TMS (Tickets Management System) - Comprehensive System Documentation

Version: 1.3.0
Date: December 20, 2025
Author: MWIMULE Bienvenu

---

1. Executive Summary

TMS (Tickets Management System) is an advanced, enterprise-grade distributed ticketing and fleet management system designed for the transportation industry. It provides a unified ecosystem that connects passengers, transport companies, drivers, and administrators.

The system automates the entire lifecycle of bus travel: from route planning, scheduling, and dynamic bus swapping to secure ticket booking, payments, QR code validation, and real-time GPS tracking. Built on a microservices architecture, it ensures high scalability, fault tolerance, and seamless integration between web, mobile, and desktop client applications.

---

2. System Architecture

The system follows a **Microservices Architecture** pattern, ensuring that each business domain (Ticketing, Payments, Users, etc.) handles its own data and logic.

2.1 Backend Services

The backend is composed of several isolated services running in Docker containers, communicating via REST APIs and RabbitMQ.

Service Name	Technology	Responsibility
Gateway	Nginx	**The Entry Point.** Reverse proxy that routes external requests from apps to internal microservices. Handles SSL, load balancing, and static path routing.
Auth Service	Python (FastAPI)	Manages User Identity (Sign up/Login), JWT Token issuance, and Role-Based Access Control (RBAC).

Service Name	Technology	Responsibility
Company Service	Python (FastAPI)	The "Brain". Manages Companies, Buses, Routes, Segments, Schedules, and Driver assignments. Handles logic like **Bus Swapping** and **Schedule Status validation**.
Ticketing Service	Python (FastAPI)	Handles Ticket creation, searching, seat availability, and generation of secure QR codes.
Payment Service	Python (FastAPI)	Processes payments (PayPal/Mobile Money). Uses **Redis** for idempotency keys to prevent double-charging and handles webhooks.
Notification Service	Python (FastAPI)	**Event-Driven.** Listens to RabbitMQ for events (Ticket Sold, Trip Cancelled) and sends SMS/Emails.
Tracking Service	Python (WebSocket)	Real-time GPS tracking of buses via WebSockets and Redis geospatial data.
AI Service	Python (FastAPI)	Powered by **Google Gemini**. Assists users with natural language queries ("Find me a bus to Kigali") and Admins with SQL generation.
QR Service	Python (FastAPI)	Dedicated service for verifying the cryptographic `HMAC-SHA256` signatures on ticket QR codes.
Super Admin Service	Python (FastAPI)	Provides system-wide analytics, company onboarding, and financial reporting.

2.2 Infrastructure Components

- **RabbitMQ:** Message Broker for asynchronous communication (e.g., Ticket Service -> Notification Service).
- **Redis:** In-memory key-value store used for:
  - Payment Idempotency (prevent duplicate transactions).
  - Real-time Bus Locations.
  - Caching frequent queries.
- **PostgreSQL:** The primary relational database for persistent storage (Users, Tickets, Buses, etc.).

---

3. Client Applications Breakdown

TMS offers interfaces for four distinct types of users.

3.1 Customer Web Portal (Frontend)

- **Tech Stack:** React.js, Vite, TypeScript, Tailwind CSS, Bootstrap.
- **Features:**

- **Search & Booking:** Search buses by Origin, Destination, and Date.
- **User Accounts:** Profile management and "My Tickets" history.
- **AI Chat Widget:** A floating assistant powered by Gemini to help find buses via text or voice.
- **Real-time Tracking:** Map view showing the live location of the bus for a booked ticket.
- **Digital Hub:** Users download/view their QR tickets directly from the portal.

3.2 Super Admin Dashboard

- **Tech Stack:** React.js, Material UI (MUI).
- **Purpose:** For the Platform Owners (TMS Admins).
- **Features:**
  - **Company Management:** Onboard new transport companies.
  - **Global Analytics:** View total revenue, total tickets sold, active buses.
  - **AI SQL Analyst:** An interface to ask questions in English ("Show me tickets sold last week") which generates and runs SQL queries safely.

3.3 Company Desktop Software

- **Tech Stack:** .NET 8 / Avalonia UI (Cross-Platform Desktop App).
- **Purpose:** The daily operational tool for Transport Company managers.
- **Features:**
  - **Fleet Management:** Add/Edit Buses and assign Drivers.
  - **Route Planning:** Define Routes (e.g., "Kigali - Musanze") and Segments.
  - **Advanced Scheduling:** Create trip schedules, enforce status checks (e.g., cannot edit "Departed" trips), and perform **Bus Swaps** in case of breakdowns.
  - **Offline Capability:** Critical operations persist locally if internet is lost.

3.4 Mobile Applications

- **Driver App:** (React Native/Expo) - Drivers login to see their assigned trips, start trips, broadcast their GPS location, and **Scan Passenger QR Codes** for boarding.
- **POS App:** (React Native/Expo) - Station agents ("Protokol") use this to sell tickets for cash at bus stations. Features **Hardened Offline Sync** with batched auditing to prevent fraud during internet outages.

---

4. Key Workflows & Logic

4.1 The Secure Booking Flow

1. **Search:** User queries "Kigali to Huye" for "Tomorrow".
2. **Locking:** When a user selects a seat, the system applies a **Row-Level Lock** (Pessimistic Locking) in the database to ensure no two users can book the last seat simultaneously.
3. **Payment:**
    - Frontend generates a unique UUID (`idempotency_key`).
    - Request sent to `payment-service` .
    - Service checks Redis: "Has this UUID been processed?"
    - If No -> Process Payment -> Save to Redis -> Return Success.
    - If Yes -> Return Cached Success immediately (Prevents double payment).
4. **Ticket Generation:**
    - System generates a QR code string.
    - Ideally, this string is **Signed** with a Secret Key (`HMAC`).
    - This signature ensures that even if a user prints the QR and changes "Standard" to "VIP", the signature validation will fail.

4.2 Real-Time Tracking Flow

1. **Driver App:** Captures phone GPS every 5 seconds.
2. **WebSocket:** Sends distinct coordinates to `tracking-service` .
3. **Redis:** Service updates the key `bus:{bus_id}:location` with the new lat/long.
4. **Customer App:** Subscribes to the bus they are booked on.
5. **UI Update:** The bus icon on the customer's map moves in real-time without refreshing the page.

4.3 Bus Swapping Intelligence

- **Scenario:** A bus breaks down 1 hour before departure.
- **Action:** Company Admin selects the trip and clicks "Swap Bus".
- **System Logic:**
    1. Validates if the new bus has equal or more seats than the *currently sold* tickets.
    2. Updates the `Schedule` record with the new `bus_id` .
    3. Triggers a generic "Bus Change" event to `notification-service` .
    4. Passengers receive an SMS: "Your bus has changed to Plate RAD 123A".

---

5. Security & Deployment

Security Measures

- **Gateway Isolation:** The outside world can ONLY access port 8000 (Nginx). All other service ports (8001, 8002, 5432) are hidden inside the Docker network.

- **JWT Implementation:** Stateless authentication. Services verify tokens without needing to check the database for every request.
- **Environment Variables:** Secrets (DB Passwords, API Keys) are injected at runtime via `.env` files, never hardcoded.

Deployment Strategy

- **Containerization:** Every service is a Docker container defined in `docker-compose.yml`.
- **Orchestration:** Docker Compose manages the lifecycle, networking, and volume persistence.
- **Updates:**
    1. `git pull` (Get code)
    2. `docker-compose build <service>` (Rebuild specific image)
    3. `docker-compose up -d` (Restart containers with zero downtime for unchanged services).

---

6. Access Information (Current Deployment)

- **Public IP:** `3.12.248.83`
- **API Gateway:** `http://3.12.248.83:8000/`
- **Customer Portal:** `http://3.12.248.83:8000/`
- **Super Admin Dashboard:** `http://3.12.248.83:8000/super-admin/`

This document serves as the primary reference for the TMS (Tickets Management System) architecture and functionality.