DayBoard Manual and Architecture Notes

Introduction

DayBoard integrates multiple services—calendars, bank transactions, commute estimates, and taxes—to provide a unified view of your daily commitments and finances. This document explains the architecture, core design decisions, and potential improvement areas. It is intended for developers and interview preparation.

High-Level Architecture

DayBoard follows a client–server model:

* **SwiftUI client** (macOS and iOS) renders the UI, stores minimal state, and communicates with the backend using HTTPS. It never stores sensitive credentials; it requests and caches data in memory and persists user preferences via Keychain.

* **Go backend** runs statelessly. It handles OAuth authorization flows, retrieves data from external APIs, normalizes and caches data in PostgreSQL, and exposes REST endpoints. Stateless design allows horizontal scaling if traffic increases.

* **PostgreSQL** stores application data (users, tokens, events, subscriptions, transactions, profiles) and reference tables (tax brackets, cost models).

* **External APIs** supply data:
    - Google and Microsoft for calendar events
    - Plaid for bank transaction data
    - Google Distance Matrix for commute time and distance
    - Cost models in the database for ride-share estimates

Design Decisions

1. **Environment variables**: All secrets (API keys, OAuth client secrets) are loaded from environment variables. Do not hard-code secrets or commit them to the repository. Use `.env` files for local development and your hosting platform’s secret management in production.

2. **OAuth flows**: The backend implements OAuth 2.0 authorization code flows for Google and Microsoft. For Plaid, the Link token exchange happens on the backend to keep secrets off the client. After authorization, tokens are encrypted before persisting.

3. **Recurring detection**: Subscriptions are inferred by analyzing transaction history. A merchant with the same amount and cadence (±2 days) is considered recurring. Users can edit or disable detected subscriptions via the UI. The algorithm is deterministic and does not rely on ML, making it easy to explain and adjust.

4. **Tax estimation**: Federal and state tax brackets are stored in tables. The estimator applies standard deduction and FICA taxes, then divides net income by pay frequency to derive per-paycheck values. This approach is transparent and easily adapted when tax laws change.

5. **Commute cost**: Since ride-share APIs restrict pricing, DayBoard uses a simple model per city: `base_fare + distance * per_mile + duration * per_minute`. Surge pricing is approximated via a slider. Users can override with actual fare quotes to refine the model.

6. **Resilience**: The backend uses retries and exponential backoff

when calling external APIs. It records timestamps to avoid duplicate inserts and uses upserts on unique keys to maintain idempotency.

7. **Privacy**: Plaid transactions and calendar events are accessible only to the authenticated user. DayBoard never requests or stores account credentials; it stores encrypted access tokens. Users can revoke access at any time.

Potential Improvements

* **Push notifications**: Currently DayBoard uses local notifications scheduled on the client. To notify users about events detected on the backend (e.g., subscription renewals discovered overnight), integrate with Apple Push Notification service (APNs). The backend can send push payloads via a secure token.

* **Expenses categories**: Extend subscription detection to classify one-off purchases (e.g., textbooks, groceries) into categories. This would turn DayBoard into a simple budgeting tool.

* **Two-way calendar sync**: Enable users to create or edit events directly in DayBoard. This would require write permissions and UI components for event creation.

* **Data exports**: Provide CSV or Excel exports of all calendar events, expenses, and commute logs for year-end reporting.

* **Third-party sync**: Add additional providers (e.g., Apple Calendar, Facebook events) to broaden coverage.

Interview Talking Points

1. **System design**: Emphasize that DayBoard is an offline-first architecture—data is cached locally on the backend and refreshed periodically. It uses idempotent writes, proper retry logic, and least-privilege OAuth scopes. Discuss trade-offs between synchronous and asynchronous fetches.

2. **Security**: Explain token encryption, environment variable management, and the decision not to implement server-side push until end-to-end encryption is in place.

3. **User experience**: The menu bar UI reduces friction by exposing the most important information in one glance. Local notifications minimize battery usage compared to polling. Provide examples of how this design improves daily workflow.

4. **Extensibility**: Highlight how the clear separation between data sources, business logic, and presentation makes it easy to add new features or providers. Mention how tax brackets or cost models can be updated via migrations.

5. **Reliability**: Mention how the backend logs errors and tracks sync state, uses exponential backoff for API calls, and avoids duplicate database entries with unique constraints.