MASTER FUNCTIONAL DESIGN DOCUMENT: Wheels On Go

Project Name: Wheels On Go, Valet Platform

Target Platform: Android (Mobile), Web (Admin)

Revision: 1.0 (Final)

Date: March 21, 2026

---

1. Project Overview

Wheels On Go is an on-demand valet driver service designed to provide car owners in Metro Manila with safe, professional, and trackable driver services. The platform's differentiator is its **Safety-First AI Suite**, which monitors driver fatigue and enforces vehicle inspections.

1.1 Project Timeline

- **Phase 1:** Design, Architecture, and Onboarding (Weeks 1-3)
- **Phase 2:** Core Booking Engine and AI Integration (Weeks 4-6)
- **Phase 3:** Admin Dashboard, QA, and Deployment (Weeks 7-9)

---

2. Architecture Solution Design

2.1 Key Design Decisions

- Modular Monolith (NestJS): The backend is structured into domain-driven modules (e.g., AuthModule, BookingModule, SafetyModule). This allows for clean separation of the BLOWBAGETS logic from the financial transactions.
- Type-Safe Data Access (Prisma): We use Prisma ORM to ensure type safety between the PostgreSQL database and the NestJS controllers. This prevents runtime errors during the high-stakes booking and payment flows.
- AI Integration (Google Gemini): The "Advanced Safety Suite" leverages Google Gemini API for analyzing driver behavior logs and providing real-time feedback on fatigue patterns based on the ai_incidents table.
- Stateless Execution: The system is hosted on Render as a stateless service. All session state is stored in PostgreSQL and JWTs, allowing the service to restart or scale without losing active booking data.

2.2 Scalability and Performance Considerations

- Efficient Querying: Prisma's fluent API is optimized with select statements to ensure that the mobile apps only fetch necessary fields, reducing data usage for drivers on limited mobile plans.
- Managed Postgres Performance: Utilizing a managed instance on Railway and Render ensures automated backups and vertical scaling for the tracking_logs table, which will grow the fastest.
- Google Maps Optimization: To manage costs, the system implements debouncing on GPS updates and uses the Google Maps Geocoding API results sparingly, caching frequently used pickup/drop-off coordinates.

2.3 Usability and User Interface Considerations

- Real-time Feedback: NestJS WebSockets (Gateway) provide users with a "live" feel during driver dispatching, reflecting the map-based view designed in Figma.
- Schema-Driven Forms: The BLOWBAGETS checklist in the Driver App is validated against the Prisma schema to ensure no safety data is missing before the trip state updates to "In Progress."

---

3. Functional Module Specifications

3.1 Onboarding & Verification

- **User/Driver Auth:** OTP-based registration via mobile number.
- **Driver KYC:** Document upload (License, NBI Clearance) with Admin approval workflow.

3.2 Booking & Dispatch Engine

- **Smart Matching:** Proximity-based broadcast to the 5 nearest drivers within a 3km - 5km radius.
- **Pricing:** Static distance-based fare calculation (
  $$Fare = Base + (KM \times Rate)$$
).

3.3 Real-Time Logistics

- **GPS Tracking:** Live turn-by-turn navigation via Google Maps API/Free Service
- **Geofencing:** Automatic status updates when the driver enters the pickup or drop-off radius.

3.4 Advanced Safety & Intelligence (AI Suite)

- **SOS Protocol (optional):** Sub-second latency emergency alerts triggering SMS, Push, and Admin notifications.

- **BLOWBAGETS Checklist:** Digital pre-trip inspection requiring photo proof of vehicle condition.

3.5 Financial & Notifications

- **Cashless/Cash Payments:** Static GCash and Credit/Debit card gateways, Cash.
- **Payout Logic:** Automated 70% Driver / 30% Platform split.
- **Push Notifications:** Event-driven alerts for booking status and safety warnings.

---

4. Operational Logic & Workflows

4.1 Trip Lifecycle (Sequence)

1. **Request:** User sets destination.
2. **Match:** Driver accepts and navigates to pickup.
3. **Check:** Driver completes BLOWBAGETS/Fatigue Detection/Breathalyzer and owner signs off.
4. **Drive:** Live GPS visible to the owner.
5. **Pay:** Automatic capture of funds and two-way rating.

---

5. Security Requirements

5.1 Built-in Security

- **Standardized Auth:** Implementation of **JWT** via `@nestjs/jwt`. For enhanced security, we will use a **short-lived Access Token** and a **long-lived Refresh Token** stored securely in the app's encrypted storage.
- **Prisma Middleware for Privacy:** We utilize Prisma middleware to automatically mask PII (like the middle part of a phone number) when records are fetched by non-admin roles.
- **Environment Security:** All **Production Keys** (Google Maps, GCash, Gemini) are managed via Railway/Render **Environment Variables**, ensuring no sensitive keys are hardcoded in the GitHub repository.

5.2 Custom Security Roles

5.2.1 Role: Valet Driver (NestJS Guard: `Role.Driver`)

- **Requirement:** Access to navigation and safety checklists.
- **Prerigatives:** * `POST /safety-check`: To submit BLOWBAGETS.
    - `PATCH /bookings/{id}`: To update status to "Arrived" or "Completed."
    - `GET /earnings`: To view personal financial summaries.

5.2.2 Role: Car Owner (NestJS Guard: `Role.User`)

- **Requirement:** Booking and tracking services.
- **Privileges:**
    - `POST /bookings`: To initiate a "Find a Driver" request.
    - `GET /tracking/{driverId}`: To view real-time location via Google Maps integration.

5.2.3 Role: System Administrator (NestJS Guard: `Role.Admin`)

- **Requirement:** Full oversight via the Admin Portal.
- **Privileges:**
    - `GET /admin/analytics`: Powered by Gemini to summarize monthly `ai_incidents`.
    - `PATCH /drivers/{id}/verify`: To move a driver from "Pending" to "Active" after document review.
