NAME: ARAVAPALLI SIREESHA

JAVA -REACT_BATCH3

Bus Ticket Reservation - Report

Problem Statement

The traditional bus ticketing system is prone to overbooking, inaccurate seat availability, and poor customer experience due to manual handling and lack of real-time updates. This leads to inefficiencies such as revenue leakage and customer dissatisfaction. The proposed solution centralizes bus and route management, real-time seat availability, booking, payment, cancellation processes, and e-ticket generation, while ensuring secure role-based access and API security with JWT (JSON Web Token) for authentication and authorization.

System Overview

The system is designed to facilitate efficient bus ticket reservations and management through a comprehensive digital platform. It covers all the essential aspects of bus ticketing, including administration, booking, payments, cancellations, and reporting. The system includes two primary user roles: **Admin** and **Customer**, each with distinct access levels and functionalities.

Admin Role:

- **Manage Buses**: Add, edit, and delete buses.
- **Manage Routes**: Create and edit routes for buses.
- **Manage Trips**: Schedule trips, assign buses, and set fares.
- **Manage Seat Inventory**: Track seat availability for each trip.
- **Generate Reports**: View and export reports on bookings and payments.
- **Access to Admin Dashboard**: Comprehensive view of all active bookings, trips, and system statistics.

Customer Role:

- **Search Trips**: Search for trips based on source, destination, and date.
- **Select Seats**: Choose seats for available trips based on real-time availability.
- **Book Tickets**: Book tickets for selected trips and pay online.

- **Download E-Tickets**: Generate and download e-tickets with QR codes.
- **Cancel Bookings**: Cancel bookings within the defined policy and receive refunds.
- **View Booking History**: View and manage past bookings.

- **Security:**
  - **Authentication**: The system uses JWT (JSON Web Token) for secure authentication and role-based authorization.
  - **Authorization**: Access to various functionalities is governed by the user's role (Admin or Customer), ensuring that only authorized users can access sensitive actions.
- **Reporting:**
  - **Booking Reports**: Provide detailed summaries of booking activities for Admins.
  - **Payment Reports**: Offer payment summaries and transaction details, along with export capabilities to PDF for both bookings and payments.
- **Cancellation & Refund Policy:**
  - The system supports a flexible cancellation policy, allowing customers to cancel bookings within the specified time frame and receive refunds or credits.

System Architecture and Development Guidelines

Backend (Spring Boot + MySQL)

- **Tech Stack**: Java 17, Spring Boot 3.5.x, Spring Security + JWT, JPA/Hibernate, MySQL, Swagger UI (springdoc-openapi).
- **Key Modules**: Authentication, Bus/Route management, Trip Scheduling, Seat Inventory, Booking/Payment, Ticketing/Cancellations, Reports.

Frontend (React + Vite + Tailwind CSS + React Router)

- **Tech Stack**: React 18, Vite 5, React Router 6, Axios, jwt-decode, Tailwind CSS.

- **Flow**: Login/Register → Store JWT → Route to Customer or Admin areas; JWT token attached to requests with Axios.

Key Functional Modules Implemented

1. **Authentication & Users**: Registration, login, JWT issuance, role management.
2. **Bus & Route Management**: Admin creates/reads buses and routes.
3. **Trip Scheduling & Seat Inventory**: Admin creates trips; public search for trips and seat listings.
4. **Booking & Payment**: Seat hold, cancellation, checkout, payment processing.
5. **Ticketing & Cancellations**: Ticket retrieval, PDF export, cancellation flow.
6. **Reports & Dashboards**: Booking and payment summaries with PDF export functionality.

Swagger API Documentation Integration

- ### Swagger UI

  Swagger UI is integrated to document and interact with the backend API. It allows you to easily test all available API endpoints from a web interface. To access the Swagger documentation, go to the following URL after starting the backend:

  Swagger UI URL: http://localhost:8080/swagger-ui/index.html

Key API Endpoints in Swagger

1. **Authentication Endpoints**:
   - POST /api/v1/auth/login (Public)
   - POST /api/v1/auth/register (Public)
2. **Booking Endpoints**:
   - POST /api/v1/bookings/hold (Protected)
   - POST /api/v1/bookings/{id}/cancel (Protected)
3. **Bus & Route Endpoints**:
   - GET /api/v1/buses (Admin)
   - POST /api/v1/buses (Admin)
   - GET /api/v1/routes (Admin)

- POST /api/v1/routes (Admin)

4. **Payment Endpoints**:
    - POST /api/v1/payments/checkout (Protected)
5. **Report Endpoints**:
    - GET /api/v1/reports/bookings (Admin)
    - GET /api/v1/reports/payments (Admin)
    - GET /api/v1/reports/bookings/pdf (Admin)
    - GET /api/v1/reports/payments/pdf (Admin)
6. **Ticket Endpoints**:
    - GET /api/v1/tickets/{bookingId} (Protected)
    - GET /api/v1/tickets/{bookingId}/pdf (Protected)
    - DELETE /api/v1/tickets/{bookingId} (Protected)
7. **Trip Endpoints**:
    - GET /api/v1/trips (Admin)
    - POST /api/v1/trips (Admin)
    - GET /api/v1/trips/{id} (Public)
    - GET /api/v1/trips/{id}/seats (Public)
    - GET /api/v1/trips/search (Public)

Extended API Guidelines

- **Base URL**: /api/v1
- **Common Errors**: 400 (validation), 401 (unauthorized), 403 (forbidden), 409 (seat conflict), 422 (payment failure), 500 (server error).
- **Endpoints**: Detailed endpoint paths provided, ranging from authentication (POST /api/v1/auth/login) to trip management (GET /api/v1/trips).

Database Design

- **Normalization**: Around 3NF.
- **Entities and Relationships**:
    - Users ↔ Bookings/Payments: One user can have multiple bookings and payments.

- Buses/Routes ↔ Trips: One bus can serve multiple trips, as can one route.
- Seat availability is derived from the Seat and BookingSeat entities on each trip.

Non-Functional Requirements

- **Security**: BCrypt password storage, signed JWT, input validation.
- **Performance**: Optimized seat hold and conflict checks in repository/service layers.
- **Reliability**: Transactional management for booking/payment statuses.
- **Scalability**: Clear separation for future service splitting (e.g., Search, Booking, Payments).
- **Auditability**: Tracking payment references and ticket numbers.

UX Implementation

- **Consistency**: Use of consistent typography and colors via Tailwind CSS.
- **Clarity & Simplicity**: Minimal search fields (source, destination, date), clear seat selection and checkout flow.
- **Feedback & Responsiveness**: Real-time seat availability and clear post-action confirmation messages.
- **Error Prevention & Handling**: Input validation on the frontend, precise status codes from the backend (401/403 handled with guards).

Challenges Faced

1. **Handling Seat Availability Conflicts**
   One of the major challenges was ensuring that seat availability was accurately reflected in real-time. The system needed to account for concurrent bookings, and preventing overbooking was a key focus. Using transaction management with a reliable conflict detection mechanism in the backend helped solve this issue.
2. **JWT Authentication and Authorization**
   Implementing stateless JWT authentication posed its own set of challenges. It required careful handling of JWT expiration, token refreshing, and role-based

access control. Ensuring that only authorized users (Admins and authenticated Customers) could access specific endpoints was a critical aspect.

3. **Integrating Payment Gateway and Ticketing Flow**
   The payment gateway integration was initially challenging due to the need for handling various payment failures, ensuring transaction integrity, and issuing e-tickets (PDF generation with embedded QR codes). We also had to handle cancellations and refunds, which involved managing complex states for each booking.
4. **UI Responsiveness and Cross-Browser Compatibility**
   The frontend, built with React, Vite, and Tailwind, needed to work seamlessly across various devices and browsers. Ensuring responsiveness, user-friendly navigation, and proper error handling in forms were challenging but were addressed with Tailwind CSS's utilities.
5. **Database and Query Optimization**
   As the data grew, optimizing the database queries, especially for seat availability and trip search, became crucial. The need to join multiple tables (Trips, Buses, Routes, Seats, etc.) efficiently without slowing down the application was resolved with optimized queries and careful use of indexes.
6. **Ensuring Robust Error Handling and Validation**
   Handling errors gracefully across both frontend and backend was another challenge. The backend had to handle multiple failure scenarios, such as payment failure, booking conflicts, and invalid user inputs. On the frontend, this was complemented with real-time form validation and displaying clear error messages to the user.

Execution Notes

- **Backend**: Ensure MySQL is running and schema is reachable.
  Run: mvn clean package -DskipTests → java -jar target/*.jar or mvn spring-boot:run.
  Swagger UI available at: http://localhost:8080/swagger-ui/index.html.
- **Frontend**:
  - Run: npm install
  - Start: npm run dev → Access at http://localhost:3000

- Update the VITE_API_BASE_URL if the backend is not running at the default location.

Appendix – Dependency Highlights

- **JWT**: io.jsonwebtoken:jjwt-*
- **OpenAPI UI**: org.springdoc:springdoc-openapi-starter-webmvc-ui
- **Database**: com.mysql:mysql-connector-j
- **Test Framework**: JUnit 5, Mockito