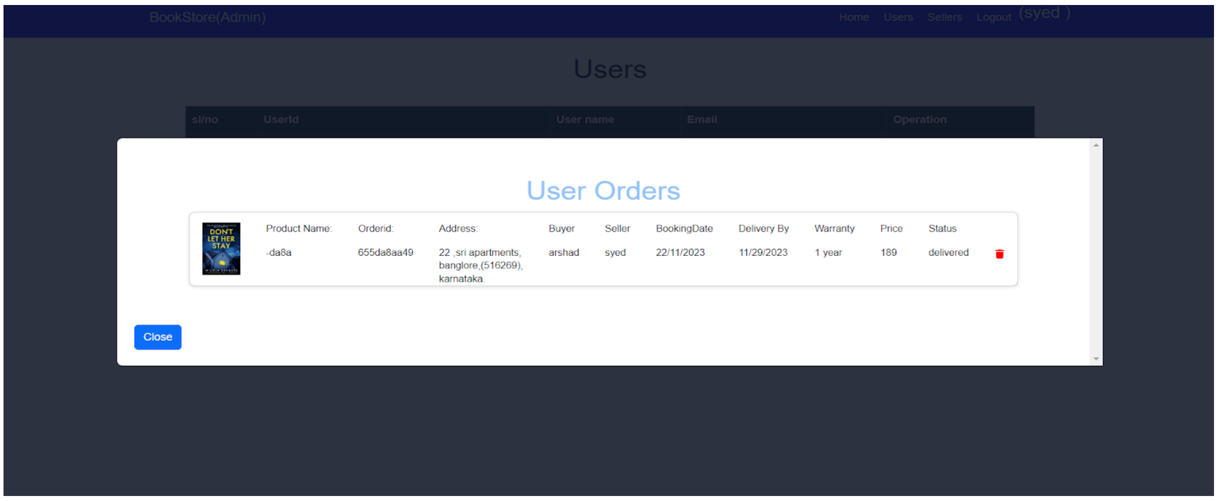
Users Page:



Users

sl/no	UserId	User name	Email	Operation
1	655d9f4c4f4ace5f198b9abf	arshad	arshad@gmail.com	view
2	655e571f62e8144c8a9cb27f	shivani	shivani@gmail.com	view
3	6580442915b2ba8ff503a659	syed	syed@gmail.com	view

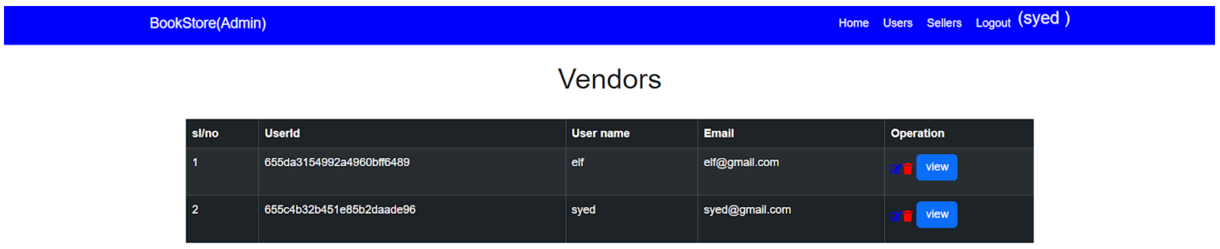
User Orders:



User Orders

Product Name:	Orderid:	Address:	Buyer	Seller	BookingDate	Delivery By	Warranty	Price	Status
-da8a	655da8aa49	22 ,sri apartments, banglore,(516269), karnataka.	arshad	syed	22/11/2023	11/29/2023	1 year	189	delivered

Sellers Page:



Vendors

sl/no	UserId	User name	Email	Operation
1	655da3154992a4960bff6489	elf	elf@gmail.com	view
2	655c4b32b451e85b2daade96	syed	syed@gmail.com	view

13.DEPLOYMENT:

The deployment process involves preparing the bookstore application for production, making it publicly accessible, and ensuring its stability across devices and users. It also includes the setup of cloud services and CI/CD pipelines to manage updates and scaling.

Deployment Steps:

Prepare the Application for Deployment:

- Run npm run build for the React frontend, ensuring that the build is optimized for production with minification and other performance improvements.

Choose Hosting Platform:

- **Frontend Deployment:** Platforms like Netlify or Vercel can be used for easy deployment of the static React application.
- **Backend Deployment:**Use cloud platforms like Heroku, AWS Elastic Beanstalk, or DigitalOcean to deploy the Express backend server.

Database Hosting:

- Use MongoDB Atlas or AWS for hosting MongoDB in the cloud. These services ensure scalability, automatic backups, and high availability.

CI/CD Configuration:

- Automate the deployment process using CI/CD tools like GitHub Actions, CircleCI, or GitLab CI. This allows for continuous integration and delivery whenever changes are pushed to the repository.

Deploy Frontend:

- Upload the build files to Netlify, Vercel, or AWS S3. These platforms handle static content efficiently and provide built-in deployment pipelines.

Deploy Backend:

- The backend can be deployed using Heroku or AWS Elastic Beanstalk, which offer scalable deployment solutions with easy management.

Environment Variables:

- Configure environment variables for production, such as database connection strings, secret keys for JWT tokens, and API keys, ensuring they are kept secure.

SSL Configuration:

- Set up SSL certificates to ensure secure HTTPS access for all users, enhancing security and trustworthiness.

Monitoring and Scaling:

- Use tools like AWS CloudWatch, Datadog, or New Relic to monitor the application's performance and ensure automatic scaling based on traffic.

Configure Domain:

- Set up custom domains with DNS management, ensuring the application can be accessed through a personalized URL.

Final Testing:

- Test the live application in different environments to ensure all features work as expected, including book search, cart functionality, user login, and checkout.

14.RESULTS

This **Book Store** platform demonstrates the versatility of the MERN stack in creating scalable e-commerce applications. Key achievements include:

1. **User-Friendly Interface**: Intuitive design for easy navigation and book searching across devices.

2. **Efficient Book Management**: MongoDB’s schema-less structure allows rapid updates and organization of diverse book genres and collections.

3. **Scalability**: Designed to support a growing number of users and inventory without compromising performance.

4. **Data Security**: Robust authentication and data encryption safeguard user information.

15.CONCLUSION:

Project Summary:

The Bookstore System provides a platform where users can browse, search, and purchase books seamlessly. It enables easy management of inventory and orders for store administrators, while customers can effortlessly browse books and complete purchases. The application successfully implements core features such as secure login, role-based access control, and a responsive UI.

Key Achievements:

The system's architecture is modular, allowing for scalability and easy maintenance. The separation of concerns between the frontend, backend, and database layers ensures clear boundaries and makes it easier to extend the system in the future.The user interface is designed with simplicity and ease of use in mind. By leveraging modern web technologies like React, the application provides an interactive and responsive experience for users, making it easy for them to browse books, search for titles, and manage their accounts.

The database schema was designed to handle relationships between books, users, and orders. Using normalization techniques ensures data integrity, while foreign key constraints maintain referential integrity.

Robust security features like encrypted passwords, JWT authentication, and data validation mechanisms were implemented to safeguard user data and ensure privacy. The system includes user roles (admin, customer) with appropriate access control to ensure only authorized actions can be performed.

A comprehensive testing approach was adopted, including unit tests, integration tests, end-to-end (E2E) tests, and API tests. Tools like Jest, React Testing Library, Cypress, Postman, and Apache JMeter were employed to ensure quality and reliability.

Reflection and Final Thoughts:

The Bookstore Management System project has been an invaluable learning experience, allowing me to apply theoretical knowledge in a real-world context while addressing various technical, operational, and design challenges. Working on this project, particularly using the MERN stack, has deepened my understanding of full-stack development, as well as the integration of both frontend and backend components.

A key takeaway from this project has been the importance of a user-centered design. Understanding the needs of bookstore customers helped me design an intuitive interface that simplifies browsing, purchasing, and managing orders. The goal was to create a seamless user experience, ensuring that customers could navigate the system easily, find books quickly, and complete purchases with minimal effort.

The project also underscored the importance of security, especially in an e-commerce setting. By implementing JWT authentication, password encryption with bcrypt, and role-based access control (RBAC), I ensured that sensitive user data remains secure. The legal and ethical considerations around handling financial data and personal information were also considered during development.

In conclusion, this project has not only helped me develop a functional and scalable bookstore system but also enhanced my technical abilities and understanding of real-world software development practices. Moving forward, I plan to introduce features like personalized book recommendations, real-time notifications, and more detailed user reviews, which will further enhance the platform's value and user experience.

16.REFERENCES

1. MongoDB - Official Documentation: https://www.mongodb.com/docs/

2. Express.js - Official Documentation: https://expressjs.com/

3. React.js - Official Documentation: https://react.dev/
React Router Documentation (for handling routing in React): https://reactrouter.com/

4. Node.js - Official Documentation: https://nodejs.org/en/docs/
Getting Started with Node.js: https://nodejs.dev/en/learn/

GITHUB REPOSITORY :

https://github.com/Karthikeyan192004/book_store_using_MERN_stack