Excellent question! It's good to keep the finish line in sight.

Based on our defined MVP scope and the progress we've made, here's a bullet-point list of the **remaining backend MVP tasks** after we (hypothetically) complete the "Delete Car" and "Set Default Car" features:

* **Car Service (for Drivers) - Finalizing:**
  + Implement "Set Default Car" API. (This is the one after "Delete Car")
* **Trip Service (Core Functionality):**
  + **Data Structures (Models & Schemas):**
    - Trip model (driver\_id, car\_id, from, to, departure\_time, price\_per\_seat, total\_seats, available\_seats, status, etc.)
    - Booking model (trip\_id, passenger\_id, seats\_booked, status, booking\_time, etc.)
    - Associated Pydantic schemas for Trip creation, Trip response, Booking creation, Booking response.
  + **CRUD Operations (for Trips and Bookings):**
    - Create Trip (Driver).
    - Get Trip by ID.
    - Search/List Trips (for Passengers, with filters like from/to/date/seats).
    - Update Trip (e.g., Driver might update departure time if no bookings, or cancel).
    - Create Booking (Passenger books seats on a Trip).
    - Get Booking by ID / Get User's Bookings.
    - Update/Cancel Booking (Passenger/Driver, with rules).
    - Function to update available\_seats on a Trip when a booking is made/cancelled.
  + **API Endpoints (Trip Management for Drivers):**
    - POST /trips (Driver creates a new trip).
    - GET /drivers/me/trips (Driver views their own created trips).
    - GET /trips/{trip\_id} (Driver views details of their specific trip).
    - PATCH /trips/{trip\_id} (Driver updates their trip - limited scope for MVP).
    - DELETE /trips/{trip\_id} (Driver cancels their trip - with rules if bookings exist).
  + **API Endpoints (Trip Interaction for Passengers):**
    - GET /trips/search (Passenger searches for available trips).
    - GET /trips/{trip\_id} (Passenger views details of a specific trip - can be same as driver's view endpoint).
    - POST /bookings (Passenger books a trip).
    - GET /passengers/me/bookings (Passenger views their bookings).
    - DELETE /bookings/{booking\_id} (Passenger cancels their booking - with rules).
* **Admin Panel Functionality (Core for MVP):**
  + **Driver & Car Verification APIs (for Admin role):**
    - GET /admin/verifications/drivers (Admin gets list of drivers pending verification).
    - GET /admin/verifications/cars (Admin gets list of cars pending verification).
    - POST /admin/verifications/drivers/{driver\_id}/approve (Admin approves a driver).
    - POST /admin/verifications/drivers/{driver\_id}/reject (Admin rejects a driver).
    - POST /admin/verifications/cars/{car\_id}/approve (Admin approves a car).
    - POST /admin/verifications/cars/{car\_id}/reject (Admin rejects a car).
  + **Admin User Creation/Management:** (How will the first admin be created? Potentially a CLI command or manual DB insertion for MVP).
  + **Basic Trip Monitoring API (for Admin role - optional MVP):**
    - GET /admin/trips (Admin views all trips).
* **Database Setup (Finalizing):**
  + Generate and apply all Alembic migrations for the complete schema.
* **Deployment & Configuration (Basic for MVP):**
  + Containerize the application (Dockerfile).
  + Basic docker-compose.yml for local development with PostgreSQL.
  + Configuration for environment variables (DATABASE\_URL, JWT\_SECRET\_KEY).
* **Testing (Ongoing, but more focused towards the end of MVP):**
  + Unit tests for critical CRUD functions and utilities.
  + Integration tests for API endpoints.

This list seems extensive, but many of the Trip Service and Admin Panel APIs will follow similar patterns to what we've already established for Users and Cars (CRUD, router, schemas, dependencies). The complexity will be in the business logic (e.g., booking rules, seat availability).

So, after "Set Default Car", the **Trip Service** is the largest remaining block of work for the backend MVP.