ElderEase Core Architecture & Feature Guide

This guide summarizes the most critical flows in ElderEase, how they work end-to-end, and the essential code you may want to review. Each section includes a short explanation and a compact code snippet to orient you in the codebase.

1) Account creation & dynamic role routing

After sign-up or login, we create/read a profile in Firestore under `users/{uid}` and also keep a compact copy in localStorage. We subscribe to the Firestore user doc so any admin edits (e.g., updated display name or status=terminated) propagate instantly. Routing sends users to the correct role home using the stored role.

Key files: `src/lib/auth.ts`, `src/App.tsx`

```
// Live user profile sync; auto-logout if terminated
// src/lib/auth.ts
userDocUnsub = onSnapshot(doc(db, "users", u.uid), async (snap) => {
  if (!snap.exists()) { setLocalProfile(null); cb(null); return; }
  const data = snap.data() as any;
  if ((data?.status || "").toLowerCase() === "terminated") {
    await signOut(auth);
    setLocalProfile(null);
    cb(null);
    return;
  }
  const profile: AuthProfile = {
    uid: u.uid,
    email: u.email ?? data.email ?? null,
    displayName: u.displayName ?? data.displayName ?? null,
    role: data.role,
    phone: data.phone ?? null,
  };
  setLocalProfile(profile);
  cb(profile);
});
```

```
// Role-based redirect to home
// src/App.tsx
const user = getCurrentUser();
if (!user) return <Navigate to="/login" replace />;
return <Navigate to={`/${user.role === 'elderly' ? 'elder' : user.role}`} replace />;
```

2) Service requests (Elder → Admin → Assignment)

Guardians submit requests with services/date/time and an optional preferred volunteer. Admins review, compute dynamic pricing, check conflicts, then create an `assignments` document and mark the request as assigned. The assignment becomes the single source of truth for schedules, notifications, and receipts.

Key files: `src/pages/elder/RequestService.tsx`, `src/pages/admin/ServiceRequests.tsx`

```
// Request payload, normalized for reliable reads downstream
// src/pages/elder/RequestService.tsx
const payload = {
  userId: currentUser?.id ?? null,
  elderName,
  services: serviceNames,
  perServiceHoursByName,
  serviceDateDisplay: format(selectedDate, "PPP"),
  serviceDateTS: serviceDayMs, // midnight millis
  startTime24: startTime, endTime24: endTime,
  startTimeText: format12h(startTime), endTimeText: format12h(endTime),
  notes: additionalNotes.trim() || null,
  preferredVolunteerEmail: preferredVolunteerEmail || null,
  preferredVolunteerName: preferredVolunteerName || null,
  status: "pending",
  createdAt: serverTimestamp(),
};
```

```
// Assignment creation and receipt generation (admin side)
// src/pages/admin/ServiceRequests.tsx
const { tier, percent } = getDynamicAdjustment(tasksDone, ratingAgg?.avg);
const demand = getDemandModifier(req);
const combinedPercent = percent + (demand.percent || 0);
// Build line items with adjusted rates
const lineItems = selectedServices.map((name) => {
  const baseRate = SERVICE_RATES[name] ?? 0;
  const hours = Math.max(0, Number(perServiceHoursByName?.[name] ?? 0));
  const adjustedRate = baseRate * (1 + combinedPercent);
  return { name, baseRate, hours, adjustedRate, amount: adjustedRate * hours };
}).filter((li) => li.hours > 0);
```

3) Date/time normalization & dynamic pricing

We store `serviceDateTS` normalized at midnight plus 24h times (`HH:mm`). Performance tier comes from tasks completed and average rating; demand tier comes from the ratio of competing requests to available matching volunteers for the same window. Combined modifiers adjust per-service base rates, and totals are saved to the receipt.

Key file: `src/pages/admin/ServiceRequests.tsx`

```
// Demand = competing requests / available matching volunteers
const getDemandModifier = (req) => {
  const day = Number(req.serviceDateTS);
  const s = toMinutes(req.startTime24), e = toMinutes(req.endTime24);
  const reqServiceIds = Array.isArray(req.services) ? req.services.map(toServiceId) :
req.service ? [toServiceId(req.service)] : [];
  const serviceMatch = (services: string[] | null | undefined) => {
    const ids = (services || []).map(toServiceId);
    return reqServiceIds.some((id) => ids.includes(id));
  };
```

```
  const availableVols = (volunteers || []).filter((v) =>
    serviceMatch(normalizeServiceLabels(v.services || [])) &&
isVolunteerAvailableForRequest(v, req)).length;
  const competing = (requests || []).filter((r) => /* same day + status + window + match */
true).length;
  const ratio = availableVols > 0 ? competing / availableVols : competing >= 1 ? 2.5 : 0;
  if (ratio >= 2.0) return { tier: "Surge", percent: 0.10, ratio };
  if (ratio >= 1.5) return { tier: "Peak", percent: 0.06, ratio };
  if (ratio >= 1.0) return { tier: "High", percent: 0.03, ratio };
  return { tier: "Normal", percent: 0, ratio };
};
```

4) Preferred volunteers & conflict-free schedules

The guardian form and admin page compute availability by querying same-day `assignments` and checking overlap of minute-intervals, including cross-midnight splitting. If a preferred volunteer is unavailable, the guardian form blocks submission with a clear toast.

Key files: `src/pages/elder/RequestService.tsx`, `src/pages/admin/ServiceRequests.tsx`

```
// Guardian side: availability in enriched list (guards on loading)
// src/pages/elder/RequestService.tsx
const rs = toMinutes(startTime), re = toMinutes(endTime);
const intervals = busyByEmail[email] || [];
let available: boolean | null = null;
if (!busyLoading && rs != null && re != null) {
  available = !intervals.some(([bs, be]) => (rs < be && bs < re));
}
```

```
// Admin side: hard availability check for the specific request
// src/pages/admin/ServiceRequests.tsx
const isVolunteerAvailableForRequest = (vol, req) => {
  const day = Number(req.serviceDateTS);
  const s = toMinutes(req.startTime24), e = toMinutes(req.endTime24);
  const intervals = (busyByDate[day] || {})[(vol.email || "").toLowerCase()] || [];
  return !intervals.some(([bs, be]) => (s < be && bs < e));
};
```

5) Notifications (guardian confirmations & receipts)

Guardians receive “Service confirmed” notifications streaming from `assignments` where they are the `elderUserId`. Each item expands to a clean, tabular receipt with confirmation number, line items, and totals.

Key file: `src/pages/elder/Notifications.tsx`

```
// Confirmations feed with receipt
// src/pages/elder/Notifications.tsx
const q = query(
  collection(db, "assignments"),
  where("elderUserId", "==", uid),
```

```
  where("status", "==", "assigned"),
  orderBy("createdAt", "desc")
);
onSnapshot(q, (snap) => setItems(snap.docs.map((d) => ({
  id: d.id,
  icon: HeartHandshake,
  title: `Service confirmed: ${Array.isArray(a.services)?a.services.join(", "):a.services}`,
  text: `${a.startTimeText} - ${a.endTimeText} on ${new
Date(a.serviceDateTS).toLocaleDateString()}`,
  badge: "Confirmed",
  tone: "info",
  receipt: a.receipt || null,
})));
```

6) Volunteer applications (public → admin triage)

The landing form validates Philippine numbers, captures selected services, and writes to `pendingVolunteers`. Admins review/approve in a searchable list. Approved applicants get auto-mapped to the companion role on sign-up (email match).

Key files: `src/components/sections/VolunteerSection.tsx`, `src/pages/admin/VolunteerApplications.tsx`

```
// PH phone normalizer used across forms
// src/components/sections/VolunteerSection.tsx
const normalizePHPhone = (input: string): string | null => {
  const d = input.replace(/\D+/g, "");
  if (d.startsWith("639") && d.length === 12) return `+${d}`;
  if (d.startsWith("09") && d.length === 11) return `+63${d.slice(1)}`;
  if (d.startsWith("9") && d.length === 10) return `+63${d}`;
  if (d.startsWith("63") && d.length === 12) return `+${d}`;
  return null;
};
```

```
// Admin triage stream
// src/pages/admin/VolunteerApplications.tsx
useEffect(() => {
  const q = query(collection(db, "pendingVolunteers"), orderBy("createdAt", "desc"));
  const unsub = onSnapshot(q, (snap) => {
    setApplications(snap.docs.map((d) => ({ id: d.id, ...(d.data() as any) })));
  });
  return () => unsub();
}, []);
```

7) Dashboards (admin & volunteer analytics)

Admin aggregates requests, assignments, ratings, and form-metrics to render trends, top volunteers, cancellations, and average form completion time (public & in-app). Volunteers see their week's upcoming hours and a paginated activity log.

Key file: `src/pages/admin/Dashboard.tsx`

```
// Form completion time averages
// src/pages/admin/Dashboard.tsx
useEffect(() => {
  const unsub = onSnapshot(collection(db, "formMetrics"), (snap) => {
    let elderSum = 0, elderCount = 0, volunteerSum = 0, volunteerCount = 0;
    snap.docs.forEach((d) => {
      const m = d.data() as any, dur = Number(m.durationMs) || 0;
      if (m.type === "elder_request_service") { elderSum += dur; elderCount += 1; }
      if (m.type === "volunteer_application") { volunteerSum += dur; volunteerCount += 1; }
    });
    setAvgElderMs(elderCount ? elderSum / elderCount : null);
    setAvgVolunteerMs(volunteerCount ? volunteerSum / volunteerCount : null);
  });
  return () => unsub();
}, []);
```

8) Volunteer list (edit & terminate with cross-collection sync)

The admin Volunteer List allows editing name/phone/address/gender/services. Edits write to both `pendingVolunteers` and matching `users` docs to keep the entire app in sync. Terminations flip `status='terminated'` in both places; `auth.ts` then blocks login and auto-logs out.

Key file: `src/pages/admin/VolunteerList.tsx`

```
// Write edits to both collections (pendingVolunteers + users)
await updateDoc(doc(db, "pendingVolunteers", editing.id), { ... });
if (editing.email) {
  const uq = query(collection(db, "users"), where("email", "==",
editing.email.toLowerCase()));
  const usnap = await getDocs(uq);
  for (const udoc of usnap.docs) {
    await updateDoc(doc(db, "users", udoc.id), { ... });
  }
}
```

```
// Terminate in both collections
await updateDoc(doc(db, "pendingVolunteers", v.id), { status: "terminated", decidedAt:
serverTimestamp() });
if (v.email) {
  const uq = query(collection(db, "users"), where("email", "==", v.email.toLowerCase()));
  const usnap = await getDocs(uq);
  for (const udoc of usnap.docs) {
    await updateDoc(doc(db, "users", udoc.id), { status: "terminated", updatedAt:
serverTimestamp() });
  }
}
```

9) My assignments & two-step completion (volunteer → guardian)

Volunteers mark an assignment completed, which sets `awaitingGuardianConfirm=true`. Guardians then confirm and rate; only after guardian confirmation do analytics (completed counts and ratings) accrue.

Key files: `src/pages/companion/MyAssignments.tsx`, `src/pages/elder/MySchedule.tsx`

```
// Volunteer marks completed; waits for guardian confirm
// src/pages/companion/MyAssignments.tsx
await updateDoc(doc(db, "assignments", assignment.id), {
  status: "completed",
  awaitingGuardianConfirm: true,
  updatedAt: serverTimestamp(),
});
```

```
// Guardian confirms and rates
// src/pages/elder/MySchedule.tsx
await updateDoc(doc(db, 'assignments', rateTarget.id), { guardianConfirmed: true, updatedAt:
serverTimestamp() });
await addDoc(collection(db, 'ratings'), {
  assignmentId: rateTarget.id, volunteerEmail: rateTarget.volunteerEmail || null,
  volunteerName: rateTarget.volunteerName || null, elderUserId: rateTarget.elderUserId ||
uid || null,
  rating: ratingValue, feedback: ratingFeedback || null, createdAt: serverTimestamp(),
});
```

10) Security rules (minimum needed for the flows above)

Signed-in users can read `serviceRequests` and `formMetrics`; public forms can create `formMetrics` entries for analytics. Assignments reads are permitted for signed-in users to support availability checks and dashboards; field-level updates remain restricted by actor.

Key file: `firebase.rules`

```
match /serviceRequests/{id} {
  allow read: if isSignedIn();
  allow create: if isSignedIn();
  allow update: if isSignedIn();
  allow delete: if false;
}

match /formMetrics/{id} {
  allow create: if true && ('type' in request.resource.data) && ('durationMs' in
request.resource.data);
  allow read: if isSignedIn();
  allow update, delete: if false;
}
```

If you want a deeper dive into any section, open the file paths above and search for the shown function names or constants—they'll lead you to the exact implementation.