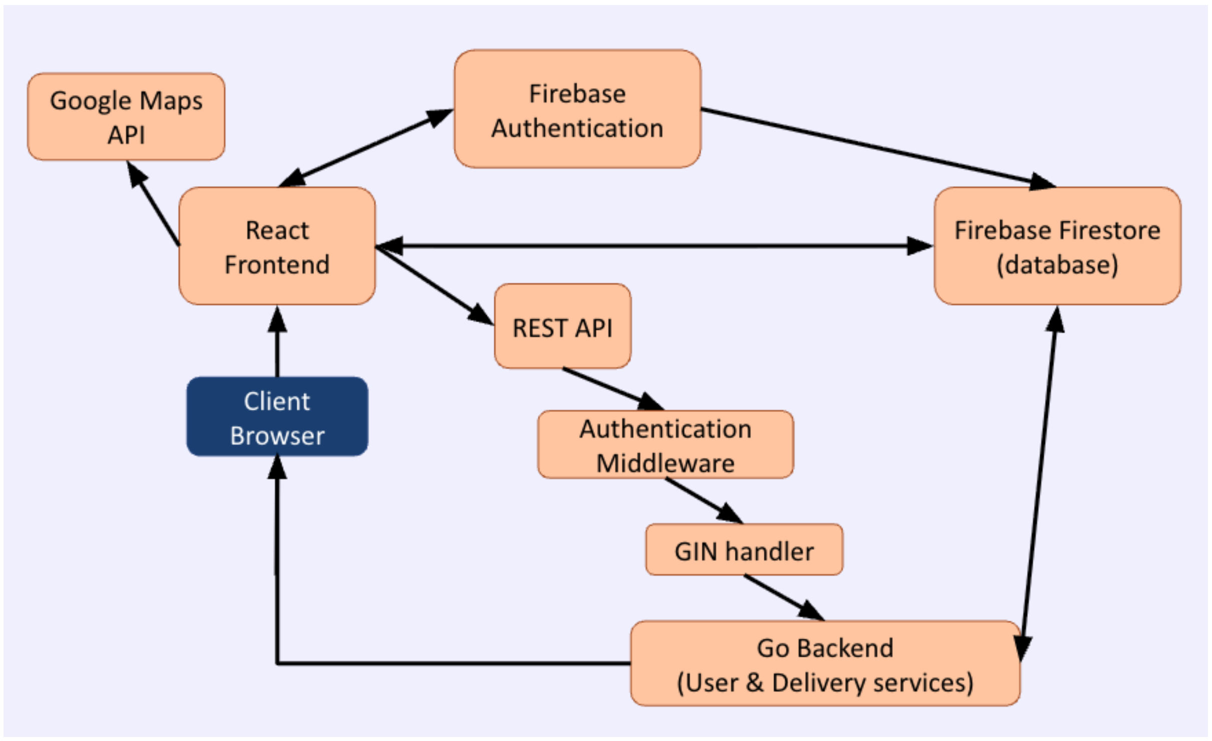
Courier Management Developers Guide

Eva Poluliakhov
Katya Stekolchick

The developers guide briefly review our high level design, and explain what the main modules are, in which files/directories they are included, and the basic interaction between them, as well as necessary tools and libraries to build our project.

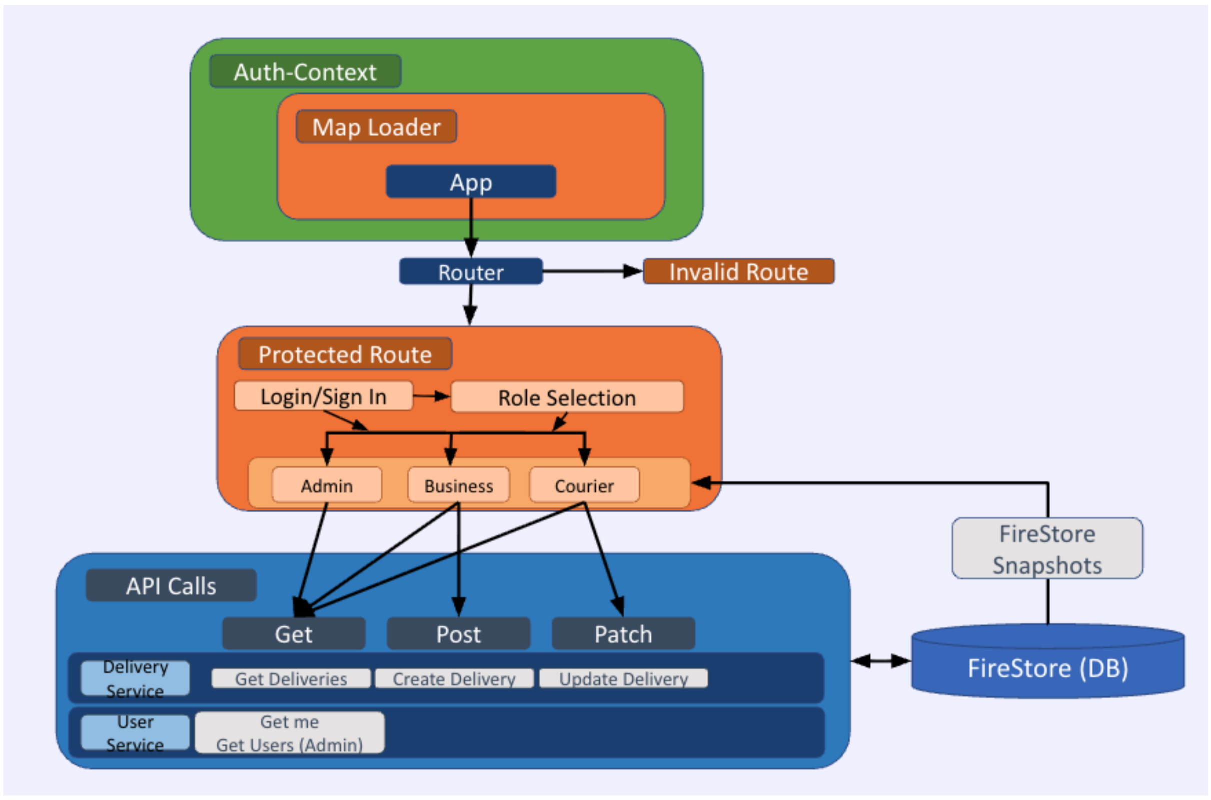
High level design


Google Maps API
Firebase Authentication
React Frontend
Firebase Firestore (database)
REST API
Client Browser
Authentication Middleware
GIN handler
Go Backend (User & Delivery services)

The system is composed of three major tiers:

1. **React frontend** running in the client's browser. This layer manages the user interface and uses Firebase Authentication for sign-in, Google Maps services for visualization and address autocomplete, and Firestore listeners for live updates. It calls the backend via a REST API for writes and for privileged read operations.
2. **Go backend** using the Gin web framework. Requests are passed through an authentication middleware, enforcing authorization based on user roles. Gin maps incoming REST endpoints to handler functions. Those handlers invoke the UserService and DeliveryService to perform operations on Firestore. Each action is performed by an authorized user with the correct role and uses atomicity in critical parts of the code such as delivery assignment. Gin takes the results and writes JSON responses with the correct HTTP status code, which are sent back to the client browser.
3. **Google Cloud Firestore** acting as a database. It stores collections for users and deliveries.

Main Modules & locations



The app is organized in two top levels: react frontend (under src/) and go backend (under backend/).

Frontend: Its primary responsibilities are user authentication (using Firebase Auth), page routing, data visualisation and map interactions.
The main modules are:

- Authentication & Context: Implemented in **src/context/AuthContext.jsx.**
  AuthContext provides global authentication and role state for the app. user, userRole and loading are exposed to all of the components and may be accessed using useAuth(). User and userRole allow the component to decide privileged routing/visuality. Loading is an indicator whether the authentication is done and the user is fetched with its data is ready to use. A function refreshUserRole() is exposed as well, to trigger refetch of the userRole from Firestore after updating it (used only in initial role selection).

- Routing: Routes for login, role selection and role based pages are defined in **src/App.js**, each component is wrapped with **src/components/ProtectedRoute.jsx** or **src/components/InvalidRoute.jsx**.
  Prior to log/sign in, the default route is the login page.
  Upon log/sign in, the user is redirected to the relevant page (role selection / role based page).
  ProtectedRoute receives the role that is privileged to access the component that is wrapped inside (the children). It uses user and userRole to verify the identity and renders it upon success or renders InvalidRoute upon failure.

InvalidRoute renders login page if user is not logged in, or the role based page if user has role (restricting only to pages of user's role) or to role selection if user role is yet set.

- Login: implemented in **src/pages/login.jsx**.
  Using Firebase Authentication, Login manages login or sign in both with email and password or google login. Based on user and userRole, it navigates the user. Navigation is blocked as long as loading, to avoid navigation to role selection while the role is fetched and loaded.

- Role Selection: implemented in **src/pages/role.jsx**.
  Triggered for new users only, upon signin. User's doc is updated in firebase, based on the selection of role and related fields. After updating, refreshUserRole(), is triggered to refetch the userRole across the app.

- Role Based Pages:
  src/pages/Admin.jsx: Using REST API, first it fetches it's own profile (get/me), fetches once all couriers (get/couriers) and listens to all fetched couriers locations updates and deliveries via onSnapshot for live visualization. This data is sent to OveriewTab, DeliveriesTab, ProfileTab and the admin can move between the tabs.

  src/pages/Business.jsx: Using REST API, first it fetches it's own profile (get/me), and listens to deliveries posted by this business via onSnapshot for live updates. This data is sent to OverviewTab, DeliveriesTab, ProfileTab and CouriersTab and the business can move between the tabs. The business info is sent to NewDeliveryTab and it uses REST API with post/deliveries.

  src/pages/courier.jsx: Displays a map showing the courier's current location and available deliveries within a configurable radius. The courier can accept a delivery, after which the UI shows the route to the pickup location and destination. The courier's device updates their location document couriers/{uid}/location periodically so admin and businesses can track progress. This page uses @react-google-maps/api for maps, Radix for a slider controlling the search radius, and canvas-confetti for success celebrations (e.g. when a delivery is delivered).

- API Integration: Implemented in **src/api/api.js.**
  The frontend communicates with the backend using the api helpers. Each Firebase IDtoken automatically attached to it's requests.

- Realtime Updates: Used in **src/pages/Admin.jsx, src/pages/Business.jsx** and **src/components/CourierMap.jsx.**
  Firestore snapshot listeners are set up in effect hooks on mount in components like Business, Admin and CourierMap. So, when a delivery document (or a courier's live location doc) changes, the UI re-renders automatically (no polling required).

- Google Maps & Places: Used in **src/index.js, src/pages/courier.jsx, src/components/Address.jsx** and **src/components/CourierMap.jsx.**
  We use <GoogleMap> component to render maps, <Marker> for pick up/drop off/courier markers and <Autocomplete> for business addresses. We also compute routes for navigation and for ETA (to pickup and destination) for the courier. We use <LoadScript> which loads the map, prior to rendering the app. That way we may access the map everywhere in the app as its already loaded.

Backend: Acts as a secure API server that performs operations on the database, based on requests from frontend. The main modules are:

- Authentication: Implemented in **backend/internal/auth/firebase_middleware.go.**
  Includes Firebase Admin setup and Gin middleware. Each request passes through auth.Middleware, which verifies the Firebase ID token. If the token is valid, the middleware puts the user's UID in the Gin context and lets the handler continue. Otherwise, it rejects the request.

- Database*:* Implemented in **backend/internal/db**.
  Wraps and shares a single Firestore client. The wrapper initializes a *firestore.Client and provides methods like Collection(). Since it embeds the native client, you can still call the standard methods like Doc(), Collection(), RunTransaction() directly on it.

- Service: Implemented in **backend/internal/service**.
  Both services are the actual logic implementation, both depend on the database wrapper.
  UserService - contains methods to fetch user info (specific for each role) and lists of users.
  DeliveryService - implements the delivery workflow logic. Operations like accept and status update are done here with Firestore transactions (for atomicity). It also handles filtering logic for listing deliveries to present the right data to each role.

- Transport: Implemented in **backend/api/openapi.yaml**, **backend/internal/transport/http/handler.go, backend/internal/transport/http/openapi.gen.go.**
  The openapi.yaml defines the API. Codegen ensures the documentation and server implementation stay in sync. openapi.gen.go includes the structure and methods generated by the yaml rules and implemented in handler.go (implements the ServerInterface from the generated code). The handlers extract user's uid from Gin context which is set in the middleware. Then the handlers decode the request into the appropriate structure format and invoke the relevant method from the relevant service (user/delivery service). Lastly the handlers return a response with the data and matching status code in JSON format.
  main.go serves as a bootstrapper, it loads environment variables, initializes Firebase and Firestore clients via NewFirestoreClient(), and wires together the services, handlers, router, and CORS configuration. Finally, it starts the Gin HTTP server on the configured port.

Database: Google Firestore serves as a cloud database, storing collections such as users and deliveries. Firestore's real-time subscriptions allow the frontend to listen to data changes (for live maps and delivery lists) without constantly polling the backend. The main modules are:

- Users Collection: Each user has a doc with their UID as key. All users have email and role fields. Some roles also have specific info - business users have businessName, businessAddress, location and more. Courier users have courierName, balance and email. Admin users have email adminName and role.

- Deliveries Collection: Each delivery has a doc with their deliveryID as key. All deliveries have the following fields: business info (name, address, location), destination info, item, payment, status, timestamps and assignment fields (assignedTo, deliveredBy). Location fields are objects with lat and lng.

- Couriers Collection: Each courier has a subcollection with it's last location update and time stamp of the update. Each document is identified by the courier UID.

Communication & Data Flow

All write operations (creating deliveries, patching status) are done through the backend REST API (ensuring validation and permission checks).

Many read operations are also done through the API (listing deliveries via GET /deliveries, getting user profile via GET /me). However, for real time UI, the frontend directly reads from Firestore with listeners (deliveries list for business/admin, courier locations for tracking).

Login Flow: Frontend uses Firebase Auth directly (backend is not involved in creating accounts or logging in). Once logged in, backend's role is to verify tokens on protected routes. If a token is invalid the backend returns 401. Token refresh is handled by Firebase SDK automatically.

Posting a Delivery: Frontend POSTs to backend, the backend writes it to Firestore, finally Firestore triggers frontend snapshot update.

Accepting a Delivery: Courier triggers an API call, backend transaction updates Firestore (atomically), Firestore triggers Business's list update (showing assigned courier/accepted status) and admin's list updates. Couriers fetch via REST periodically rather than subscribe with listeners to deliveries, to avoid heavy real time load.

Updating Status: Similar to accepting a delivery. Additionally, backend updates courier's balance which the courier frontend fetches on demand (after it was marked as delivered).

Tracking: The courier app continuously writes to Firestore their couriers/{uid}/location doc. The admin and business have listeners on those docs and update map markers accordingly (backend isn't involved here).

Tools & libraries

- **Frontend:** React(Node & npm**)**, React Router, @react-google-maps/api.

- **Backend:** Go, oapi-codegen, Gin.

- **Firebase:**

    - Firebase JS SDK: Authentication and Firestore client.
    - Firebase Admin SDK: Verifies ID tokens and accesses Firestore.
    - Firestore: SDK to read and write data from Firestore.

- **UI/UX Libraries:**
    - Mantine: Ready to use components and theming.
    - Tailwind: Rapid, consistent layout and styling, configured via tailwind.config.js
    - Radix UI Slider: Used on the Courier page to control the search radius.
    - Framer Motion: Animation library for component transitions and presence, used for smooth UI state changes and overlays like the DeliveryCard.
    - Canvas-Confetti: For celebratory bursts triggered on success actions, like "Accept" / "Delivered".
    - Recharts: Charting library used for analytics/visualizations. Used in the business and admin dashboards.

- **Postman**: to test API endpoints independently from the UI.