**Book-Store using MERN Stack**

# Introduction

# The UKEC Book Store App is a comprehensive platform designed to simplify the way users explore, purchase, and manage books in the digital age. By combining an intuitive interface with robust functionality, UKEC caters to a diverse audience, including avid readers, book collectors, and sellers. The app streamlines the book-buying experience while offering tailored solutions for sellers to list and manage their inventory effectively.

# TEAM MEMBERS

DIVYA S 513421104008

AFRA LIKATH G 513421104002

KOSURU VAISHNAVI 513421104016

SRIJA R 513421104042

# Purpose and Goals

**Purpose**

The primary purpose of the UKEC Book Store App is to provide a seamless and efficient platform for buying, selling, and managing books online. By leveraging the MERN stack, the app ensures a responsive, scalable, and user-friendly experience for readers, sellers, and administrators alike. UKEC bridges the gap between traditional book shopping and modern e-commerce, offering a centralized solution for book enthusiasts and businesses.

**Goals**

**Enhance User Convenience:**

Simplify book browsing, selection, and purchasing with intuitive navigation and secure transactions.

**Empower Sellers:**

Provide robust tools for sellers to list, manage, and monitor their inventory and sales effectively.

**Ensure Operational Efficiency:**

Equip administrators with comprehensive control over system management, user data, and book listings.

**Promote Accessibility:**

Make books accessible to users from diverse locations, catering to various genres and formats (e.g., e-books and physical books).

**Foster Engagement:**

Encourage user participation through feedback and ratings to improve platform quality.

**Build a Scalable System:**

Create a robust platform that can accommodate growth in user base, book inventory, and features.

UKEC aims to revolutionize the online book store experience by creating a holistic, user-centric ecosystem for readers, sellers, and administrators.

# Key Features and Functionalities

**1. User Features**

**User Registration and Authentication:**

* + Users can register securely and log in to access the platform.
  + Authentication ensures only authorized users can interact with the app.

**Book Browsing and Search:**

* + Users can search for books based on titles, authors, genres, or popularity.
  + View book details, including descriptions, authors, genres, price, and availability status.

**Book Selection and Purchase:**

* + Add books to a shopping cart and specify quantities.
  + Secure checkout process for completing purchases.

**Order Confirmation and History:**

* + Users receive a confirmation page or notification with order details.
  + View past and ongoing orders, track shipments, and manage purchases.

**Feedback and Reviews:**

* + Users can write reviews, rate books, and provide feedback on their shopping experience.

**2. Seller Features**

**Seller Registration:**

* + Sellers can create accounts and register their business.

**Book Management:**

* + Sellers can list new books, update existing details, and manage inventory levels.

**Order Fulfillment:**

* + View and process customer orders for timely shipping and delivery.

**3. Administrator Features**

**User and Seller Management:**

* + Admins can manage user and seller accounts, including updates, deactivations, and monitoring activity.
  + Block users or sellers if necessary to maintain platform integrity.

**Book and Inventory Management:**

* + Add, update, and remove books from the platform.
  + Monitor stock levels to ensure availability.

**Order Oversight:**

* + Track order statuses, resolve disputes, and ensure smooth order fulfillment.

**System Management:**

* + Oversee platform security, performance, and operational configurations.

**4. General Functionalities**

**Cart Management:**

Add, remove, or modify books in the cart before checkout.

**Shipping Details:**

Input and manage shipping information for accurate delivery.

**Payment Options:**

Select preferred payment methods (e.g., credit cards) for secure transactions.

**Dashboard Insights:**

Provide administrators with reporting and analytics, including sales trends, popular books, and user demographics.

**Integration with APIs:**

Enable third-party integrations for payment processing, shipping logistics, and book recommendations.

These features ensure a streamlined experience for all stakeholders, focusing on usability, scalability, and functionality to support users, sellers, and administrators effectively.

# Frontend Architecture

The front-end architecture of the **UKEC Book Store App**, built using **React.js**, is designed to provide a seamless and interactive user experience. The architecture follows a component-based structure and emphasizes scalability, reusability, and responsiveness.

**1.Component Structure**

The front end is divided into reusable components, ensuring modularity and maintainability:

**Header:** Contains navigation links (Home, Browse Books, Cart, Login/Signup).

**Footer:** Displays additional information such as links to policies, terms, and contact details.

**Home Page:** Highlights featured books, genres, and a search bar for quick access.

**Search Component:** Enables users to search books by title, author, or genre.

**Book Details Page:** Displays detailed information about a selected book, including reviews, price, and add-to-cart options.

**User Profile:** Allows users to manage account settings, view order history, and provide feedback.

**Cart Page:** Displays selected books, their quantities, and the total cost, with an option to proceed to checkout.

**Checkout Page:** Collects shipping details and payment information.

**Admin Dashboard:** Manages books, users, sellers, and orders with access to analytics and reporting.

**2. State Management**

**React Context API or Redux:**

* + Manages global states like user authentication, cart data, and admin privileges.
  + Ensures efficient data flow between components (e.g., cart updates and user profile changes).

**3. Routing**

**React Router**:

Provides dynamic navigation and page transitions.

Example routes:

* + - /home: Home Page
    - /search: Search Results
    - /book/:id: Book Details Page
    - /cart: Cart Page
    - /profile: User Profile
    - /admin: Admin Dashboard

**4. API Integration**

**Axios** or **Fetch API**:

Handles communication with the back-end REST API for data operations.

Key API endpoints:

* + - GET /books: Fetch all books for display.
    - POST /auth: User or seller authentication.
    - POST /orders: Place a new order.
    - PUT /users: Update user profile details.

**5. Styling**

**CSS Frameworks**:

Bootstrap or Tailwind CSS for responsive and consistent UI design.

**Custom Styling**:

Additional styling using CSS Modules or Styled Components for fine-grained control.

**6. Performance Optimization**

**Code Splitting**:

Use **React.lazy** and **Suspense** to load components only when required, reducing initial load time.

**Lazy Loading**:

Implement lazy loading for images and large datasets to improve performance.

**Caching**:

Use service workers or client-side caching mechanisms to store frequently accessed data locally.

**7. Responsive Design**

**Mobile-First Approach**:

Ensure compatibility across devices (mobile, tablet, desktop) using media queries.

**8. Error Handling**

**Global Error Boundaries**:

Ensure graceful error handling with fallback UIs for broken components.

**Notification System**:

Display success/error messages (e.g., for order confirmation or login failures).

This architecture ensures the front end of UKEC remains scalable, efficient, and user-friendly while maintaining a clean codebase and seamless interaction with the back end.

# Backend Architecture and Key Components

The back end of **UKEC Book Store App** is built using the **MERN stack** (MongoDB, Express.js, React.js, and Node.js). It is designed to handle robust and scalable functionality for seamless user experiences.

**Back-End Architecture**

**Server Framework**:

* + The back-end server is implemented using **Express.js**, a lightweight and flexible framework for Node.js.
  + Manages all API routes, middleware, and business logic.

**Database**:

* + **MongoDB** serves as the primary database, offering a flexible schema for managing books, users, orders, and inventory.
  + Implements relationships using collections like Users, Books, Orders, and Inventory.

**Authentication**:

* + Uses **JWT (JSON Web Tokens)** for secure user authentication and session management.
  + Includes middleware to protect routes and enforce role-based access control (e.g., User, Seller, Admin).

**Data Models**:

* + Defined using **Mongoose**, allowing for schema validation and database interactions.
  + Core models:
  + User: Manages user details and credentials.
  + Book: Stores book metadata like title, genre, and author.
  + Order: Tracks purchase details and status.
  + Inventory: Handles stock and availability.

**API Routes**:

* + Organized by functionality for scalability:
  + /auth: Handles user registration, login, and logout.
  + /books: Manages book listing, search, and updates.
  + /orders: Tracks orders and their statuses.
  + /admin: Provides admin controls for managing users, sellers, and books.

**Middleware**:

* + **Authentication Middleware**: Verifies JWT tokens for protected routes.
  + **Error Handling Middleware**: Captures errors and provides standardized responses.

**Error Handling and Logging**:

* + Implements error-handling middleware for consistent responses.
  + Uses logging libraries like **Winston** for debugging and monitoring.

**Performance Optimization**:

* + Uses **Redis** (optional) for caching frequently accessed data, like popular books or genres.
  + Implements pagination and query optimization for large datasets.

**Key Components**

**User Management**:

* + Secure user registration and authentication.
  + Role-based permissions for users, sellers, and admins.

**Book Listings**:

* + CRUD operations for books, allowing sellers to manage inventory.
  + Advanced search and filters for users to explore books.

**Order Processing**:

* + Manages order creation, tracking, and history.
  + Ensures inventory updates after purchases.

**Admin Controls**:

* + Manages users, sellers, and book listings.
  + Generates analytics reports for sales and performance.

**API Documentation**:

* + Comprehensive documentation for all APIs using tools like **Swagger**.

**Security Features**:

* + Implements encryption for sensitive data and prevents common vulnerabilities like SQL injection and CSRF.

The back-end architecture of **Ukec Book Store App** ensures a secure, scalable, and high-performance platform to meet business needs and provide an optimal user experience.

# Data Flow Overview

The data flow for the **Ukec Book Store App** illustrates how data moves between different components of the back-end architecture, ensuring seamless interaction between users, the server, and the database.

**1. User Interaction**

* **Input**: Users interact with the front end (React) by performing actions such as searching for books, logging in, or placing an order.
* **Output**: The front end sends HTTP requests (e.g., GET, POST) to the back end through RESTful API endpoints.

**2. API Gateway (Express.js)**

* Acts as the central point for handling requests.
* **Request Processing**:
  + Routes the incoming requests to the appropriate endpoint (e.g., /auth/login, /books, /orders).
  + Middleware ensures data validation, authentication, and authorization.
* **Response**: Sends a JSON response back to the client with the requested data or an error message.

**3. Middleware**

* **Authentication**: Verifies JWT tokens to ensure users are authorized to access specific endpoints.
* **Validation**: Checks request payloads (e.g., ensuring valid email formats or required fields are present).
* **Error Handling**: Logs errors and provides standardized error messages.

**4. Business Logic**

* **Processing**: The back end processes the request using route handlers in Express.js.
* Example:
  + A request to /books retrieves book data from the database.
  + A request to /orders validates inventory before creating an order.

**5. Database Operations (MongoDB)**

* **Interaction**: Data models (Mongoose) handle CRUD operations in MongoDB.
* **Data Access**:
  + **Users Collection**: Handles user profiles, credentials, and roles.
  + **Books Collection**: Stores book details, genres, authors, and ratings.
  + **Orders Collection**: Tracks purchase details like buyer information, items purchased, and status.
  + **Inventory Collection**: Manages stock levels for books.
* **Output**: Returns queried data or updates the database accordingly.

**6. Data Processing and Response**

* **Processing**: Combines data from multiple sources if necessary (e.g., user orders with book details).
* **Formatting**: Converts raw data into a user-friendly JSON format.
* **Output**: Sends the processed data as a response to the front end.

**7. Front-End Update**

* The back end sends the response to the front-end app.
* Users see updates based on their actions (e.g., a new book added to the cart, a confirmation of a successful order, or a list of search results).

# Database Schemas

**1. User Schema**

Stores information about users, including customers, sellers, and admin roles.

const mongoose = require('mongoose');

const userSchema = new mongoose.Schema({

name: { type: String, required: true },

email: { type: String, required: true, unique: true },

password: { type: String, required: true },

role: { type: String, enum: ['customer', 'seller', 'admin'], default: 'customer' },

createdAt: { type: Date, default: Date.now },

});

module.exports = mongoose.model('User', userSchema);

**2. Book Schema**

Manages book details such as title, author, genre, and availability.

const bookSchema = new mongoose.Schema({

title: { type: String, required: true },

author: [{ type: String, required: true }],

genre: [{ type: String }],

description: { type: String },

price: { type: Number, required: true },

quantity: { type: Number, required: true },

ratings: { type: Number, default: 0 },

createdAt: { type: Date, default: Date.now },

});

module.exports = mongoose.model('Book', bookSchema);

**3. Order Schema**

Tracks customer orders, including details about the purchased books and their status.

const orderSchema = new mongoose.Schema({

userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },

items: [

{

bookId: { type: mongoose.Schema.Types.ObjectId, ref: 'Book', required: true },

quantity: { type: Number, required: true },

},

],

totalAmount: { type: Number, required: true },

status: { type: String, enum: ['pending', 'shipped', 'delivered', 'cancelled'], default: 'pending' },

createdAt: { type: Date, default: Date.now },

});

module.exports = mongoose.model('Order', orderSchema);

**4. Inventory Schema**

Manages stock levels for books in the store.

const inventorySchema = new mongoose.Schema({

bookId: { type: mongoose.Schema.Types.ObjectId, ref: 'Book', required: true },

location: { type: String },

quantity: { type: Number, required: true },

condition: { type: String, default: 'new' },

});

module.exports = mongoose.model('Inventory', inventorySchema);

**5. Review Schema**

Captures user feedback and ratings for books.

const reviewSchema = new mongoose.Schema({

userId: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },

bookId: { type: mongoose.Schema.Types.ObjectId, ref: 'Book', required: true },

rating: { type: Number, min: 1, max: 5, required: true },

comment: { type: String },

createdAt: { type: Date, default: Date.now },

});

module.exports = mongoose.model('Review', reviewSchema);

**6. Genre Schema**

Stores genre details and associations with books.

const genreSchema = new mongoose.Schema({

name: { type: String, required: true, unique: true },

books: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Book' }],

});

module.exports = mongoose.model('Genre', genreSchema);

1. **Schema Interactions with MongoDB**

The **Ukec** back end uses **Mongoose** to interact with the MongoDB database. Mongoose provides a schema-based approach for structuring and manipulating data, enabling CRUD operations and efficient query handling. Below are examples of how the defined schemas interact with MongoDB.

**1. Basic CRUD Operations**

**a. Creating a New Document**

* **Example**: Adding a new book to the database.

**b. Reading Documents**

* **Example**: Fetching all books in a specific genre.

**c. Updating a Document**

* **Example**: Updating the quantity of a book in inventory.

**d. Deleting a Document**

* **Example**: Removing a book from the database.

**2. Advanced Queries**

**a. Populating References**

* **Example**: Fetching all orders with user and book details.

**b. Filtering and Sorting**

* **Example**: Getting top-rated books.

**3. Aggregation Pipelines**

* **Example**: Aggregating sales data to find total revenue per book.

**4. Using Middleware**

Mongoose provides **middleware** for pre- and post-hooks on schema methods.

* **Example**: Hashing a user's password before saving.

**5. Transactions**

For complex operations, transactions ensure database consistency.

* **Example**: Processing an order while updating inventory.

**6. Error Handling**

Ensure proper error handling for database interactions.

* **Example**: Catching validation errors during user creation.

These interactions showcase the capabilities of Mongoose in managing data efficiently while leveraging MongoDB's flexibility for the **Ukec Book Store App**.

1. **Relationships**
   * User - Complaint Relationship: Each complaint is tied to a user (userId).
   * Complaint - Assigned Complaint Relationship: Each assignment links a complaint to an agent.
   * User - Message Relationship: Messages reference users as senders and receivers, establishing communication about specific complaints.

This architecture keeps data interactions efficient, enables easy tracking of complaint status, and facilitates role-based access. Each schema is purposedriven, supporting a specific part of the complaint management process.

**Relationships:**

* **User ↔ Order**: One-to-Many. A user can place multiple orders.
* **Book ↔ Inventory**: One-to-One. Each book entry corresponds to an inventory record.
* **Book ↔ Review**: One-to-Many. A book can have multiple reviews from different users.
* **Book ↔ Genre**: Many-to-Many. A book can belong to multiple genres, and a genre can include multiple books.

These schemas provide a scalable and efficient structure for the **Ukec Book Store App** database.

# Setup Instructions for Complaint Management and Registration System

Follow these steps to set up and run the **Complaint Management and Registration System** locally or on a server. This guide assumes you are using the **MERN stack** (MongoDB, Express.js, React, Node.js).

**1. Prerequisites**

* **Node.js** (v16 or later) installed on your system.
* **MongoDB** database (local or cloud-based, e.g., MongoDB Atlas).
* A code editor (e.g., Visual Studio Code).
* Git (optional, for cloning the repository).

**2. Backend Setup**

**a. Clone the Repository**

git clone <repository-url>

cd <project-directory>/backend

**b. Install Dependencies**

Install the required Node.js packages using npm or yarn:

bash

npm install

**c. Configure Environment Variables**

Create a .env file in the backend directory and define the following variables:

PORT=5000

MONGO\_URI=mongodb://localhost:27017/complaint-system # Replace with your MongoDB URI

JWT\_SECRET=your\_jwt\_secret # Replace with a strong secret

**d. Start the Backend Server**

Run the following command to start the server:

npm start

The backend will run at http://localhost:5000 by default (or the specified port in .env).

**3. Frontend Setup**

**a. Navigate to the Frontend Directory**

cd ../frontend

**b. Install Dependencies**

Install the required React packages:

bash

npm install

**c. Configure API Base URL**

Open the src/config.js file and set the API base URL:

javascript

export const API\_BASE\_URL = "http://localhost:5000/api"; // Replace with your backend's URL if deployed

**d. Start the React Development Server**

Run the following command:

bash

npm start

The frontend will run at http://localhost:3000.

**4. Database Setup**

**a. Start MongoDB**

* If using a local MongoDB server:

bash

mongod

* If using MongoDB Atlas, ensure the database URI is configured in .env.

**b. Initialize Data (Optional)**

Run a script (if provided) to initialize the database with sample data:

bash

node seed.js

**5. Testing the Application**

**Access the Application**

* Open the frontend in your browser: http://localhost:3000.
* Test backend routes using tools like **Postman** or **cURL**.

**Default Roles**

* **Admin**: Manage user complaints, view statistics.
* **User**: Register complaints, track complaint status.

**6. Deployment (Optional)**

**a. Backend Deployment**

* Deploy the backend to platforms like **Heroku** or **AWS**.
* Set up environment variables on the server (e.g., Heroku Config Vars).

**b. Frontend Deployment**

* Build the React app:

bash

npm run build

* Deploy the build folder to **Netlify**, **Vercel**, or **AWS S3**.

**c. Update API Base URL**

Ensure the frontend's API base URL points to the deployed backend.

**7. Troubleshooting**

**Common Issues**

1. **Backend Port Conflicts**: Change the PORT in .env if 5000 is already in use.
2. **Database Connection Errors**: Verify MONGO\_URI and ensure MongoDB is running.
3. **CORS Issues**: Enable CORS in the backend by configuring middleware.

# Folder Structure

A well-organized folder structure is crucial for maintaining scalability and clarity in a **Book Store Management System** using the MERN stack. Below is a recommended folder structure:

**Root Directory**

book-store-management/

├── backend/

├── frontend/

├── README.md

├── package.json

└── .gitignore

**1. Backend**

backend/

├── controllers/

├── models/

├── routes/

├── middlewares/

├── config/

├── utils/

├── validations/

├── tests/

├── .env

├── server.js

└── package.json

**Folder and File Descriptions**

* **controllers/**: Contains logic for handling requests.
  + userController.js
  + bookController.js
  + orderController.js
  + authController.js
* **models/**: Defines Mongoose schemas and models.
  + User.js
  + Book.js
  + Order.js
  + Inventory.js
* **routes/**: Defines API endpoints.
  + userRoutes.js
  + bookRoutes.js
  + orderRoutes.js
  + authRoutes.js
* **middlewares/**: Holds custom middleware.
  + authMiddleware.js (authentication and authorization)
  + errorMiddleware.js (error handling)
* **config/**: Contains configuration files.
  + db.js (MongoDB connection setup)
* **utils/**: Reusable utility functions.
  + generateToken.js (JWT token generation)
  + logger.js (logging errors and activity)
* **validations/**: Validation logic for inputs.
  + userValidation.js
  + orderValidation.js
* **tests/**: Unit and integration tests for backend.
  + userController.test.js
  + bookController.test.js
* **Root Files**:
  + **.env**: Environment variables (e.g., MONGO\_URI, JWT\_SECRET).
  + **server.js**: Entry point for the backend server.

**2. Frontend**

frontend/

├── public/

├── src/

│ ├── components/

│ ├── pages/

│ ├── hooks/

│ ├── context/

│ ├── services/

│ ├── utils/

│ ├── assets/

│ ├── App.js

│ ├── index.js

│ └── config.js

├── .env

├── package.json

└── README.md

**Folder and File Descriptions**

* **public/**: Static files like HTML templates and favicons.
  + index.html
  + favicon.ico
* **src/**:
  + **components/**: Reusable UI components.
    - Navbar.js
    - Footer.js
    - BookCard.js
    - OrderSummary.js
  + **pages/**: Main application pages.
    - HomePage.js
    - BookDetailsPage.js
    - CartPage.js
    - CheckoutPage.js
    - AdminDashboard.js
  + **hooks/**: Custom React hooks.
    - useAuth.js
    - useFetch.js
  + **context/**: React Context for global state management.
    - AuthContext.js
    - CartContext.js
  + **services/**: API service files for communicating with the backend.
    - authService.js
    - bookService.js
    - orderService.js
  + **utils/**: Utility functions.
    - formatPrice.js
    - validateInput.js
  + **assets/**: Static assets like images and icons.
    - /images/
    - /icons/
  + **Root Files**:
    - **App.js**: Root React component.
    - **index.js**: Entry point for React app.
    - **config.js**: API base URLs and other configs.
* **.env**: Environment variables (e.g., API base URL).

**3. Database**

If your project involves database initialization or seed data:

backend/

└── data/

├── books.json

├── users.json

└── seed.js

* **data/**: Contains seed data for MongoDB.
* **seed.js**: Script to populate initial data.

**4. Additional Tools**

* **Testing**:
  + Use **Jest** and **Supertest** for backend testing.
  + Use **React Testing Library** for frontend testing.
* **Deployment**:
  + Store production-ready builds in a /build/ directory after building the frontend:

cd frontend

npm run build

This structure ensures scalability and maintainability for the **Book Store Management System**.

# Running the Application

1. **Frontend**
   * Build the frontend
   * Deploy the /build folder to **Netlify**, **Vercel**, or similar platforms.
2. **Backend**

 Deploy the backend to **Heroku**, **AWS**, or similar platforms.

 Set environment variables on the deployment platform.

# API Documentation

**Overview**

This API provides functionality for managing books, users, orders, and reviews in the UKEC Book Store. It supports operations like adding, updating, deleting, and retrieving books, as well as user authentication and order management.

Base URL:

https://api.UKEC.com

Endpoints

1. Books

a. GET /api/books

Description: Retrieve a list of all books in the store.

Response:

{

"status": "success",

"data": [

{

"id": "book123",

"title": "The Great Gatsby",

"author": "F. Scott Fitzgerald",

"price": 19.99,

"genre": "Classic",

"imageUrl": "https://linktoimage.com/gatsby.jpg"

},

{

"id": "book124",

"title": "1984",

"author": "George Orwell",

"price": 15.99,

"genre": "Dystopian",

"imageUrl": "https://linktoimage.com/1984.jpg"

}

]

}

b. GET /api/books/

Description: Retrieve details of a specific book.

Parameters:

* id (required): The ID of the book to retrieve.

Response:

{

"status": "success",

"data": {

"id": "book123",

"title": "The Great Gatsby",

"author": "F. Scott Fitzgerald",

"price": 19.99,

"genre": "Classic",

"description": "A novel about the American Dream and its corruption in the 1920s.",

"imageUrl": "https://linktoimage.com/gatsby.jpg"

}

}

c. POST /api/books

Description: Add a new book to the store.

Request Body:

{

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"price": 18.99,

"genre": "Fiction",

"description": "A novel about racial injustice in the Deep South.",

"imageUrl": "https://linktoimage.com/mockingbird.jpg"

}

Response:

{

"status": "success",

"message": "Book added successfully",

"data": {

"id": "book125",

"title": "To Kill a Mockingbird",

"author": "Harper Lee",

"price": 18.99,

"genre": "Fiction",

"description": "A novel about racial injustice in the Deep South.",

"imageUrl": "https://linktoimage.com/mockingbird.jpg"

}

}

d. PUT /api/books/

Description: Update an existing book's details.

Parameters:

* id (required): The ID of the book to update.

Request Body:

{

"title": "To Kill a Mockingbird (Updated)",

"price": 20.99

}

Response:

{

"status": "success",

"message": "Book updated successfully"

}

e. DELETE /api/books/

Description: Delete a book from the store.

Parameters:

* id (required): The ID of the book to delete.

Response:

{

"status": "success",

"message": "Book deleted successfully"

}

2. Users

a. POST /api/users/register

Description: Register a new user.

Request Body:

{

"username": "johndoe",

"email": "johndoe@example.com",

"password": "password123"

}

Response:

{

"status": "success",

"message": "User registered successfully",

"data": {

"id": "user123",

"username": "johndoe",

"email": "johndoe@example.com"

}

}

b. POST /api/users/login

Description: Log in an existing user.

Request Body:

{

"email": "johndoe@example.com",

"password": "password123"

}

Response:

{

"status": "success",

"message": "Login successful",

"token": "jwt\_token\_here"

}

3. Orders

a. GET /api/orders

Description: Retrieve a list of orders placed by the authenticated user.

Response:

{

"status": "success",

"data": [

{

"id": "order123",

"userId": "user123",

"bookId": "book123",

"quantity": 2,

"totalPrice": 39.98,

"status": "shipped"

}

]

}

b. POST /api/orders

Description: Create a new order.

Request Body:

{

"bookId": "book123",

"quantity": 2

}

Response:

{

"status": "success",

"message": "Order placed successfully",

"data": {

"id": "order124",

"bookId": "book123",

"quantity": 2,

"totalPrice": 39.98,

"status": "pending"

}

}

4. Reviews

a. POST /api/reviews

Description: Add a review for a book.

Request Body:

{

"bookId": "book123",

"userId": "user123",

"rating": 5,

"reviewText": "Amazing book, a true classic!"

}

Response:

{

"status": "success",

"message": "Review added successfully"

}

Authentication

a. JWT Token

Description: Use JSON Web Tokens (JWT) for authentication. The token is required in the header of authenticated requests.

Header Example:

makefile

Authorization: Bearer jwt\_token\_here

Error Handling

If an error occurs, the API will return an error response.

Example:

{

"status": "error",

"message": "Invalid credentials"

}

This API documentation provides the basic structure and endpoints for the UKEC Book Store. You can customize the paths, models, and responses as per the application's needs.

# Authentication

Authentication is the process of verifying the identity of a user, device, or system. It ensures that the entity attempting to access a resource is genuinely who it claims to be.

**Methods**:

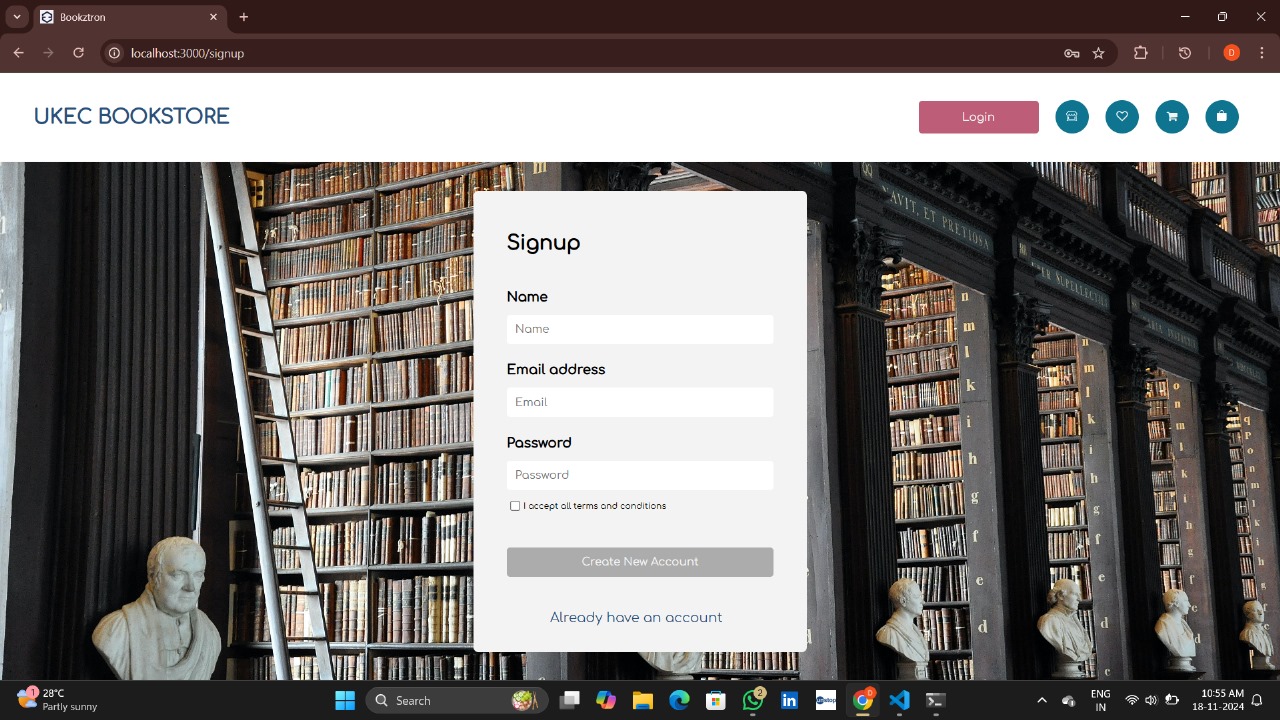
* **Username and Password**: The most common method, where a user provides a username and password to verify their identity.
* **Token-based Authentication (JWT)**: After the user logs in, a JSON Web Token (JWT) is issued. This token is sent with each subsequent request to authenticate the user.
* **OAuth**: An authorization framework that allows third-party applications to access user data without exposing their credentials.

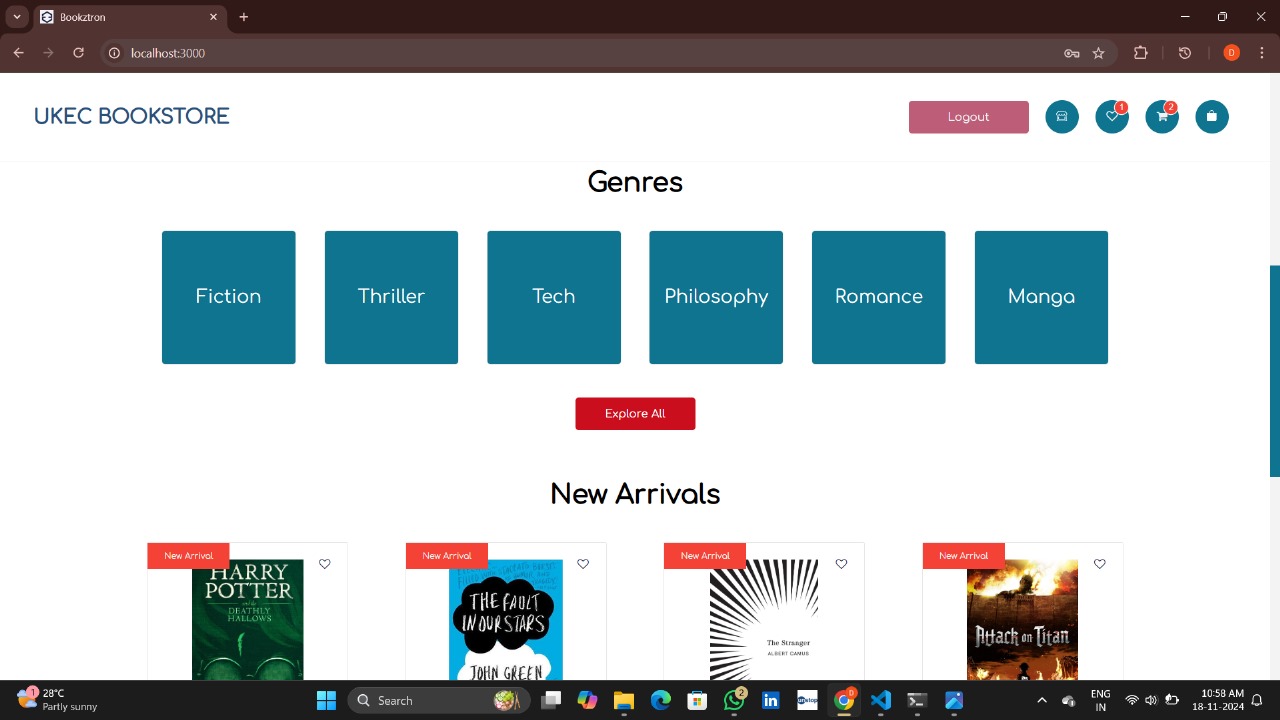
# Authorization

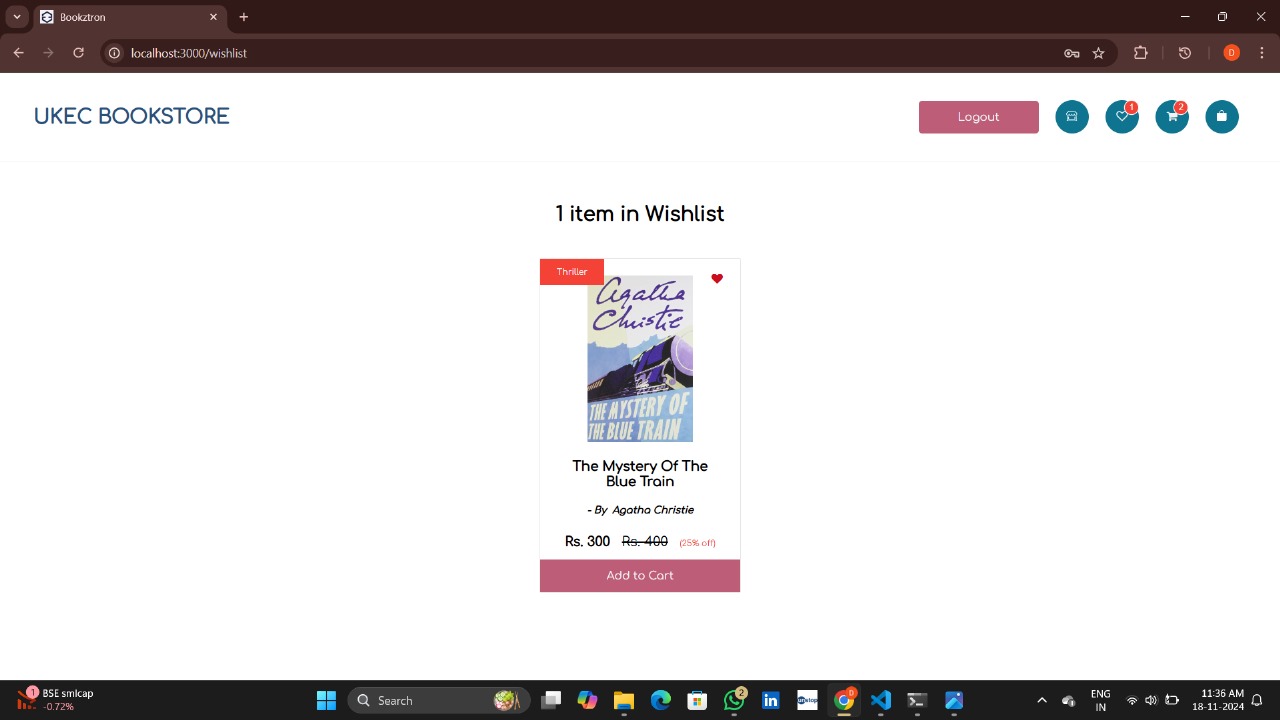
Authorization is the process of granting or denying access to specific resources or actions based on the authenticated user’s permissions or roles. After a user is authenticated, the system decides whether they are permitted to perform an action or access certain data. Methods:

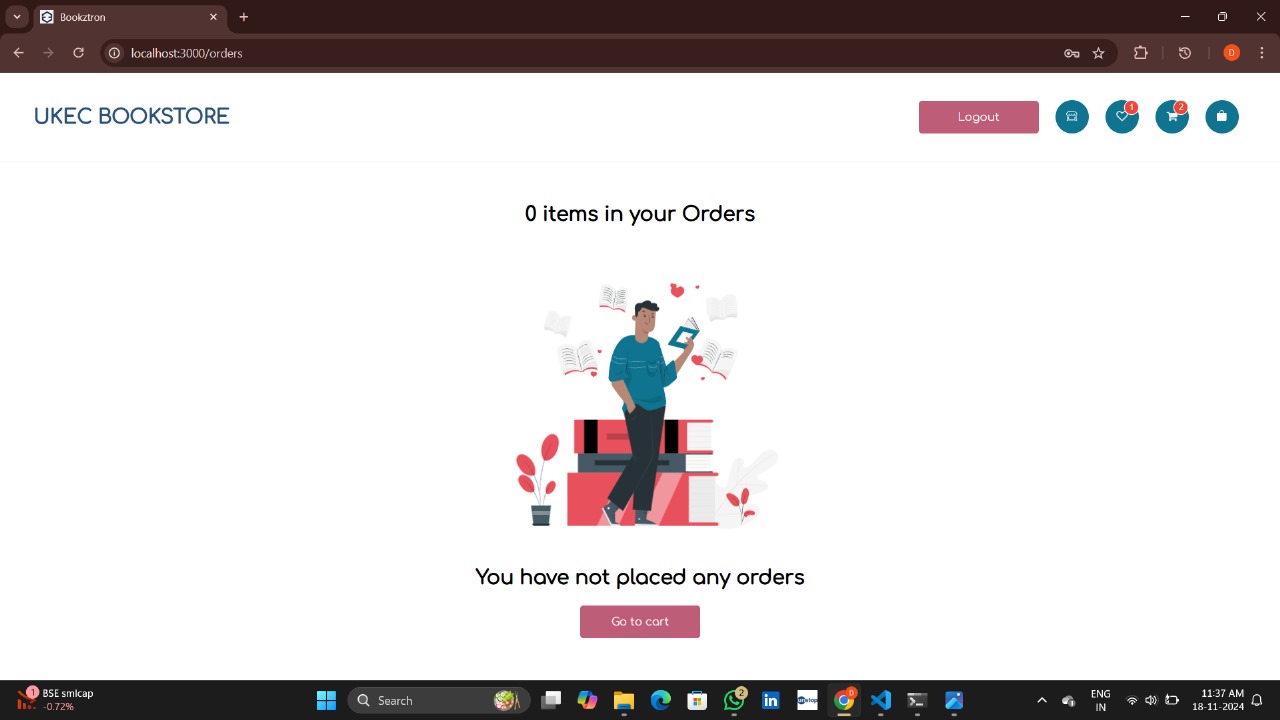
* **Role-based Access Control (RBAC)**: Users are assigned roles (e.g., admin, user) that define their level of access to resources.
* **Permission-based Access Control**: Permissions are explicitly assigned to users or groups, defining what actions they can perform (e.g., read, write, delete).

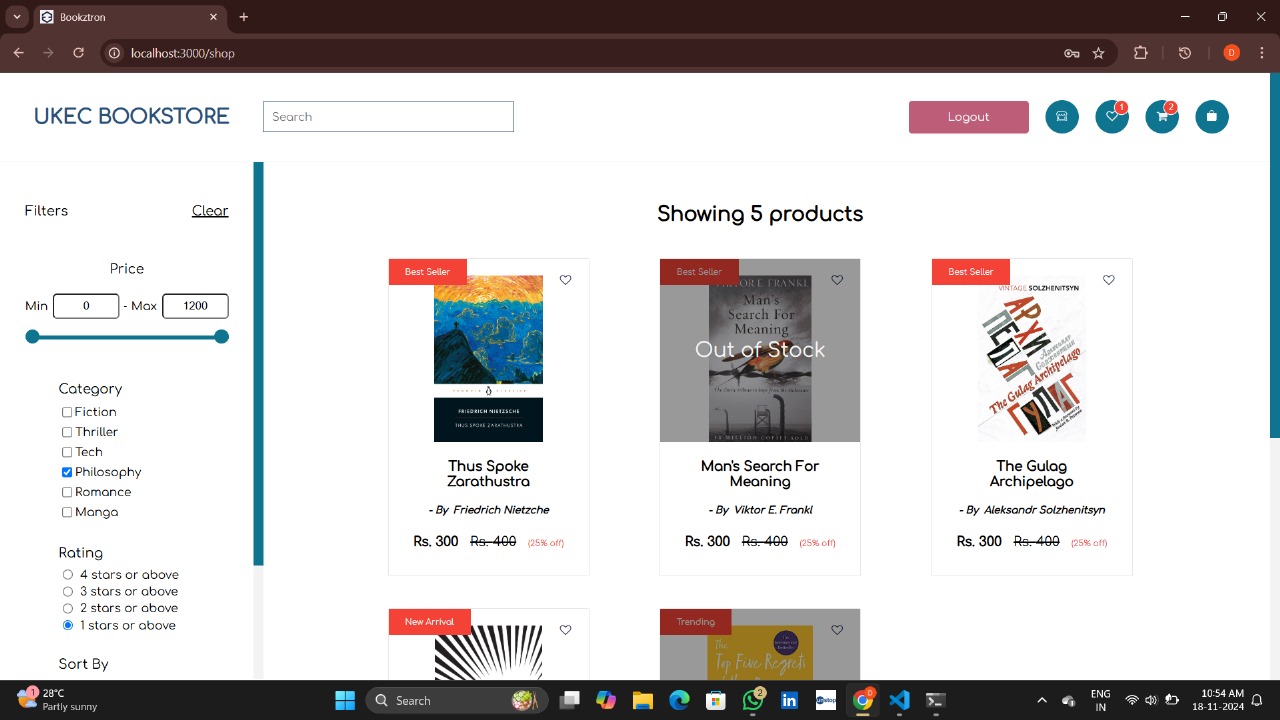
# User Interface

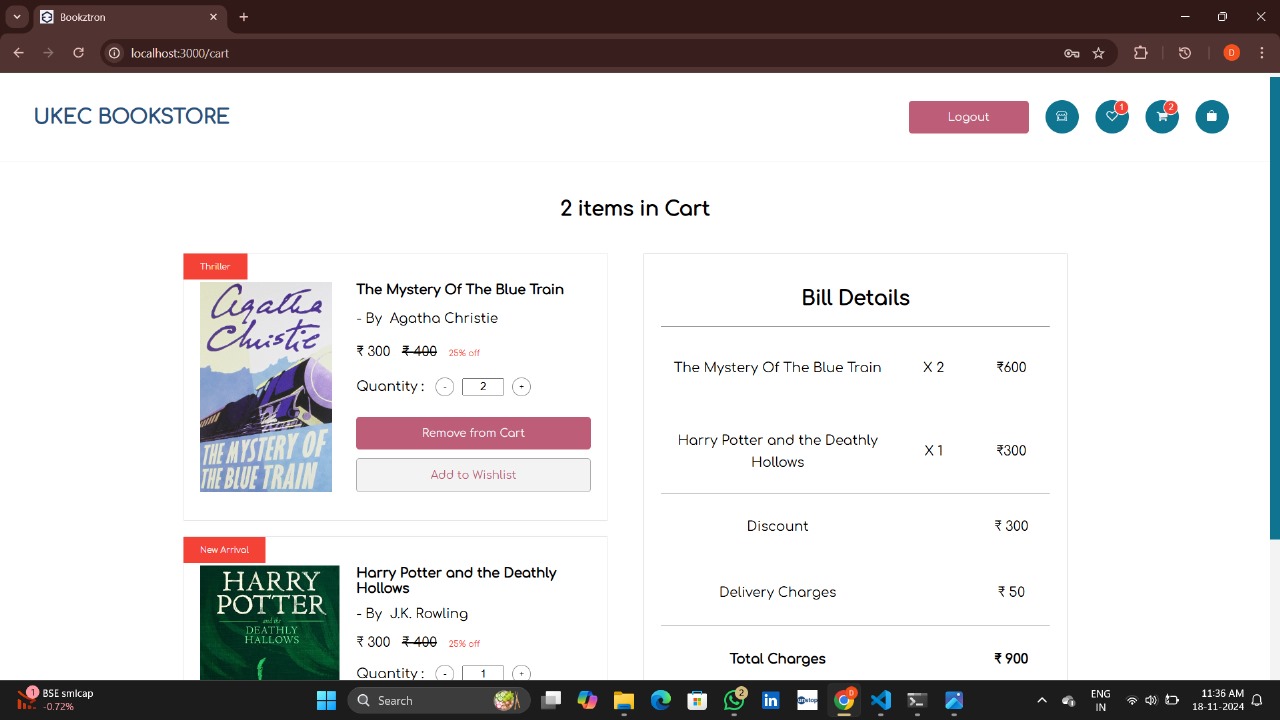












# Testing Strategy and Tools Used

Testing is an essential part of ensuring the functionality, performance, and reliability of the application. In this project, the following testing approaches and tools are used:

**Testing Strategies**:

 **Unit Testing**: Tested individual API routes and functions (e.g., user registration, login) using **Jest** and **Supertest**.

 **Integration Testing**: Validated interactions between APIs, database (MongoDB), and middleware using **Mocha** and **Chai**.

 **End-to-End (E2E) Testing**: Simulated user workflows (e.g., registration to purchase) using **Postman** and **Cypress**.

 **Performance Testing**: Monitored API response times and stress-tested with tools like **JMeter** and **Artillery**.

 **Security Testing**: Identified vulnerabilities (e.g., authentication, SQL injection) using **OWASP ZAP** and **Postman**.

**Tools Used:**

* Jest & Supertest: For unit and API testing.
* Postman: For manual endpoint testing.
* Mocha & Chai: For integration testing.
* Cypress: For automating user flows.
* JMeter & Artillery: For load and performance tests.
* OWASP ZAP: For security validation

# Demo

[Video Link](https://drive.google.com/file/d/1umlmbdQIWj_AliezHxdqvBDcGlSBdhNB/view?usp=sharing)  <https://drive.google.com/file/d/1J5BQG8ilGZkAG6QdzyRAtnNiF_saeKJ8/view?usp=drivesdk>

**Known Issues**

1. **Slow Response for Large Datasets**: APIs slow down when fetching large data due to missing pagination.
2. **Token Expiration Handling**: Expired JWT tokens aren't managed smoothly, causing user disruptions.
3. **Order Conflicts**: Concurrent orders occasionally lead to inventory mismatches due to missing transaction handling.
4. **Limited Error Logging**: Critical API errors lack detailed logs for troubleshooting.
5. **Validation Gaps**: Inconsistent input validation, e.g., weak email or password checks.
6. **CORS Issues**: Front-end requests sometimes fail due to misconfigured CORS policies.
7. **Logout Issues**: Token invalidation during logout is not fully implemented.

These issues are under review and will be resolved in future updates.

# Future Enhancements

1. **Advanced Search and Filters**: Add functionality for more refined book searches based on multiple criteria like price, ratings, and availability.
2. **Recommendation System**: Implement AI-driven book recommendations based on user preferences and purchase history.
3. **Improved Performance**: Optimize database queries and caching for faster API response times.
4. **Enhanced Security**: Introduce multi-factor authentication (MFA) and stricter password policies.
5. **Real-Time Notifications**: Enable notifications for order updates and promotional offers using WebSockets.
6. **Analytics Dashboard**: Build a comprehensive dashboard for admins to monitor sales, inventory, and user activity.
7. **Localization and Multilingual Support**: Support multiple languages to cater to a global audience.

These enhancements will improve functionality, scalability, and user experience.