📘 Driver's Klub Backend - Production Documentation

Version: 4.0.0 (Microservices Architecture) **Date:** January 12, 2026 **Authors:** Driver's Klub Engineering Team **Status: LIVE / PRODUCTION**

📑 Table of Contents

---

1. Executive Summary

The **Driver's Klub Backend** is a mission-critical logistics platform designed to manage the end-to-end lifecycle of inter-city and intra-city electric cab services. It acts as the central nervous system connecting:

- **Fleets**: Companies or individuals owning vehicles.
- **Drivers**: The workforce operating the vehicles.
- **Customers**: End-users booking rides via mobile apps.
- **Aggregators**: External demand sources like **MakeMyTrip (MMT)** and MojoBoxx.

The system is engineered for **high availability**, **strict consistency** (ACID-compliant), and **real-time orchestration** between internal fleets and external fulfillment providers.

Key Capabilities

- **Hybrid Fulfillment**: Automatically routes bookings to internal drivers or external providers (MojoBoxx) based on availability.
- **Rapido Integration**: Automated status synchronization to manage driver availability across platforms.
- **Compliance-First**: Enforces strict constraints (T-1 Booking, KYC validation, Vehicle Fitness).
- **Authentication**: Secure **OTP-based login** (No passwords) for all user roles.
- **Dynamic Pricing**: Rule-based pricing engine supporting multipliers for Airport/Rental/Outstation trips.
- **Granular RBAC**: Role-Based Access Control for Super Admins, Ops, Managers, and Drivers.
- **Regional Enforcement**: Strict `Origin City` validation (e.g., DELHI NCR).
- **Payment System**: Complete payment & payout system with Easebuzz integration, supporting rental and payout models.
- **Cash Management**: Drivers declare cash/UPI collections daily during check-out for reconciliation.

---

2. System Architecture

The application follows a **Microservices Architecture** with 6 independent services behind an API Gateway, designed for horizontal scalability and independent deployment.

High-Level Components

```
graph TD
    Client[Mobile App / Web Dashboard] --> LB[Load Balancer]
    MMT[MakeMyTrip Server] --> LB
    LB --> API[Express.js API Gateway]

    subgraph "Core Domain Layer"
        API --> Auth[Auth Module (OTP)]
        API --> Trip[Trip Engine]
        API --> Ops[Ops Module]
    end

    subgraph "Orchestration Layer"
        Trip --> Alloc[Allocation Service]
        Alloc --> Inte[Internal Fleet]
        Alloc --> Ext[External Provider Adapter]
    end

    subgraph "Data Persistence"
        Trip --> DB[(PostgreSQL)]
        Auth --> DB
    end

    subgraph "External Integrations"
        Ext --> Mojo[MojoBoxx API]
        Ext --> MMT_API[MMT Webhooks]
        Auth --> Exotel[Exotel SMS]
    end
```

Data Flow Pattern

1. **Request Entry**: All requests hit src/app.ts and are routed via `src/modules/*`.
2. **Validation**: Joi/Zod schemas validate inputs (including City/Vehicle constraints).
3. **Service Layer**: Business logic resides in `*.service.ts` files inside modules or `src/core`.
4. **Orchestration**: `TripOrchestrator` manages the lifecycle and provider selection.
5. **Persistence**: `Prisma Client` performs ACID transactions against PostgreSQL.

Provider Lifecycle Mapping

The system normalizes external provider statuses to internal states:

Internal State	MMT Status	Internal Driver
`CONFIRMED`	`paid/confirmed`	`DRIVER_ASSIGNED`
`STARTED`	`on_trip`	`STARTED`
`COMPLETED`	`billed`	`COMPLETED`
`CANCELLED`	`cancelled`	`CANCELLED`

---

3. Technology Stack

Component	Technology	Version	Purpose
Runtime	Node.js	v18+	Event-driven Javascript Runtime
Framework	Express.js	v4.x	REST API Framework
Language	TypeScript	v5.x	Static Typing & Compliance
Database	PostgreSQL	v15+	Relational Data Store
ORM	Prisma	v5.x	Database Access & Migrations
Auth	JWT + OTP	-	Stateless Authentication
External	Exotel / MMT	-	SMS & Partner Integration

---

4. Directory Structure & Codebase Navigation

The project follows a **Feature-Based** structure:

```
src/
├── app.ts                  # Entry Point & Middleware Chain
├── server.ts               # Server startup
├── worker.ts               # Background worker for provider sync
├── core/                   # Shared Business Logic
│   ├── constraints/        # Trip validation rules
│   ├── payment/            # Payment System (Rental, Payout, Penalties, Incentives,
Virtual QR)
│   ├── pricing/            # Pricing Engine
│   ├── provider/           # Provider integrations
│   └── trip/               # Trip Orchestration & Validators
├── middlewares/            # Auth, Logging, Rate Limiting
├── modules/                # Feature Modules (Routes/Controllers)
│   ├── auth/               # OTP Logic (No Registration)
│   ├── users/              # User Management (Admin-Only Creation)
│   ├── drivers/            # Driver Profiles (Admin-Only Creation)
│   ├── fleets/             # Fleet Onboarding
│   ├── fleetManager/       # Fleet Manager Management
│   ├── vehicles/           # Asset Management
│   ├── assignments/        # Driver-Vehicle Assignments
│   ├── attendance/         # Driver Attendance & Check-in/out
│   ├── trips/              # Driver App APIs
│   ├── payment/            # Payment & Payout Endpoints (Driver & Admin)
│   ├── pricing/            # Pricing Calculator
│   ├── partner/mmt/        # MakeMyTrip Integration
│   └── webhooks/           # Easebuzz Webhooks (Payment Gateway & Virtual Accounts)
├── adapters/               # External Integrations
│   ├── easebuzz/           # Easebuzz Payment Gateway Integration
│   └── providers/          # Provider adapters (MojoBoxx, MMT)
```

```
├── shared/                 # Shared code (enums, errors)
└── utils/                  # Helpers (Logger, ApiResponse)
```

---

5. Database Schema & Data Models

Core Entities

The schema (`prisma/schema.prisma`) revolves around the **Ride** (Unified Trip) entity. A major refactor (Dec 2025) consolidated all legacy `Trip` logic into `Ride` .

1. **User**: Identity layer (Phone + Role). No public registration - created by admins only.
2. **Driver**: Profile linked to User and Fleet.
3. **Fleet**: The supply partner (Vendor).
4. **FleetManager**: Managers who oversee fleet operations.
5. **HubManager**: Managers who oversee specific fleet hubs/locations.
6. **FleetHub**: Physical hub locations for fleet operations.
7. **Vehicle**: Physical asset managed by Fleet.
8. **Ride**: The central transaction unit (formerly Trip).
    - **Fields**: `pickupLocation` , `dropLocation` , `tripType` , `status` , `price` , `distanceKm` .
    - **Geofencing**: `pickupLat` (Float), `pickupLng` (Float) added for granular location validation.
    - **Orchestration**: Linked to `RideProviderMapping` for External Providers (MMT).
9. **TripAssignment**: The specific link between a `Ride` and a `Driver` .
    - One Ride can have multiple assignments (history), but only one ACTIVE assignment.
    - Tracks `bookingAttempted` and `status` (`ASSIGNED` , `COMPLETED`).
10. **Assignment**: Daily driver-vehicle assignments (roster).
11. **Attendance**: Tracks Driver Check-In/Check-Out and Duty Status.
    - Fields: `checkInTime` , `checkOutTime` , `status` , `selfieUrl` , `odometerStart` , `odometerEnd` , `cashDeposited`
    - Statuses: `PENDING` , `APPROVED` , `REJECTED` , `CHECKED_OUT`
12. **Break**: Tracks driver breaks during active attendance.
    - Fields: `id` , `attendanceId` , `startTime` , `endTime`
    - Linked to Attendance model
13. **RideProviderMapping**: Links internal rides with external provider bookings.
14. **Otp**: OTP verification records.
15. **RefreshToken**: JWT refresh tokens.
16. **DriverPreference**: Stores active, admin-approved preferences.
    - Fields: `id` , `driverId` , `preferences` (Json), `approvedBy` , `approvedAt` , `createdAt` , `updatedAt`
17. **DriverPreferenceRequest**: Tracks all preference change requests (Approval Workflow).
    - Fields: `id` , `driverId` , `currentPreference` (Json), `requestedPreference` (Json), `status` (PENDING/APPROVED/REJECTED), `rejectionReason`
18. **PreferenceDefination**: Config-driven definition table for system-wide preferences.
    - Fields: `key` (PK), `displayName` , `description` , `category` (TRIP/PAYOUT/SHIFT), `approvalRequired` , `defaultValue` , `isActive`

---

6. Core Business Flows

A. Driver Fulfillment (Admin-Led Dispatch)

1. **Trip Creation**: Admin creates a `Ride`.
2. **Assignment**: Admin assigns the Trip to a specific Driver.
3. **Start Trip (Strict Logic)**:
   - Driver can ONLY start trip within **2.5 Hours** of pickup time.
   - Early start is blocked with `400 Bad Request`.
4. **Arrive at Pickup**:
   - **Geofence**: Must be within **500m** of `pickupLat` / `pickupLng`.
   - **Time**: Must be within **30 mins** of pickup time.
5. **Complete**: Driver completes the trip (Status: `COMPLETED`).

B. Partner Fulfillment (MMT)

1. **Search**: MMT polls `/partner/mmt/search` for inventory.
2. **Block**: MMT reserves a car (`BLOCKED` status).
3. **Confirm**: MMT confirms payment (`CREATED` status).
4. **Webhooks**: Backend pushes status updates (Driver Assigned, Started, Completed) to MMT automatically.

C. Rental Model Flow (Financial)

1. **Plan Creation**: Fleet Manager creates `RentalPlan` (e.g., Weekly, ₹3000).
2. **Plan View**: Driver views available plans via `GET /payment/rental/plans`.
3. **Subscription**: Driver initiates payment (`POST /payment/rental`).
4. **Payment**: Driver pays via Easebuzz PG.
5. **Activation**:
   - System validates payment.
   - System activates `DriverRental` record.
   - Driver status updates to `RENTAL`.

D. Payout Model Flow (Financial)

1. **Collection**: Driver collects cash/QR payments daily.
2. **Reconciliation**: Admin reconciles `DailyCollection` (Cash vs QR).
3. **Calculation**: External Excel sheet calculation for net earnings.
4. **Disbursement**: Admin uploads CSV (`POST /bulk-payout`).
5. **Execution**: System triggers Easebuzz Wire Transfer (IMPS) to driver's bank account.

E. InstaCollect Orders (Ad-Hoc Payments)

1. **Order Creation**: Admin creates a `PaymentOrder` (e.g., for ad-hoc invoices or bulk payments).
2. **QR Generation**: System generates a unique Dynamic QR (Easebuzz Virtual Account) for the order.
3. **Payment**: Customer scans and pays (supports partial payments).
4. **Reconciliation**:
   - Webhook differentiates payment type ('ORDER' vs 'VEHICLE').
   - System updates `collectedAmount` and `remainingAmount`.
   - Status transitions: `PENDING` -> `PARTIAL` -> `COMPLETED`.
5. **Ledger**: All payments are recorded as `Transaction` records linked to the `PaymentOrder`.

F. Attendance & Duty Management

1. **Check-In ("On Duty")**:
    - Driver captures **Selfie** and **Odometer Start**.
    - Status: `PENDING` (Manager Approval Required).
    - Validation: Cannot check in if already checked in or on duplicate device.
2. **Duty Operations**:
    - Driver accepts/completes trips.
    - Can take **Breaks** (stops receiving trips).
3. **Check-Out ("Off Duty")**:
    - Driver enters **Odometer End**.
    - **Cash Declaration**: Driver explicitly declares `cashDeposited` (Total Cash + Personal UPI collection).
    - Status: `CHECKED_OUT`.
    - System Link: Automatically marks driver OFFLINE on Rapido.

G. Rapido Status Synchronization

1. **Trigger**: Occurs on Login, Trip Completion, internal Break Start/End, and every 5 minutes (Worker).
2. **Logic Check**:
    - **Go Offline**: If on Internal Trip, on Break, or upcoming assignment in < 45 mins.
    - **Go Offline**: If **NOT Checked In** (Strict Policy).
    - **Go Online**: If Idle and no upcoming conflicts.
3. **Execution**: System sends status change request to Rapido API.
4. **Edge Case Handling**:
    - **External Conflicts**: Drivers active on MMT/Internal are forced OFFLINE on Rapido.
    - **Manual Override**: Webhooks detect if a driver manually forces ONLINE and reverts it if rules are violated.
    - **Retry Queue**: API failures are queued in DB and retried by a background worker.

---

7. API Reference (Canonical)

The **Single Source of Truth** for all API endpoints, request/response schemas, and error codes is the **API_REFERENCE.md** file.

🔗 Complete API Reference

Team Specific Guides

- **Flutter Driver App Guide**
- **React Admin Dashboard Guide**

Scope of Contract:

1. **Authentication**: OTP-based flows.
2. **Trip Module**: Driver App interactions.
3. **Admin Ops**: Fleet/Vehicle management.
4. **Partner**: MMT Integration specs.

---

8. Setup, Testing & Operations

Local Development

1. `npm install`
2. `npx prisma generate`
3. `npm run dev` (API Gateway runs on Port 3000, services on 3001-3006)

Testing

- **Unit Tests**: `npm test`
- **API Tests**: `npm run test:api` (Runs Dredd/Supertest against endpoints)

Deployment

- **Platform**: Render / AWS
- **Build**: `npm run build` -> `dist/`
- **Process Manager**: PM2 or Docker Container.

8.4 Environment Configuration

Key variables for production:

```
RAPIDO_PRE_TRIP_BUFFER_MINUTES=45
WORKER_ENABLED=true
WORKER_SYNC_INTERVAL_MS=300000
```

---

9. Production Readiness & Validation (Dec 2025)

✅ Stability & Integrity Checks

- ☑ **Trip Assignment Transaction**: `TripAssignmentService` now atomically updates `Ride` status to `DRIVER_ASSIGNED` within a transaction.
- ☑ **Error Handling**: `AdminTripController` hardened with `try/catch` blocks to prevent unhandled rejections during Assignment/Reassignment.
- ☑ **Concurrency**: Database transactions ensure no double-booking of drivers or trips.

✅ End-to-End Verification

The system has passed the **Full Flow Test Protocol** (`npm run test:full`):

1. **Auth**: Admin & Driver Login (OTP Bypass).
2. **Attendance**: Driver Check-in/out + Admin Approval.
3. **Dispatch**: Admin Create -> Assign -> MMT Webhook Trigger.
4. **Execution**: Driver Start -> MMT Webhook -> Complete -> MMT Webhook.

✅ Partner Integration (MMT)

- **Inbound**: Fully mapped to `MMTController` (Search, Block, Confirm, Cancel, **Reschedule Block/Confirm**). **Secured via Basic Auth**.

- **Outbound**: All hooks (Assignment, Start, Arrive, Complete, Cancel, **Location Update**) implemented.

End of Document