Federated Music Player

Jade Keira Melody Ellis

jkme2@kent.ac.uk

School of Computing

University of Kent

June 15, 2026

Abstract

As streaming services dominate the modern music consumption landscape, listeners often find themselves constrained by heavily limited local music players and centralized platforms with limited licensed libraries that inadequately compensate artists. This report introduces a project codenamed PlayerBrainz, a federated music library server that aims to bridge the gap between local music collections and the rich metadata available through the MetaBrainz ecosystem. By storing music libraries locally and establishing a robust federation mechanism, users can connect their independently hosted servers, share access to their libraries, and discover new music collaboratively without relying on an intermediary streaming provider. This report details the design and implementation of a scalable Rust back-end and corresponding GraphQL API capable of distributed federation, paired with a SvelteKit administrative front-end.

1 Introduction

The transition from physical media and locally managed music collections to centralized streaming platforms has fundamentally altered how users discover, consume, and manage music. While platforms like Spotify offer unparalleled convenience and expansive libraries, they introduce rigid constraints. Users sacrifice ownership of their music files (with tracks frequently disappearing from user's libraries as licensing changes), and are subject to recommendation systems driven by platform profitability rather than pure artistic alignment. Furthermore, centralized streaming largely disconnects listeners from the underlying creative ecosystems, commoditising music by divorcing it from its context, authorship, and performers.

The MetaBrainz Foundation, through its interconnected databases like MusicBrainz and ListenBrainz, has chronicled detailed and intricate relationships spanning millions of musical acts globally. However, modern music players predominantly utilise a naïve schema consisting solely of simple textual fields like "Title", "Artist", and "Album", which frequently run into limitations displaying real information, omitting the rich collaborative tapestry of the music industry.

To counter these limitations, this project conceptualises and implements PlayerBrainz, a federated infrastructure designed to expose the full depth of MusicBrainz relationships directly alongside the user's audio files. The vision extends beyond a single local media player; it proposes a distributed, relationship-driven alternative to commercial streaming services. By allowing users to host their music on a private server and federating these servers using standardized cryptographic protocols, individuals can share their libraries with family and close friends, retain absolute ownership of their files, and explore the rich, multifaceted universe of music organically.

This report documents the architectural design, technological choices, and the practical implementation of the PlayerBrainz backend and frontend systems.

2 Previous Work

2.1 The MetaBrainz Foundation

The MetaBrainz Foundation is a non-profit organisation that maintains a number of complementary projects related to music metadata. These were deemed of particular relevance to the project, and are described in more detail below.

2.1.1 MusicBrainz

MusicBrainz (Swartz, 2002) is a collaborative structured database of music metadata. It is the most comprehensive and widely used music metadata database, and is the basis for many other projects in the music metadata space.

The MusicBrainz database schema (The MusicBrainz Authors, 2025) provides a structured and accurate representation of music metadata as encountered in the real world. Naïve approaches to music metadata often assume a simple schema where each track has a single title, artist and album, and that albums have a single artist - each of these would be keyed by its human readable name. In reality, music metadata is much more complex. In contemporary music, it is common for tracks to have multiple artists, and for the performing name of artists to change over time. Credits are often done in unique formats. In addition to this, the same track may be released on multiple albums or singles, with multiple different recordings. The MusicBrainz schema accounts for this complexity and provides a structured way to represent it. It also provides a way to link to external databases, including streaming providers. For this reason, it was chosen as the basis for the schema used in this project, with some adjustments to account for the particular use case of on-disk music files.

2.1.2 MusicBrainz Picard

MusicBrainz Picard is an application for tagging and organising audio files using metadata from the MusicBrainz database. It uses multiple techniques, including audio fingerprinting techniques, to match audio files to corresponding entities in the MusicBrainz database, and then adds metadata to them according to configured rules. The project maintains a schema of tag mappings (The Picard Authors, 2026), which can be used by third party programs. Notably, Picard embeds the MusicBrainz identifiers (MBIDs) of several entities by default.

2.2 Music Servers

Other programs exist which provide the basic functionality of exposing a music library over the network with an API for the purpose of streaming. However, most use somewhat naïve approaches to music metadata, with many of the shortcomings listed earlier. Additionally, only Funkwhale (Berriot, 2017) has any kind of federation functionality - however, it has a particular focus on publicizing uploaded music using the ActivityPub social protocol.

2.2.1 Subsonic

These music servers have a *de facto* common API known as the Subsonic API (The Subsonic Authors, 2005). This is a JSON and XML API providing a wide variety of related endpoints and functionality for music, podcast and other audio streaming. The OpenSubsonic project (The OpenSubsonix Authors, 2023) is an attempt to formalise this to deal with some of the ambiguities and limitations of the Subsonic API. However, it was decided that avoiding implementing this API would provide greater flexibility to experiment with this project, and it was not necessary for the goals of the project.

2.3 Federated Protocols

In modern federated software, there are two primary relevant protocols – the Matrix protocol (The Matrix Authors, 2026), and the ActivityPub protocol (Lemmer-Webber et al., 2018). Both are comprehensive standards including key parts such as server discovery, HTTP signatures, user identifiers and more to provide everything needed to implement software in their respective ecosystems. The Matrix protocol has a focus on synchronizing state between untrusted servers, with byzantine fault tolerant properties. The ActivityPub protocol focuses on subscription & replication of structured data from the originating servers.

Given that the application is intended to synchronize data between trusted, cooperating systems, it was decided that neither protocol could be used in its existing form.

3 System Overview

3.1 Back-end

The server is required to return large amounts of deeply nested data in arbitrary structures. For example, a music library page might need to display various tracks, albums, releases, and other related metadata. Because each potential screen design needs to display different subsets of data (e.g., some views require the track year or album while others do not), and because ideally third party apps would be able to use the API, it would be impossible to design traditional REST endpoints that deeply returns all necessary data without also returning a significant amount of unnecessary data. Conversely, relying on smaller, limited endpoints would necessitate a large number of API requests to populate a single page, resulting in a slow and poor user experience.

GraphQL was selected to solve these problems. It allows clients to query exactly the data they need in one request, including deeply nested struc-

tures. When examining the Rust ecosystem for GraphQL implementations, `async-graphql` appeared to be the most popular and established asynchronous library. It provided all the necessary features and functionality for the application and was therefore chosen.

For database interactions, SeaORM was selected. It is the leading async-native Rust object-relational mapper (ORM) and is built upon SQLx, which is the leading asynchronous SQL database client library for Rust.

3.2 Front-end

To demonstrate the API provided by the back-end, a front-end application was necessary. The SvelteKit meta-framework was chosen based on existing developer familiarity and the known positive performance properties of its implementation. The user interface was built without the use of a CSS framework, relying on handwritten basic styles.

To integrate with the back-end API, a SvelteKit-compatible GraphQL client library named Houdini was selected. Initially, Houdini provided everything necessary, featuring ergonomic code generation and context drilling. However, a significant limitation was discovered later in the project: when Houdini is used with SvelteKit, it relies on one global Houdini instance. As a result, it can only connect to a single GraphQL endpoint at a time, making logging into and querying arbitrary selfhosted servers significantly more difficult.

3.3 Federation

When making federated requests between instances, a robust way of authenticating the server making the request is needed, in addition to the standard transport-layer security (TLS) authentication provided for the response. Mutual TLS was considered but was deemed impractical due to a lack of broader ecosystem support and the difficulty of acquiring certificates issued by the existing public Certificate Authority infrastructure.

Other request signature schemes were examined, leading to the decision to use RFC 9421 HTTP Signatures. This standard is well-specified, and an existing Rust implementation, the `httpsig` library, was available to be adjusted and integrated into the application. Using a standardised RFC also provides potential future interoperability benefits, even if those are not immediately visible at this stage of the project. The `cynic` GraphQL library is used as the client in the back-end to execute these federated queries, and the application utilises a '.well-known' scheme for endpoint discovery.

4 Implementation

4.1 Schema

The MusicBrainz schema provided a strong foundation for the application. However, it had a few limitations. Primarily, the MusicBrainz Recording entity was not granular enough to match an actual on-disk recording, because a Recording would be the same regardless of encoding, bitrate, or starting periods of silence. It was decided that a new entity called a 'Mastering' would be created, with each instance of a Mastering being sample-for-sample identical. As a simplification, each file on disk would be its own unique Mastering.

In addition to this, some adjustments to the positioning of cover art and lyrics were made. Cover art corresponded to a release, however on-disk music files can additionally have media embedded within the file, which must be represented. Lyrics are not present in the MusicBrainz schema, and although the Work entity is technically the most correct place to store them, they are attached to a Mastering in typical filesystem representations - even ignoring that not all Recordings have had corresponding Works created and linked.

4.2 Scanner

Producing a robust and efficient scanner was a particular point of difficulty and iteration. The scanner was originally prototyped as a routine that recursively walked the music directory, upserting music files as it found them. However, this handled two important cases poorly. It could not handle music files being inserted or updated during the lifetime of the server, and it could not handle music files being deleted. For this reason, a new design was created. It also had a number of other limitations, like only being able to handle a single on-disk library.

Particular care was taken regarding filesystem watching. It is known that watching large numbers of files is not reliable (Lar, 2022), and the server may be offline for periods of time. Therefore, an incremental update scheme was chosen that could be run from scratch, and was resilient to file watch events being incomplete or incorrect.

The new design comprised of two parts. The first part was the file reader process. This process first recursively scanned each directory it was configured to watch. It would simultaneously initialize a filesystem watcher, which would start processing events once the recursive scan was done. Directories would be walked in a depth-first manner, sending first the items found in a directory down a channel, and then upon the completion of a particular directory an event marking that directory as complete. The watcher would then mark particular directories as modified and re-scan the subdirectory.

The second part was a state machine that received events from the reader process. Relying on the precondition that the channel was a serial depth-first search of the music file, it would upsert the Mastering entities into the database and collect other data. Upon receiving the directory complete notification, it would remove entities that had not been sent, with the assumption that they had been deleted, and finalize the transaction with any other data.

Of some note were the issues regarding extracting the tags from the metadata. The upstream library used for audio container parsing seemed to have inconsistent support for metadata across file types, with some man-

ual parsing required to consistently extract recording IDs from mp3 containers. Additionally, some tags appeared to be mapped incorrectly, for example `MusicBrainzRecordingId` never appeared in real files, while `MusicBrainzTrackId` contained the actual recording ID. For this reason, the tag extraction code is somewhat more complex than it would otherwise be, and may need to be adjusted in the future if the upstream library is updated.

4.3 API

The project exposed a discoverable and introspectable API using the GraphQL protocol and conventions. This API provided basic user management, library management and federation functionality. This API is unified between unauthenticated, authenticated local access, and authenticated federated access. Each subsection of the graph had access control configured as appropriate. Local user authentication followed a simple session token scheme, with defined expiration.

4.4 Federation

The federation protocol took some superficial inspiration from the Matrix protocol, exposing files at well-known endpoints to direct federated servers to the correct API endpoint. Remote servers then make an unauthenticated request to that endpoint to retrieve one or more ed25519 keys associated with the server. Once those keys are retrieved and stored, they may then be used to authenticate requests from that server. However, this was where the similarities ended. The signature scheme used was an RFC 9421 signature of the request with a sha512 digest of the body, rather than a custom HTTP signature scheme, and the federation API was exposed over a GraphQL API described earlier rather than a RESTful JSON API.

Requests to the API are authenticated using signed requests. The server fetches the public keys of the requesting server by first requesting the JSON file under the `/.well-known/playerbrainz/server` path on the host. That

responds with a JSON object containing a URL under the `graph_endpoint` property, from which it can request the appropriate keys using the GraphQL protocol. Using these keys, the request is then checked, and the server can be authenticated. If the server is authorized to access an endpoint, it can then be used.

5 Results

The project successfully delivered a robust foundation for a federated music library system, alongside a functional administrative front-end. Significant focus was placed on the core systems, establishing a strong base for future development.

5.1 Backend

The Rust-based back-end server successfully implements a GraphQL API leveraging existing ecosystem libraries where appropriate.

A prominent achievement of the backend is the implementation of the resilient file scanner. The scanner reliably reads the local filesystem, extracts advanced metadata conforming to the adjusted MusicBrainz schema, and tracks incremental changes.

5.2 Federation

Federation functionality was implemented according to the designed architecture. The system successfully exposes endpoints for node discovery via the `/.well-known/playerbrainz/server` path. Remote servers are able to request necessary public keys, and subsequent cross-instance queries are securely verified using RFC 9421 HTTP Signatures. This allows authenticated exchange of data between cooperating servers, enabling the core vision of a federated music library ecosystem.

5.3 Front-end Administration

The SvelteKit front-end provides the basic administrative user surface for the server, using Houdini to drive state management and GraphQL API connectivity. This includes authentication and basic CRUD functionality for managing the users and libraries on the server.

While front-end prototypes for actual music playback and track listing were explored (visible in isolated components of the codebase), they were deferred in the final artifact in favour of establishing a solid federation and API core. This is further discussed in the Future Work section.

6 Future Work

6.1 MusicBrainz Enrichment

One strongly considered functionality of the project was the ability to enhance the API with metadata retrieved from the MusicBrainz database. This would have allowed an integrated and accessible way of using that information in downstream apps.

Due to limitations and strict rate limits in the MusicBrainz API, functionality to retrieve & synchronise metadata with the MusicBrainz database was not developed. A new MusicBrainz API that would relieve these limitations is in development as a part of Google Summer of Code 2026, but would not be completed by the conclusion of this project. Using a local mirror of the database was considered as an alternative solution, but it was decided that it would be impractical due to the large download size, and it would be better to wait for other work to mature.

6.2 Unified library API

During development, an API over the local library was exposed. However, this was removed as federation functionality was developed as it only adequately exposed local, already-downloaded music files. Developing an API

that consistently exposes remote and local media would make the project more appealing to developers and end users.

6.3 ListenBrainz Integration and Recommendations

A significant future goal for PlayerBrainz would be to more tightly integrate with the ListenBrainz project. ListenBrainz provides a rich source of user listening history, loved/hated tracks, and social feeds. By natively importing this data, PlayerBrainz could offer personalized recommendations and social features that enhance the user experience and increase music discoverability. Further on from that, using learnings from the Troi recommendation framework, PlayerBrainz could implement music recommendation and radio features that provide functionality familiar to users from commercial streaming services.

7 Acknowledgment

Robert Kaye, who never got to see the end of this project. Dan Bard, who has been a great support throughout this. The rest of the MetaBrainz team, who have patiently put up with my nonsense.

References

(2022). Large scale watching of paths appears to drop/miss events · Issue #412 · notify-rs/notify.

Berriot, A. (2017). Funkwhale.

Lemmer-Webber, C., Jessica, T., Shepherd, E., Guy, A. and Prodromou, E. (2018). ActivityPub.

Swartz, A. (2002). MusicBrainz: A semantic Web service. *IEEE Intelligent Systems*, 17(1), pp. 76–77.

The Matrix Authors (2026). Matrix Specification version v1.18.

The MusicBrainz Authors (2025). MusicBrainz Database Schema. https://wiki.musicbrainz.org/MusicBrainz_Database/Schema.

The OpenSubsonix Authors (2023). OpenSubsonic. https://opensubsonic.netlify.app/.

The Picard Authors (2026). Appendix A: Tag Mapping. https://picard-docs.musicbrainz.org/en/latest/appendices/tag_mapping.html.

The Subsonic Authors (2005). Subsonic. https://www.subsonic.org/pages/api.jsp.