Product Requirements Document (PRD)

Project: Frontend Web Application — TypeScript + React (Vite) + Tailwind

Owner: Diwakar Patel / Frontend Team

Created: 2026-02-25

Purpose: This PRD describes the goals, scope, user personas, features, UX flows, technical arch implementation details, testing, deployment, and success metrics for a scalable, secure, and performant frontend web application built with TypeScript, React (Vite), Tailwind CSS and related (React Hook Form, Zod, TanStack Query, Recharts).

---

1. Executive Summary

Build a modern, maintainable, and accessible frontend that serves as the primary user interfac web product (for example: mental-health tracker, educational dashboard, or community platform). app will be fast to develop and iterate on, robust in type-safety, and production-ready with pipeline targeting Vercel.

Key qualities: **Type-safe** (TypeScript + Zod) - **Fast developer experience** (Vite, React), **Reliable data handling** (TanStack Query) - **Small, consistent UI** (Tailwind + component system) - **Secure auth** (JWT in HTTP-only cookies, refresh flow)

---

2. Objectives & Success Metrics

Objectives

1. Deliver a responsive, accessible, and polished UI that meets product goals and user needs.
2. Ship features with low bugs and clear typing to reduce runtime errors.
3. Keep first-load time under 1.5s on 3G simulated mobile and TTFB optimized for CDN.
4. Implement scalable state and data fetching patterns for offline resilience and caching.

Primary Metrics

- Time to Interactive (TTI) — target < 2s on mobile.
- Core Web Vitals: LCP < 2.5s, CLS < 0.1, FID < 100ms.
- Error rate (JS exceptions) < 0.5% of user sessions.
- API success rate > 99%.
- Developer cycle time for PR → Merge < 24 hours (CI passing).

---

3. User Personas

1. **Student/Consumer** — mobile-first, low bandwidth, expects a quick onboarding, simple analytics (mood/stress charts), and secure login.
2. **Teacher/Coach** — uses dashboard features on desktop, needs charts, export, and filters.
3. **Admin** — manages content, users, and sees system observability. Requires role-based access.

---

4. Key Features & Requirements

4.1 Core Features

- **Authentication**: Sign up / Login / Logout, social login optional, password reset, email verification.
- **User Profile**: View/edit profile, preferences (dark mode), privacy controls, export data.
- **Data Entry Forms**: Daily mood, sleep, notes — implemented with React Hook Form + Zod.
- **Dashboard**: Overview cards, time-range filters, and Recharts visualizations for trends.
- **Notifications**: In-app notifications and email preferences.
- **Admin Panel**: User management, role assignments, content moderation.

4.2 Non-functional Requirements

- **Accessibility**: WCAG 2.1 AA compliance for key flows.
- **Internationalization**: i18n-ready (e.g., react-i18next) with English + regional languages.
- **Offline & Resilience**: Cached data with TanStack Query; graceful degraded UI when offline.
- **Security**: CSRF protection, JWT in HTTP-only cookies, secure refresh token rotation.

---

5. UX & UI Guidelines

5.1 Design System

- Build a small component library (Atoms → Molecules → Organisms). Each component lives in `src/components/<Component>/` with `index.tsx`, `styles.css`, `test.tsx`, `stories.tsx` (if using Storybook).
- Tokens: colors, spacing, typography defined in Tailwind config and a `tokens.ts` file for cross-use.

5.2 Pages & Flows

- **Landing** → Auth → Onboarding → Dashboard → Detail Page → Settings.
- Single-primary CTA per page.
- Mobile-first responsive design with progressive enhancement for desktop features.

5.3 Motion & Effects

- Use subtle motion (Framer Motion) for micro-interactions: button presses, modals, and list reordering. Keep animations small to avoid layout shifts.

---

6. Technical Architecture

6.1 Directory Structure (recommended)

```
src/
  api/               # typed API clients
  assets/
  components/
  features/          # domain-driven feature folders
  hooks/
  pages/
  routes/
  styles/
  utils/
  App.tsx
  main.tsx (Vite entry)
```

6.2 Core Libraries

- **Language**: TypeScript (strict mode enabled)
- **Build**: Vite
- **Framework**: React
- **Styling**: Tailwind CSS + Tailwind config for tokens
- **Forms**: React Hook Form + Zod (for both frontend and backend schema sharing)
- **Data fetching**: TanStack Query (React Query)
- **Routing**: React Router v 6
- **Charts**: Recharts
- **State**: React Context for auth + small local reducers/hooks. Avoid global state for UI unless necessary.
- **Testing**: Vitest + React Testing Library; Playwright for E 2 E.
- **Linting**: ESLint + Prettier + TypeScript rules
- **Storybook**: Optional, recommended for component-driven development.

6.3 API Contract Guidance

- Use typed API clients (openapi-generator or hand-rolled fetch wrappers) returning typed responses.
- Prefer small REST endpoints or GraphQL (if multiple clients & complex queries). Example endpoint patterns:
- `GET /v1/dashboard?range=30d`
- `POST /v1/users/profile`
- `POST /v1/auth/login`
- Centralize fetch `src/api/fetch.ts` that handles `401` -> refresh token logic (with queueing) and integrates with TanStack Query's `queryFn`.

6.4 Auth Flow (recommended)

- Login -> server returns access token (short lived) in HTTP-only cookie and refresh token (HTTP-only cookie).
- Frontend uses `GET /v1/auth/me` to fetch user profile.
- On `401`, `fetch.ts` calls `POST /v1/auth/refresh` and retries original request; fallback to redirect to login if refresh fails.

---

7. Validation & Error Handling

- Validate forms with Zod schemas mirrored on backend.
- Show user-friendly error messages; log structured errors to Sentry or similar.
- Use TanStack Query error boundaries and `useMutation` error handling.

---

8. Accessibility

- Use semantic HTML and ARIA attributes where needed.
- Ensure color contrast & keyboard navigability.
- Provide skip links and focus management for modals.
- Test with axe-core and manual keyboard + screenreader checks.

---

9. Performance & Optimization

- Code-splitting per route (React.lazy + Suspense) and keep initial bundle small.
- Use Vite production optimizations (esbuild minify), Brotli/Gzip on CDN (Vercel provides it).
- Use image optimization (Next/Image style or third-party service); prefer SVG for icons.
- Use service-worker caching for static assets and runtime caching for API responses where appropriate.
- Implement LCP optimization: preload key fonts, critical CSS, and render content above the fold quickly.

---

10. Security

- Store JWT tokens in HTTP-only, Secure cookies; use `SameSite=Strict` / `Lax` as appropriate.
- CSRF: server side should enforce anti-CSRF tokens for state-changing operations, or ensure double submit cookie pattern.
- Input sanitization; never trust client-side validation alone.
- Use security headers (CSP, HSTS) via CDN.
- Protect admin routes with RBAC checks server-side; frontend should not rely on client checks for access control.

---

11. Testing Strategy

- **Unit tests**: Vitest + RTL for components and hooks. 80% coverage target for core modules.
- **Integration tests**: Test form flows, auth flows, and data fetching using mocked network via MSW (Mock Service Worker).
- **E2E tests**: Playwright to cover critical user journeys (signup, login, record entry, view dashboard, admin actions).
- **Performance tests**: Lighthouse CI for PR gating and tracking.

---

12. CI/CD & Deployment

- Host on **Vercel** for frontend; use preview deployments for each PR.
- GitHub Actions pipeline:
- `lint` (ESLint), `type-check` (tsc), `test` (Vitest), `build` (Vite), `lighthouse-ci`.
- On merge to `main`: production deploy to Vercel.
- Secrets: store in GitHub secrets / Vercel environment variables.

---

13. Observability & Monitoring

- Runtime errors to Sentry or LogRocket (session replay optional).
- Track usage & funnels with PostHog/Amplitude/GA4.
- Monitor Core Web Vitals via Web Vitals library and integrate with analytics.

---

14. Roadmap & Milestones (suggested)

M0 — Setup & Foundations (1–2 weeks) Vite project, TypeScript strict mode, Tailwind config, ESLint, Prettier, Vitest setup. - Basic auth flow stub and API mock infra (MSW). - Compone skeleton.

M1 — Core UX (2–4 weeks) Auth pages, onboarding, profile page. - Form flows (React H Form + Zod). - Dashboard skeleton + Recharts integration.

M2 — Stability & Production (2–3 weeks) Error monitoring, CI gating, Lighthouse CI, accessibility audit. - Real API integration, refresh-token flow.

M3 — Polish & Admin (2–3 weeks) Admin panel, export features, multi-language support, advanced charts.

M4 — Scale & Optimize (ongoing) Performance budget enforcement, progressive web app (optional), A/B testing.

---

15. Risks & Mitigation

- **Risk:** Third-party libraries (Recharts, TanStack Query) upgrade breaks behavior.
- **Mitigation:** Dependabot + automatic PR testing; pin major versions.
- **Risk:** Auth token flows cause inconsistent `401` handling.
- **Mitigation:** Centralized fetch wrapper with request queuing.
- **Risk:** Accessibility regressions.
- **Mitigation:** Integrate axe checks into CI and manual accessibility sprints.

---

16. Open Questions

- Will backend expose OpenAPI/GraphQL schema for direct typed client generation?
- Is social login required at launch (Google/Facebook/Apple)?
- What level of offline functionality is mandatory (full offline write vs read-only)?

---

17. Appendix

- **Coding conventions:** Use `src/` absolute imports (`tsconfig.paths`), `kebab-case` filenames for components.
- **Naming:** components prefixed by domain (e.g., `ProfileEditForm` in `features/profile/`).
- **Useful resources:** TanStack Query docs, React Hook Form docs, Zod docs, Recharts docs.

If you'd like, I can also: - Generate a component checklist and starter repo structure (Vite React + TanStack Query) as code. - Produce a Storybook-ready component library scaffold.