GSoC 2026 Proposal: Playlists Sorting and Organization

Project: ListenBrainz - Playlists Sorting and Organization
Estimated Length: 175 hours
Mentors: Ansh, Monkey
Applicant: Shreshth Sharma

---

1. Synopsis

ListenBrainz users who rely heavily on playlists currently face significant friction: they can't search their playlists from their own page, can't organize them beyond a flat paginated list, can't sort tracks within a playlist, and can't access their curated MusicBrainz collections. This project addresses all of these pain points through four interconnected sub-projects.

What I will build:

#	Feature	Ticket(s)	Hours
1	User-level playlist search	-	~20h
2	Playlist organizing with tags	LB-1302	~60h
3	Track ordering within playlists	LB-1374	~35h
4	MusicBrainz collections as playlists	LB-1231, LB-961	~60h

Rather than tackling the sub-projects in numerical order, I have structured the execution timeline strategically (1 -> 3 -> 2 -> 4) to mitigate review bottlenecks. Sub-projects 1 (Search) and 3 (Sorting) are tightly scoped, self-contained features. Completing them first establishes early momentum and allows for quick, reviewable PRs. Sub-project 2 (Tags) is the largest core architectural change, but it heavily benefits from having the Sub-project 1 search infrastructure already in place. Finally, Sub-project 4 (MB Collections) is an independent feature loop that can be safely parallelized while the massive Tag PR undergoes review cycles.

Codebase Familiarity

I have studied the existing playlist infrastructure in depth:

- **Database layer** - **listenbrainz/db/playlist.py** handles SQL **(search_playlists_for_user(), get_playlists_for_user()**, recordings CRUD).
- **Data models** - Pydantic models in model/playlist.py (**Playlist, PlaylistRecording**) and **serialize_jspf()**.
- **API views** - **webserver/views/playlist_api.py** exposes REST endpoints (**/create**, **/search**, **/item/move**).
- **Frontend** - **Playlists.tsx** (client-side lodash **orderBy**, grid/list toggles via **jotai**, pagination) and **Playlist.tsx** (drag-and-drop via **ReactSortable**).
- **Schema** - TimescaleDB tables: **playlist** (with **additional_metadata** JSONB), **playlist_recording** (**position**), and **playlist_collaborator**.

2. Detailed Deliverables

Sub-Project 1: User-Level Playlist Search

Goal: Allow users to search within their own playlists directly from **/user/<username>/playlists**.

Current state: Right now, users cannot search within their own playlists, despite the backend already being equipped for it. The global playlist search at **listenbrainz.org/search/** works well, and the backend function **search_playlists_for_user()** already exists in **listenbrainz/db/playlist.py** (using PostgreSQL's **pg_trgm** extension for fuzzy matching). We just need to wire these existing pieces up to the user profile page.

What needs to change

Backend Implementation:

The existing **GET /1/playlist/search** endpoint will be extended to accept an optional **user_name** query parameter. To avoid unintended side effects, the default behavior of **/search** remains unchanged when **user_name** is absent.

1. **Fixing Scope**: Currently, **search_playlists_for_user()** includes all public playlists. I will modify this (via an **include_public=False** parameter) to strictly scope results to playlists owned, collaborated on, or created for that user.
2. **Auth-Aware Output**: The API will wrap results via **playlist.is_visible_by(user_id)** so unauthenticated users see public results, while owners securely access private playlists.
3. **API Edge Cases & Pagination**: Since the database treats count=0 as **LIMIT NULL**, I will enforce pagination bounds at the API layer. Empty results cleanly return **[]**, invalid users return **404**.

4. **Rate Limiting**: I will add a 300ms debounce on the frontend to respect the **@ratelimit()** middleware.

Frontend Integration Strategy:

Instead of building a search interface from scratch, I will add a reusable search hook in **Playlists.tsx**. Since the current implementation relies on server-loaded data via React Router loaders (**useLoaderData**) and local component state, I will extend this by adding a debounced search input and an async fetch flow for search results.

Implementation Details & UX Behavior:

- **Debouncing**: Add a search input to Playlists.tsx with a discrete **searchQuery** state and a 300ms debounce to respect the API limits.
- **State Control & UI Consistency**: The moment a search query becomes active, the component will trigger the async search fetch layer. To keep the UX smooth, I will explicitly manage a loading state that preserves the previous results on screen until the new search data arrives, taking care not to overwrite the original props-derived playlist state.
- **Pagination Isolation**: The main playlist view uses URL parameters (**?page=...**) for server reloads. To prevent state conflicts, the active search will use isolated, component-level pagination (resetting to page 1) that applies only to the search results and avoids any URL mutations.
- **Sorting Logic**: Sorting on the search results will be applied client-side, strictly to the current page of results (mirroring the existing **orderBy** logic on **this.state.playlists**).
- **Clear Reversions**: Clearing the search input will drop the async search state and instantly restore the original playlist layout by pulling from the initial loader data. Since the search operates on isolated component-level state and **never mutates the URL's ?page= parameter**, a user on page 5 who searches and then clears will return to page 5, not page 1.

Testing & Performance Validation:

- **Query Performance**: I will use **EXPLAIN ANALYZE** to confirm queries actively leverage the **pg_trgm** GIN/GIST indexes and avoid sequential scans.
- **Backend Tests**: Add tests to **test_playlist_api.py** asserting visibility boundaries (owners retrieve private/collaborative lists; guests see only public).
- **Frontend Tests**: Write Jest tests to verify the 300ms debounce, loading state rendering, and pagination isolation.

Sub-Project 2: Playlist Organization with Tags

Goal: Allow users to organize their growing collections of playlists using creator managed, playlist-specific tags. A playlist can belong to multiple tags.

Design Decision: Playlist Tags vs. MusicBrainz Tags

Unlike MusicBrainz tags which are crowd-sourced globally and publicly editable by any user, playlist tags in ListenBrainz will follow a strict **Creator-Only** ownership model. ListenBrainz playlist tags are for personal organization. As a result:

- **Only the playlist creator** can add, remove, or rename tags. Collaborators cannot modify tags, even on collaborative playlists. This is enforced via a strict `playlist.creator_id == user["id"]` check (mirroring the existing security pattern at `playlist_api.py` for editing core playlist metadata), rather than the broader `is_modifiable_by()` which also grants collaborator access.
- Tags are stored directly in the internal ListenBrainz database, completely bypassing the external MusicBrainz API sync.
- Tags are **visible** to others (if the playlist is public) as read-only informational metadata, but not **editable** by anyone other than the creator. The tag editing UI (the "+" icon, the Tag Manager Modal) will be hidden from non-creators using the existing `isPlaylistOwner(playlist, currentUser)` check.

(Note: If the project later requires both creators and collaborators to apply their own distinct, private tags to the same shared playlist, the `TEXT[]` array approach would need to be replaced with a normalized junction table (`playlist_user_tag(user_id, playlist_mbid, tag)`). I am intentionally choosing the `TEXT[]` approach and creator-only semantics to keep this 175-hour project tightly scoped. And also after looking at the community discussions that resulted in the outcome that tags should only be modified by creators.)

Additionally, following the existing ListenBrainz design, tags are kept attached directly to the playlist (not user+playlist). For collaborative playlists, only the playlist creator can modify tags, which aligns with how playlist metadata editing is already handled. This keeps the model simple and also allows tags to be reused in global search.

Architectural Implementation

I will implement this using native Postgres array types and a GIN index to keep reads quick. Since playlist filtering is a high-read, low-write operation, avoiding aggregation queries improves performance.

1. Database Layer Models & Migrations (TEXT[] vs Junction Table): I chose to use a native PostgreSQL **TEXT[]** column rather than a normalized **playlist_tag** junction table. These are creator-managed personal tags, not a shared global taxonomy. Since the creator-only ownership model restricts tags to the playlist owner, there is no shared tag entity to normalize. While a junction table forces the database to scan, group, and count (**HAVING COUNT(DISTINCT tag)**) to find a playlist with multiple tags, a **TEXT[]** array paired with a GIN index resolves set-intersection queries instantly via a single bitmap index scan.

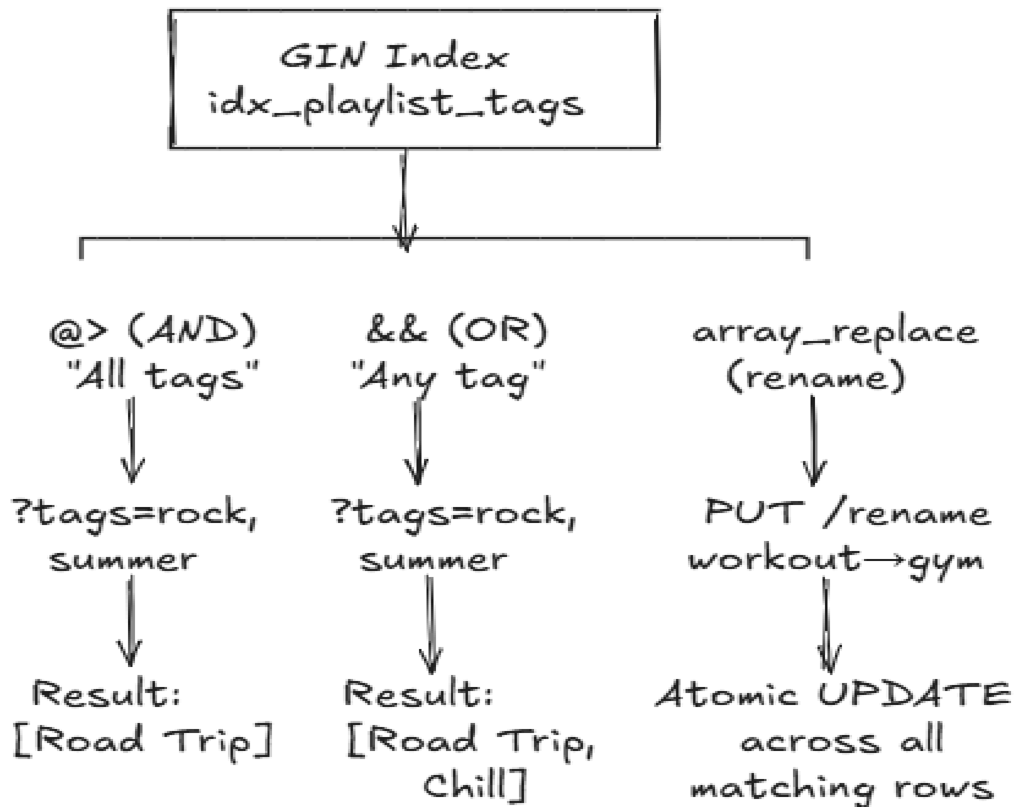
*(Note on Future Extensibility: If ListenBrainz later requires global shared tags or tag descriptions, this **TEXT[]** array acts as a denormalized cache, which can then be safely backed by a new **playlist_tag_definition** metadata table without breaking existing query patterns).*

```sql
ALTER TABLE playlist.playlist ADD COLUMN tags TEXT[] DEFAULT '{}';

-- Creating a GIN index (using default array operators for strict AND containment)
CREATE INDEX idx_playlist_tags ON playlist.playlist USING GIN (tags);
```

*Tags will be normalized to lowercase via the API. A maximum limit of 20 tags per playlist will be enforced at the API level. This limit prevents GIN index bloat (which inflates **VACUUM** costs and degrades performance) and ensures the frontend UI pill-bar remains usable at a glance without complex overflow scrolling.*

playlist.playlist		
id (PK)	name	tags TEXT[]
1	Road Trip	{rock, summer}
2	Gym Mix	{workout, energy}
3	Chill	{chill, summer}
4	Focus	{study, chill}



2. Query Modification (The Core Filtering Logic): To support searching for playlists via multiple tags (e.g., **/user/<username>/playlists?tags=workout,chill**), I will modify **get_playlists_for_user()** to use PostgreSQL's native array containment operator (**@>**).

By default, filtering multiple tags assumes intersection (**AND** logic). However, **OR** logic filtering is also supported via the overlap operator:

```
-- Identifies playlists that contain ALL specified tags (AND logic)
AND pl.tags @> :tags::text[]

-- Identifies playlists that contain ANY of the tags (OR logic)
AND pl.tags && :tags::text[]
```

Both **@>** and **&&** operators are natively accelerated by the GIN index without any schema changes. Tags will be passed as a parameterized array to avoid SQL injection and ensure optimal query plan reuse.

3. Pydantic Model & JSPF Serialization (listenbrainz/db/model/playlist.py): The base **Playlist** model will be updated to explicitly map the new column: **tags: List[str] = []**. I will extend the **serialize_jspf()** method so that **self.tags** is injected into the JSPF **extension** dictionary, exposing the data to the frontend API without requiring secondary database fetches.

4. API Endpoints & Race Condition Handling (listenbrainz/webserver/views/playlist_api.py):All tag endpoints enforce **playlist.creator_id == user["id"]** (creator-only). I will implement the following:

- **POST /1/playlist/<mbid>/tags** Appends tags atomically using a single **UPDATE** with inline deduplication, avoiding any read-modify-write cycle:

```sql
UPDATE playlist.playlist
   SET tags = ARRAY(SELECT DISTINCT unnest(tags || :new_tags::text[])),
       last_updated = now()
 WHERE mbid = :playlist_mbid
   AND creator_id = :user_id;
```

  The **tags || :new_tags** concatenates the existing array with the incoming tags, **unnest + DISTINCT** deduplicates, and **ARRAY(...)** reconstructs the result, all in a single atomic statement. No read step, no row lock needed.

- **DELETE /1/playlist/<mbid>/tags/<tag>** (Uses **array_remove** to strip a specific tag. The endpoint is idempotent).

```sql
UPDATE playlist.playlist
   SET tags = array_remove(tags, :tag),
       last_updated = now()
 WHERE mbid = :playlist_mbid
   AND creator_id = :user_id;
```

- **PUT /1/user/<user_name>/playlist-tags/rename** (Renames a tag across **all** of the user's playlists atomically. Body: **{ "old_tag": "workout", "new_tag": "gym" }**.

```
UPDATE playlist.playlist
  SET tags = ARRAY(
        SELECT DISTINCT unnest(array_replace(tags, :old_tag, :new_tag))
      ),
      last_updated = now()
WHERE creator_id = :user_id
  AND tags @> ARRAY[:old_tag]::text[];
```

  This handles the edge case where a playlist already has both the old and new tag by deduplicating in-place. The GIN index on **tags** makes the **@>** containment check fast even across thousands of playlists.
- Extend **GET /user/<username>/playlists** (To parse the **?tags=** query parameters and apply the appropriate **@>** or **&&** filters).

Frontend UI & UX Integration

UI Components & API Replacement: I will reuse the existing autocomplete **AddTagSelect** component structure (from **TagsComponent.tsx**) currently utilized on Artist pages. However, I will strip out its external **submitTagToMusicBrainz()** logic and tag-voting UI states. Instead, I will wire the component to point to the newly created internal **/1/playlist/<mbid>/tags** API.

Playlist Dashboard: For the main Playlists view, a permanent, horizontally scrolling pill-filter bar below the search input will be implemented (See below in UI mockups). It will manage its own **selectedTags** state, pushing the actively selected filters into the **/user/<user>/playlists?tags=** API call to isolate the rendered playlist grid. Selecting or deselecting a tag will immediately trigger a refetch via React Query, keeping the UI responsive without full page reloads.

Sub-Project 3: Track Ordering Within Playlists

Goal: Allow users to sort tracks within a playlist by date added, track title, artist name, release date, or randomly, in addition to the existing manual drag-and-drop reordering.

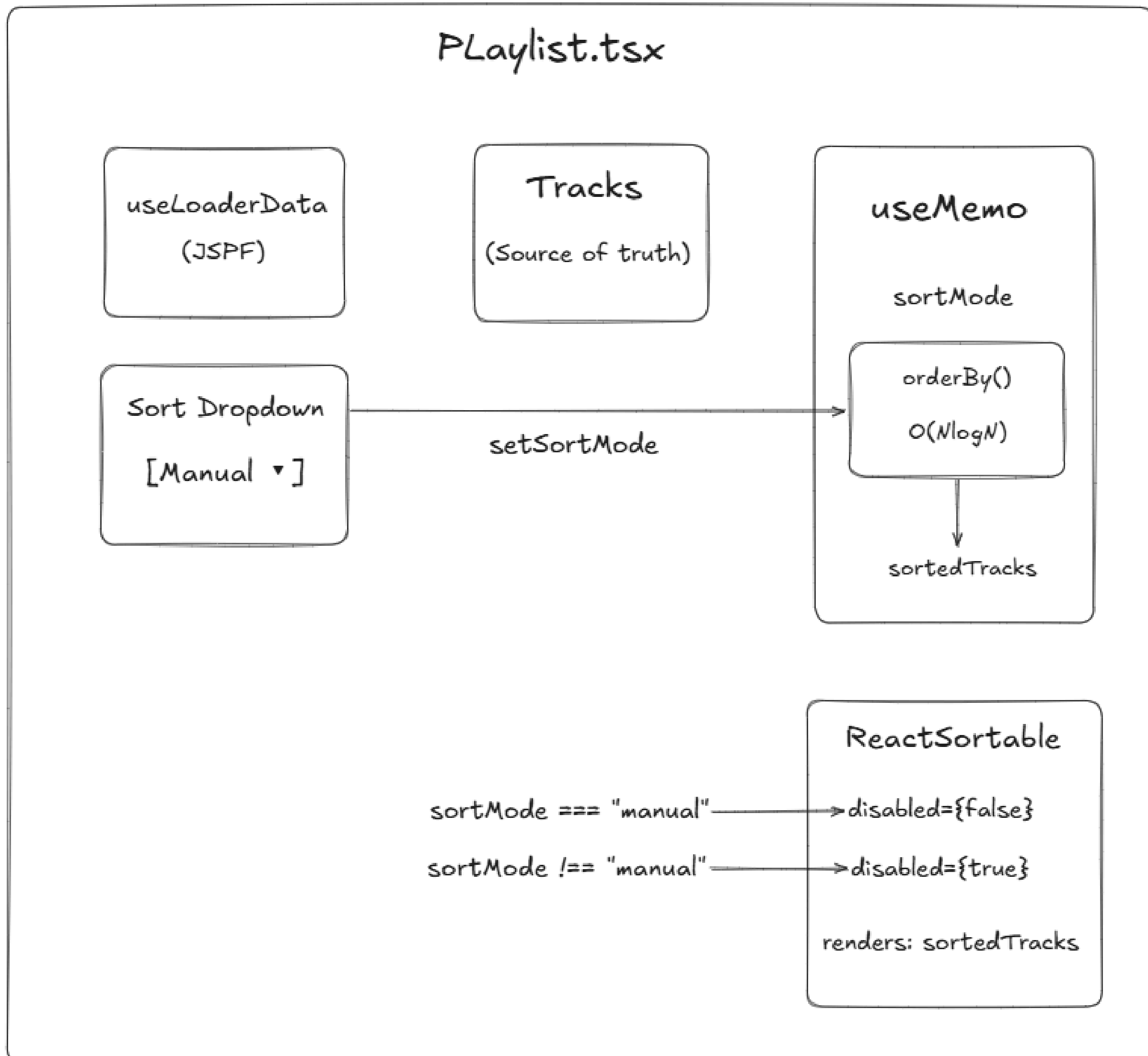
Current state: Right now, the individual playlist page `Playlist.tsx` relies completely on ReactSortable for manual drag-and-drop reordering. If a user has a massive playlist and just wants to view it alphabetically or see their most recently added tracks first, they are out of luck. Furthermore, when a track is moved, the `movePlaylistItem` function triggers `APIService.movePlaylistItem()` to persist the change one-by-one. If a user is drastically reorganizing a playlist, this approach creates unnecessary network overhead.

Important architectural context: Tracks within a single playlist are **not** paginated. `get_by_mbid()` calls `get_recordings_for_playlists()` which fetches all recordings with `WHERE playlist_id IN :playlist_ids ORDER BY position` (no `LIMIT`/`OFFSET`). The `serialize_jspf()` method serializes every recording into the `"track"` array, and `Playlist.tsx` renders them all in a single `ReactSortable` list. This means the sort will always apply to **all** tracks in the playlist, not a subset.

Architecture: View-Only Sorting

Following recent community discussions regarding UI complexity, I will keep track sorting strictly as a frontend, view-only feature, preserving the existing manual drag-and-drop system as the sole source of database truth.

- **Temporary UI Sorting (Frontend):** I will introduce a "Sort By" dropdown above the tracklist (Manual, Date Added, Title, Artist, Release Date, Random). React's `useMemo` hook will handle the sorting logic, instantly returning the derived track list based on the selected criteria without hitting the database.
- **Preserving Manual Order:** "Manual Order" will remain the default state. When a user is in Manual mode, the existing `ReactSortable` drag-and-drop UI and its associated `APIService.movePlaylistItem()` persistence will function exactly as they do today.
- **Disabling Conflicts:** When a user selects any dynamic sort (e.g., Artist, Date Added), the view becomes strictly read-only and the drag-and-drop handles are disabled. This completely prevents race conditions and avoids building unnecessary, complex backend reordering endpoints.



Frontend Modifications (`Playlist.tsx`)

I will add a **`useMemo`**-backed **`<select>`** dropdown (Manual, Date Added, Title, Artist, Release Date, Randomize) above the tracklist, positioned next to the existing "Play all" button. Selecting a dynamic sort instantly renders the derived list in $O(N \log N)$ time and safely locks the drag-and-drop UI to prevent accidental reordering.

State Management:

```
type SortMode = "manual" | "date_added" | "title" | "artist" | "release_date" | "random";
const [sortMode, setSortMode] = useState<SortMode>("manual");
```

Sorting Logic (`useMemo`):

The sorted track list is a derived state computed via **`useMemo`**. It recomputes only when **`tracks`** or **`sortMode`** changes:

```
const sortedTracks = useMemo(() => {
if (sortMode === "manual") return tracks;  // passthrough, no sort
if (sortMode === "random") return [...tracks].sort(() => Math.random() - 0.5);

const sorters: Record<string, (t: JSPFTrack) => string | number> = {
  title:        (t) => (t.title || "").toLowerCase(),
  artist:       (t) => (t.creator || "").toLowerCase(),
  date_added:   (t) => {
    const addedAt = t.extension?.[PLAYLIST_TRACK_EXTENSION_URI]?.added_at;
    return addedAt ? new Date(addedAt).getTime() : 0;
  },
  release_date: (t) => {

// release_date comes from the recording metadata extension
    const releaseDate = t.extension?.[PLAYLIST_TRACK_EXTENSION_URI]
                          ?.additional_metadata?.release_date;
    return releaseDate ? new Date(releaseDate).getTime() : 0;
    },
  };

  return orderBy([...tracks], [sorters[sortMode]], [
    sortMode === "date_added" || sortMode === "release_date" ? "desc" : "asc"
  ]);
}, [tracks, sortMode]);
```

Drag-and-Drop Lock:

The **`ReactSortable`** component's **`disabled`** prop controls whether drag handles are active:

```
<ReactSortable
  handle=".drag-handle"
  list={sortedTracks as (JSPFTrack & { id: string })[]}
  onEnd={movePlaylistItem}
  setList={(newState) => setPlaylist({ ...playlist, track: newState })}
  disabled={sortMode !== "manual"}  // locks drag when sorting is active
>
```

Additionally, the drag handle icon will be visually hidden via CSS when **sortMode !== "manual"**:

```css
.playlist-page[data-sort-active="true"] .drag-handle {
opacity: 0.3;
pointer-events: none;
cursor: default;
}
```

Sort Dropdown UI:

Positioned next to the existing “Play all” button in the tracks header section:

```jsx
<h3 className="header-with-line">
  Tracks
  {Boolean(playlist.track?.length) && (
  <>
    <button type="button" className="btn btn-info btn-rounded play-tracks-button" ...>
        <FontAwesomeIcon icon={faPlayCircle} fixedWidth /> Play all
    </button>
    <select
    className="form-select sort-dropdown"
    value={sortMode}
    onChange={(e) => setSortMode(e.target.value as SortMode)}
  >
    <option value="manual">Manual Order</option>
    <option value="date_added">Date Added</option>
    <option value="title">Track Title</option>
    <option value="artist">Artist Name</option>
    <option value="release_date">Release Date</option>
    <option value="random">Randomize</option>
    </select>
  </>
  )}
</h3>
```

Data Availability for Sort Fields:

Sort Field	Data Source	Available?
Manual Order	Default **position** from DB	Already present
Date Added	**extension.added_at** in JSPF track extension	Already serialized by **serialize_jspf()** at **model/playlist.py:167**
Track Title	**track.title**	Populated by **fetch_playlist_recording_metadata()**
Artist Name	**track.creator**	Populated by **fetch_playlist_recording_metadata()**
Release Date	**additional_metadata** on the recording	Needs to be included in the metadata fetch. The existing **get_playlist_recordings_metadata()** already queries the MB database. I will extend it to also pull **release.date** via the existing join path (**recording** → **track** → **medium** → **release**). This data will be injected into each recording's **additional_metadata** dict and serialized via the existing **serialize_jspf()** extension path.
Random	Frontend **Math.random()**	No data needed

I also mapped out a few edge cases:

- **Missing metadata**: Tracks with **null** title/artist/release_date sort to the bottom of the list.
- **Sort persistence**: The selected sort mode is **not** persisted across page reloads (it resets to "Manual Order"). This is intentional. The sort is a view-only ephemeral state.
- **Interaction with "Add a track"**: When a new track is added via **SearchTrackOrMBID** while a dynamic sort is active, the new track is appended to the underlying **tracks** array (manual order), and **useMemo** automatically re-derives the sorted display. The user sees the track appear in its correct sorted position.

Testing:

- Frontend: Jest tests verifying the sort dropdown component, the **`useMemo`** React state logic for accurate sorting, and the locking/unlocking states of the drag-and-drop UI when in read-only mode.

Performance Considerations & Limitations: Since **`Playlist.tsx`** currently lacks DOM virtualization, rendering all 500+ re-sorted tracks will cause a brief main-thread stall during React's reconciliation pass. The **`useMemo`** sort itself is fast $O(N \log N)$, ~500 comparisons takes <1ms), but the React DOM diff/re-render of 500 **`PlaylistItemCard`** components is the actual bottleneck. This is an **existing** limitation and even the current manual drag-and-drop re-renders all 500 cards. My sorting feature inherits this constraint rather than introducing it. Implementing chunked rendering or a virtualization library like **`react-virtuoso`** alongside **`react-sortablejs`** is a substantial architectural shift that falls outside the 175-hour scope, but isolating this view-only sorting logic establishes the clean state-management foundation required for when that broader virtualization refactor occurs in the future.

Sub-Project 4: MusicBrainz Collections as Playlists

Goal: Allow users to view, play, and import their MusicBrainz collections directly within ListenBrainz.

User Flow:

- User navigates to **/musicbrainz-collections** to view their synced MB collections.
- User can toggle a "Hide" state for utility collections (e.g., Audiobooks).
- User clicks a collection to view a read-only tracklist fetched live from MB.
- User clicks "Import" to clone the tracks into a native, editable ListenBrainz playlist.

Technical Implementation: Direct Database Integration via MB_DATABASE_URI

Important Distinction: Live View vs Imported Playlist MusicBrainz collections will **not** be stored in the LB database directly. The endpoint strictly serves a live, read-only view backed by the MusicBrainz database. Maintaining separate endpoints and ignoring **playlist** primary keys creates a cleanly isolated boundary. "Importing" a collection is completely optional and copies the data into standard editable ListenBrainz playlists functionality. Note on Caching: While locally caching MB metadata seems optimal, I intentionally skip caching to mirror existing LB patterns (e.g., **entity_pages.py**), protecting against staleness but retaining the option for short-lived caches if bottlenecks arise.

Initial designs for this feature assumed fetching data via the public MusicBrainz REST API. However, the public API enforces a strict 1 request/second rate limit, which would cause severe throttling when users browse massive collections.

By inspecting the codebase (specifically **listenbrainz/webserver/__init__.py**), I verified that ListenBrainz already maintains a direct, read-only connection to the MusicBrainz database via **app.config["MB_DATABASE_URI"]**.

I will bypass the REST API entirely and execute direct SQL queries against this connection. Because the queries are isolated to the **musicbrainz** schema, we avoid cross-database (FDW) bottlenecks. While this introduces a dependency on the MB schema, this approach aligns with existing LB features that already rely on the same schema. Given its long-term stability, the practical risk is low, though the query layer will remain isolated to simplify future adaptations if needed.

1. The Endpoints: I'll introduce a new blueprint (**musicbrainz_collection_api.py**) with the following routes:

Step 1: User Links Their MusicBrainz Account

ListenBrainz already requires users to authenticate via MusicBrainz OAuth. When a user visits **/user/<username>/musicbrainz-collections**, the backend knows their MusicBrainz **editor.id**

through the existing user → MB account mapping. This mapping is already used throughout the codebase (e.g., in **db_user** for MusicBrainz ID lookups).

Step 2: Fetching Collections List

Endpoint: **GET /1/user/<user_name>/musicbrainz-collections**

Backend flow:

1. Resolve **user_name** → **user_id** via **db_user.get_by_mb_id()**.
2. Look up the user's MusicBrainz **editor.id** from the existing LB ↔ MB user mapping.
3. Open a read-only connection to the MB database using the existing **MB_DATABASE_URI** pattern (exactly as **entity_pages.py** and **playlist_api.py** do):

```
with psycopg2.connect(current_app.config["MB_DATABASE_URI"]) as mb_conn, \
        mb_conn.cursor(cursor_factory=DictCursor) as mb_curs:
```

4. Execute a direct SQL query against the MusicBrainz schema:

```sql
SELECT ec.id,
       ec.gid AS collection_mbid,
       ec.name,
       ec.description,
       ec.public,
       ect.entity_type,
       (SELECT COUNT(*)
          FROM musicbrainz.editor_collection_recording ecr
         WHERE ecr.collection = ec.id) +
       (SELECT COUNT(*)
          FROM musicbrainz.editor_collection_release ecrl
         WHERE ecrl.collection = ec.id) AS item_count
  FROM musicbrainz.editor_collection ec
  JOIN musicbrainz.editor_collection_type ect
    ON ec.type = ect.id
 WHERE ec.editor = :mb_editor_id
   AND ect.entity_type IN ('recording', 'release')
 ORDER BY ec.name;
```

5. Cross-reference with the LB **playlist.mb_collection_prefs** table to check **is_hidden** status:

```
SELECT collection_mbid, is_hidden
  FROM playlist.mb_collection_prefs
 WHERE user_id = :lb_user_id;
```

6. Since cross-database JOINs aren't feasible here (the MB collections and LB preferences live in entirely separate databases), the Python backend fetches the MB collections list, fetches the LB `is_hidden` preferences, and merges them in memory via a dictionary keyed by `collection_mbid`. Hidden collections are filtered out unless `?show_hidden=true` is passed.
7. Return JSON:

```json
{
  "collections": [
    {
      "collection_mbid": "abc-123",
      "name": "My Favorite Albums",
      "description": "A curated list of my all-time favorite albums.",
      "entity_type": "release",
      "item_count": 42,
      "is_hidden": false,
      "public": true
    }
  ],
  "total_count": 5
}
```

Step 3: Fetching Collection Items (Recordings)

Endpoint: `GET /1/musicbrainz-collection/<collection_mbid>`

Backend flow:

1. Validate `collection_mbid` as a valid UUID.
2. Open MB database connection (same `psycopg2.connect` pattern).
3. Determine the collection's entity type:

```
SELECT ec.id, ect.entity_type
  FROM musicbrainz.editor_collection ec
  JOIN musicbrainz.editor_collection_type ect
    ON ec.type = ect.id
 WHERE ec.gid = :collection_mbid;
```

4. **For recording collections** - direct query:

```
SELECT r.gid AS recording_mbid,
       r.name AS recording_name,
       r.length AS duration_ms,
       ac.name AS artist_credit
  FROM musicbrainz.editor_collection_recording ecr
  JOIN musicbrainz.recording r
    ON ecr.recording = r.id
  JOIN musicbrainz.artist_credit ac
    ON r.artist_credit = ac.id
 WHERE ecr.collection = :collection_id
 ORDER BY ecr.position
 LIMIT 2000;
```

5. **For release collections** - multi-join to resolve playable recordings (mirrors the join logic in `listenbrainz/db/recording.py`):

```
SELECT r.gid AS recording_mbid,
       r.name AS recording_name,
       r.length AS duration_ms,
       ac.name AS artist_credit,
       rel.gid AS release_mbid,
       rel.name AS release_name,
       m.position AS medium_number,
       t.position AS track_position
  FROM musicbrainz.editor_collection_release ecrl
  JOIN musicbrainz.release rel
    ON ecrl.release = rel.id
  JOIN musicbrainz.medium m
    ON m.release = rel.id
  JOIN musicbrainz.track t
    ON t.medium = m.id
  JOIN musicbrainz.recording r
    ON t.recording = r.id
  JOIN musicbrainz.artist_credit ac
    ON r.artist_credit = ac.id
 WHERE ecrl.collection = :collection_id
 ORDER BY rel.name, m.position, t.position
 LIMIT 2000;
```

6. **Serialize to JSPF format**: The backend converts the SQL results into the standard JSPF playlist structure so the frontend can render them identically to normal playlists:

```
def serialize_collection_as_jspf(collection_metadata, recordings):
    """Convert MB collection data to standard LB JSPF format."""
    tracks = []
    for rec in recordings:
        track = {
            "identifier": [PLAYLIST_TRACK_URI_PREFIX + str(rec["recording_mbid"])],
            "title": rec["recording_name"],
            "creator": rec["artist_credit"],
        }
        if rec.get("duration_ms"):
            track["duration"] = rec["duration_ms"]
        extension = {}
        if rec.get("release_mbid"):
            extension["release_identifier"] = (
                PLAYLIST_RELEASE_URI_PREFIX + str(rec["release_mbid"])
            )
        track["extension"] = {PLAYLIST_TRACK_EXTENSION_URI: extension}
        tracks.append(track)

    return {
        "playlist": {
            "title": collection_metadata["name"],
            "creator": collection_metadata["editor_name"],
            "identifier": f"https://musicbrainz.org/collection/{collection_metadata['collection_mbid']}",
            "track": tracks,
            "extension": {
                PLAYLIST_EXTENSION_URI: {
                    "public": collection_metadata["public"],
                    "is_mb_collection": True,
                    "entity_type": collection_metadata["entity_type"],
                }
            }
        }
    }
```

7. **Scale & Pagination**: Instead of building custom pagination just for collections, I will mirror the existing `GET /1/playlist/<mbid>` architecture, which fetches all tracks unpaginated to supply the frontend visualizer `MBCollectionPage.tsx`. To prevent excessive payload sizes and frontend rendering bottlenecks, the endpoint will enforce a configurable upper bound (e.g., 2000 records), which can be adjusted based on performance testing during implementation.

2. Frontend View (Read-Only Rendering): When a user navigates to `/user/:username/musicbrainz-collections/:collectionMbid`, the frontend will hit the `GET` endpoint. The backend will serialize the SQL results into the standard ListenBrainz JSPF playlist format. The frontend will render this using a new `MBCollectionPage.tsx` component, a read-only variant of the standard Playlist view (disabling drag-and-drop and track deletion, as the source of truth remains MusicBrainz).

Step 4: The "Import to ListenBrainz" Pipeline

Endpoint: **`POST /1/user/<user_name>/musicbrainz-collection/<collection_mbid>/import`**

Backend flow (Architectural upgrade): While the existing **`/playlist/create`** JSPF pipeline can be reused for smaller collections, large collections (e.g., 10,000+ tracks) may exceed practical HTTP payload limits. To address this, I propose an optimized backend import endpoint that performs a direct database-to-database transfer, bypassing serialization overhead while preserving identical playlist semantics.

1. The frontend simply triggers this dedicated endpoint with the user's auth token - no JSPF payload needed.
2. The Python backend opens a read connection to the MB database and queries the full collection in a streaming/batched manner (the configurable upper bound only applies to the read-only frontend view, not the import). It then opens the LB database connection and performs efficient bulk inserts.
3. It creates a new native ListenBrainz playlist row holding the imported metadata.
4. It bulk-inserts the recording MBIDs into the local **`playlist.playlist_recording`** table, completely bypassing HTTP serialization constraints.
5. This creates an independent, editable copy - the imported playlist has no ongoing link to the MB collection. Returns the new playlist MBID.

Step 5: "Hide Collection" Flow

Endpoint: **`POST /1/user/<user_name>/musicbrainz-collection/<collection_mbid>/hide`**

Backend flow:

1. Validate auth, ensure **`user_name`** matches the authenticated user.
2. Upsert into the LB-side prefs table:

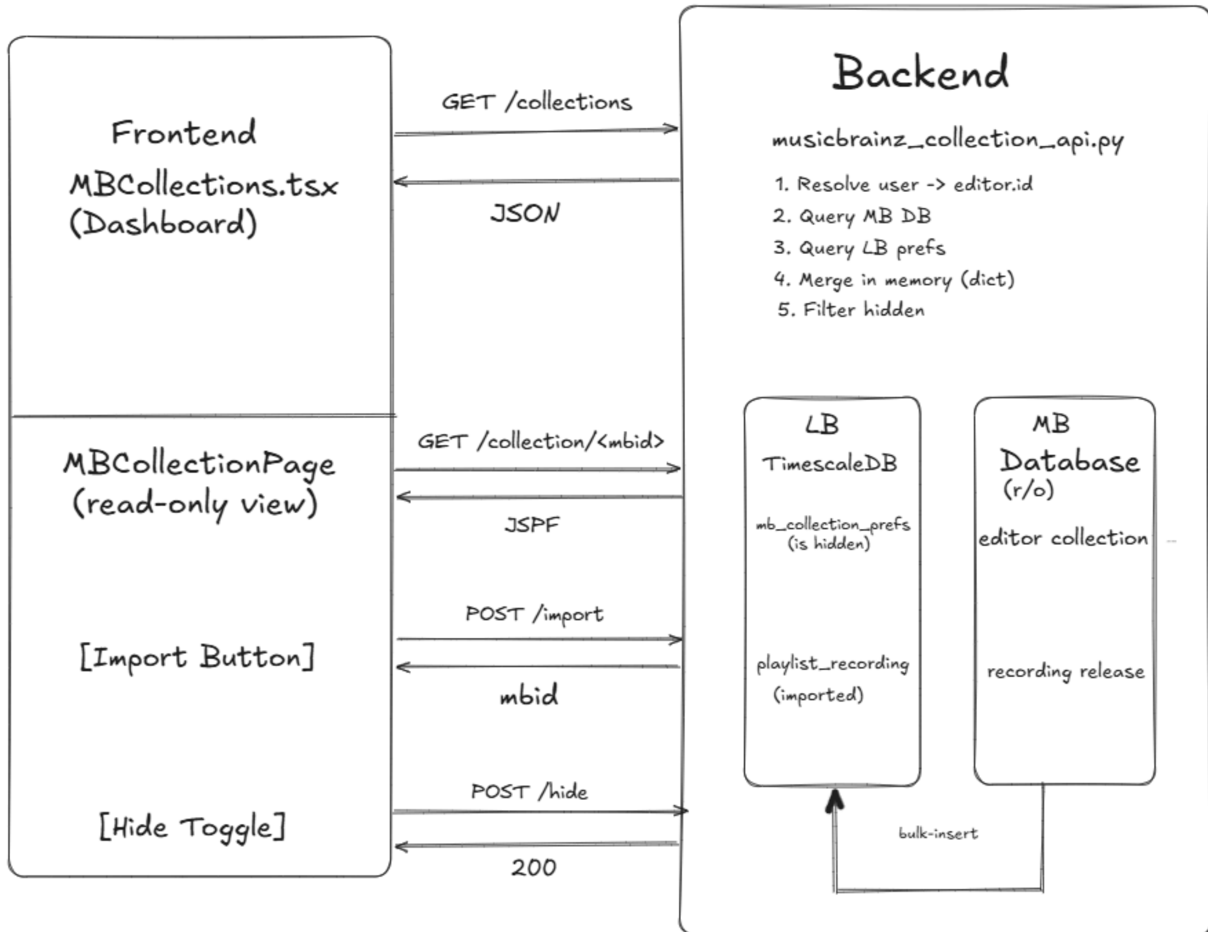
```sql
-- Schema (LB database, not MB)

CREATE TABLE playlist.mb_collection_prefs (
    user_id     INTEGER NOT NULL REFERENCES "user"(id),
    collection_mbid UUID NOT NULL,
    is_hidden   BOOLEAN NOT NULL DEFAULT FALSE,
PRIMARY KEY (user_id, collection_mbid)
);

-- Upsert
INSERT INTO playlist.mb_collection_prefs (user_id, collection_mbid, is_hidden)
VALUES (:user_id, :collection_mbid, :is_hidden)
ON CONFLICT (user_id, collection_mbid)
  DO UPDATE SET is_hidden = :is_hidden;
```

3. Return **200 OK**. The next time the collections list is fetched, hidden collections are filtered out.

Data Flow




```
READ paths:  Frontend → LB API → MB DB (read-only) → JSPF → Frontend
WRITE paths: Frontend → LB API → LB DB only (prefs, imported playlists)
             We NEVER write to the MB database.
```

1. GET /collections → LB API queries MB DB for editor_collection → returns JSON
2. GET /collection/<mbid> → LB API queries MB DB for recordings → Unpaginated JSPF (limit 2000) → returns JSON
3. POST /collection/<mbid>/import → LB Backe nd executes native bulk-insert from MB DB to LB DB → creates independent LB playlist
4. POST /collection/<mbid>/hide → Writes to LB's mb_collection_prefs table only

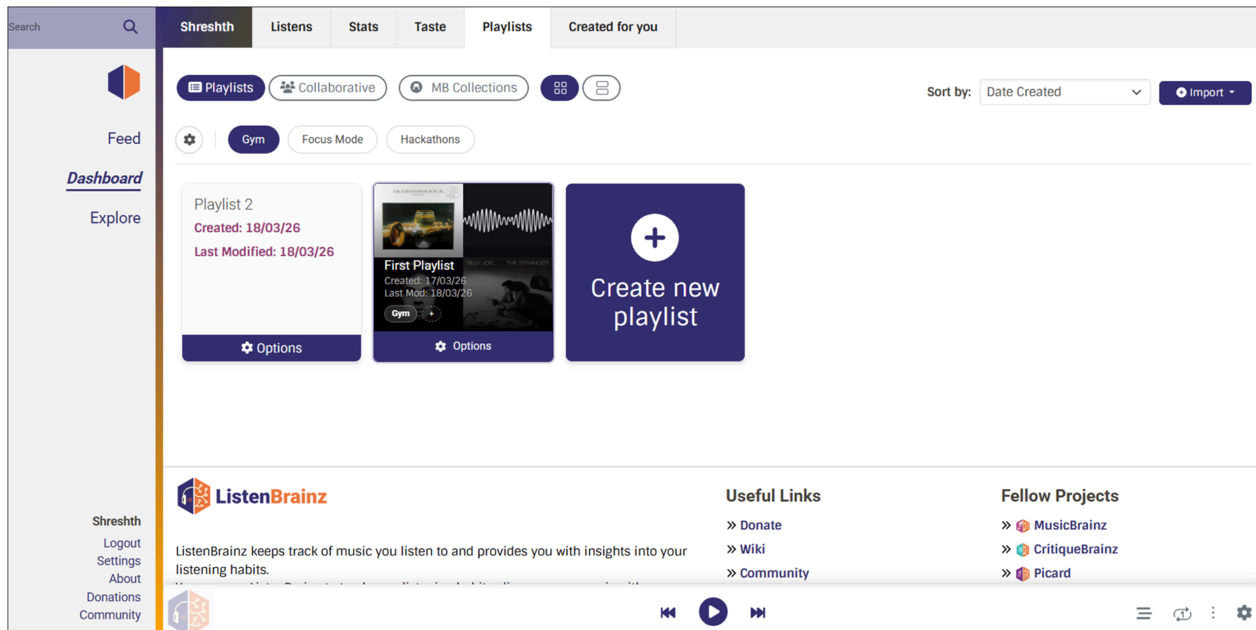
Key Design Decisions

Decision	Rationale
Direct MB SQL, not REST API	Public MB API enforces 1req/s. **MB_DATABASE_URI** is already used across **entity_pages.py**, **playlist_api.py**, **art_api.py**, etc.
JSPF output format	Reuses the existing frontend rendering pipeline. **MBCollectionPage.tsx** is a read-only variant of **Playlist.tsx**.
No caching	Mirrors existing LB patterns. MB data is served live to avoid staleness.
Separate prefs table in LB	The "hide" state is LB-specific. We never write to the MB database.
Separate blueprint	**musicbrainz_collection_api.py** keeps collection logic isolated from playlist CRUD.

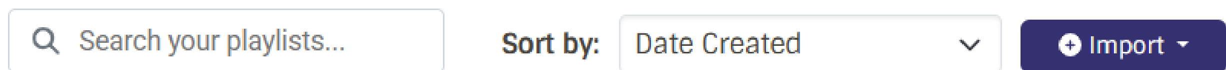
3. UI Mockups

Sub-Projects 1 & 2: Playlist Tags & Search

Entire Page with tags and search



Search



Tags (Labels) and Search Labels

Key UI Elements:

- **Search Bar:** Integrated cleanly into the top controls, enabling debounced user-level playlist search without leaving the dashboard (Sub-Project 1).
- **Horizontal Tag Filter Bar:** A space-efficient, scrollable strip of tags replacing the traditional sidebar. Preserves the main grid width while keeping tags immediately discoverable.
- **Interactive Tag Pills:** Displayed at the bottom of playlist cards, alongside a fast-action "+" icon to instantly assign new tags without digging into dropdown menus.
- **Instant Filtering:** Clicking a tag pill in the top filter bar will instantly isolate the grid to match that tag.

Tag Manager Modal

Manage Labels

Create new label...

Add

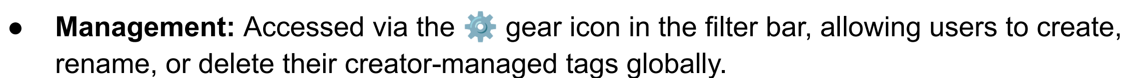
YOUR LABELS

Gym

Focus Mode

Hackathons

Close



- **Management:** Accessed via the ⚙️ gear icon in the filter bar, allowing users to create, rename, or delete their creator-managed tags globally.

Sub-Project 3: Track Ordering

View-Only Sorting Dropdown

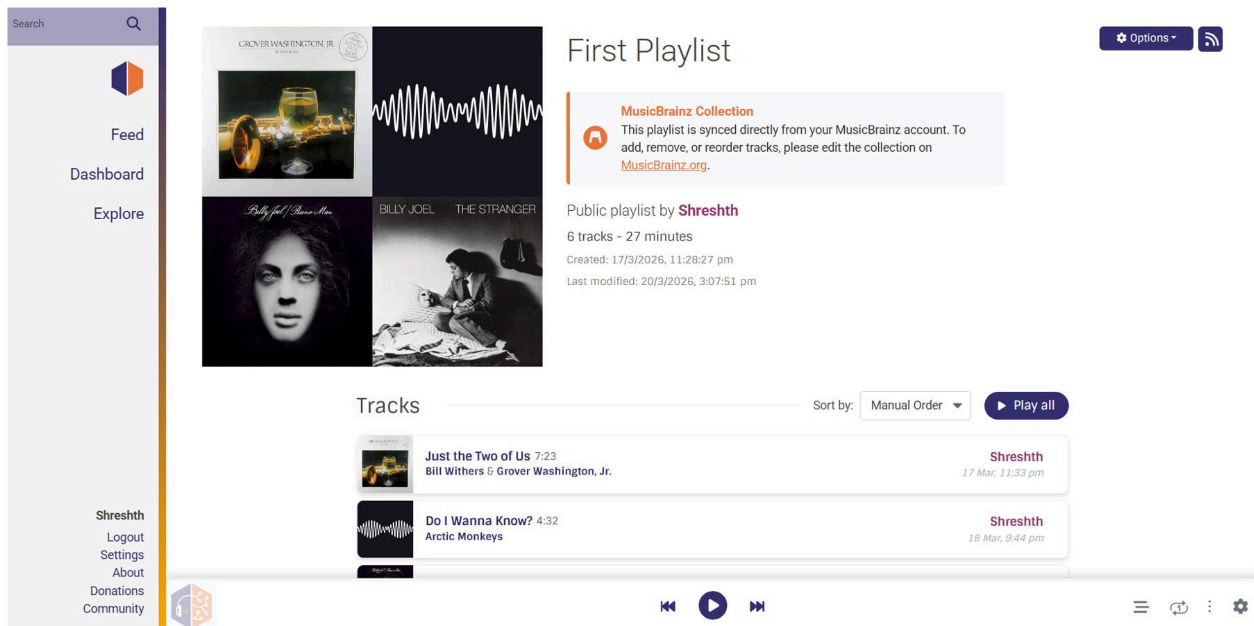
Key UI Elements:

- **Sort Dropdown:** Positioned intuitively next to the "Play all" button, providing clear dynamic sort options (Date Added, Track Title, Artist Name, Release Date, Randomize).
- **State Management:** When a dynamic sort is selected, the view becomes strictly read-only, safely disabling the drag-and-drop handles.
- **Manual Order Preservation:** Users can easily return to the "Manual Order" state to re-enable the standard **`ReactSortable`** drag-and-drop workflow.

Sub-Project 4: MusicBrainz Collections as Playlists

Collections Dashboard & Detail Views

- **Collections Dashboard**



- **Collection Detail View**

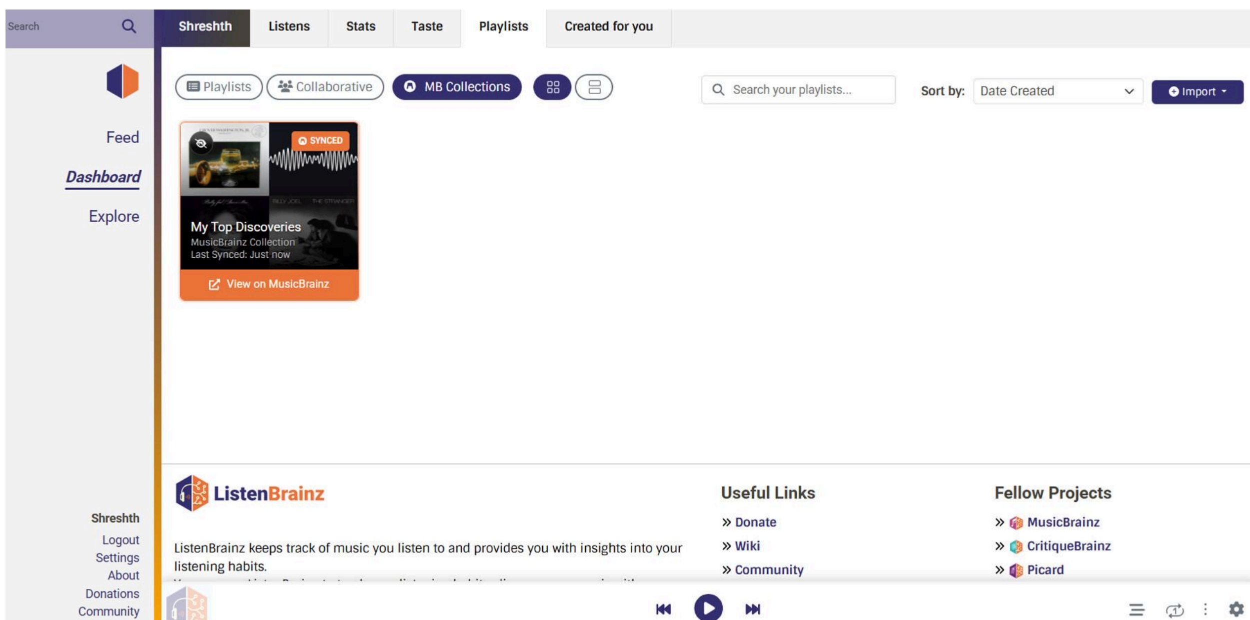
MusicBrainz Collection

This playlist is synced directly from your MusicBrainz account. To add, remove, or reorder tracks, please edit the collection on MusicBrainz.org.

Key UI Elements:

- **UI Isolation:** A dedicated "MusicBrainz Collections" tab keeps live, read-only external collections conceptually separated from standard, editable ListenBrainz playlists.
- **Actionable Cards:** Collection cards display vital metadata (item count, entity type) and feature a direct "Hide" toggle to allow users to filter out unwanted utility collections (e.g., audiobooks) from their dashboard.
- **Read-Only Rendering:** The collection detail page renders exactly like a standard playlist, but safely removes the drag-and-drop hooks and delete actions, reinforcing that the source of truth remains MusicBrainz.

Final Layout Vision: The mockup below represents the unified Playlists dashboard integrating these features. *(One thing to note: Following mentor discussions regarding frontend performance, the 4-grid SVG cover art shown on these cards is explicitly excluded from this project. As MusicBrainz returns SVGs that individually load album art, attempting to render hundreds of these images synchronously would bottleneck the client and cause a severely degraded user experience).*



4. Timeline

The timeline is structured to deliver incremental, reviewable features early, while reserving sufficient buffer for iteration and integration. The goal is to target the smaller, isolated features first (Search and Sorting), to establish a steady rhythm of reviewed and merged PRs before diving into the complex database migrations required for Labels and Collections.

Community Bonding (Weeks 0-1)

- Set up full local development environment (Docker, TimescaleDB, NodeJS)
- Meet with mentors to confirm design decisions:
    - Finalize any minor edge-case UI behaviors with mentors before coding begins
- For code familiarization and getting my hands in, I'll also begin Sub-Project 1 (search) as a warm-up

Week 1-2: Sub-Project 1 - User-Level Playlist Search (20h)

Week	Tasks
1	Modify **playlist_api.py** search endpoint to accept **user_name** param. Write backend tests for owner vs guest viewing.
2	Build **usePlaylistSearch** hook. Add debounced search bar to **Playlists.tsx**. Implement isolated pagination logic. Write Jest tests and open PR.

Deliverable: PR for user-level playlist search - under review / merged.

Weeks 3-4: Sub-Project 3 - Track Ordering (35h)

Week	Tasks
3	Focus on frontend React state. Build "Sort By" dropdown components and integrate **useMemo** hooks for client-side sorting logic.
4	Implement view-only lock for drag-and-drop handles during active dynamic sorts. Write Jest tests for component behaviors and open a PR for review.

Deliverable: PR for track sorting - under review / merged.

Weeks 5-8: Sub-Project 2 - Playlist Tags (60h)

Week	Tasks
5	Write DB migration to add **tags TEXT[]** column and GIN index to the playlist table. Implement backend array-mutation functions.
6	Implement **add_tags_to_playlist** batch API endpoint and other required endpoints. Write API tests.
7	Integrate existing autocomplete Tags component for Playlist view. Build horizontal tag-filter bar.
8	Add tag pills to **PlaylistCard.tsx**. Styling. Add frontend tests and open a PR for review.

Deliverable: PR for playlist tags - under review / merged.

Weeks 9-12: Sub-Project 4 - MB Collections (60h)

Week	Tasks
9	Investigate MB database schema mapping. Implement direct SQL queries in LB backend to fetch **editor_collection** metadata and resolve **editor_collection_recording** and **editor_collection_release** items.
10	Build **musicbrainz_collection_api.py** endpoints. Implement the **playlist.mb_collection_prefs** table for the "Hide Collection" functionality.
11	Build **MBCollections.tsx** tab and **MBCollectionPage.tsx** read-only views. Build the dedicated backend import endpoint (**POST /1/user/<user_name>/musicbrainz-collection/<collection_mbid>/import**) with bulk-insert logic.
12	Integration testing for cross-database querying. Final PR and dedicated buffer time for mentor review iteration cycles across all features.

Deliverable: PR for MB collections - under review / merged.

Communication Protocol: I will maintain regular communication with mentors through weekly updates, sharing progress, blockers, and design decisions.

Stretch Goals (Buffer Permitting)

- **Unified Playlist Search**: Extend the global pg_trgm search to include custom playlist tags from the TEXT[] column to improve discoverability.
- **Nested Tag Trees**: Extend tags into a structured model using parent-child relationships queried via recursive CTEs if hierarchical organization is required.
- **In-Playlist Filtering**: Add a client-side search bar inside Playlist.tsx to instantly filter tracks without additional API calls.
- **Collection Auto-Refresh**:Implement a periodic background refresh mechanism to maintain data freshness if collection caching is introduced.

5. About Me

Personal Information

- **Name**: Shreshth Sharma
- **Email**: shreshth013@gmail.com
- **GitHub**: github.com/Shreshtthh
- **IRC/Matrix**: shreshtthh:matrix.org
- **Timezone**: IST (UTC+5:30)
- **LinkedIn:** https://www.linkedin.com/in/shreshth-sharma-a76128280/
- **University**: National Institute of Technology, Hamirpur

Other Information

- I have an Acer Nitro V15 (16 GB RAM, i5-13420H (2.10 GHz), 64-bit operating system, x64-based processor) for the purpose of the SOC.

- I started programming first in high school, with the classic “Hello World” program. At the time, we were mostly taught algorithmic questions and OOP principles. My actual journey of building projects began in college where-in I’ve participated and built in a lot of hackathons!

- I love listening to all kinds of music. If I had to be more specific, I think pop is something I consistently enjoy but calm music is always appreciated as well. Here are some tracks I frequently visit with MBIDs
  - Do I Wanna Know? (Arctic Monkeys) - aa5db513-c948-4ed1-a86c-58c58d456af7
  - Piano Man (Billy Joel) - ad703ade-23fd-4314-911e-eefd1db59743
  - Thinkin Bout You (Frank Ocean) - 198cee24-a93c-4565-a5ca-48d9265a5f17

- Applying for LB but I’ve always been interested in reading Sci-fi, Tech, Fantasy.

Personal Projects

I spend most of my free time building full-stack applications, primarily using collegiate and national hackathons as a forcing function to learn entirely new tech stacks over a weekend. I love taking on projects that require solving complex architectural problems. Two recent examples include:

  - **ChainInsight** (GitHub - github.com/Shreshtthh/ChainInsight): Won the Web3 main track at the IQAI ADK-TS Hackathon), an AI-driven analytics platform that uses a network of Node.js/TypeScript backend agents to simulate and analyze complex on-chain data workflows, visualizing the results on a Vite/React dashboard.
  - I've also built infrastructure projects like **StreamPay** (github.com/Shreshtthh/StreamPay) [won third place at Somnia AI Hackathon] and **StealthPay** (github.com/Shreshtthh/StealthPay)), which involved extensive work with Next.js and secure data routing.
  - **ZK-Seep** (github.com/Shreshtthh/ZK-Seep), a decentralized, multiplayer version of the traditional Indian card game Seep. To ensure the gameplay was completely trustless and players couldn't cheat by looking at each other's hands, I implemented zero-knowledge proofs. This required writing cryptographic circuits in Noir, building the backend contracts in Rust, and wiring it all up to a highly interactive React/TypeScript frontend to manage the complex, real-time game state. Ended up winning honorable mention at Stellar Hacks : ZK Gaming.
- These hackathon environments taught me how to wire up full-stack applications and integrate complex APIs rapidly under intense 48-hour deadlines. However, I am actively pursuing GSoC because I want to transition out of that "rapid prototyping" mindset. Contributing to ListenBrainz has already shown me the value of respecting existing architecture, writing defensible tests, and truly understanding the *why* behind a system rather than just rushing to a finish line. I am excited to apply my execution speed toward writing clean, maintainable, production-quality code.

Relevant Experience

- **Frontend (React & TypeScript)**: Built a custom game engine in TypeScript resolving complex state synchronization (WebRTC) with sub-100ms latency. I'm highly comfortable using strict TS to keep data models predictable. Recently refactored LB React components (PR #3553).
- **Backend (Python, Node.js, PostgreSQL)**: Designed REST APIs capable of batching 500+ concurrent state updates. Comfortable abstracting Python services (PR #3587), writing performant SQL, and handling complex edge cases (N+1 queries, race conditions).
- **Prototyping**: Won 6 hackathons (Somnia AI, ADK-TS, OneHack 2.0 | Gamify Edition, Stellar Hacks : ZK Gaming) requiring rapid full-stack execution under 48-hour deadlines. I am now focused on applying this execution speed to production-grade open-source codebases.
- **Community (GitHub Campus Expert)**: Deeply passionate about open-source developer ecosystems, event organization, and mentoring which is why I really appreciate MetaBrainz culture and it strongly resonates with me.

ListenBrainz Contributions

My time contributing has given me a deep understanding of the full stack:

Merged PRs:

- **PR #3587**: LB-1889: Import loved tracks from LibreFM. Abstracted shared API logic into a new `AudioscrobblerService` base class to eliminate code duplication.
- **PR #3553**: LB-1928: UX improvements for Listen timestamps. Redesigned UX in React/SCSS by replacing clunky toggles with an always-enabled DateTimePicker grid.

Open/Reviewed PRs:

- **PR #3611**: LB-1941: Show playlist track collage as opengraph image. Built a dynamic endpoint using Pillow to compose 1280x630 PNG grids on the fly from cover art.
- **PR #3621**: LB-1836: Fix JSPF spec violations. A full-stack serialization update to enforce strict JSPF array compliance while implementing defensive client unwrapping.

Why This Project

To be honest, I haven't always been a die-hard music fan, but being involved in campus and developer communities over the last few years has really shifted my perspective. Seeing how people bond over shared tracks has made me realize how deeply personal digital music curation is. For many, music isn't just audio; it's identity. Playlists and collections are how people map out their memories and share them with the world. The philosophy of open-sourcing these listening habits and taking control of that data away from walled gardens and giving it back to the listeners, is what originally drew me to ListenBrainz. I want to help build the tools that make that curation experience as seamless as possible.

Availability

- I can commit approximately **25-30 hours per week** during the coding period.
- I have no planned vacations; I will be available on Matrix/IRC daily.
- Note on AI Tools: I aim to avoid relying on generative AI for core logic (features, queries, architecture) during this project to ensure deep, authentic learning. Every line I submit will be solely owned and understood by me. However, if they are ever used in my contributions, it will be explicitly disclosed.