CAReLESS

Problem Statement

We have to organize lifts for people from their houses to church. Communication is a problem.

Driver Problems

- Who must the driver pick up?
- Where?
- When?

Passenger problems

- Who fetches the passenger?

- Where?
- When?

Current Solution:

- Send poll on whatsapp on a group to see who requires a lift
- Send messages on driver group
- Then manually arrange each passenger to a driver
- Send arrangements to passengers on the group

Problem with Current Solution

- Wait for passengers to answer poll
- Wait for driver to answer important questions:
    - Where
    - When
    - How many seats are open
- Wait for Admin to allocate passengers to drivers and to determine pickup spots and time
- Late Passengers answering polls (after the allocation of passengers to drivers)
- Dependent on whatsapp group
- Takes lot of time to arrange passenger-driver assignment

Summary of Problem Statement:

Slow Communication and redundant messaging

Why should we solve this problem?

- CV
- Scalable
- Convenient (if implement correctly considering lazy users)

Scope

Gather passenger and driver (+-) static personal information.

Choose a medium to gather data.

Calculate pickup points, driver-passenger mapping, optimize driver route.

Generate messages to send to passengers (personally or groups) and drivers.

Ensure user friendly convenient experience.

Walkthrough 1

The new passenger wants a lift.

The admin sends a link through Whatsapp to them.

The passenger registers on the link and joins a group

When Someone makes an event, a poll will be sent to the group, to confirm who will go to the event.

Another poll will be sent to confirm which drivers will be available.

Pull driver information to calculate routes and pickup points. Admin can override.

Generate a text message to inform everyone of arrangements.

ER Diagram

Users → Drivers & Passengers

Events

Address (Address of users)

Cars (Perhaps different cars per users)

Groups

Entities

User

- Fields:
    - `id` (Primary Key)
    - `name`
    - `phone_number`

    - `Address_ID (FK)`
    - `created_at`
    - `updated_at`
    - (optional) `profile_picture`
- Notes: A user can be a driver, a passenger, or an admin. An admin user ensures the allocation process works smoothly and the drivers and passengers are notified on time.

(thus overlapping and complete supertype)

Driver (Subtype of User)

- Fields:
    - `id` (Primary Key)

Passenger (Subtype of User)

- Fields:
    - `id` (Primary Key)

Group

- Fields:
    - `id` (Primary Key)
    - `name`
    - `description`
    - (optional) `picture`
    - `created_at`
    - `updated_at`

Event

- Fields:
    - `id` (Primary Key)

    - `name`
    - `description`
    - `date`
    - `start_time`
    - `end_time`
    - `is_recurring`
    - `Address_ID` (FK)
    - `Group_ID` (FK)

Address

- Fields:
    - `id` (Primary Key)
    - `street`
    - `city`
    - `Province`
    - `postal_code`
    - `country`
    - (optional) `latitude`, `longitude`
- Notes:
    - Addresses should be normalized and reusable.
    - Use google maps to determine location.
    - If User is unsure about address open google maps (for current address)

Car

- Fields:

- `id` (Primary Key)
- `driver_id` (Foreign Key to Driver)
- `make`
- `model`
- `year`
- `license_plate`
- `seat_capacity`

- Notes: A driver can have multiple cars.

Passenger_Allocation (Composite entity for Passenger-Driver-Event relationship)

- Fields:
  - `driver_booking_id` (**FK1**, **PK**)
  - `passenger_id` (**FK2**, **PK**)
  - `pickup_id` (**FK3)**
  - `is returning`

Address_List

- Fields:
  - `User_ID` (**FK1**, **PK**)
  - `Address_ID` (**FK2**, **PK**)
  - `Rank` (Ranks preferred address)

Driver_Booking (Drivers in Event)

- Fields:
  - `id` (**PK**)
  - `driver_id` (**FK1**)
  - `event_id` (**FK2**)

- `car_id` (Foreign Key to Car)
- `available_seats`
- `is_returning`
- (optional) `notes` (e.g., "no pets", "leaving at 5 PM sharp")

- Notes:
  - One RideOffer per driver per event per car.
  - **driver_id and event_id** must have a unique constraint

PickupPoint (Point and time for Drivers in Event)

- Fields:
  - `id` (Primary Key)
  - `address_id`
  - `time`
  - `passenger_count`

TO DO

- **Look at DBMS**
  - How does an online database work?
  - What is the cheapest (MySQL, Postgres, Azure)?

- **Git**
  - Create a repo

- **Application software**
  - Coding language (not that important)
  - What GUI program will we use
  - Will whatsapp bot work?
  - Online App or local?

Skeleton Structure

1. I hosted my MySQL database on TiDB Cloud for free

2. I used an ORM (Prism) to interface with the DB
3. I am using Next.js (React) to build the frontend and javascript (Express.js) for the backend

Core functionality

Entity	CRUD Operations	Custom/Business Logic Functionalities
User	Create User (Registration), Read User (Profile view), Update User (Edit details), Delete User (Account deactivation)	**Authentication/Authorization:** Login, Logout, Change Password, Determine User Role (Driver, Passenger, Both). **Communication:** Send personal notification/message.
Group	Create Group (Privatize community), Read Group Update Group (Edit details), Delete Group (Remove group for user or delete it entirely)	**Create group:** Send an invite link or ask members to join via app. **Events by groups:** View upcoming events by groups.
Driver	Create Driver (Link to User), Read Driver (View Driver profile), Update Driver (Edit status, e.g., "active"), Delete Driver (Remove Driver status)	**Availability:** Set/Change Driver status for an Event (via `Driver_Booking`). **Car Management:** Link/unlink available `Car` records.
Passenger	Create Passenger (Link to User), Read Passenger (View Passenger profile), Update Passenger, Delete Passenger (Remove Passenger status)	**Request Lift:** Signal intent to attend an Event (managed by subsequent allocation).

Event	Create Event (Admin), Read Event (View details), Update Event (Change time/location), Delete Event (Cancel)	**Event Management:** Send polls/notifications to Users about the Event. Close/Open lift sign-ups. **Reporting:** View list of signed-up Drivers and Passengers.
Address	Create Address (New location), Read Address, Update Address (Edit street name), Delete Address (If no longer linked to a User or Event)	**Geocoding:** Convert street address to `latitude, longitude` (via external API). **Validation:** Verify address format and existence.
Car	Create Car (Add new vehicle), Read Car, Update Car (Change capacity, license plate), Delete Car	**Capacity Check:** Verify that a car's `seat_capacity` is correctly used when creating a `Driver_Booking`.
Address_List	Create (Add preferred address), Read (View user's preferred addresses), Update (Change address or `Rank`), Delete (Remove address preference)	**Preference Management:** Sort/filter addresses based on `Rank` for default pickup point selection.
Driver_Booking	Create (Driver offers a ride for an Event), Read (View all ride offers for an Event), Update (Change `available_seats`, `car_id`, or notes), Delete (Driver cancels offer)	**Seat Management:** Automatically decrement/increment `available_seats` when a `Passenger_Allocation` is created/deleted. **Unique Constraint Check:** Enforce that a driver can only have one booking per event.

Passenger_Allocation	Create (Admin/System assigns passenger to driver), Read (View individual assignments), Update (Re-assign passenger to a different driver/pickup), Delete (Passenger cancels lift or is de-allocated)	**Route Optimization Trigger:** When a new allocation is created, potentially trigger a recalculation of the driver's route and pickup times. **Communication:** Send allocation details to passenger (Who, Where, When).
Pickup	Create (System suggests a pickup point), Read (View pickup point details), Update (Admin overrides time/address), Delete (System removes unused pickup point)	**Route Generation:** Link to `Driver_Booking` and `Passenger_Allocation` to calculate the optimal route for the driver, determining `address_id` and `time`. **Count Management:** Automatically update `passenger_count` based on linked `Passenger_Allocation` records.

App Workflow and Screens

Based on the core functionalities and entities, the application workflow can be broken down into three main user experiences: **Passenger**, **Driver**, and **Admin**.

1. Passenger Workflow

The passenger primarily focuses on registering, setting their address, and requesting/confirming lifts for events.

Screen/Step	Goal	Key Interactions/Functionalities Used
Registration/Login	Initial access to the system.	*User*: Create User, Read User, Authentication.
Profile Setup	Ensure the system knows where to fetch the passenger.	*User*: Update User (name, phone). *Address_List*: Create, Read, Update, Delete (set preferred/home address).
Event List	See upcoming events requiring lifts.	*Event*: Read Event.
Event Details/Lift Request	Indicate desire to attend and request a lift.	*Passenger*: Request Lift (internally triggers a pending allocation).
Lift Confirmation	View and confirm the allocated ride details.	*Passenger_Allocation*: Read (Who, Where, When). *User*: Communication (Receive allocation message/notification).
My Lifts	View current and past scheduled rides.	*Passenger_Allocation*: Read. *Driver_Booking*: Read (view assigned driver's car/notes).
Cancel Lift	Cancel the request or confirmed ride.	*Passenger_Allocation*: Delete (Re-allocates seat).

2. Driver Workflow

The driver primarily focuses on registering their vehicle(s) and declaring their availability for specific events.

Screen/Step	**Goal**	**Key Interactions/Functionalities Used**
Registration/Login	Initial access.	*User*: Authentication. *Driver*: Create Driver (if not already set).
Profile Setup	Ensure the system knows where to fetch the passenger.	*User*: Update User (name, phone). *Address_List*: Create, Read, Update, Delete (set preferred/home address).
Car Management	Add or update vehicles and their capacities.	*Car*: Create Car, Read Car, Update Car, Delete Car.
Event List/Availability	See upcoming events and declare availability to drive.	*Event*: Read Event. *Driver*: Set/Change Driver status.
Offer Ride	Specify vehicle and available seats for an event.	*Driver_Booking*: Create, Update (Set car_id, available_seats, is_returning). *Car*: Capacity Check.
My Event Bookings	View rides offered and their current passenger allocations.	*Driver_Booking*: Read. *Passenger_Allocation*: Read (View assigned passengers). *Pickup*: Read (View pickup points and times).
Route/Pickup View	See the optimized route and planned pickup sequence.	*Pickup*: Route Generation.
Cancel Ride Offer	Withdraw the offer to drive for an event.	*Driver_Booking*: Delete (Triggers re-allocation of passengers).

3. Admin Workflow (Management)

The admin oversees event creation, manages users, and handles the core task of passenger-driver assignment and optimization.

Screen/Step	Goal	Key Interactions/Functionalities Used
Admin Dashboard	Overview of current events, open requests, and system status.	*Event*: Read. Summarized data from Driver_Booking and Passenger_Allocation.
Event Management	Create, modify, or cancel events.	*Event*: Create Event, Update Event, Delete Event, Event Management (Open/Close sign-ups).
Allocation Console	Core screen for matching passengers and drivers manually.	*Driver_Booking*: Read (availability). *Passenger*: Request Lift (list of unassigned passengers).
Route Calculation & Assignment	The process where the system suggests the best match.	*Passenger_Allocation*: Create, Update, Delete. *Pickup*: Create (System suggests), Update (Admin override). *Passenger_Allocation*: Route Optimization Trigger.
Communication/Notification Center	Send out final arrangements.	*User*: Communication (Send notifications/messages to passengers and drivers).

Data Flow Summary

1. **Event Creation:** Admin creates **Event**.
2. **Driver Offer:** Driver creates **Car** and then **Driver_Booking** for the **Event**.
3. **Passenger Request:** Passenger signals interest (Implicit **Passenger_Allocation** pending status).
4. **Admin/System Allocation:** The system calculates optimal **Pickup** points and creates/updates **Passenger_Allocation** records linking the passenger, driver, and pickup point.
5. **Notification:** System sends arrangement details (**User**: Communication).

Screens to Build Checklist (Suggested Development Order)

This checklist organizes the required screens based on a logical progression for building the core application, starting with foundational user and data management.

1. Foundational Data & Core User (Admin Focus)

- [x] **Registration/Login Screen:** For all User types (Passenger/Driver/Admin). (*Interactions: User: Create/Read, Authentication*)
- [ ] **Admin Dashboard:** Initial overview of system status and quick access to management tools. Almost like a (*Interactions: Event: Read, Summarized data*)
- [ ] **Event Management Screen:** For a group admin to create, update, and manage events. (*Interactions: Event: CRUD, Event Management*).
- [ ] **Address Management Screen (Admin/User):** To manage and geocode reusable Address entities. (*Interactions: Address: CRUD, Geocoding, Validation*)
- [ ] **Group Dashboard(Any User):** Create, join, or leave a group. Update group details if you are the admin of a group.

2. Driver Setup & Core Functionality

- [ ] **Profile Setup Screen (Driver/Passenger):** Where users set their personal details and preferred addresses. (*Interactions: User: Update, Address_List: CRUD*)
- [ ] **Car Management Screen:** For Drivers to add and manage their vehicles and capacities. (*Interactions: Car: CRUD*)
- [ ] **Driver Event List/Availability Screen:** For Drivers to see events and set their availability. (*Interactions: Event: Read, Driver: Set Status*)
- [ ] **Offer Ride Screen:** Where the Driver specifies the Car, available seats, and if they are returning for an Event. (*Interactions: Driver_Booking: Create/Update, Car: Capacity Check*)

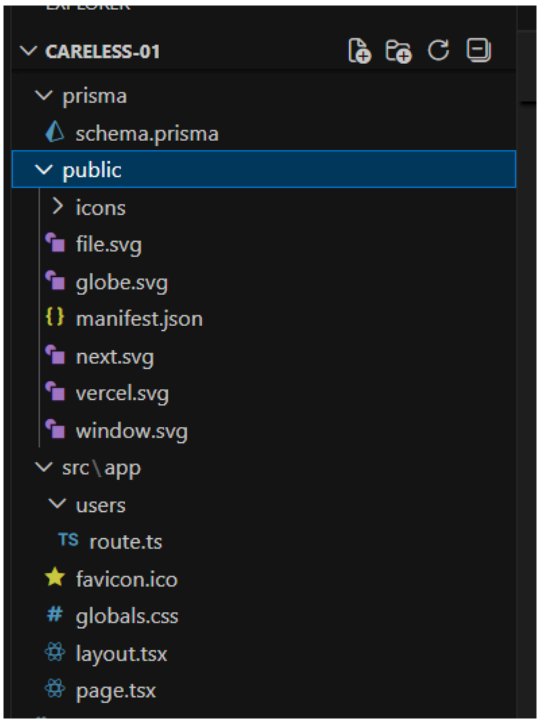
3. Passenger Core Functionality

- [ ] **Event List Screen (Passenger):** View upcoming events. (*Interactions: Event: Read*)
- [ ] **Event Details/Lift Request Screen:** Passenger indicates they want a lift for an event. (*Interactions: Passenger: Request Lift*)
- [ ] **My Lifts Screen:** View current and past scheduled rides. (*Interactions: Passenger_Allocation: Read, Driver_Booking: Read*)
- [ ] **Cancel Lift Screen:** Passenger can cancel a request or confirmed ride. (*Interactions: Passenger_Allocation: Delete*)

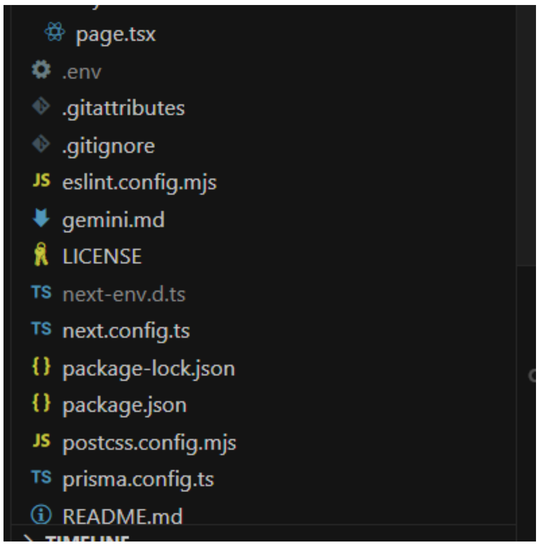
4. Allocation, Optimization, and Confirmation (Admin/System Focus)

- ☐ **Allocation Console Screen (Admin):** Core screen for manual or automated matching of unassigned passengers to available drivers. (*Interactions: Driver_Booking: Read, Passenger: Request Lift (unassigned list)*)
- ☐ **Route Calculation & Assignment Screen (Admin):** Where the Admin/System triggers and reviews route optimization and finalizes assignments, including pickup points/times. (*Interactions: Passenger_Allocation: CRUD, Pickup: Create/Update*)
- ☐ **Lift Confirmation Screen (Passenger):** Passenger views and confirms the allocated ride details (Who, Where, When). (*Interactions: Passenger_Allocation: Read, User: Communication*)
- ☐ **My Event Bookings Screen (Driver):** View offered rides, assigned passengers, and planned pickups/routes. (*Interactions: Driver_Booking: Read, Passenger_Allocation: Read, Pickup: Read*)
- ☐ **Route/Pickup View Screen (Driver):** Detailed view of the optimized pickup sequence for a driver's specific booking. (*Interactions: Pickup: Route Generation*)
- ☐ **Communication/Notification Center (Admin):** Tool for sending out final arrangement messages to users. (*Interactions: User: Communication*)

Current Repo Folder Structure



```
CARELESS-01
  prisma
    schema.prisma
  public
    icons
    file.svg
    globe.svg
    manifest.json
    next.svg
    vercel.svg
    window.svg
  src \ app
    users
      route.ts
    favicon.ico
    globals.css
    layout.tsx
    page.tsx
```



```
    page.tsx
  .env
  .gitattributes
  .gitignore
  eslint.config.mjs
  gemini.md
  LICENSE
  next-env.d.ts
  next.config.ts
  package-lock.json
  package.json
  postcss.config.mjs
  prisma.config.ts
  README.md
```

Proposed Folder Structure

```
├── .github/
├── node_modules/
├── package.json              # Monorepo root dependencies
├── tsconfig.json             # Root TypeScript configuration
│
├── /apps/
│   └── /web/                 # The Next.js (React) Frontend application
│       ├── node_modules/
│       ├── public/           # Next.js static assets
│       ├── src/              # <--- YOUR PROPOSED STRUCTURE IS HERE
│       │   ├── app/          # Application entry point/routing
│       │   ├── assets/       # Static files, images, fonts
│       │   ├── components/   # Shared UI components
│       │   ├── config/       # Global configurations
│       │   ├── features/     # Feature-based modules (e.g., /features/user, /features/events)
│       │   ├── hooks/        # Shared custom hooks
│       │   ├── lib/          # Reusable libraries/config (e.g., API client setup)
│       │   ├── stores/       # Global state management (e.g., Zustand, Redux)
│       │   ├── testing/      # Test utilities and mocks
│       │   ├── types/        # Local frontend types (shared types are in /packages/types)
│       │   └── utils/        # Shared utility functions
│       ├── package.json
│       └── next.config.js
│
│   └── /api/                 # The Dedicated TypeScript Backend service (e.g., Express/Koa)
│       ├── node_modules/
│       ├── src/
│       │   ├── controllers/  # Handlers for routes
│       │   ├── middleware/   # Authentication, logging, etc.
│       │   ├── models/       # If not using Prisma directly, or for complex logic
│       │   └── routes/       # Express/Koa route definitions
│       ├── package.json
```

```
|       └── tsconfig.json
|
└── /packages/
    ├── /database/            # Contains Prisma schema, migrations, and client
    |    ├── prisma/
    |    └── src/
    |
    └── /types/               # Shared TypeScript types for data models/entities
        └── src/
```

Key Changes and Why They Work

1. **Frontend Organization (`/apps/web/src`)**: Your structure is incorporated directly into the `src` folder of the `web` application. This is a best practice for clean, feature-focused React development.
2. **Backend Separation (`/apps/api`)**: Since you specified **TypeScript for the backend**, it's better to isolate it into its own application/service (`/apps/api`). This allows it to run independently (e.g., on a dedicated server), use a different framework (like Express or NestJS), and have its own build configuration, completely decoupled from the Next.js frontend.
3. **Monorepo Shared Packages (`/packages/types`, `/packages/database`)**: Keeping the database ORM (Prisma) and the core data model types in separate packages ensures that both the `/apps/web/` (if it needs direct database access or just types) and the `/apps/api/` services can share the same source of truth for your data layer.
4. **Why the web subdirectory?** I can now make any subdirectory, one for mobile, one for pc to build multiple interfaces for the api and database. This will make maintenance easier later.

Sidebar app Navigation

Menu Item	Description	Access Role
--- General Setup ---		
Profile Setup	Manage user personal details and preferences (Address, Contact).	All

Menu Item	Description	Access Role
Groups	View joined groups, create new ones, or join via ID.	All
--- Passenger Core ---		
Event List	View upcoming events and request lifts.	Passenger, Driver
My Lifts	View current, past, and pending scheduled rides.	Passenger
Lift Confirmation	Final confirmation details for a booked lift.	Passenger
--- Driver Core ---		
Car Management	Register and manage vehicles, capacity, and details.	Driver
Driver Event List	View events and declare availability to drive.	Driver
Offer Ride	Specific flow for a driver to offer a lift for an event.	Driver
My Event Bookings	Dashboard for drivers to see confirmed passenger allocations.	Driver
Route View	Visual map and optimized route details for a driver.	Driver
--- Admin Management ---		
Admin Dashboard	System overview, key statistics, and management entry points.	Admin Only
Event Management	Create, update, and manage church events and event sign-ups.	Admin Only

Menu Item	Description	Access Role
Address Management	Manage reusable addresses (Geocoding/Validation tool).	Admin, Driver
Allocation Console	Interface for manual/auto assignment and optimization of lifts.	Admin Only
Communication Center	Messaging interface for coordination and sending arrangements.	Admin, (Future: All)
--- Hidden/Secondary Screens ---		
Event Details	Detailed view of a specific event (Likely accessed from Event List).	All

Features for the future:

- Chatroom for the group
- Payment system
- Add events to a person’s Google calendar (will require their email address).
- Group suggestions (AI suggest people who can be in a group, if users are willing to share travelling plans)