Peer-to-Peer Campus Ride Sharing API (Layered Go Backend, gRPC, Layered Architecture)

1. Project Overview

1.1 Purpose

Build a modular, production-quality backend API for a campus-only ride sharing platform, strictly using layered architecture to separate HTTP/gRPC handlers, business services, repositories (data access), dependency injection (DI), and database entities (GORM structs). This document is modeled on the attached address book SRS, extended for a more scalable and maintainable design.

1.2 Scope

- All API endpoints are exposed as gRPC (or REST if required).
- Multi-layered backend: Handler (API), Service (business logic), Repository (DB access), DI (dependency wiring), and DB (GORM models).
- Supports university-only registration/auth, geo-based matching, chat, reviews, and real-time features.
- Fully documented to be consumed by frontend or mobile teams independently.

2. Functional Requirements

2.1 Core Features

Feature	Handler Layer	Service Layer	Repository Layer	DB Layer (GORM Structs)
Authentication	AuthHandler (gRPC)	AuthService	UserRepository	User
Profile management	UserHandler	UserService	UserRepository	User
Ride offers/requests	RideHandler	RideService	RideRepository	RideOffer, RideRequest
Matchmaking	MatchHandler	MatchService	MatchRepository	Match
Messaging/Chat	ChatHandler	ChatService	ChatRepository	ChatMessage
Live Location	LocationHandler	LocationService	LocationRepository	UserLocation
Reviews/Rating	ReviewHandler	ReviewService	ReviewRepository	Review

3. Layered Architecture

3.1 Handler Layer

- Accepts and validates API/gRPC/REST requests.
- Maps request/response to standard Protobuf/JSON and delegates to the service layer.
- Handles authentication/authorization by extracting JWT/session info.

```
type RideHandler struct {
    RideService ride.Service // via DI
}
func (h *RideHandler) CreateOffer(ctx context.Context, req *pb.RideOfferRequest) (*pb.Rid
```

3.2 Service Layer

- Business logic and validation: all transactional logic, permission checks, aggregation, and orchestration.
- Calls repository methods and performs required computations.

```
type RideService struct {
    RideRepo  ride.Repository
    UserRepo  user.Repository
}
func (s *RideService) CreateOffer(ctx context.Context, params CreateOfferParams) (*RideOf
```

3.3 Repository Layer

- Data access only. Uses GORM for PostgreSQL or MySQL, and MongoDB/Redis drivers for auxiliary data (e.g., chat, locations).
- Implements low-level CRUD and geohash-based search using efficient queries.

```
type RideRepository interface {
    CreateRideOffer(ctx context.Context, offer *RideOffer) error
    ListNearbyOffers(ctx context.Context, geohashPrefix string, radius int) ([]RideOffer,
}
```

3.4 Dependency Injection (DI)

- Composes service and handler dependencies, typically via google/wire or fx.
- Ensures testability (mockable layers), separation of lifecycle concerns.

```
func InitRideHandler(db *gorm.DB, ...) RideHandler {
    rideRepo := NewRideRepository(db)
    rideService := NewRideService(rideRepo, ...)
    return NewRideHandler(rideService)
}
```

3.5 DB Layer (GORM Structs)

- Defines persistent entities with table mappings, indices, and constraints.

```
type User struct {
    ID        string `gorm:"primaryKey"`
    Name      string
    Email     string `gorm:"uniqueIndex"`
    PhotoURL  string
    Geohash   string
    LastSeen  time.Time
}
type RideOffer struct {
    ID        string `gorm:"primaryKey"`
    DriverID  string
    FromGeo   string
    ToGeo     string
    Fare      float64
    Time      time.Time
    Seats     int
    Status    string
}
```

4. API Endpoints/Sample gRPC Methods

Handler	Method (gRPC/REST)	Description
AuthHandler	Login, Logout, Refresh	Auth with Google, JWT management
UserHandler	GetProfile, UpdateProfile	CRUD user profile
RideHandler	CreateOffer, ListOffers	CRUD for rides, geo-search
RideHandler	RequestRide, ListRequests	CRUD for requests, geo-search
MatchHandler	RequestToJoin, Accept, Reject	Handle ride matching
LocationHandler	StreamLocation	Update/fetch live user location
ChatHandler	Send, StreamMessages	CRUD/send live chat messages
ReviewHandler	SubmitReview, ListReviews	Ratings and feedback for rides

5. HTTP/gRPC Responses and Error Handling

- **Standardized responses** for every endpoint (success, not found, unauthorized, bad input, server error).
- **Status codes** (for REST): 200, 201, 400, 401, 404, 500.
- **Protobuf error fields** or gRPC Status errors for gRPC APIs.
- All errors logged in the service layer, mapped to client-safe responses in the handler.

6. Technical Requirements

- **Tech Stack:**
  - Go (Golang)
  - gRPC + Protobuf for APIs (REST optional via grpc-gateway)
  - GORM for SQL
  - MongoDB/Redis for chat, session, geo, as needed
  - DI/Wire for dependency management
- **Project Structure:**

```
backend/
├── cmd/
├── api/            # gRPC/REST handlers
├── service/
├── repository/
├── db/             # GORM models + migrations
├── di/             # Dependency injection setup
├── proto/
├── utils/
└── main.go
```

7. Implementation Phases

1. Project & DB schema setup with Makefile, environment config.
2. GORM models/entities, DB migration scripts.
3. Repository: CRUD and geohash-index based queries.
4. Service: Business logic, request validation, state transitions.
5. Handler: gRPC/REST endpoints, auth middleware.
6. DI setup for wiring all layers.
7. End-to-end API tests and sample client scripts.

8. Deliverables

- Codebase organized in strict layered modules.
- gRPC proto files and auto-generated API documentation.
- Full database schema and migration scripts.
- README with clear API usage.
- Test suite covering major flows and error scenarios.

Summary:
This SRS delivers a scalable, testable backend API for ride sharing, adopting a layered architecture appropriate for modern Go microservices. Every function—from geo-matching to chat—follows clear separation of concern, making it easily consumable by any frontend/mobile app team, allowing independent development and maintenance.