Productivity App - Technical Walkthrough

Repository: addes4/productivity | Commit: a91177c | Generated: 2026-02-19 10:52

1. What Was Built

This project is a local-first weekly planning application that combines goal planning, calendar booking management, recurrence support, drag-and-drop editing, overlap visualization, and secure Google Calendar import.
The app is designed to help a user translate weekly goals (for example study, exercise, or work sessions) into concrete time blocks while respecting existing commitments and user-defined constraints.
- Frontend stack: React 18 + TypeScript + Vite + Tailwind CSS + date-fns + dnd-kit.
- Backend helper: a dedicated Node server for OAuth and Google Calendar API proxying.
- Persistence model: browser localStorage for app state, encrypted token file for Google refresh tokens.

2. High-Level Architecture

2.1 Frontend Application Shell

The frontend is orchestrated from src/App.tsx. It composes the calendar grid, side panel, settings modal, event modal, and report modal. App.tsx also wires UI actions to store operations and import/sync flows.
- Main route and interaction hub: src/App.tsx.
- Central state and mutators: src/store/useStore.ts.
- View components: src/components/*.tsx.

2.2 Data and Domain Model

The data model in src/types.ts separates three core concepts: CalendarEvent (bookings), ActivityGoal (intent), and PlannedBlock (scheduled output). This separation keeps planning logic deterministic and UI rendering predictable.
- CalendarEvent supports source tags (manual/import/google), all-day flags, and recurrence metadata.
- ActivityGoal supports weekly targets, sessions-per-week mode, day/time preferences, travel buffer, and color.
- PlannedBlock stores status (planned/done/missed/partial), lock state, and mini-session markers.

3. Scheduling Engine (How Planning Actually Works)

3.1 Entry Point and Scope

Planning runs through planWeek(...) in src/utils/planWeek.ts, triggered by runPlanWeek(...) in useStore. The week start is normalized to Monday to keep day indexing consistent across all calculations.
- Locked blocks in the active week are preserved.
- Unlocked blocks in the active week are regenerated.
- Blocks outside the active week are kept unchanged.

3.2 Constraint Synthesis

For each day, the planner builds blocked intervals from existing calendar events, locked planned blocks, travel buffers after locked blocks, and sleep windows (including cross-midnight handling).
- Work-hour framing can limit scheduling to a configured daily interval.
- Allowed days, earliest start, and latest end are enforced per goal.
- Minimum break minutes are applied when selecting and reserving slots.

3.3 Goal Ordering and Slot Scoring

Goals are sorted by priority (high to low), then by shorter session duration first. Candidate slots are scored by preferred time of day (morning/lunch/evening/any) and basic duration fit.
- This strategy increases success for high-priority and tighter-fit goals.
- The algorithm greedily places one session at a time in the best available slot.

3.4 Session Count Logic and Fallback

Each goal computes intended session count using sessionsPerWeek when provided, otherwise via weeklyTargetMinutes/sessionMinutes. If the week is full and Minimum Viable Day is enabled, the planner attempts 10-minute mini sessions.
- Unschedulable remainder generates a ConflictReport with actionable suggestion text.
- Mini sessions are labeled (isMini=true) for transparent UX.

3.5 Key Behavioral Rule: Gym Distribution

A dedicated daily cap ensures gym goals are not scheduled multiple times on the same day. The implementation tracks gym sessions per day globally and also per goal where sessionsPerWeek is explicit.

- Outcome: selecting 3 gym sessions per week spreads sessions across different days when possible.

4. Calendar Rendering and Interaction Model

4.1 Week Grid and All-Day Lane

src/components/CalendarGrid.tsx renders a fixed header, a dedicated all-day row, and an hourly timeline. Week-number pseudo-events are filtered out from normal event rendering.

- Timed events and planned blocks are rendered in day columns.
- All-day events are shown in compact chips at the top day lane.

4.2 Overlap Visualization (1/2, 1/3, 1/4 Splits)

The utility src/utils/overlapLayout.ts computes overlap groups and assigns per-item columns. CalendarGrid applies this to both bookings and planned blocks so collisions remain visible.

- Two overlaps: each item gets roughly half width.
- Three overlaps: each item gets roughly one third.
- Four overlaps: each item gets roughly one fourth.
- A hard guard rejects operations that would create more than 4 concurrent items.

4.3 Drag-and-Drop Editing

Drag-and-drop is unified for planned blocks and calendar events. Drop targets are hour cells encoded as day+hour identifiers. On drop, new ranges are validated before persistence.

- Planned block move validates sleep-window overlap and 4-way concurrency limit.
- Calendar event move also respects the global 4-way concurrency limit.
- Dragging across days is supported by target-cell interpretation, not just vertical delta.

4.4 Compact Rendering for Short Blocks

Very short visual blocks now prioritize activity/event name instead of time text. This avoids clipped or unreadable labels and preserves semantic clarity in dense schedules.

5. Booking Lifecycle: Manual, Recurring, Import, Google

5.1 Manual and Recurring Bookings

AddEventModal supports weekly recurrence by weekday selection. Recurring templates store recurrenceDays and optional recurrenceExDates (date exceptions).

- When recurrence is enabled, at least one weekday is required.
- Single-instance deletion writes an exception date instead of deleting the whole series.
- When a single recurring instance is dragged, the parent series receives an exception and a detached one-off event is created.

5.2 Weekly Expansion of Recurrence

src/utils/recurringEvents.ts expands recurrence templates into concrete instances for the visible week. Each instance carries recurrenceParentId and recurrenceInstanceDate for targeted edits.

5.3 iCal Import

ICS import parses incoming events and deduplicates against existing entries using a normalized key based on lowercase title + start + end. Imported events are marked source='import'.

5.4 Google Calendar Import

Google events are fetched for the active week through the backend proxy, normalized into the app schema, categorized by source calendar, deduplicated, and merged into state as source='google'.

- Auto-sync is attempted once per week navigation while connected.
- Manual sync remains available for explicit refresh.

6. Security and Backend Design for Google OAuth

server/googleCalendarProxy.mjs exists to avoid exposing secrets in the frontend. OAuth client secret and token encryption key remain server-side. The frontend only uses /api/* endpoints.

- OAuth session state uses short-lived server memory + HttpOnly cookie.
- Return URL is sanitized to frontend origin to prevent open redirects.
- Refresh tokens are encrypted at rest using AES-256-GCM with key material derived from env secret.
- Connections are persisted in server/data/google-connections.json via atomic write.

- Disconnect revokes token (best effort), removes stored connection, and clears cookie.
- CORS is restricted to configured frontend origin.

7. User-Configurable Behavior

Settings allow work-hour boundaries, sleep windows, minimum break, max activities per day, office days, and distinct color themes per booking source (manual/import/google).

- Color values are validated to strict #RRGGBB format before storage.
- Goal forms support location, priority, allowed days, and time-of-day preference tuning.

8. Reliability Guards and Edge Cases

- Concurrency guard prevents creating or moving into >4 overlapping items.
- Invalid ranges (end <= start) are filtered during processing and rendering.
- Week-number helper events are excluded from normal booking logic.
- Cross-midnight sleep windows are treated as true blocked ranges.
- Google invalid_grant clears stale connection and prompts reconnect flow.

9. File-Level Map for Key Responsibilities

```
src/App.tsx                          -> top-level orchestration and import/sync wiring
src/store/useStore.ts                -> state CRUD, persistence, planner trigger
src/utils/planWeek.ts                -> scheduling algorithm and conflict reporting
src/utils/overlapLayout.ts           -> overlap math, column layout, concurrency cap
src/utils/recurringEvents.ts         -> weekly recurrence expansion
src/components/CalendarGrid.tsx      -> calendar render pipeline + drag/drop logic
src/components/AddEventModal.tsx     -> booking and recurrence authoring UI
src/components/SettingsPanel.tsx     -> behavior and color configuration UI
server/googleCalendarProxy.mjs       -> OAuth flow, token storage, Google API proxy
```

10. End-to-End Flow (Practical Sequence)

- User adds bookings manually or imports via ICS/Google.
- User defines goals with constraints and preferences.
- Planner computes free slots, places sessions, and emits conflicts if needed.
- Calendar shows schedule with overlap splitting and source-specific colors.
- User drags blocks/events to adjust plan; validators enforce rules.
- User tracks completion states and reviews weekly progress report.

11. How to Run and Verify

Frontend

```
npm install
npm run dev
```

Google proxy (optional)

```
cp .env.server.example .env.server
npm run dev:server
```

Production build check

```
npm run build
```

12. Summary

The implemented system is more than a static calendar UI: it is a constraint-aware planning engine with interactive schedule editing, recurrence management, overlap-safe layout, and secure Google synchronization. The architecture keeps planning logic isolated, state transitions explicit, and sensitive credentials off the client.