

Architecture Plan: Thick Client Music Tracker (Rust/WASM)

1. Core Architecture: The "Thick Client"

This model moves the "backend" logic entirely into the browser. The application is compiled to WebAssembly (WASM), allowing it to run near-native code on the client side. It treats ListenBrainz as the upstream synchronization source and the browser's IndexedDB as the local "source of truth" for statistical queries.

Proposed Stack

- **Language:** Rust (targeting wasm32-unknown-unknown).
- **Frontend Framework: Leptos.**
 - *Reasoning:* Leptos uses fine-grained reactivity (signals) rather than a Virtual DOM. This results in extremely performant DOM updates and a smaller binary size compared to VDOM-based frameworks, fitting the efficiency philosophy often preferred by Arch users.
- **Matching Engine: interzic** (from the **Alistral** project).
 - *Reasoning:* You specifically requested Alistral. interzic is the library crate within the Alistral ecosystem responsible for "translation" and linking tracks across services. It is designed to handle the exact "fuzzy matching" logic needed to normalize "David Bowie - Starman (2012 Remaster)" into the canonical "Starman" entity.
- **Local Database: IndexedDB.**
 - *Library:* rexie (a Rust wrapper for IndexedDB) or glo-storage.
 - *Reasoning:* It provides persistent, transactional storage capable of holding hundreds of thousands of listen records without performance degradation.
- **HTTP Client:** reqwest (with wasm feature enabled).

Alternatives to Consider

- **Framework: Dioxus.** It has a similar syntax to React (RSX) but runs in Rust. It has excellent support for desktop targets (via Tauri).
- **Database: SQLite WASM.** Instead of IndexedDB, you could compile SQLite to WASM and run a real SQL database in the browser (persisted via the Origin Private File System).

2. Rate Limiting Strategy (The "Polite" Middleware)

ListenBrainz enforces strict rate limits (~1 request/second/IP). To prevent the application from crashing or getting banned during a sync, the HTTP client should include a dedicated "middleware" layer.

Proposed Logic: Dynamic Sleep

Instead of a hard-coded delay, the client should inspect the headers returned with every response, specifically X-RateLimit-Reset-In.

1. **Interception:** On every response, the client parses X-RateLimit-Remaining and X-RateLimit-Reset-In.
2. **Decision:**
 - o If Remaining > 0, the next request proceeds immediately.
 - o If Remaining == 0, the client parses X-RateLimit-Reset-In (e.g., "1.2"), adds a small safety buffer (e.g., 100ms), and asynchronously sleeps for that duration before releasing the lock for the next request.
3. **429 Handling:** If the API returns HTTP 429 (Too Many Requests), the middleware should automatically capture the X-RateLimit-Reset-In header from that error response, sleep, and retry the request transparently.

3. Data Synchronization: The Two-Lane Highway

The application needs to handle two distinct data flows: real-time updates and historical backfilling.

Lane A: The Speedometer (Real-Time)

- **Mechanism:** Polling GET /1/user/{username}/playing-now.
- **Interval:** Every 30 seconds.
- **State Detection:** This endpoint returns the currently playing track. The app stores the recording_msid (MessyBrainz ID) in memory. If the MSID changes between polls, the app infers a track has finished and triggers an immediate "Sync" (Lane B) and a "Coach" prompt.

Lane B: The Warehouse (Historical Sync)

- **Mechanism:** Pagination using GET /1/user/{username}/listens.
- **Logic:**
 1. On startup, the app queries IndexedDB: "*What is the timestamp of the most recent listen I have stored?*" (max_local_ts).
 2. It calls the API with min_ts={max_local_ts} and count=1000 (the maximum allowed batch size).
 3. It inserts the new records into IndexedDB and repeats until no new results are returned.

4. Metrics Engine: The "Bowie Specific" Workout

All metrics are calculated locally against the IndexedDB store. This bypasses the ListenBrainz daily stat update delay.

Normalization (via Alistral/Interzic)

Before stats are calculated, incoming metadata is passed through interzic.

- **Problem:** Streaming services tag "Ziggy Stardust" as "Ziggy Stardust (2012 Remaster)" or "Ziggy Stardust (40th Anniversary)".
- **Solution:** interzic attempts to resolve these variations to a single canonical MusicBrainz ID (MBID). The stats engine then groups by this *Canonical MBID* rather than the raw string name, ensuring your "Album Completion Rate" isn't broken by remaster tags.

Core Metric Logic (The "Bowie Filter")

All calculations below are applied strictly to tracks where `artist_name == "David Bowie"`.

Resolution	Songs Metric	Albums Metric	Minutes Metric
Real-time (Pace)	Count in last 60 mins	N/A	Total duration in last 60 mins
Daily	Total count today	Unique <code>release_group_mbid</code> today	Sum of duration today
Weekly	Total count this week	Unique <code>release_group_mbid</code> this week	Sum of duration this week
Monthly	Total count this month	Unique <code>release_group_mbid</code> this month	Sum of duration this month
All-Time	Total count in DB	Total unique <code>release_group_mbid</code> in DB	Total sum of duration in DB

Advanced Gamification Logic

- **Album Completion Rate:** Compares the number of unique tracks listened to in a specific session against the total known tracks for that `release_group` (fetching track lists via Alistral's MusicBrainz integration).
- **The "Sprint" (Hour Pace):** Calculates the current "Heart Rate" by looking at the density of listens in the last 15-minute window and projecting to an hourly rate.
- **Discipline/Focus Check:** Tracks if the user switches albums mid-play. If `current_release` changes before X% of the album is finished, the Coach provides negative feedback.

Potential Alternatives to Discuss

- **Normalization:** Should we use Alistral's remap-mbid logic to permanently write cleaner IDs to your personal ListenBrainz database (using the alistral CLI tool separately), or only normalize virtually inside the WASM app?
- **In-Memory vs. Disk:** For the "Daily" stats, should we keep them in memory for even faster UI updates, or always query IndexedDB to ensure consistency across browser

refreshes?

5. Deployment

The final artifact is a static bundle (index.html, package_name.js, package_name.wasm).

Hosting

- **GitHub Pages:** Free, integrates with your version control, and serves static files efficiently.
- **Vercel / Cloudflare Pages:** Similar to GitHub Pages but often faster builds/deploys.

Access

- **PWA (Progressive Web App):** By adding a manifest.json and a Service Worker, the web page can be "installed" on Android/iOS. It will appear as an icon on your home screen and launch without the browser UI, feeling like a native app.