Full Stack Development with MERN

1. Introduction

- **Project Title:** Bookstore Webpage
- **Team Members:**

  - Dhirubhai - A talented Full Stack Developer, who contributes to both frontend and backend development, ensuring everything works seamlessly.

  - Karan - The Frontend Developer of our team, responsible for designing a smooth and interactive user experience.

  - Mohammed Kaif - Our Backend Developer, who expertly handles server-side logic, APIs, and database management.

  - Aakash - The Frontend Developer of our team, responsible for designing a smooth and interactive user experience.

2. Project Overview

- **Purpose:**

  1. To provide a centralized online platform where users can easily find and purchase books.
  2. To enhance the user experience with a clean, responsive design and simple navigation.
  3. To ensure secure handling of user data and payment transactions.
  4. To support administrators in managing book inventory, orders, and customer inquiries efficiently.
  5. To build a scalable system capable of handling a growing number of users and book listings.

- **Features:**

  1. User Management: Supports user registration, login, profile updates, and authentication.
  2. Book Catalog: Allows users to browse a wide range of books with options to filter by category, author, and price.
  3. Order Management: Enables users to place orders, view order history, and receive order confirmations.
  4. Admin Panel: Empowers admins to manage book listings, monitor sales, and handle user queries.
  5. Security: Implements password encryption, secure payment gateways, and data protection measures.

3. Architecture

- **Frontend:**

1. The user interface includes a home page, book catalog, search functionality, and user accountpages.
2. Features JavaScript for dynamic updates and responsive design.
3. Integrates TailWind for styling and responsiveness.
4. There are list of packages inside the frontend part.
    - Package -lock.json
    - package.json
    - vite.config.js

- **Backend:**

1. Node.js and Express.js handle routing, user authentication, and data management.

2. APIs include endpoints like `/api/books`, `/api/users`, `/api/orders` for CRUD operations.

3. Uses Mongoose ODM for MongoDB interactions

4. There are list of packages inside the backend part.
    - Index.js
    - Route.js
    - Model.js

- **Database:**

1. Collections: Users, Books, Orders, Cart.

2. Performs CRUD operations for adding new books, updating stock, and managing user data.

3. Stores user details for registration, login, and role-based access control.

4. Maintains records of all books, including details like title, author, genre, price, and stock.

5. Temporarily saves items users add to their cart before checkout.

6. Stores order details, including purchased items, total cost, and order status (pending, shipped, delivered).

7. Allows admins to manage inventory (add/update books) and monitor customer orders.

4. Setup Instructions

- **Prerequisites:**

1. Mongodb (Community Edition or Atlas)
2. node.js (LTS version)
3. Express.js (for backend server)
4. HTML, CSS, JavaScript (for frontend development)
5. Visual Studio Code

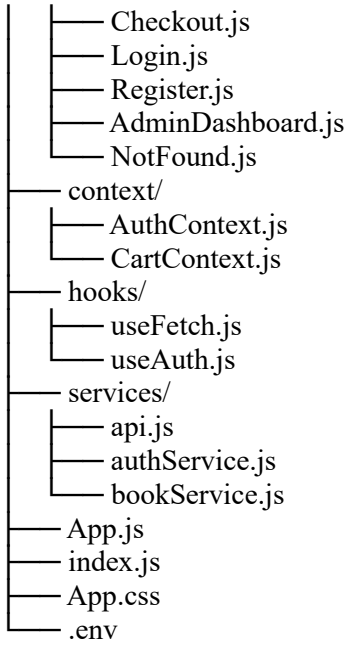
- **Installation:**

1. Install Node.js and npm (Node Package Manager) on your local system.
2. Set up a MongoDB database, either locally or through MongoDB Atlas (cloud-based solution).
3. Basic understanding of HTML, CSS, JavaScript for frontend development.
4. Familiarity with REST APIs and handling JSON data exchanges.
5. Optional: Experience with Git for version control
6. Clone the GitHub repository using `git clone <repository-link>`.
7. Navigate to the project directory: `cd bookstore-webpage`.
8. Install backend dependencies: `npm install`.
9. Start the MongoDB service using `mongod` (for local setup) or connect to MongoDB Atlas.
10. Start the server with `npm start` or `nodemon server.js`.
11. Open `index.html` in your browser or use Live Server extension in VS Code for frontendtesting.
12. Access the application at `http://localhost:3000`.

5. Folder Structure

- **Client:**

```
src/
├── assets/
│   ├── images/
│   └── styles/
├── components/
│   ├── Navbar.js
│   ├── Footer.js
│   ├── BookCard.js
│   ├── SearchBar.js
│   ├── CartItem.js
│   └── ProtectedRoute.js
├── pages/
│   ├── Home.js
│   ├── Books.js
│   ├── BookDetails.js
│   ├── Cart.js
```



1. **assets**:
   - Contains static assets like images and styles (CSS files).

2. **components**:
   - Reusable UI Components:
   - Navbar.js: Handles site navigation with links to different pages.
   - Footer.js: Displays footer information.
   - BookCard.js: Displays book details in a card format.
   - SearchBar.js: Provides search functionality to filter books.
   - CartItem.js: Manages individual items in the shopping cart.
   - ProtectedRoute.js: Restricts access to certain routes based on user authentication.

3. **pages**:
   - Different Pages of the Application:
   - Home.js: Displays the homepage with featured books.
   - Books.js: Lists all available books with filters and search.
   - BookDetails.js: Shows detailed information about a selected book.
   - Cart.js: Displays items added to the shopping cart.
   - Checkout.js: Handles the checkout process.
   - Login.js & Register.js: User authentication forms.
   - AdminDashboard.js: Dashboard for admin functions like managing books and orders.
   - NotFound.js: A 404 page for handling invalid routes.

4. **context**:
   - Global State Management(using React Context API):
   - AuthContext.js: Manages user authentication state and tokens.
   - CartContext.js: Manages the shopping cart state across the app.

5. **hooks**:
  - Custom Hooks for reusable logic:
   - useFetch.js: Handles API requests and data fetching.
   - useAuth.js: Provides authentication utilities like checking if a user is logged in.

6. **services**:
  - API Integration:
   - api.js: Configures Axios for making HTTP requests.
   - authService.js: Handles user login, registration, and authentication.
   - bookService.js: Manages API calls related to book data (fetching, adding, updating).

7. **App.js**:
  - The main entry point of the React application. It sets up React Router for navigation and wraps components with necessary contexts.

8. **index.js**:
  - The starting point of the React app. Renders the entire application into the root element of the HTML.

9. **App.css**:
  - Global styles for the entire application.

10. **.env**:
   - Stores environment variables like API URLs and secret keys.

- **Server:**

```
backend/
├── config/
│   └── db.js
├── controllers/
│   ├── authController.js
│   ├── bookController.js
│   ├── cartController.js
│   └── orderController.js
├── middleware/
│   ├── authMiddleware.js
│   └── errorHandler.js
```

```
├── models/
│   ├── User.js
│   ├── Book.js
│   ├── Cart.js
│   └── Order.js
├── routes/
│   ├── authRoutes.js
│   ├── bookRoutes.js
│   ├── cartRoutes.js
│   └── orderRoutes.js
├── utils/
│   ├── generateToken.js
│   └── sendEmail.js
├── .env
├── server.js
├── package.json
└── README.md
```

Explanation of Key Folders and Files:

1. config:

- Database Configuration:
- db.js: Manages the connection to MongoDB using Mongoose.
- Contains the database connection string and handles any errors during connection.

```
javascript
const mongoose = require('mongoose');
const connectDB = async () => {
   try {
      await mongoose.connect(process.env.MONGO_URI);
      console.log('MongoDB connected');
   } catch (error) {
      console.error(`Error: ${error.message}`);
      process.exit(1);
   }
};
module.exports = connectDB;
```

2. controllers:

- Business Logic for Each Resource:
- authController.js: Handles user registration, login, and authentication.

- bookController.js: Manages operations like fetching books, adding new books, updating, or deleting them.
- cartController.js: Manages user shopping cart (adding/removing items).
- orderController.js: Handles order creation, tracking, and updates.
- Example: bookController.js

```
javascript
const Book = require('../models/Book');
exports.getBooks = async (req, res) => {
    const books = await Book.find({});
    res.json(books);
};
```

3. middleware:

- Custom Middleware Functions:
- authMiddleware.js: Protects routes by verifying JSON Web Tokens (JWT) for authenticated access.
- errorHandler.js: Global error handler to catch and respond with appropriate HTTP error messages.
- Example: authMiddleware.js

```
javascript
const jwt = require('jsonwebtoken');
const authMiddleware = (req, res, next) => {
    const token = req.header('Authorization').split(' ')[1];
    if (!token) return res.status(401).json({ msg: 'No token, authorization denied' });
    try {
        const decoded = jwt.verify(token, process.env.JWT_SECRET);
        req.user = decoded;
        next();
    } catch (err) {
        res.status(401).json({ msg: 'Token is not valid' });
    }
};
module.exports = authMiddleware;
```

4. models:

- MongoDB Schema Definitions:
- User.js: Defines the user schema with fields like name, email, password, and role.
- Book.js: Defines book details (title, author, genre, price, stock).
- Cart.js: Stores items added by users to their cart.

- Order.js: Stores order details including user info, purchased items, and order status.
- Example: User.js

```
javascript
const mongoose = require('mongoose');
const userSchema = new mongoose.Schema({
    name: String,
    email: { type: String, unique: true },
    password: String,
    role: { type: String, default: 'user' }
});
module.exports = mongoose.model('User', userSchema);
```

5. routes:

- Defines API Endpoints:
- authRoutes.js: Routes for user authentication (e.g., /api/auth/login, /api/auth/register).
- bookRoutes.js : Routes for book-related operations (e.g., /api/books, /api/books/:id).
- cartRoutes.js: Routes for managing shopping cart items (e.g., /api/cart).
- orderRoutes.js: Routes for handling orders (e.g., /api/orders).
- Example: bookRoutes.js

```
javascript
const express = require('express');
const router = express.Router();
const { getBooks, addBook } = require('../controllers/bookController');
router.get('/', getBooks);
router.post('/', addBook);
module.exports = router;
```

6. utils:

- Helper Functions:
- generateToken.js: Generates JWT tokens for user sessions.
- sendEmail.js: Utility for sending emails (like order confirmations).

7. server.js:

- Main Entry Point of the Backend:
- Initializes Express, connects to MongoDB, and sets up routes and middleware.

```
javascript
const express = require('express');
const connectDB = require('./config/db');
```

```
const bookRoutes = require('./routes/bookRoutes');
const app = express();

connectDB();
app.use(express.json());
app.use('/api/books', bookRoutes);
const PORT = process.env.PORT || 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

8. .env:

- Environment Variables:
- Stores sensitive information like database URI, JWT secret, and API keys.

```
MONGO_URI=mongodb://localhost:27017/bookstore
JWT_SECRET=your_jwt_secret
PORT=5000
```

9. package.json:

- Dependencies and Scripts:
- Lists all packages (like Express, Mongoose, Bcrypt) and defines start scripts.

```
json
{
  "name": "bookstore-backend",
  "version": "1.0.0",
  "scripts": {
    "start": "node server.js",
    "dev": "nodemon server.js"
  },



    "dependencies": {
    "express": "^4.18.1",
    "mongoose": "^6.9.2",
    "jsonwebtoken": "^8.5.1",
    "bcryptjs": "^2.4.3"
  }
}
```

6. Running the Application

- **Frontend:**

1. Open a terminal and navigate to the backend directory:

```
bash
cd backend
```

2. Install backend dependencies:

```
bash
npm install
```

3. Set up environment variables:

- Create a .env file in the backend directory with the following content:

```
MONGO_URI=mongodb://localhost:27017/bookstore
JWT_SECRET=your_jwt_secret
PORT=5000
```

4. Start the backend server:

```
bash
npm start
```

- Alternatively, if you have nodemon installed (for auto-restart on file changes):

```
bash
npm run dev
```

5. Backend server running at:

The backend server should now be running on http://localhost:5000.

- **Backend:**

1. *Open a new terminal* (or tab) and navigate to the frontend directory:

bash

cd frontend

2. *Install frontend dependencies*:

bash

npm install

3. *Start the frontend server*:

bash

npm start

4. *Frontend server running at*:

- The frontend server should now be running on http://localhost:3000**.**

7. API Documentation

1. Get All Books
- Endpoint: /api/books
- Method: GET
- Parameters: None
- Description: Fetches a list of all available books.
- Example Response:

```
json
[
  {
    "id": 1,
    "title": "The Great Gatsby",
    "author": "F. Scott Fitzgerald",
    "price": 10.99,
```

```
    "stock": 5
  },
  {
    "id": 2,
    "title": "1984",
    "author": "George Orwell",
    "price": 8.99,
    "stock": 3
  }
]
```


2. Get Book by ID

- Endpoint: /api/books/:id
- Method: GET
- Parameters:
- id (Path) - The ID of the book to fetch.
- Description: Fetches a single book based on its ID.
- Example Response:

json


```
{
  "id": 1,
  "title": "The Great Gatsby",
  "author": "F. Scott Fitzgerald",
  "price": 10.99,
  "stock": 5
}
```


3. Create a New Book

- Endpoint: /api/books
- Method: POST
- Parameters:
- title (String) - The title of the book.
- author (String) - The author of the book.
- price (Number) - The price of the book.
- stock (Number) - The number of copies available.
- Description: Creates a new book record.
- Example Request:

json


```
{
  "title": "Brave New World",
  "author": "Aldous Huxley",
  "price": 11.99,
  "stock": 4
}
```

- **Example Response:**
  json
  {
    "message": "Book added successfully",
    "book": {
      "id": 3,
      "title": "Brave New World",
      "author": "Aldous Huxley",
      "price": 11.99,
      "stock": 4
    }
  }

4. Update Book by ID

- Endpoint: /api/books/:id
- Method: PUT
- Parameters:
  - id (Path) - The ID of the book to update.
  - title, author, price, stock (Body) - Fields to update.
  - Description: Updates the details of a book.
  - Example Request:
    json
    {
      "price": 12.99,
      "stock": 2
    }

- Example Response:
  json
  {
    "message": "Book updated successfully",
    "book": {
      "id": 1,
      "title": "The Great Gatsby",
      "author": "F. Scott Fitzgerald",
      "price": 12.99,
      "stock": 2
    }
  }

5. Delete Book by ID

- Endpoint: /api/books/:id
- Method: DELETE
- Parameters:
- id (Path) - The ID of the book to delete.
- Description: Deletes a book based on its ID.
- Example Response:
  json
  {
    "message": "Book deleted successfully"
  }

8. Authentication

1. *User Registration*:

- When a new user registers, their details (name, email, and password) are collected.

- The password is securely hashed before being stored in the database.

- After successful registration, a JWT token is generated and sent back to the user for automatic login.

2. *User Login*:

- Existing users log in by providing their email and password.

- The application verifies the credentials against stored data.

- If the credentials are correct, a JWT token is generated and

3. *JWT Token*:

- The JWT token serves as a digital signature, confirming the user's identity.

- It is stored on the client side (usually in local storage or cookies) and included in the headers of subsequent API requests.

- The token typically includes user information (like user ID and role) and an expiration time.

4. *Protected Routes*:

- Certain parts of the application (e.g., managing the cart, placing orders, accessing the admin dashboard) are restricted to authenticated users.

- The backend verifies the JWT token on every request to these routes.

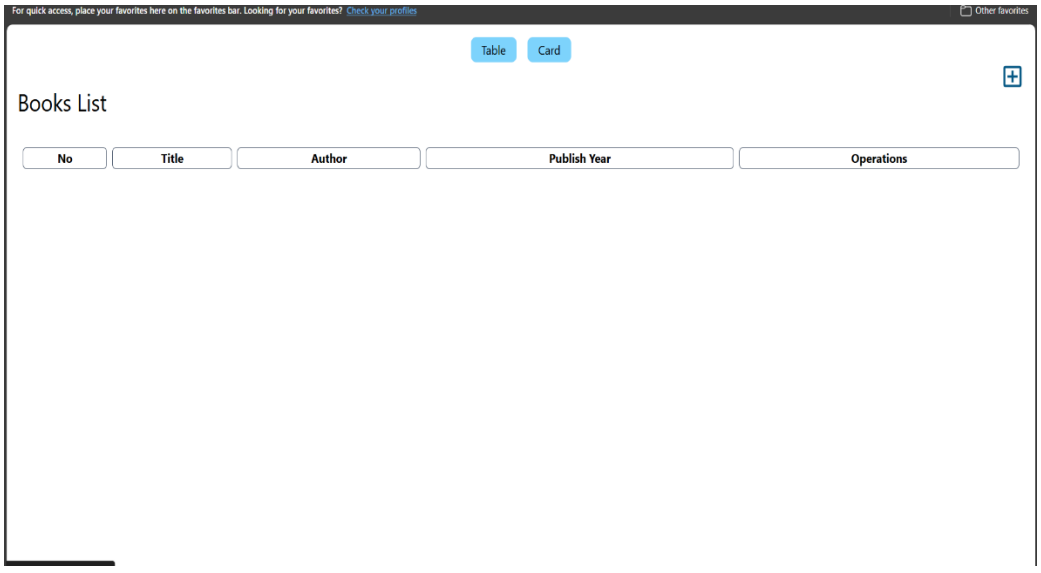
- If the token is valid, the user is granted access; otherwise, they receive an authorization error.

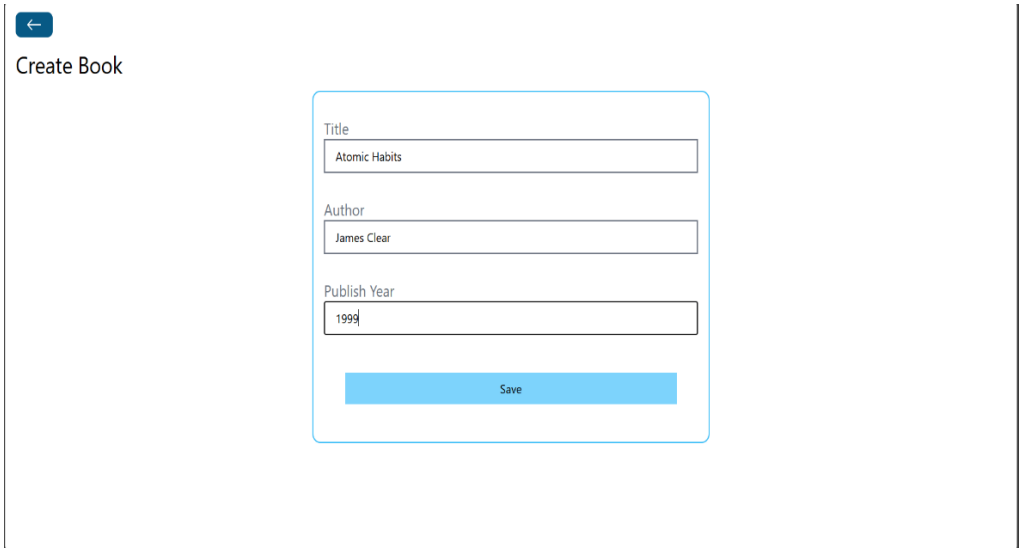
5. *Role-Based Access Control*:

- The system also uses roles (like "user" and "admin") to control access to specific features.

- For example, only users with the "admin" role can add, update, or delete books from the inventory.

9. User Interface




Create Book
Title
Atomic Habits
Author
James Clear
Publish Year
1999
Save

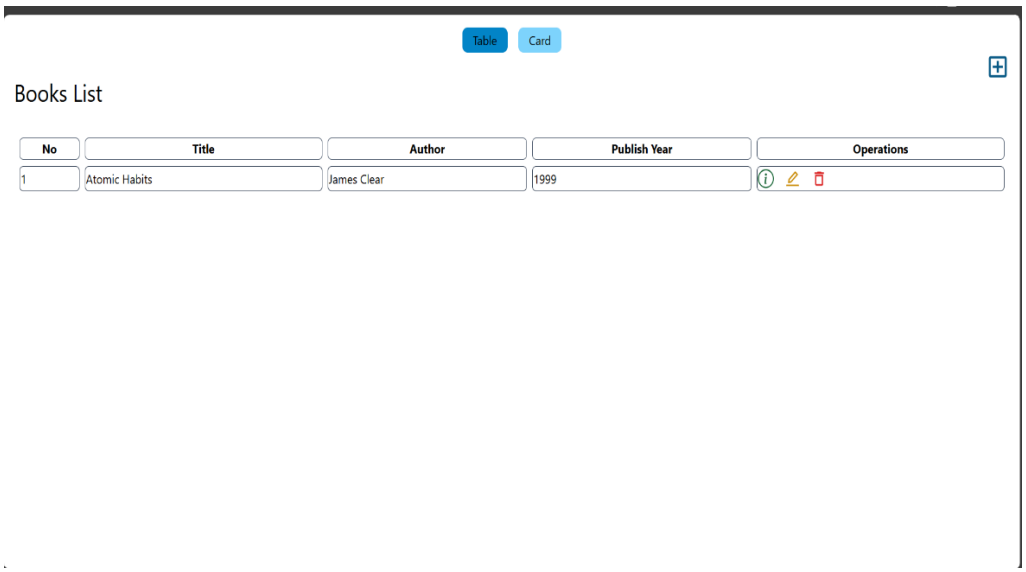
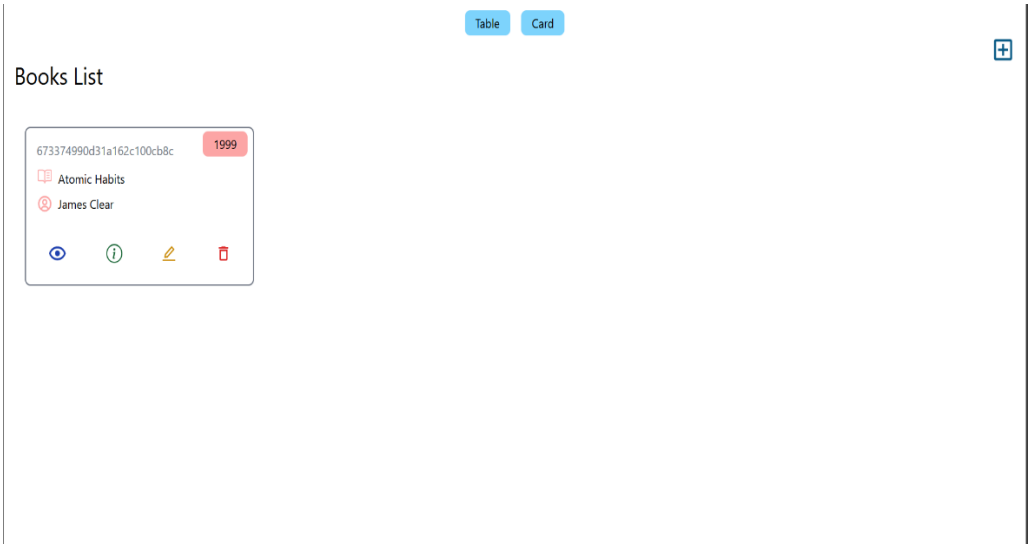


Table
Card
Books List
No
Title
Author
Publish Year
Operations
1
Atomic Habits
James Clear
1999


Create Book
Title
Author
Publish Year
Save



10. Testing

1. *Unit Testing*:

   - Unit tests focus on testing individual components of the application in isolation. This includes testing backend utility functions, middleware, and frontend React components to verify that they behave as expected.

   - For example, helper functions like calculating the total price of items in a cart, or ensuring a React component renders correctly with given props.

   - Tools used: *Jest* for both frontend and backend JavaScript code.

2. *Integration Testing*:

   - Integration tests verify that different parts of the application work together as expected. This includes testing API endpoints to ensure they correctly interact with the database and return accurate responses.

   - This level of testing checks the communication between the server, database, and client to ensure that data flows correctly between them.

   - Tools used: *Mocha* (testing framework), *Chai* (assertion library), and *Supertest* (for testing HTTP endpoints).

3. *End-to-End (E2E) Testing*:

   - E2E tests simulate real user interactions with the application, covering the entire flow from start to finish. This includes testing scenarios like user registration, login, browsing books, adding items to the cart, and completing a purchase.

- The goal is to verify that the application behaves correctly in a real-world environment and that all integrated parts function together seamlessly.

- Tools used: *Cypress* (optional for comprehensive E2E testing).

- Tools Used for Testing

- Jest: A popular testing framework for JavaScript, ideal for unit testing both backend logic and React components.

- Mocha: A flexible testing framework primarily used for backend testing.

- Chai: An assertion library that pairs well with Mocha, providing readable and expressive assertions.

- Supertest: A tool for testing HTTP endpoints, making it suitable for integration testing of REST APIs.

- Cypress: A robust tool for conducting E2E tests to simulate real user interactions across the entire application.

- Setting Up Testing in the Bookstore Application

1. *Backend Testing with Mocha, Chai, and Supertest*:

   - Install necessary dependencies: *Mocha, **Chai, **Supertest, and **Jest*.

   - Create test files for various backend functionalities such as user authentication, book management, and order processing.

   - Integration tests focus on API routes to ensure they interact correctly with the database, verifying responses like successful user registration, login, or book retrieval.

2. *Backend Unit Testing with Jest*:

   - Use Jest to test isolated backend functions like utility methods, ensuring they produce the correct outputs based on given inputs.

   - Example: Testing a utility function that calculates the total price of items in a shopping cart.

3. *Frontend Testing with Jest and React Testing Library*:

   - Set up Jest along with the React Testing Library to test React components individually.

- These tests ensure components render correctly and handle user interactions as expected, such as verifying that a book card component displays the correct title, author, and price.

4. *End-to-End (E2E) Testing with Cypress* (Optional):

- Install Cypress for comprehensive testing of user flows throughout the application.

- Create tests that simulate user scenarios, such as navigating the website, registering, logging in, browsing books, adding items to the cart, and completing a purchase.

- E2E tests help identify issues that may not be caught by unit or integration tests, ensuring a smooth user experience.

11. Screenshots or Demo:

Link:

https://drive.google.com/file/d/14c5J-UoVGEkbNLDANQQpbZNKPrA3eTxW/view?usp=drive_link

12. Known Issues:

1. Search Functionality Limitations:

The search feature is basic and only matches exact strings. It doesn't support partial matches, fuzzy search, or case-insensitive searches, which can result in limited search results.

2. No Payment Gateway Integration:

Currently, the checkout process doesn't include a real payment gateway. Orders are placed without actual payment validation, which limits the application's use for real-world scenarios.

3. Lack of Pagination:

The book catalog does not support pagination, which can lead to slow load times and performance issues if the database contains a large number of books.

4. Limited Error Handling:

Some API endpoints have minimal error handling, which may cause generic error responses. For example, database connection issues or invalid data inputs may not provide detailed feedback.

5. Mobile Responsiveness:

Although the design is responsive, certain elements (like the admin dashboard) may not render correctly on smaller screens, leading to a suboptimal user experience on mobile devices.

6. JWT Expiry Handling:

When a user's JWT token expires, the application does not automatically redirect them to the login page. This can result in failed API requests without clear feedback to the user.

7. Concurrency Issues in Cart Management:

If multiple users update the same book in their carts simultaneously, it may cause race conditions, resulting in incorrect cart quantities or stock inconsistencies.

8. No Support for Book Reviews and Ratings:

The current version lacks a feature for users to leave reviews or ratings for books, limiting user engagement and feedback.

9. Security Vulnerabilities:

Basic security measures like rate limiting, input sanitization, and prevention of SQL/NoSQL injection are implemented but not fully tested, which may leave the application vulnerable to attacks.

10. Session Persistence:

Users are logged out if the page is refreshed because the session data stored in local storage may not always persist reliably, impacting the overall user experience.

13. Future Enhancements

- Implement fuzzy search with case-insensitivity.
- Integrate a payment gateway like Stripe or PayPal.
- Add pagination and infinite scroll for book listings.
- Enhance error handling with descriptive messages.
- Improve mobile responsiveness for admin features.
- Implement automated token refresh for expired JWTs.
- Introduce a book review and rating system.
- Conduct a thorough security audit to address vulnerabilities.

14. Conclusion:

The Online Bookstore Webpage is a robust solution for digital book sales, providing users with a seamless shopping experience. Future enhancements could include adding a recommendation system, implementing advanced security measures, and integrating analytics for sales data insights. This project demonstrates the effective use of the MERN stack for building scalable and user-friendly web applications.