Product Requirements Document (PRD) — blabout-monorepo
Subtitle: Input for TestSprite standardization and test generation

Document info
- Version: 0.1
- Date: 2025-09-13
- Owner: Project team
- Repository: /Users/aloe/Code/blabout-monorepo

1. Overview
The project is a modern full-stack monorepo:
- Frontend: React 18 + TypeScript + react-scripts (dev) with TailwindCSS, Shadcn UI, Redux Toolkit, TanStack Query, Kinde Auth.
- Backend: Rust (Axum) + ParadeDB (PostgreSQL-compatible), WebSockets + Redis, JWT auth.
- Mobile: Expo + React Native + Tamagui.
- Desktop: Electron + React.

Shared utilities live in /shared.

Primary objective: produce a standardized PRD that enables automated generation of comprehensive tests across web, backend, mobile, and desktop.

2. Goals
- Define key user flows and acceptance criteria.
- Enumerate functional and non-functional requirements.
- Specify test coverage expectations and environments to guide automated test creation.
- Provide enough structure for TestSprite to standardize and derive test plans.

Non-goals
- Detailed UI design specs.
- Finalized API schemas for every endpoint (can be discovered or stubbed).

3. Users and roles
- End user: signs in, navigates main dashboard, uses core features.
- Admin/maintainer: manages configuration, monitors deployments and health.
- Developer: runs local services, contributes features, and maintains tests.

4. In scope
- Web app authentication, routing, and core dashboard interactions.
- API request/response flows with error handling and retries.
- Real-time updates over WebSockets.
- Basic mobile and desktop launch flows and feature parity checks where applicable.

Out of scope (initially)
- App store submission and desktop store-specific flows.

- Advanced analytics configuration and third-party billing integrations.
5. Key user journeys (web)
- Sign in with Kinde and land on the workspace dashboard.
- Navigate between main sections with protected routes.
- Perform a primary action (for example, create or update a resource).
- Observe real-time updates (presence or notifications).
- Handle errors gracefully (network failure, 401/403, validation).
6. Acceptance criteria (examples)
- Authentication
- Given a logged-out user, when they click Sign in and complete Kinde flow, then they see their authenticated dashboard and a valid session is established.
- Given an expired token, when making API calls, then tokens are refreshed automatically or user is redirected to login without data loss.
- Routing and authorization
- Protected routes are not accessible without auth; direct URL hits redirect to login.
- Data interactions
- Create, read, update, delete operations reflect consistently in UI and API, including optimistic UI where configured.
- Failure cases show user-friendly error toasts and do not leave UI in inconsistent state.
- Real-time
- When a server-side event occurs, connected clients receive updates within a reasonable latency and UI reflects changes without manual refresh.
- Accessibility
- Keyboard navigation works across major flows; ARIA attributes present on interactive elements; color contrast meets WCAG AA.
7. Functional requirements
- Authentication using Kinde across clients.
- Secure API access with JWT and token refresh.
- WebSocket connectivity for live updates; automatic reconnection; presence indicators.
- Consistent state via Redux Toolkit + TanStack Query with cache invalidation patterns.
- Error handling with user-visible feedback (toasts, inline messages).
- Routing with React Router; guarded routes for authenticated pages.
8. Non-functional requirements
- Performance: Web pages render within 3 seconds on a cold load on baseline hardware; animations run smoothly (~60 fps).
- API: P95 latency under 200 ms for typical CRUD endpoints under expected dev/stage loads.
- Reliability: Automatic reconnection for real-time; transient errors retried with exponential backoff.
- Security: JWT validation, CORS, rate limiting, secure secrets handling.

- Accessibility: WCAG 2.1 AA targets for key flows.
- Cross-platform: Basic verification for mobile and desktop launches.

9. Environment and URLs (development)
- Frontend: http://localhost:3000 (react-scripts dev)
- Backend API: http://localhost:3001 (Axum)
- Mobile: Expo local dev ports
- Desktop: Electron dev environment

Note: Exact endpoints may vary; TestSprite may auto-discover or rely on mocks.

10. Testing requirements (by platform)

Frontend
- Unit tests: component rendering and interactions; Redux slices and thunks; query hooks; form validation; error states; auth flows; route protection.
- Integration tests: store + component interactions; API happy-path and failure; WebSocket connection lifecycle; file upload/download if applicable.
- E2E tests: authentication; primary user flows across dashboard; cross-browser basics; responsive behavior; performance assertions where feasible.

Backend
- Unit tests: handlers, services, validation, auth middleware, error mapping, WebSocket handlers.
- Integration tests: DB operations and transactions; cache/Redis interactions; real-time event propagation; auth end-to-end; file operations if applicable.
- Load tests: endpoint performance (CRUD), WebSocket concurrency thresholds, DB query tuning baselines.

Mobile
- Unit tests: components, navigation, state, API integration, offline behavior.
- Integration tests: push notifications, deep links, auth, basic file access.

Desktop
- Unit tests: main process functions, IPC, file operations, updater hooks, menu actions.
- Integration tests: cross-platform checks, system tray, auto-update simulation, native API calls.

11. Coverage targets
- Critical auth and data mutation paths: 100 percent.
- Utilities and pure functions: 90 percent or higher.
- Overall: minimum 80 percent with focus on critical flows.
- E2E: cover all main user journeys end-to-end.

12. Tooling preferences
- Frontend: Vitest + React Testing Library, Playwright for E2E.
- Backend: Rust built-in tests; add load testing tool of choice.
- Mobile: Jest for RN, Detox for E2E if available.

- Desktop: Electron test utilities; Spectron-like automation or Playwright with electron runner.
13. Risks and assumptions
- Some endpoints and flows may be placeholders or TBD; TestSprite can standardize and infer tests from structure and acceptance criteria.
- Real-time tests may require mocks if live infra is unavailable.
- CI integration can be added after initial suite generation.
14. Success metrics
- Automated suite validates all acceptance criteria for primary flows.
- Regressions are caught in CI within minutes of PRs.
- Developers can run npm run test:all locally with deterministic results.

Appendix: references
- See project README for architecture details and service commands.
- WARP.md outlines development and deployment workflows and environment guidelines.