Restaurant Inventory Management System

Project Report

Submitted By:- Aryan Dhiman

UID:- 23BCS12638

Section:-KRG-3b

Introduction & Abstract

The Restaurant Inventory Management System (RIMS) is a full-stack web application developed to efficiently manage restaurant data, menus, and customer orders. It employs a modern RESTful API architecture with separate frontend and backend components. The system's primary goal is to provide a secure and centralized digital platform where administrators can manage restaurant listings and customers can securely place and track orders1. It utilizes Spring Boot for the secure backend and React for a responsive user interface.

Project Objectives

The primary objectives for the RIMS project were to design and develop a complete, functional management system with the following goals:

Build a secure and scalable backend using Spring Boot 3.1.5 with Java 17.

Create a responsive and intuitive frontend using React 18.2.02.

Implement Role-Based Access Control (RBAC) for USER and ADMIN roles.

Allow administrators to manage (create, read, update, delete) restaurant and menu data.

Enable customers to place and track orders through a defined lifecycle.

Maintain data flexibility and scalability using a MongoDB (NoSQL) database.

System Architecture

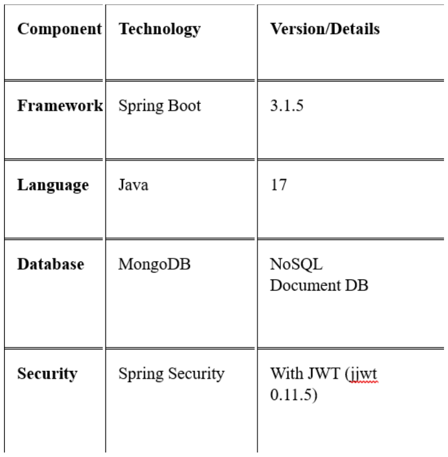
RIMS follows a three-tier architecture with a clear separation of concerns, built upon a RESTful API foundation:

Frontend (Client Layer): Built using React, it is responsible for user interaction, rendering, and communicating with the backend API via Axios3. It runs on the default port 3000.

Backend (Application Layer): Developed with Spring Boot 3.1.5 (Java 17), it serves as the secure application layer, handling business logic, data validation, authentication (JWT), and API request processing4. It runs on server port 8080.

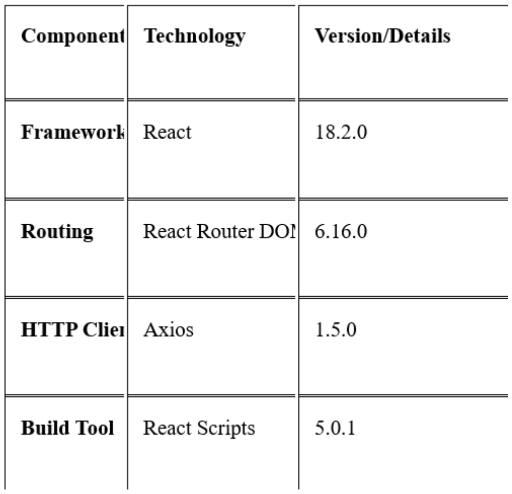
Database (Data Layer): MongoDB serves as the data store, accessed exclusively by the backend. It uses a document-based schema ideal for embedding complex data structures like MenuItem arrays within Restaurant documents and OrderItem arrays within Order documents.

Technology Stack (Detailed) Backend (Port 8080)



Component	Technology	Version/Details
Framework	Spring Boot	3.1.5
Language	Java	17
Database	MongoDB	NoSQL Document DB
Security	Spring Security	With JWT (jjwt 0.11.5)

Frontend (Port 3000)



Component	Technology	Version/Details
Framework	React	18.2.0
Routing	React Router DO	6.16.0
HTTP Clie	Axios	1.5.0
Build Tool	React Scripts	5.0.1

Features and Functionality

Authentication & Authorization

User Registration & Login: Users register with username, email, and password. Login generates a secure JWT token for stateless authentication.

Role-Based Access Control (RBAC): Access is governed by USER and ADMIN roles.

Protected Routes: Both frontend routes and backend API endpoints are protected using the JWT token and role checks.

Restaurant Management (ADMIN Role)

Full CRUD operations (Create, View, Update, Delete) on restaurant entities.

Each restaurant entity includes name, description, address, phone, imageUrl, and an embedded array of menu items.

Menu Management

Menu items are embedded within the corresponding Restaurant document, ensuring atomicity.

Each MenuItem contains name, description, price, and category.

Order Management

Order Creation: Users can create new orders with multiple items.

Order Tracking: Orders follow a lifecycle with statuses:

PENDING, CONFIRMED, DELIVERED, CANCELLED. Admins can update the status.

Order Data: Each order includes userId, restaurantId, totalAmount (calculated on the backend), status, and orderDate.

User Interface Features

Dual Dashboards: Separate views for USER (browsing, ordering) and ADMIN (management).

Dark Mode: Implemented via a DarkModeToggle and a DarkModeContext for theme management.

Responsive Design: Ensures optimal usability across various devices.

Database Design

The system uses MongoDB, a NoSQL document database, enabling flexible schema design, particularly for nested data.

Rationale: Embedding menu and items improves read performance, as related data is retrieved in a single query, which aligns well with the high read-frequency nature of menu and order display.

API Endpoints

The backend provides a set of secure RESTful API endpoints under the base URL http://localhost:8080.

Authentication Endpoints

POST /api/auth/register - User registration (Public)

POST /api/auth/login - User login and JWT token generation (Public)

Restaurants Endpoints

GET /api/restaurants - Get all restaurants (USER, ADMIN)

GET /api/restaurants/{id} - Get restaurant by ID (USER, ADMIN)

POST /api/restaurants - Create restaurant (ADMIN)

PUT /api/restaurants/{id} - Update restaurant (ADMIN)

DELETE /api/restaurants/{id} - Delete restaurant (ADMIN)

Orders Endpoints

POST /api/orders - Create new order (USER)

GET /api/orders/user/{userId} - Get user's order history (USER)

GET /api/orders/restaurant/{restaurantId} - Get orders for a specific restaurant (ADMIN)

GET /api/orders/{id} - Get order by ID (USER, ADMIN)

PUT /api/orders/{id}/status - Update order status (ADMIN)

Security Implementation

The system's security is anchored in Spring Security and the JWT standard.

JWT-based Authentication: Tokens are generated on login and validated on every protected API call using the JwtAuthenticationFilter.

Password Encryption: User passwords are encrypted using a strong algorithm managed by Spring Security before storage.

Role-based Authorization: Endpoints are protected to ensure only users with the required roles (ADMIN or USER) can access them.

CORS Configuration: Explicit Cross-Origin Resource Sharing (CORS) is enabled for the frontend host (localhost:3000) to allow secure communication with the backend (localhost:8080).

Token Expiration: JWT tokens are set to expire after

86400000ms (24 hours), requiring re-authentication to mitigate security risks from long-lived tokens.

Frontend Architecture

The React application uses a component-based architecture for high reusability and maintainability.

Components: Logic and UI are encapsulated in modules like Login, Dashboard, and reusable elements like RestaurantCard and DarkModeToggle.

Contexts: The Context API is used for global state management, specifically:

AuthContext: Handles the authentication state and JWT token persistence.

DarkModeContext: Manages the application-wide theme state.

Service Layer: The api.js file serves as the centralized service layer, handling all HTTP requests via Axios, separating API logic from UI components.

Styles: Separation of concerns is maintained by using dedicated CSS files for each component.

Backend Architecture

The Spring Boot backend adheres to the industry-standard Layered Architecture (MVC variant).

Models (POJOs): Define the data structure for MongoDB collections (User, Restaurant, Order).

Repositories: Interfaces (UserRepository, RestaurantRepository) extending Spring Data MongoDB to handle data access and persistence with minimal boilerplate code.

Services: (UserService, RestaurantService, OrderService) Implement the core business logic, validation, and transaction management.

Controllers: (AuthController, RestaurantController,

OrderController) Handle incoming API requests, validate DTOs, and coordinate with the Service layer.

Security & Configuration: Dedicated classes for JWT utility (JwtUtil), security configuration (SecurityConfig), and application initialization (DataSeeder).

User Interface Design

The UI is designed to be intuitive, responsive, and role-aware.

Dual-Role Design: Separate, optimized dashboards ensure users only see relevant information (e.g., users see Restaurant Cards and order history; admins see management tables and forms).

Order Flow: The OrderModal component provides a focused, step-by-step interface for placing orders.

Theme Management: The Dark Mode Toggle allows for immediate theme switching, enhancing user preference and accessibility.

Responsiveness: The layout dynamically adjusts to screen sizes, ensuring a consistent experience on desktop and mobile devices.

Testing Considerations

A robust testing strategy ensures system reliability and security.

• Unit Tests (Service Layer): Use JUnit and Mockito to verify the correctness of business logic in Service classes (e.g., order total calculation, status transition validity).

Integration Tests (Controller Layer): Use Spring Boot Test to verify that API endpoints handle requests, security rules (RBAC), and data interactions correctly.

E2E (End-to-End) Tests: Use tools like Cypress or Playwright to simulate full user flows (e.g., user registration, admin creating a restaurant, user placing an order) to validate frontend-backend integration.

Deployment Considerations

The microservices-inspired architecture facilitates independent deployment.

Backend: The Spring Boot application is packaged as an executable JAR and can be deployed to a cloud server (e.g., AWS, Azure) alongside a managed MongoDB database instance (e.g., MongoDB Atlas).

Frontend: The React application is built into static assets (npm run build) and can be deployed easily to static hosting providers (e.g., Netlify, Vercel).

Configuration: The MongoDB connection string

(mongodb://localhost:27017/restaurant_inventory) and the JWT secret key must be configured as secure environment variables during deployment.

Future Enhancements

The current RIMS provides a strong foundation. Future development could include:

Live Inventory Tracking: Implementing detailed stock levels for ingredients (true inventory management) instead of just menu item management.

Real-time Notifications: Integrating WebSockets to instantly alert administrators of new order placements.

Payment Gateway Integration: Adding a secure third-party payment provider (e.g., Stripe) to enable transaction processing.

Reporting & Analytics: Developing a dedicated dashboard for generating reports on sales, popular items, and order volume.

Conclusion

The Restaurant Inventory Management System successfully achieves its goal of providing a secure, role-based platform for restaurant and order management. By utilizing the modern, robust combination of Spring Boot, React, and MongoDB, the project delivers a scalable and maintainable solution that fulfills all core objectives. The modular design and strong security foundation make RIMS an excellent professional application ready for expansion into a production environment.