Virtual Queue System

A PROJECT REPORT

Submitted by

Garvi Dabas(23BCS11346)

inpartialfulfillmentfortheawardofthedegree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE ENGINEERING

Chandigarh University

Aug-2025

INDEX

Virtual Queue System

1. Introduction and Project Objective

This report details the architecture, design, and implementation of the Virtual Queue Management System, a full-stack, enterprise-style web application designed to replace physical waiting lines.

The primary objective of this project was to solve the problem of physical, crowded waiting rooms. This system allows users to view wait times and join a queue remotely, while businesses (such as a clinic, pharmacy, or service center) can manage their customer flow, serve tokens, and send notifications, all from a web application.

The system is architected as a decoupled, three-tier application, consisting of:

- A standalone React (Vite) frontend for the user interface.
- A stateless Spring Boot (Java 17) backend serving a secure REST API.
- A cloud-hosted PostgreSQL database (managed by Supabase) for data persistence.

This report will cover the in-depth implementation of each layer, from the security architecture and database design to the frontend state management and final deployment.

2. System Architecture and Technology Stack

2.1 Technology Stack

The application is built on modern, professional technologies chosen for security, scalability, and performance.

Component	Technology / Library
Frontend (Client)	React (Vite), React Router, Material-UI (MUI), Axios
Backend (Server)	Java 17, Spring Boot, Spring Security, Spring Data JPA
Database	PostgreSQL (Cloud-hosted via Supabase)
Notifications	Resend API (via Java HttpClient)
Authentication	JSON Web Tokens (JWT)
Deployment	Docker, Render (for Backend), Vercel (for Frontend)

2.2 High-Level Architecture

The system follows a classic three-tier architecture. This decoupled model ensures that the frontend and backend are completely independent, communicating only through a stateless REST API. This makes the system easier to maintain, scale, and test.

- **Tier 1: Presentation (React Frontend):** Hosted on Vercel, this is what the user sees and interacts with. It manages UI state and makes API calls.

- **Tier 2: Logic (Spring Boot Backend):** Hosted on Render, this Docker container handles all business logic, security, and database operations.

- **Tier 3: Data (PostgreSQL Database):** Hosted on Supabase, this tier is responsible for data persistence.

---

3. Backend Implementation (Spring Boot)

The backend is a secure REST API built with Spring Boot. It is responsible for all business logic, data persistence, and security. The architecture follows a standard Controller-Service-Repository pattern.

3.1 Data Layer (Models & Repositories)

This layer communicates directly with the PostgreSQL database using Spring Data JPA.

- **Models (Entities):** Plain Old Java Objects (POJOs) annotated with @Entity. The primary models are User, Queue, and Token, which map directly to the database tables.

- **Repositories:** These are interfaces (e.g., UserRepository, QueueRepository) that extend JpaRepository. Spring automatically provides all common CRUD (Create, Read, Update, Delete) operations, as well as custom "magic" methods like findByEmail(String email).

3.2 Service Layer (Business Logic)

This is the "brain" of the application, marked with the @Service annotation. This layer is responsible for coordinating all operations.

- **AdminService:** Contains logic for creating/updating queues, serving the next token, and skipping tokens.

- **CustomerService:** Contains logic for joining a queue (including checking for duplicates) and canceling a token (including ownership verification).

- **NotificationService:** Uses @Async to run on a separate thread, allowing it to send an email via the Resend API without blocking the main application flow.

3.3 Controller Layer (API Endpoints)

This is the "front door" of the backend. These classes, annotated with @RestController, define the API endpoints that the React frontend can call.

- **AuthController:** Handles public @PostMapping requests for /api/auth/login and /api/auth/register.

- **AdminController:** Secures all its endpoints (e.g., /api/admin/**) to only allow users with ROLE_ADMIN. It handles requests for creating queues, serving tokens, etc.

- **QueueController & TokenController:** Handle requests for customers, such as joining a queue or checking token status.

---

4. Security Architecture (Spring Security & JWT)

Security is a core pillar of the application, implemented using a stateless JSON Web Token (JWT) flow. This approach ensures that the backend does not need to store session data, making it highly scalable.

4.1 Part 1: The Login Flow (Authentication)

When a user attempts to log in from the React frontend, the following occurs:

1. The frontend sends a POST request with an email and password to the /api/auth/login endpoint.

2. AuthController passes these credentials to Spring Security's AuthenticationManager.

3. The AuthenticationManager uses UserDetailsServiceImpl (our custom service) to find the user in the database via UserRepository.

4. It then uses the PasswordEncoder bean to securely compare the provided password with the hashed password in the database.

5. If successful, JwtUtils (our utility class) generates a new, signed JWT. This token contains the user's email and role (e.g., ROLE_ADMIN) as "claims."

6. This JWT is sent back to the React frontend, which stores it in localStorage.

4.2 Part 2: API Request Validation (Authorization)

Once logged in, every API request from the frontend is secured as follows:

1. The React apiClient (Axios) uses an interceptor to automatically read the JWT from localStorage and attach it to the request in the Authorization: Bearer <token> header.

2. The request arrives at the backend and is immediately intercepted by our custom JwtAuthFilter.

3. This filter validates the token's signature using JwtUtils and our secret key.

4. If the token is valid, the filter extracts the user's email, loads their details, and sets their identity in the Spring Security context.

5. The request is finally passed to SecurityConfig, our "rulebook." This file enforces rules like .requestMatchers("/api/admin/**").hasRole("ADMIN").

6. If the user has the correct role for the endpoint, the request is allowed to proceed to the controller. If not, a 403 Forbidden error is returned.

---

5. Frontend Implementation (React)

The frontend is a modern, responsive Single Page Application (SPA) built with React and Vite. It provides a fast, interactive experience by managing its own UI state and fetching data from the backend API.

5.1 State Management (Context API)

Global application state is managed using React's Context API, which avoids the need for complex libraries and allows any component to access crucial data.

- **AuthContext.jsx**: This provider manages the user's authentication status. It holds the current user object and the login() and logout() functions. It also reads from localStorage on app load to maintain the user's session.

- **QueueContext.jsx**: This provider manages queue-related data, such as the list of active queues for the homepage and the specific token a user received after joining.

5.2 API Communication (Axios)

All communication with the Spring Boot backend is handled through a central Axios instance in apiClient.js.

- **Base URL**: A single baseURL is configured to point to the deployed backend (e.g., https://virtual-queue-system-backnd.onrender.com/api).

- **Interceptor**: The client uses an axios.interceptors.request to automatically attach the JWT from localStorage to every outgoing request, as described in the security section.

- **API Service**: All API calls are defined as clean, async functions in api.js (e.g., getAllQueues(), updateQueue()), which are then imported by the page components.

5.3 Component & Page Structure

The app is organized by function, separating pages from reusable components.

- **Pages**: Components that represent an entire page, such as HomePage.jsx, AdminDashboardPage.jsx, and QueueManagementPage.jsx.

- **Components**: Reusable "bricks" used to build pages, such as Navbar.jsx, QueueCard.jsx, and the EditQueueModal.jsx.

---

6. Deployment & CI/CD

A major achievement of this project was the successful deployment of the full-stack application to a live, production-grade environment. This involved a decoupled CI/CD (Continuous Integration / Continuous Deployment) pipeline.

6.1 Backend: Docker & Render

The Spring Boot backend is deployed on Render.

1. A multi-stage Dockerfile was written to containerize the application.

2. **Stage 1 (Build):** Uses a full Java JDK image to build the project and create the executable .jar file.

3. **Stage 2 (Run):** Copies the .jar file into a minimal Java JRE image. This creates a small, secure, and efficient final image for deployment.

4. Render is connected to the GitHub repository. A push to the main branch automatically triggers a new Docker build and deployment.

6.2 Frontend: Vercel

The React frontend is deployed on Vercel.

1. Vercel is connected to the GitHub repository.

2. A push to the main branch automatically triggers Vercel to build the React project.

3. The resulting static files (HTML, CSS, JS) are deployed to Vercel's global CDN, providing extremely fast load times for users.

6.3 Secret Management & CORS

All secrets (database passwords, API keys) are handled securely using Environment Variables on both Render and Vercel.

- **Backend (Render):** Holds the database credentials, JWT secret, and Resend API key. It also holds a FRONTEND_URL variable.

- **Frontend (Vercel):** Holds a single VITE_API_URL variable, pointing to the live Render backend URL.

- **Security**: The Spring Boot SecurityConfig.java is configured to only accept Cross-Origin Resource Sharing (CORS) requests from the FRONTEND_URL, securing the API from unauthorized web clients.

---

7. Case Study: Debugging the 'Active/Inactive' Bug

During development, a persistent bug was encountered where a queue, after being set to "Active" in the edit modal, would revert to "Inactive."

This bug was a classic full-stack problem. A deep-dive analysis revealed a mismatch between the JSON key being sent by the React frontend and the key expected by the Spring Boot backend.

- The Frontend (EditQueueModal.jsx) was sending a JSON key active.
- The Backend DTO (UpdateQueueRequest.java) had a field private boolean isActive; but its *setter method* was named public void setActive(boolean active).
- **The Problem**: Spring Boot's JSON mapping (Jackson) uses setter methods, not field names, to map incoming data. Because the setter was named setActive, it correctly looked for the active key.
- **The Bug**: The bug was in the AdminController.java, which was incorrectly calling request.isActive() on the DTO. The isActive field was never set (because the setter was setActive), so it defaulted to false.
- **The Fix**: The controller was changed to call the correct getter that matched the field (request.isActive()), and the DAdjustments were made to align the setter/getter/field naming, resolving the data-binding mismatch. This case study highlights the importance of end-to-end data tracing.

---

8. Conclusion and Future Work

This project successfully delivers a complete, secure, and deployed full-stack application. It demonstrates a mastery of modern web technologies, from backend API design and security to frontend state management and cloud deployment.

Future work to enhance this system could include:

- Real-Time Updates (WebSockets): Implementing WebSockets (e.g., using Spring's STOMP) to push live updates to the customer's status page, removing the need for manual refreshing.
- Admin Statistics: Adding a statistics panel to the admin dashboard to show data like average wait times, total customers served, and peak hours.