Service Responsibility and Boundary Definition

Target Audience: Backend Developers, Architects
Goal: Enforce loose coupling and high cohesion by strictly defining service boundaries.

---

1. Core Philosophy

- **Share Nothing:** Services must not share database schemas or tables.
- **Interact via Interfaces:** All communication must occur via public APIs (REST) or Domain Events (RabbitMQ).
- **Single Responsibility:** A service should have one reason to change.

---

2. Service Definitions

2.1 Auth Service (`/services/auth-service`)

- **Role:** Identity Provider (IdP).
- **Owns:**
  - User Credentials (Password hashes).
  - JWT Issuance & Signing Keys.
  - Role Definitions (RBAC).
- **Does NOT Own:**
  - User Profiles (Addresses, Preferences) -> *User Service*.
  - Driver Licenses -> *Company Service*.
- **Boundary Rule:** If it involves verifying *who* someone is, it belongs here. If it involves *what* they are doing, it is elsewhere.

2.2 Company Service (`/services/company-service`)

- **Role:** Domain Core / Fleet Management.
- **Owns:**
  - Company Metadata.
  - Assets (Buses, physical layout).

- Topography (Routes, Stops).
- Planning (Schedules, Trips).

- **Does NOT Own:**
  - Ticket Inventory (Seat locking) -> *Ticketing Service*.
  - Live GPS Data -> *Tracking Service*.
- **Boundary Rule:** This service defines the *static* and *planned* world. It does not handle the *transactional* selling of that world.

2.3 Ticketing Service (`/services/ticketing-service`)

- **Role:** Sales & Inventory engine.
- **Owns:**
  - Ticket Records (Status: Reserved, Paid, Used).
  - Seat Inventory (Row-level locking).
  - Pricing Logic (Fare calculation).
- **Does NOT Own:**
  - Payment Gateways -> *Payment Service*.
  - Bus Capacity Definition -> *Company Service*.
- **Boundary Rule:** This is the "Cash Register" logic. It cares about filling seats, not about the bus engine.

2.4 Payment Service (`/services/payment-service`)

- **Role:** Financial Processor.
- **Owns:**
  - Transactions (Records of money movement).
  - Integrations (M-Pesa, Stripe, PayPal).
  - Idempotency Keys (Preventing double-charge).
- **Does NOT Own:**
  - Order Fulfillment (Delivering the ticket).
- **Boundary Rule:** It answers one question: "Did we get the money?" It should be agnostic to *what* was bought.

2.5 Notification Service (`/services/notification-service`)

- **Role:** Communication Gateway.
- **Owns:**
  - Message Templates (SMS, Email).
  - Provider Integration (Twilio/AWS SNS).
  - Delivery Logs.
- **Does NOT Own:**

    - Business Logic (Deciding *when* to send).
- **Boundary Rule:** Dumb pipe. It receives a command "Send X to Y" and executes. It does not decide functionality.

2.6 Tracking Service (`/services/tracking-service`)

- **Role:** Telemetry & Geospatial.
- **Owns:**
    - WebSocket Connections (Live stream).
    - Redis Geospatial Index.
    - Trip History (Breadcrumbs).
- **Does NOT Own:**
    - Schedule adherence logic (Comparing planned vs actual) - *Shared responsibility, but storage is here.*

---

3. Communication Patterns

3.1 Synchronous (REST API)

- **Use when:** The client needs an immediate answer or the operation is a localized query.
- **Example:** `GET /trips/{id}` (Frontend asks Company Service).
- **Constraint:** Avoid Service-to-Service HTTP chains (Service A calls B, B calls C). This creates latency spikes and brittleness.

3.2 Asynchronous (Event-Driven)

- **Use when:** An action in one domain triggers side effects in others.
- **Pattern:** Publisher/Subscriber via RabbitMQ.
- **Example:**
    - Payment Service: *Publishes* `PaymentSuccess` .
    - Ticketing Service: *Subscribes* -> Issues Ticket.
    - Notification Service: *Subscribes* -> Sends Receipt.

---

4. Anti-Patterns (Strictly Forbidden)

4.1 The "Shared Database"

- **Violation:** Service A querying Service B's tables directly.
- **Why:** Tighly couples schemas. If B changes a column, A breaks.
- **Fix:** Service B must expose an API or publish data changes via events.

4.2 The "God Service"

- **Violation:** Putting everything into `Company Service` because "it's easier".
- **Why:** Creates a monolith that is hard to scale and deploy.
- **Fix:** Continually refactor. If `Company Service` starts handling payments, extract it.

4.3 "Distributed Monolith"

- **Violation:** Services that are physically valid but logically coupled (e.g., they must be deployed together).
- **Why:** Defeats the purpose of microservices.
- **Fix:** Ensure API versioning and backward compatibility.

Target Audience: Backend Engineers, System Architects, DevOps
Purpose: High-level technical understanding of the system topology, boundaries, and data flow.

---

1. Architectural Pattern

The TMS uses a **Microservices Architecture**. The system is decomposed into vertical business domains (e.g., Ticketing, Payments, Fleet), each managed by an isolated service.

- **Communication:**
  - **Synchronous:** HTTP/REST (FastAPI) for direct client requests.
  - **Asynchronous:** AMQP (RabbitMQ) for inter-service consistency and event propagation.
  - **Real-time:** WebSockets & Redis Pub/Sub for fleet tracking.
- **Storage:** Per-service logical isolation (logical separation within shared PostgreSQL instance for simplicity in current deployment, but architecturally distinct).
- **Infrastructure:** Containerized (Docker), orchestrated via Docker Compose (support for K8s).

---

2. System Topology

2.1 The Entry Point (API Gateway)

- **Component:** Nginx Reverse Proxy

- **Role:** Single ingress point for all external traffic.
- **Responsibilities:**
  - SSL Termination.
  - Path-based routing (e.g., `/api/v1/auth` -> Auth Service).
  - Static content serving for Web Frontend.
  - CORS handling.

2.2 Core Backend Services (Python/FastAPI)

Service	Responsibility	Key Interactions
Auth Service	Identity Provider (IdP). Issues/Verifies JWTs. Manages RBAC.	All services (via Token Validation).
Company Service	**Domain Core.** Manages Fleets, Routes, Stops, Schedules, and Driver assignments.	Publishes `ScheduleCreated`; Consumes `BusLocationUpdated`.
Ticketing Service	Inventory/Sales. Handles Booking locking, Seat selection, and QR generation.	Publishes `TicketSold`; Calls `PaymentService` for status.
Payment Service	Financial Transaction Processor. Handles Mobile Money/PayPal integrations.	Publishes `PaymentSuccess`; Consumes `TicketLocked`.
Notification Service	**Event Consumer.** Sends SMS/Emails based on system events.	Consumes `TicketSold`, `TripCancelled`.
Tracking Service	Real-time GPS ingest and broadcasting.	WebSockets to Clients; Redis Geospatial Index.
AI Service	LLM Integration (Google Gemini) for natural language queries.	Read-only access to DB schemas for SQL generation.
QR Service	Cryptographic verification of ticket validity.	Isolated for security/performance.

2.3 Data & Infrastructure Layer

- **Message Broker (RabbitMQ):**
  - Decouples services.
  - Examples: `PaymentService` emits `payment.confirmed` -> `TicketingService` finalizes ticket -> `NotificationService` SMS.
- **In-Memory Store (Redis):**
  - **Caching:** Frequently accessed Route/Schedule data.
  - **Distributed Locking:** Prevents double-booking of seats (`SETNX`).
  - **Geospatial:** Stores live bus coordinates.
- **Database (PostgreSQL):**

    - Relational persistence. Heavily relies on Foreign Keys and ACID transactions within service boundaries.

---

3. Data Flow Diagrams (Conceptual)

3.1 Flow: Ticket Booking (Sync + Async)

1. **Client** requests seat lock -> **Gateway** -> **Ticketing Service**.
2. **Ticketing Service** acquires Redis Lock on `seat_id`.
3. **Client** initiates payment -> **Payment Service**.
4. **Payment Service** confirms transaction -> Publishes `PaymentSuccess` to RabbitMQ.
5. **Ticketing Service** consumes event -> Updates DB status to `CONFIRMED`.
6. **Notification Service** consumes event -> Sends SMS to User.

3.2 Flow: Live Fleet Tracking

1. **Driver App** captures GPS -> POSTs to **Tracking Service**.
2. **Tracking Service** updates `bus:{id}:geo` in Redis.
3. **Web Client** (via WebSocket) subscribes to `trip:{id}`.
4. **Tracking Service** pushes updates from Redis to WebSocket subscribers.

---

4. Client Ecosystem

The architecture supports multiple diverse clients, all consuming the same REST APIs.

1. **Public Web Portal (React):** SEO-optimized, customer-facing booking engine.
2. **Super Admin Dashboard (React/MUI):** Platform-wide analytics and tenancy management.
3. **Company Operations (Desktop .NET/Avalonia):** Cross-platform desktop app for heavy duty scheduling and fleet management (offline capable).
4. **Driver App (React Native):** Focused interface for Trip Start/Stop and QR Scanning.
5. **POS App (React Native):** High-throughput, offline-first ticket sales for agents.

---

5. External Integrations

- **Google Gemini (AI):** Used for Natural Language Processing in the Chat Assistant.
- **Payment Gateways:** Interface for Mobile Money (M-Pesa, MTN) and PayPal.
- **SMS Gateway:** Cloud provider integration for outbound messaging.

6. Deployment Model

- **Current State:** Docker Compose monolith on a single Virtual Machine (AWS/Azure).
  - Port 80/443 exposed via Nginx.
  - Services communicate on internal Docker bridge network.
- **Scalability Path:**
  - Stateful components (Postgres, Redis, RabbitMQ) moved to managed cloud services (RDS, ElastiCache).
  - Stateless services (Python containers) migrated to Kubernetes (EKS/AKS) or Serverless (Cloud Run).

7. Key Engineering Constraints & Decisions

- **Synchronization:** Offline-first clients (POS) sync via batch endpoints when connectivity is restored. Conflict resolution favors the Server.
- **Consistency:** "At least once" delivery via RabbitMQ. Idempotency keys required for all critical mutations (Payments, Booking).
- **Security:** Services do not trust each other implicitly; JWTs are passed and validated at each boundary (Zero Trust principles).