“Online BookStore”

Submitted By- Kritika Ola

Trainer- Mr. Naveen Kumar

Date- 16 Mar, 2025

Batch- C1

Table Of Contents:

Project Description-

The Online Bookstore Web Application is a comprehensive full-stack project aimed at providing users with an intuitive and efficient platform to explore, search, and purchase a wide variety of books and manga online. The front end of the application is developed using React.js, incorporating Material-UI components to ensure a modern, responsive, and user-friendly interface. Key frontend features include a dynamic navigation bar with search functionality, a shopping cart with real-time item count updates, and user authentication for both customers and administrators. The application also implements responsive design principles to deliver a seamless experience across desktop and mobile devices.

On the backend, we are building a robust RESTful API using C# with ASP.NET Core. This backend is responsible for handling all the core business logic, including user registration, login authentication, role-based authorization, product management, order processing, and secure transactions. It is designed following best practices to ensure security, scalability, and maintainability.

The backend services are connected to a Microsoft SQL Server database, which serves as the primary data store for the application. The database schema is designed to efficiently manage data related to users, books, manga, orders, payment details, and administrative operations. We are utilizing Entity Framework Core to interact with the SQL Server database, streamlining data access and management. The system is structured to allow easy integration of additional features in future iterations, such as payment gateways, real-time notifications, and analytics.

Additional project goals include implementing dynamic search functionality, enabling users to search for products by keywords, categories, or authors. The system is also designed to support real-time cart updates, order tracking, and administrative reporting features. Future enhancements may include integrating third-party payment gateways, real-time stock tracking, customer reviews and ratings, activity logs for system monitoring, and personalized recommendations powered by machine learning.

Overall, this project delivers a complete e-commerce solution that combines a polished front-end interface with a powerful backend and a reliable relational database. It is aimed at providing a smooth and secure user experience for both customers and administrators.

Project Architecture-

In this project, I have developed the frontend using React.js, a powerful JavaScript library for building dynamic and responsive user interfaces. The frontend structure is organized to ensure scalability and maintainability.

1. Folder Structure:

- The main files are inside the src/ folder, which contains the pages like Home.js, Login.js, Register.js, AdminDashboard.js, and UserDashboard.js.
- I also have other components like Contact.js and Payment.js that are styled using separate CSS files (Contact.css and Payment.css).
- The App.js file handles routing between different pages, and index.js is the entry point of the React application.

2. Key Functionalities:

- For client-side routing, I am using react-router-dom, which allows smooth navigation between pages without reloading.
- The Navbar component includes a search bar, a cart icon with real-time item count updates, and navigation links for quick access to various sections like Books, Manga, Contact, and Orders.
- There is a search functionality that takes the user query and displays results on a separate Result.js page using query parameters.
- For state management, I am using React's built-in useState and useEffect hooks. API calls to the backend are made using fetch or axios.
- The UI is built with Material-UI (MUI) components for responsiveness, along with custom CSS for styling specific pages to prevent style clashes.

3. User Experience:

- The interface is fully responsive and mobile-friendly.
- Users can browse and search for books and manga, add items to the cart, manage their profile, place orders, and navigate seamlessly through the application.

Backend Architecture

For the backend, I am using ASP.NET Core in C#, which is a cross-platform, high-performance framework. The backend is designed to follow a clean architecture with separation of concerns, making it modular and easier to maintain.

1. Folder Structure:

- Controllers:
    - AuthController.cs: Manages authentication operations like user registration and login, and generates JWT tokens for secure authentication.
    - BooksController.cs: Handles all CRUD operations related to books.
    - UsersController.cs: Manages user data and administrative actions such as user role management.
- Data:
    - LibraryDbContext.cs: This is the Entity Framework Core (EF Core) database context. It configures the relationships and maps C# models to SQL Server tables.
- Migrations:
    - Contains EF Core migration files like InitialCreate and AddPasswordSaltToUser. These handle database schema creation and updates over time.
- Models:
    - User.cs and Book.cs are the domain models representing core data objects.
    - UserDto.cs and UserLoginDto.cs are Data Transfer Objects used to securely transfer user information between the client and server.
    - LibraryManagementDatabaseSettings.cs manages configuration settings related to the database.
- Services:
    - UserService.cs and BookService.cs hold the core business logic for user management and book operations. These are injected into the controllers using Dependency Injection.

Authentication & Authorization:

- I implemented JWT-based authentication to ensure secure API access.
- Role-based authorization is also in place. Admin users have additional privileges such as managing users, books, and orders, while regular users can browse and place orders.

3. Database:

- The project uses Microsoft SQL Server as the relational database.
- Entity Framework Core (EF Core) is used as the ORM tool to manage database interactions. It allows me to query and save data easily by working with strongly typed C# objects.
- Migrations are applied to keep the database schema in sync with the application models.

4. Configuration:

- Configuration settings such as connection strings and JWT settings are stored in appsettings.json and appsettings.Development.json.
- Program.cs handles service registration and configures middleware like authentication, authorization, and error handling.

✅ Frontend + Backend Integration

The React frontend communicates with the backend through HTTP requests using RESTful APIs. The frontend makes GET, POST, PUT, and DELETE requests to various endpoints exposed by the ASP.NET Core controllers.

- JWT tokens are included in the Authorization header for secure endpoints.
- The backend responds with JSON data, which the frontend uses to dynamically render the UI.
- Role-based permissions are enforced both on the backend and frontend to ensure secure and appropriate access.

Frontend Component Breakdown

The frontend of the application is built using React.js. It is structured into different components, each with a specific responsibility. The main focus areas are state management, routing, and UI components.

1. State Management

- The application uses React's built-in useState and useContext hooks to manage state.
- User authentication state, theme settings, and user roles are managed globally using the Context API.
- Local component state is used for handling form inputs, loading states, and modals.

2. Routing

- Routing is handled using react-router-dom.
- There are separate routes for:
    - Home (/)
    - Login (/login)
    - Register (/register)
    - Admin Dashboard (/admin-dashboard)
    - User Dashboard (/user-dashboard)
- Protected routes are implemented to ensure only authenticated users can access certain pages.
- Role-based routing ensures that Admins and Users are redirected to their respective dashboards.

3. UI Components

- Components are modular and reusable.
- Major components include:
    - Navbar.js (Contains navigation links and theme toggle)
    - Login.js (Handles user authentication)
    - Register.js (Handles user registration)
    - AdminDashboard.js (Displays admin functionalities like user management)
    - UserDashboard.js (Displays user functionalities like borrowing books)
- Styling is handled with CSS Modules to avoid style clashes. Bootstrap is used for consistent and responsive design.

📌 Backend Component Breakdown

The backend is built using ASP.NET Core (C#). It exposes a RESTful API that interacts with a SQL Server database.

1. Controllers

- AuthController.cs: Handles user authentication (login and registration).
- UsersController.cs: Manages user operations, including role assignments.
- BooksController.cs: Manages book-related operations, including CRUD functionalities.

2. Services

- UserService.cs: Contains business logic for user management and authentication.
- BookService.cs: Contains business logic for handling books.
- These services are injected into controllers using Dependency Injection (DI).

3. Data Layer

- LibraryDbContext.cs: Represents the database context using Entity Framework Core.
- The Models folder contains:
    - User.cs & UserDto.cs: Define user entities and data transfer objects.
    - Book.cs & BookSQL.cs: Define book entities.
    - LibraryManagementDatabaseSettings.cs: Holds database configuration settings.

API Design

1. Authentication Mechanism

- Authentication is implemented using JWT (JSON Web Tokens).
- Passwords are hashed and salted before being stored in the SQL Server database.
- Tokens are returned on successful login and are required for accessing protected endpoints.

2. Endpoints-

Endpoint	Method	Description
/api/auth/register	POST	Registers a new user
/api/auth/login	POST	Retrieves a list of users (Admin Only)
/api/users	GET	Retrieves a list of users (Admin Only
/api/users/{id}	GET	Retrieves a user by ID
/api/users/promote/{id}	GET	Promotes a user to Admin
/api/books	GET	Retrieves all books
/api/books/{id}	ADD	Adds a new book (Admin Only)
/api/books	Post	Adds a new book (Admin Only)
/api/books/{id}	PUT	Updates book information (Admin Only)
/api/books/{id}	DELETE	Deletes a book (Admin Only)

Main Components-

1. React

What it is:
React is the core framework I used to build the frontend of my application. It's a JavaScript library that helps in creating fast, responsive user interfaces through components.

How I used it:
I structured the entire frontend using React. I broke down the UI into reusable components like Navbar, Sidebar, ThemeToggle, and different pages such as Home, Login, Register, AdminDashboard, and UserDashboard. Each component is built as a functional component. I handled state locally where it made sense, using the useState and useEffect hooks—for example, managing form inputs and API calls.

2. Redux

What it is:
Redux is a state management library that helps manage and centralize the app's global state.

How I used it:
I used Redux to manage the global state across my frontend. Things like authentication status, user roles, and theme preferences are stored in a central Redux store. For example, when a user logs in, I dispatch a LOGIN_SUCCESS action to update the authentication state, store the JWT token, and set the role of the user (Admin or User). Components like the Navbar and Dashboard use useSelector to read from the Redux store, so they update automatically based on the user's state. I also implemented Redux Thunk to handle asynchronous operations, such as API requests for logging in and fetching user data.

3. Routing (react-router-dom)

What it is:
react-router-dom is the library I used for client-side routing in my React app. It allows me to navigate between different pages without reloading the page.

How I used it:
I set up routing to navigate between different pages of the app—like the Login, Register, Admin Dashboard, and User Dashboard. I used BrowserRouter, Routes, and Route to define the paths. For role-based routing, I created protected routes so that only authenticated users can access the dashboard pages. Admin and User dashboards are separated based on roles, which I control using the authentication data from Redux.

4. REST API

What it is:
A REST API (Representational State Transfer API) allows the frontend and backend to communicate over HTTP using standard HTTP methods like GET, POST, PUT, and DELETE.

How I used it:
I built a REST API using C# and ASP.NET Core on the backend. On the frontend, I used fetch and axios (if you prefer axios, we can mention it) to make HTTP requests to my backend endpoints. For example, I send POST requests to the login and register endpoints, and GET requests to fetch the books or user data. All my API requests are secured using JWT tokens, which I include in the headers for authentication and authorization.

5. JWT Authentication

What it is:
JWT (JSON Web Token) is a secure way to transmit information between parties as a JSON object. It's often used for authentication.

How I used it:
I implemented JWT-based authentication in my project. When a user logs in, the backend generates a JWT token, which I store in localStorage on the frontend. I attach this token to the Authorization header when making protected API requests. I also decode the token to get user roles and enforce role-based access in the app, like preventing non-admin users from accessing admin routes.

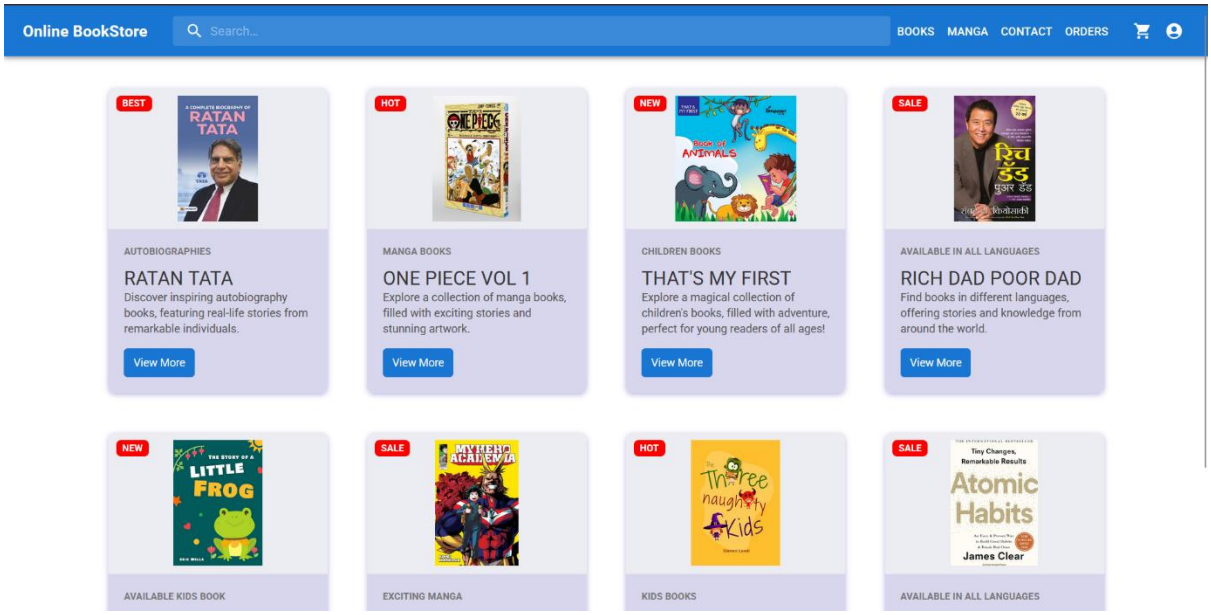
6. Form Handling

What it is:
Form handling in React involves capturing and managing user input, validating it, and submitting it to the backend.

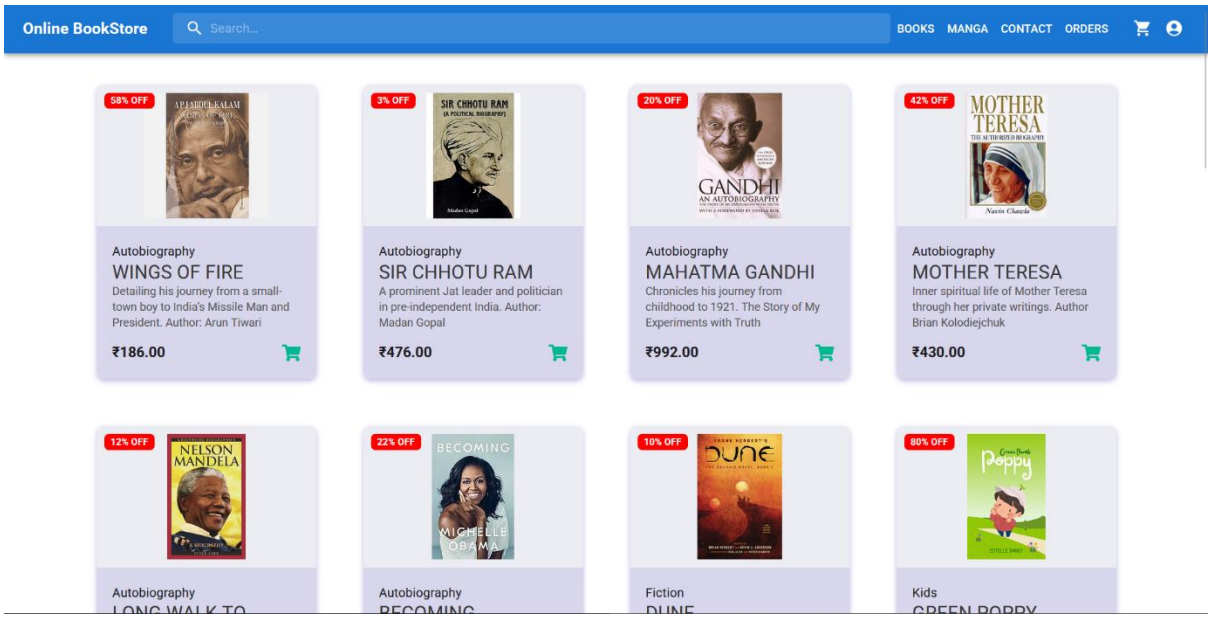
How I used it:
I created controlled components for forms like Login, Register, and adding new stock data. I used useState to manage the form input values and wrote functions to handle onChange and onSubmit events. I also performed basic validations on the frontend before sending data to the backend through API calls. For example, in the registration form, I check if passwords match before making the API request.

Screen-Shots:

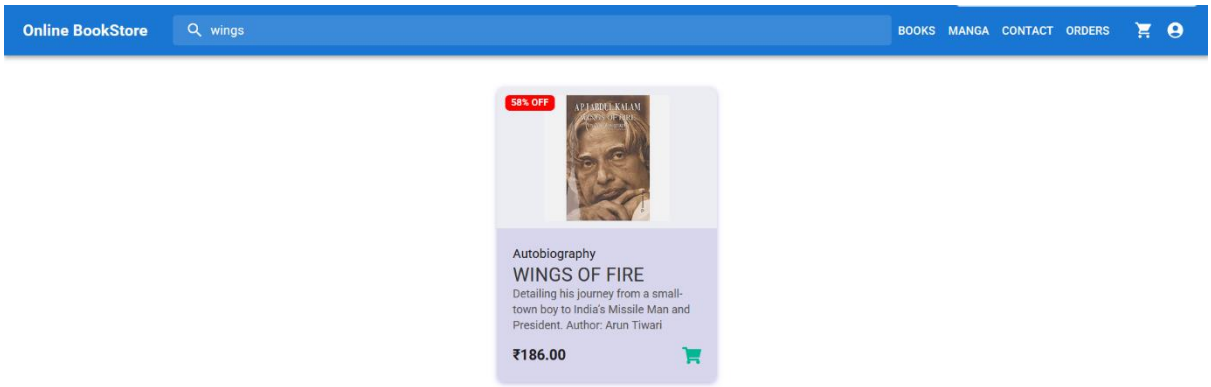
Home Page-



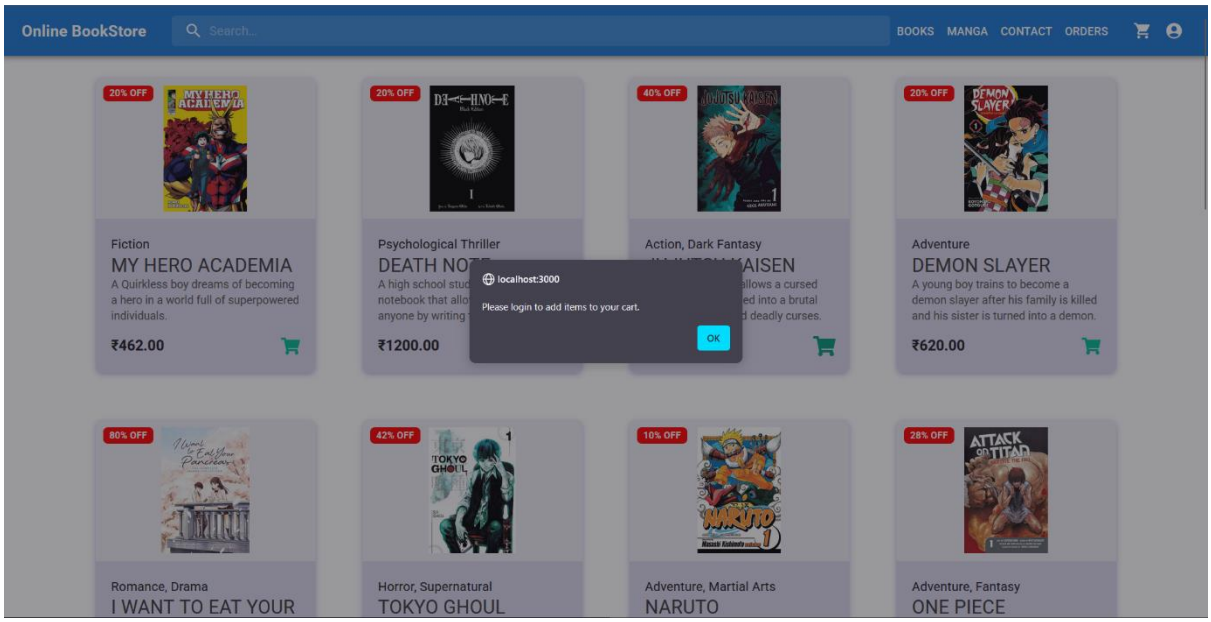
Books Page



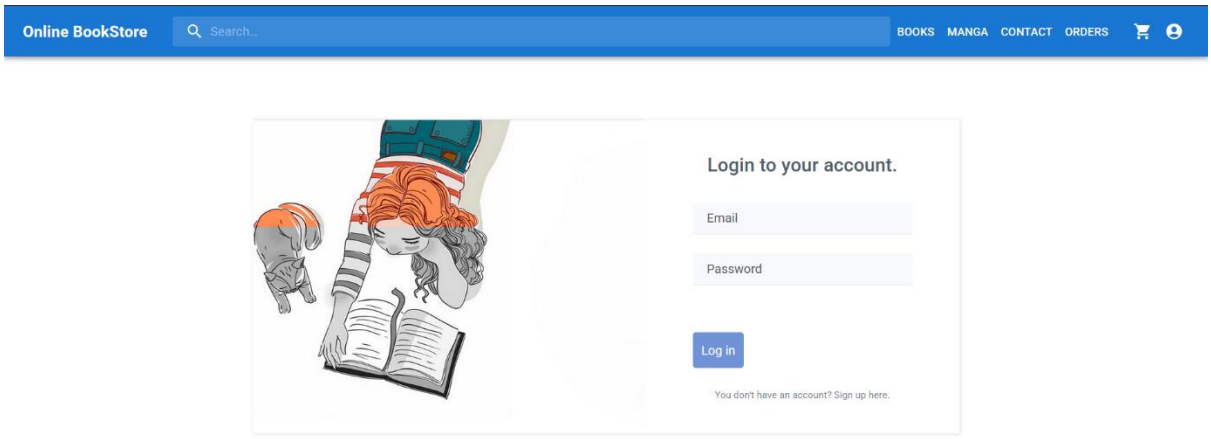
User Can Browse Books-



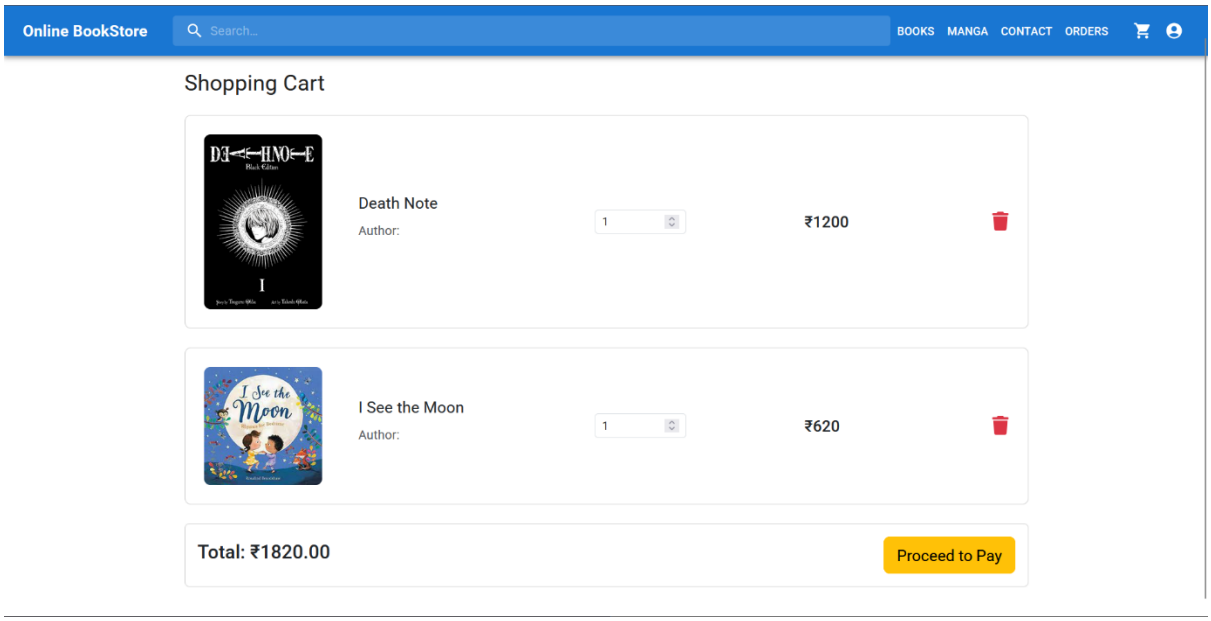
User can only add things to cart, once he's logged in..



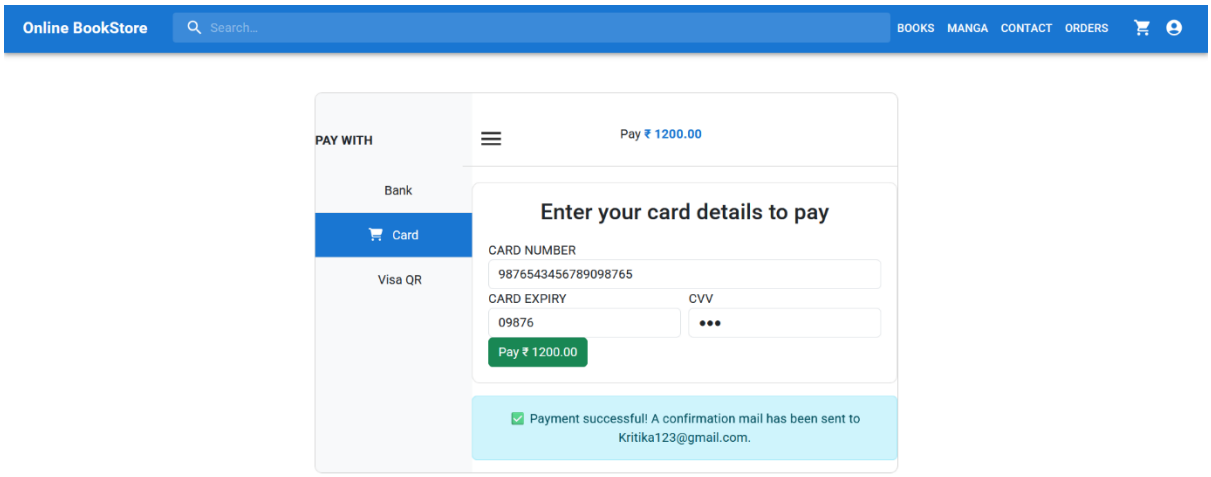
Login Page



Cart Page-



Secure Payment-



Database Design & Storage Optimization

Entity-Relationship Diagram (ERD):
In my project, I designed the database schema in SQL Server to support a Library Management System. The ERD reflects the main entities and their relationships. Here's a breakdown of how I structured it:

1. **User Table**
    - Stores user information such as UserId, Username, PasswordHash, PasswordSalt, Role (Admin/User), etc.
    - It connects to borrowings if we track borrowed books per user.
2. **Book Table**
    - Stores data about books like BookId, Title, Author, ISBN, Category, AvailabilityStatus.
    - It relates to categories and borrowings.
3. **Category Table**
    - Helps classify books into different genres or categories (CategoryId, CategoryName).
4. **Borrowing Table** (if implemented in the future)
    - Links users and books to keep a record of which user borrowed which book, with BorrowingId, UserId, BookId, BorrowDate, ReturnDate.
5. **Admin/User Roles**
    - Managed through the Role field in the User table to control access between AdminDashboard and UserDashboard.

I used **Entity Framework Core** for ORM, which handles these relationships in code. I've also applied data annotations and Fluent API configurations to manage relationships like one-to-many (one category has many books) and foreign key constraints.

Optimization Techniques for Efficient Queries:

Here's how I made the database operations efficient:

1. ✅ **Indexes**
    - I added indexes on frequently searched columns like Username (in Users) and Title (in Books).
    - This speeds up search queries on login and book searches.

2. ✅ **Primary and Foreign Keys**
    - Defined clear PK and FK relationships to maintain referential integrity and enable fast joins.
3. ✅ **Stored Procedures (optional)**
    - (If you did this) I created stored procedures for repetitive tasks like user login validation or fetching books by category.
4. ✅ **Pagination**
    - On the backend APIs for listing books, I implemented pagination using Skip() and Take() in LINQ.
    - This avoids sending too much data to the frontend and keeps the UI snappy.
5. ✅ **Asynchronous Queries**
    - I used async methods with EF Core (await _context.Users.ToListAsync()) to avoid blocking threads and improve performance.
6. ✅ **Password Hashing & Salting**
    - For security, I stored hashed and salted passwords instead of plain text.
    - I added salt to the User entity and handled hashing in the AuthController and UserService.
7. ✅ **Caching** (Optional/Future)
    - Planning to implement caching for frequently accessed data like book lists using in-memory caching or Redis for better scalability.

Github Repositories:

FrontEnd- https://github.com/KritikaOla/Frontend-BookStore

Backend- https://github.com/KritikaOla/Server