FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	System for Management of Personal Music Libraries using Knowledge Graphs Technology
Student:	Bc. Ondřej Viskup
Supervisor:	Ing. Milan Dojčinovski, Ph.D.
Study program:	Informatics
Branch / specialization:	Web Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

With the growing popularity of music streaming platforms several problems are starting to arise. Portability of information across streaming services is limited. Synchronization of information among different services is almost impossible. Last but not least, local music user libraries are not well integrated with the online music platforms. The ultimate goal of the thesis is to create a centralized system for music library management using the concept of Knowledge Graphs. The system will enable users to integrate and synchronize music libraries across different music platforms.

Objectives:

- Identify relevant data sources from the music domain (e.g. Spotify, local music libraries, YouTube).
- Create an RDF model for the music domain and create a user-centric, music-oriented knowledge graph by integrating various data sources.
- Enhance the knowledge with information from related knowledge graphs, such as DBpedia or Wikidata.
- Design and implement a system for management of music knowledge graphs.
- Implement a lightweight user interface for the system.
- Test and document the developed solution.

Electronically approved by Ing. Jaroslav Kuchař, Ph.D. on 15 December 2023 in Prague.

Master's thesis

SYSTEM FOR MANAGEMENT OF PERSONAL MUSIC LIBRARIES USING KNOWLEDGE GRAPHS TECHNOLOGY

Bc. Ondřej Viskup

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Milan Dojčinovski, Ph.D.
May 9, 2024

Citation of this thesis: Viskup Ondřej. *System for Management of Personal Music Libraries using Knowledge Graphs Technology.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Figures

List of Tables

I would like to thank my supervisor for his valuable advice and insights during the preparation of my thesis assignment. I would also like to thank my girlfriend, family and friends for their support during my studies.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.
I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work pursuant of Section 60 (1) of the Act.

In Prague on May 9, 2024

Abstract

With the rising popularity of multimedia streaming services, the challenge of user data storage becomes more prominent. The main objective of this thesis is to design and develop a system that allows users to manage music libraries and playlists outside streaming services and integrate them with various services that provide additional information through Knowledge Graphs technology and Linked Data. The system consists of a back-end server, front-end client, RDF store, and database for user credentials. Furthermore, it outlines the design of the system, which includes the system architecture, components of the back-end server and front-end client, and RDF store. It also reviews current solutions and relevant technologies, including the Semantic Web, the RDF, the SPARQL, and several music ontologies. Finally, software validation was performed and evaluated against the set requirements, and future directions were suggested.

Keywords Semantic Web, RDF, SPARQL, ontology, knowledge graph, knowledge base, Wikidata, web application

Abstrakt

S rostoucí popularitou multimediálních streamovacích služeb se do popředí dostává problém ukládání uživatelských dat. Hlavním cílem této práce je navrhnout a vyvinout systém, který uživatelům umožní spravovat hudební knihovny a playlisty mimo streamovací služby a integrovat je s různými službami, které poskytují další informace prostřednictvím technologie znalostních grafů a Linked Data. Systém se skládá z backendového serveru, frontendového klienta, úložiště dat ve formátu RDF a databáze pro přihlašovací údaje uživatelů. Dále je zde nastíněn návrh systému, který zahrnuje architekturu systému, součásti backendového serveru, frontendového klienta a úložiště dat ve formátu RDF. Rovněž nabízí přehled současných řešení a příslušných technologií, včetně sémantického webu, RDF, SPARQL a několika hudebních ontologií. Nakonec byla provedena validace systému, který byl vyhodnocen na základě stanovených požadavků a byly navrženy podněty pro budoucí práci.

Klíčová slova sémantický web, RDF, SPARQL, ontologie, znalostní graf, znalostní báze, Wikidata, webová aplikace

List of abbreviations

API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CORS	Cross-Origin Resource Sharing
CSV	Comma-separated values
FOAF	Friend-of-a-Friend
GUI	Graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IRI	International Resource Identifier
ISO	International Organization for Standardization
ISRC	International Standard Recording Code
JSON	JavaScript Object Notation
OWL	Web Ontology Language
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
SPARQL	SPARQL Protocol and RDF Query Language
TSV	Tab-separated values
UI	User interface
UML	Unified Modeling Language
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
XML	Extensible Markup Language
XPath	XML Path Language
XSLT	eXtensible Stylesheet Language Transformations
W3C	World Wide Web Consortium

Chapter 1
Introduction

1.1 Motivation

In recent years, the use of multimedia streaming services has increased and the number of active users continues to increase. Service providers such as **Spotify**, **Apple Music**, **Youtube Music**, etc. are competing to attract even more users to use their services.

As online streaming becomes increasingly popular, there is the dilemma of storing user data. Users who do not have complete control over their data may encounter some problems. Although the services that users are actively using are currently working perfectly, there is no guarantee that they will continue to work this way for years to come. If the service provider suddenly stops the platform, the user's music library and the playlists can be lost forever if they are not properly stored. Backup of user data is an issue, and service providers rarely provide backup options. Another problem is integration. If the user uses multiple platforms, the music library may have duplicates, incorrect data, or can be, in general, unorganized. If music comes from different sources (streaming platforms, audio files from user devices, or digital albums purchased online), it is almost impossible to synchronize the user's music library on multiple platforms.

In addition, initiatives or community projects such as **Wikidata**, **DBpedia**, or **MusicBrainz** provide extensive information about artists, albums, recordings, and music in general. Integrating these services into the user's music library can provide valuable and interesting information about the user's music preferences and listening patterns.

1.2 Thesis Objectives

The main objective of this thesis is to create a system that allows users to manage music libraries and playlists outside streaming services and integrate them with various services that provide additional information through the Knowledge Graphs technology. By doing so, users can fully control their music collection data, regardless of the service they use. Moreover, users will be able to integrate their music library with existing knowledge graphs to access more information about their favorite albums, artists, and tracks.

1.3 Thesis Structure Overview

In the second chapter, the essential knowledge required to grasp the thesis topic will be explained. This section delves into the areas of music and knowledge graphs, illustrated with examples. It also reviews current solutions and relevant technologies including the *Semantic Web*, *RDF*, *SPARQL*, and several music ontologies.

Subsequently, the third chapter discusses the analysis of requirements, which involves identifying the intended audience and different use case scenarios. This section specifies both the functional and non-functional requirements that are crucial for the development of the system. In addition, it outlines the design of the system, which includes the system architecture, components of the back-end server, *RDF* store, database for user credentials, and front-end client.

The fourth chapter details the implementation process. The chapter discusses the organization, structure, and the development of the code for the back-end server and the front-end client. In addition, it covers aspects such as error handling, data type libraries, *RDF* store configuration, and integration of external data sources such as *Wikidata*.

The fifth chapter evaluates and discusses the developed software, pointing out the strengths and limitations of the system. Furthermore, it explores potential areas for future enhancement and expansion of the system.

The thesis is ended by the *Conclusion* chapter which summarizes the key contributions and the findings of the thesis.

Chapter 2

Background and Related Work

2.1 Music

Music as a concept can be described as the art of arranging sounds over time to produce continuous, coherent, and stimulating compositions such as melody, harmony, and rhythm. It can also be any aesthetically pleasing or harmonious sound or a combination of the above-mentioned sounds. [1]

The origin of the word *music* can be traced back to the ancient Greek word *mousike*, which comes from the word *Moûsa*, an ancient Greek deity of the arts. [2]

2.2 Knowledge Graphs

Although the term knowledge graph is one of the fundamental concepts of the Semantic Web, there is no single definition of it.

According to *IBM*, knowledge graphs, also known as semantic networks, represent the network of entities in the real world, such as objects, events, situations and concepts, and explain the relationship between them. This information is usually stored in a graph database and is referred to as a graph structure. Knowledge graphs are composed of three main components: *nodes*, *edges*, and *labels*. Any object, place, or person can be a node. The edge defines the relationship between the nodes. [3]

Furthermore, knowledge graphs combine several powerful data management technologies such as **databases** (because data can be retrieved through complex queries), **graphs** (because data can be analyzed using graph algorithms), and **knowledge bases** (because data can be interpreted as their semantics are embedded in them). In addition, knowledge graphs comply with the standards set by the World Wide Web Consortium (W3C) in the Resource

Description Framework (RDF), which allows extensive data representation using RDF schemas, taxonomies, and vocabularies. Finally, knowledge graphs are optimized for querying through very large datasets that contain billions of facts or properties. All of this makes them one of the most suitable frameworks for data integration, unification, linking, and reuse. [4]

In practice, knowledge graphs are used to analyze financial markets, to create contextually aware recommendation systems, for semantic search engines, business data analysis or to build and enhance conversational AI chatbots used in various fields such as tourism, energy, financial technology, education, etc. [4][5]

Some examples of frequently mentioned knowledge graphs are: **DBpedia** (2007), **Wikidata** (2012), Google's **Knowledge Graph** (2012), Facebook's **Entity Graph** (2013), or Microsoft's **Satori** (2016). [6]

Long-standing efforts are also being made to create a knowledge base that captures general information about artists, songs, albums, and music-related data in general, including the relationships between them. In the following section, some of the selected projects that evolve around the creation of a music knowledge base will be briefly introduced.

Knowledge bases can be divided into two different groups depending on the organization behind the knowledge base and the ease of access to data. The first group is **open data sources**, characterized by the fact that there is often a community or non-profit organization behind them, which often provides data without restrictions for any purpose. The open data sources are then divided into two subgroups: *general knowledge bases* and *music-oriented knowledge bases*. The second group is **industry API**[1]**s**, which are often maintained by private companies and often do not allow access to the entire knowledge base at once, but only on demand. This can be done either by registering the application that consumes the API or by using an API key to both of which various constraints can be applied (data categories, number of queries per second, etc.).

2.2.1 Open Data Sources

Wikidata

Wikidata, a member of open data sources, was created in October 2012. It is a collaborative, multilingual, community-driven project whose objective is to collect structured data to provide a knowledge base to anyone in the world. *Wikidata data* are free and published under the *Creative Commons* license, which allows copying, modifying, and distributing without asking permission. The data are curated by the editors, but anyone can contribute. *Wikidata* claims to be a *secondary knowledge base*, meaning that any statement in it is

[1]**Application Programming Interface (API)** is an application interface that provides data and functionality to other applications. **IBM: What is an API?** [access. 2024-04-17]

supported by its source and connection to different databases. The main use case of *Wikidata's knowledge base* is to support other applications from the *Wikimedia movement* such as *Wikipedia*, *Wikimedia Commons*, and the *other wikis*. Other than that, it can serve as a basis for creating a custom knowledge base, using a subset of *Wikidata's knowledge base*. [7]

As of *April 2024*, there are more than **1.55 billion** item statements in *Wikidata's knowledge base*[2], making it one of the largest knowledge bases in existence.

DBpedia

Another member of the open data source group is **DBpedia**. It was created in 2007 as an academic project by Sören Auer, Jens Lehmann, and Christian Bizer, together with the support of the company *OpenLink*. Since then, *DBpedia* has offered structured information to anyone who can access the Internet. It is a crowd-sourced community effort aimed at extracting structured content and information created in various *Wikimedia* projects. In addition, *DBpedia* serves the information as *Linked Data*, enabling information to be easily collected, organized, searched, and utilized. [8]

The largest *DBpedia knowledge graph*, called *DBpedia Largest Diamond*, contains **1.45 billion** triples from *DBpedia* and other sources. [9]

MusicBrainz

The last open source to be presented in this chapter is **MusicBrainz**. While the other two above-mentioned knowledge bases are general knowledge bases, *MusicBrainz* is a knowledge base focused on the music domain. It is an open-source music information encyclopedia produced and maintained by the international community of music fans who appreciate both music and music metadata. Its database stores all kinds of information about music, from artists and their releases to works and composers, and much more. Most of the data are licensed under the *CC0* license, which does not impose any restrictions on who and for what purpose is accessing and using the data. [10]

2.2.2 Music Services

There are several services which offer some information and metadata about music along with their core services, such as *music streaming*, *selling music*, *connecting music enthusiasts*, etc.

Last.fm

One of the most used service providers when it comes to accessing music domain-specific knowledge bases is **Last.fm**. *Last.fm* offers its users the ability

[2]from **Grafana** [access. 2024-04-17]

to track the music they listen to. Users can run the service in the background and collect information about the songs they listen to. Users can then view various statistics about their listening sessions, including their top tracks, artists, and albums. *Last.fm* also uses the data it collects to make music recommendations that users might like. [11]

To be able to access the *Last.fm's API*, one must create an *API account*. The *Last.fm's API* provides endpoints to access information about albums, artists, tracks, most popular artists, and tracks (globally and by country) and its main focus — the information about what music its users are listening to including their favorite albums, artists, and tracks. [12]

Spotify

Another company offering access to its music knowledge base is **Spotify**. *Spotify* is an audio streaming platform with more than **100 million** tracks and more than **600 million** users. Users can create and manage playlists and discover new tracks. [13]

There are multiple *APIs* that developers could access. To access its music knowledge base, Spotify provides *Web API* through which information about albums, artists, genres, playlists, and tracks can be accessed. In addition, users are able to manipulate their playlists using the *API*, including creating playlists, adding or removing a track to or from it, or deleting it. Another functionality which could user control through the *API* is music player, including obtaining recently played tracks. [14]

Other notable streaming services include **Youtube Music**, **Apple Music**, **Deezer**, and **Tidal**. Each provides an *API* for accessing music metadata.

Bandcamp

Another group of services are services for selling music. **Bandcamp** is a digital music platform and community where music fans can explore, engage with, and directly support their favorite artists. The website states that on average more than 80% of the money spent on music goes directly to the artists or their label. *Bandcamp* has an *API*, but as of April 2024, it currently does not support access to music metadata. [15]

Another prominent platform for music sales is **Discogs**. This service operates as a marketplace for physical album formats such as CDs, DVDs, vinyl, and cassettes, and as a community forum. Additionally, *Discogs* provides a public *API* for accessing music metadata.

2.3 Recommendation Systems

Recently, there have been many articles and papers on the idea of using music knowledge graphs in the field of content recommendation systems. This thesis will not focus on aspects of recommendation systems, but only on datasets that researchers used for their experiments. Most of the works mentioned using the *Last.fm data set* as one of the main sources for their knowledge graphs. Other sources used were *MusicBrainz* or the *Freesound dataset*. However, since the main aim of the recommendation system is to recommend the user the content that is the most similar to the content they enjoy, researchers prioritized the creation of the datasets which reflect the relationship between the item (in this case, music recordings, albums, artists, etc.) instead of focusing on the creation of the more general music-oriented dataset. [16][17]

2.4 Existing Solutions

In this section, current technologies in the music domain that manage music libraries will be examined. Depending on the main capabilities, the technologies can be divided into five different categories: *multimedia players*, *MP3 tag managers*, *playlist transfer services*, *cloud-based music services*, and *discovery and recommendation platforms*.

Multimedia Players

Multimedia players are essential for managing and organizing music libraries, often serving as the main interface for users to engage with their music collections. Numerous multimedia players provide extensive library management tools that enable users to arrange their music directly in the application. Features may include the ability to create playlists, organize tracks by different criteria (e.g. artist, album, genre) and handle metadata (e.g. modify tags, album art). Typically, multimedia players show metadata tags for each track, detailing the artist, album, track title, genre, and album art. This metadata can often be edited by users within the player, facilitating the correction of errors or the addition of new details to enhance organization.

Some examples of *multimedia players* include:

- **Foobar2000** — Provides comprehensive library management options, featuring support for multiple metadata formats, a customizable interface, and sophisticated playlist management tools like dynamic playlists and playback statistics tracking. In addition, it supports numerous audio formats and benefits from a robust community of users who create plugins to enhance its features. It is compatible with *Windows*, *MacOS*, and mobile platforms (*Android*, *iOS*).
- **Clementine** — Provides an easy-to-use interface coupled with library management capabilities. This includes compatibility with online music platforms like *Spotify* and internet radio stations, along with tools for generating dynamic playlists, modifying metadata, and acquiring album art-

work. It supports various operating systems, including *Windows*, *MacOS*, and several *Linux distributions* such as *Ubuntu*, *Fedora*, and *Debian*.

- **iTunes** — Apple's multimedia platform and digital storefront. It provides music library management tools, such as playlist creation, track organization by artist, album, and genre, and synchronization with *iOS* devices. In addition, it includes support for buying and downloading music from the *iTunes Store*, and for using the *Apple Music* streaming service to listen offline.
- **VLC Media Player** — Flexible multimedia player compatible with numerous audio formats. Although primarily designed for video playback, it also provides music library management tools, including playlist creation, metadata-based track organization, and album art visualization. This player is lightweight, supports multiple platforms (*Windows*, *MacOS*, *Linux*, *Android*, and *iOS*), and is free to use.

MP3 Tag Managers

MP3 tag managers are specialized software designed to efficiently organize and modify the metadata of audio files. Details on MP3 metadata are further discussed in Section **2.5.8**. These tools allow for manual adjustments of metadata including song title, artist, album, and genre, promoting precision and personalization in music collections. They support *batch editing*, enabling modifications across multiple files at once, which simplifies managing extensive music libraries. These applications typically feature automated tagging options, using online databases like *MusicBrainz* or *Discogs* to automatically fetch and update metadata, thereby improving both efficiency and uniformity. Moreover, their integration with online databases provides access to the latest metadata, and capabilities to export/import metadata in formats like *CSV/TSV*[3] or *XML*[4] enhance the ease of transferring and backing up metadata across various libraries or platforms.

Some examples of *MP3 tag managers* include:

- **Mp3tag** — Efficient and easy-to-use MP3 tag editor compatible with *Windows* and *MacOS*. It enables batch editing of metadata and retrieval of metadata from online sources like *MusicBrainz* and *Discogs*, promoting both precision and completeness. In addition, it provides capabilities such

[3]**Comma-separated values (CSV)** or **Tab-separated values (TSV)** are text files formats that uses commas or tabs to separate values, and newlines to separate records. **CSV or TSV** [access. 2024-04-18]

[4]**Extensible Markup Language (XML)** is a markup language that allows developers to define arbitrary tags - tags that can be used to build a tree structure. It is a popular format for searching, sharing, and backing up data. **XML introduction — Mozilla** [access. 2024-04-18]

as personalized scripting for automated tasks and broad file format compatibility for different audio types.

- **MusicBrainz Picard** — Cross-platform, open-source tool for MP3 tagging that employs *acoustic fingerprinting* technology *AcoustID* to accurately identify and tag audio files. This tool is capable of automatically aligning audio files with the *MusicBrainz* database to fetch metadata such as artist, album, and track title. Additionally, *Picard* facilitates batch processing and scripting to customize tagging processes, and it offers plugins to enhance its features.

- **TagScanner** — Application designed for the management and organization of music collections on *Windows*. It includes numerous functionalities for batch metadata editing, such as formatting tags, renaming files using tag data, and acquiring album artwork from the Internet. *TagScanner* accommodates multiple audio file formats and offers features to normalize and refine metadata in music collections.

- **Kid3** — An open-source MP3 tagging tool that works with *Linux*, *Windows*, and *macOS*. This tool offers a user-friendly and clear interface for editing metadata, with features for mass editing and synchronizing tags. *Kid3* accommodates multiple audio formats and allows for customization of tag fields and templates according to user requirements.

Playlist Transfer Services

Playlist transfer services provide a range of functionalities that simplify the movement of music collections between various streaming services. Bulk playlist transfers are also possible, with features such as duplicate removal and playlist consolidation. These services support major platforms like *Spotify*, *Apple Music*, *Deezer*, and *Tidal*. Customization features are abundant, offering users the ability to adjust the transfer to fit their needs, including maintaining the order of tracks or omitting certain tracks. Automatic synchronization features are also available to maintain updated playlists across different services. The interfaces of these services are user-friendly, making the transfer process straightforward. While the free versions offer basic features, the premium plans provide advanced features like limitless transfers and more sophisticated customization options. Some of the prominent playlist transfer services include **TuneMyMusic**, **FreeYourMusic**, **SongShift**, and **Soundiiz**.

Other Services

Some music services such as *cloud-based music services* and *discovery and recommendation platforms* such as *Spotify*, *Apple Music*, *YouTube Music*, and *Last.fm* were already mentioned previously.

2.5 Relevant Technology

In this section, several relevant technologies for the project will be described.

2.5.1 Semantic Web

One of the technologies that the thesis and the implementation of the system depend on is the **Semantic Web**. The concept of *Semantic Web* was created by Tim Berners-Lee in the late 1990s as an extension of the *World Wide Web*. The main problem that it solves is that machines cannot read, access, and interpret the content and information on the Internet. It is achieved by adding structure and semantic information to general unstructured content. Additional information enable machines to interpret and process content on the Internet more intelligently and efficiently, allowing the creation of programs that could automate tasks such as *knowledge discovery* or *data integration*. [18]

2.5.2 Resource Description Framework

Another technology, which the system is using is the **Resource Description Framework** (RDF). *RDF* is a framework for expressing information about resources that allows a representation of information in machine-processable form. Resources can be any type of document, person, physical object, or abstract concept. The main idea behind *RDF* is that any statement can be represented as a triple in the following format:

```
<subject> <predicate> <object> .
```

The subject and object represent the two resources in a relationship, while the predicate signifies the character of their relationship. Any set of triples, called *RDF graph*, can be represented as a connected directed graph, where nodes are subjects and objects, and arcs are predicates. [19]

There are **three types of data** defined in *RDF* [19]:

- **International Resource Identifiers** (IRIs): *IRIs* are the generalization of *Uniform Resource Identifier* (URI). *URI* is a sequence of *ASCII*[5] characters used to identify, locate, and access a resource on the Internet. *IRIs* support the usage of *non-ASCII characters*, making it easier for users of non-Latin scripts to represent resource identifiers. *IRIs* can be used as *subjects*, *predicates*, and *objects*. Some examples of *IRIs* are: [20]
  - `https://www.example.com/page?query=example`

[5]**American Standard Code for Information Interchange (ASCII)** is a 7-bit character code where each individual bit represents a unique character. ASCII encodes letters of the English alphabet, digits, and symbols such as punctuation marks, etc. **ASCII table** [access. 2024-04-18]

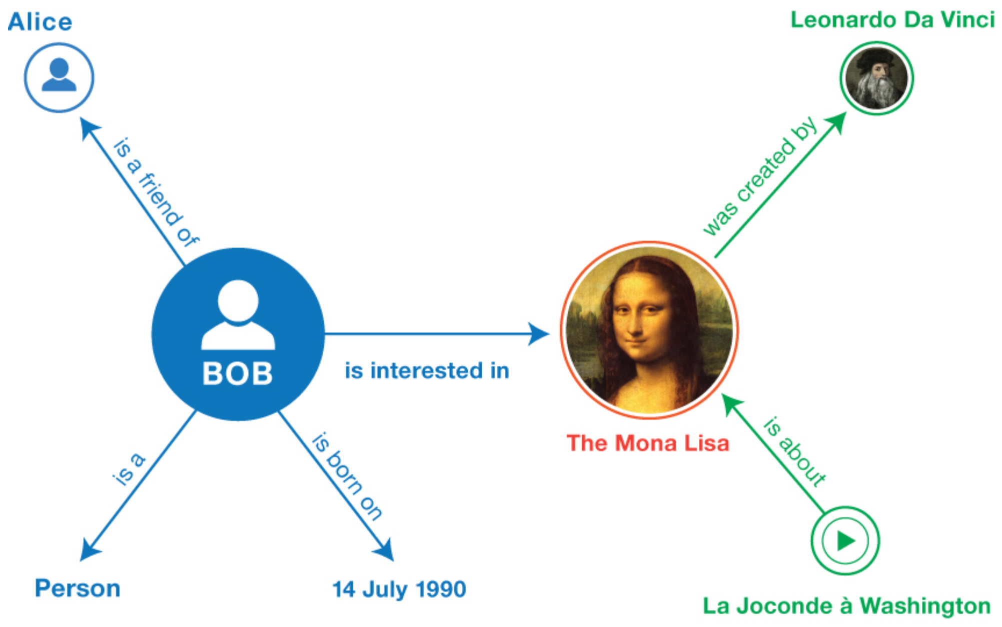


Figure 2.1 Example of a RDF graph [19]

  - `https://ja.wikipedia.org/wiki/%E6%97%A5%E6%9C%AC%E8%AA%9E`

- **literals**: Literals are basic values that are not *URIs* (strings, dates, numbers, etc.). They are usually paired with *data types*[6]. String literals can optionally be associated with a *language tag*, which specifies the language of the provided string. Literals can only be used as *objects* in a *RDF triple*. Some examples of literals are as follows.
  - ”Hello, world!” (string literal)
  - 42 (integer literal)
  - 2024-04-18 (date literal)
  - ”Bonjour tout le monde!”@fr (string literal with a language tag)
- and **blank nodes**: A *blank node* is a special kind of an identifier. It can be used when there is no need to define a global identifier (e.g. an *IRI*) for a resource. In a *RDF triple*, it can be used as *subject* or as *object*.

As stated previously, *RDF triples* can be structured into *RDF graphs*. These *RDF graphs* can be identified by an *IRI*. Subsequently, numerous *RDF graphs* can be structured into *RDF datasets*. An *RDF dataset* can contain several

[6] *RDF* supports the usage of **XML Schema data types**, such as *string*, *boolean*, *dateTime*, *integer*, etc.

named graphs and a maximum of one unnamed graph, sometimes referred to as *default graph*. [19]

Various *serialization formats* are available to record *RDF triples*. Regardless of the serialization format used to represent a *RDF graph*, all result in identical triples, making them logically indistinguishable. Some of the most popular serialization formats are the following: [19]

- **N-Triples** and **Turtle** and their extensions **N-Quads** and **TriG** from the *Turtle family of RDF languages*:
  - **N-Triples** is the most basic format for expressing *RDF triples*. In an *N-Triples* document, each line represents one triple. It follows the following schema; each part of the triple is separated by white space, and the triple ends with full stop:

    ```
    <subject> <predicate> <object> .
    ```

    The full *IRIs* are contained within *angle brackets* `<>`. The data types of literals can be specified by employing the *delimiter* `^^` as a suffix.
    Some examples of using *N-Triples* are: [19]

    ```
    <http://example.org/bob#me> <http://www.w3.org
        /1999/02/22-rdf-syntax-ns#type> <http://xmlns.com/
        foaf/0.1/Person> .
    <http://example.org/bob#me> <http://schema.org/
        birthDate> "1990-07-04"^^<http://www.w3.org/2001/
        XMLSchema#date> .
    ```

  - **Turtle** is an extension of *N-triples*. It adds syntactic sugar, such as support for name-space prefixes, lists, and short-hands for data-typed literals, which makes it more readable for humans. Although the text is more readable and easier to write, syntactic shortcuts make the format harder to parse.
    Some examples of using *Turtle* are: [19]

    ```
    BASE   <http://example.org/>
    PREFIX foaf: <http://xmlns.com/foaf/0.1/>
    PREFIX schema: <http://schema.org/>
    PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

    <bob#me>
      a foaf:Person ;
      schema:birthDate "1990-07-04"^^xsd:date .
    ```

  In addition, the *TriG* and *N-Quad* extensions enable the definition of named graphs in a *RDF document*.

- **JSON-LD** is a format that offers a *JSON*[7] syntax for *RDF graphs* and *datasets*. *JSON-LD* has the ability to convert *JSON documents* into *RDF* with only slight modifications.
- **RDFa** is a type of *RDF syntax* that allows the integration of *RDF data* into *HTML*[8] and *XML* documents. For example, this allows search engines to collect these data during web crawling and use them to improve search results.
- **RDF/XML** provides an *XML-based* structure for *RDF graphs*. When *RDF* was first developed in the late 1990s, this was the only syntax that was used. Subsequently, the additional formats mentioned above were embraced and standardized.

2.5.3 SPARQL Protocol and RDF Query Language

To be able to create, read, and manage the added data efficiently, the system uses the **SPARQL Protocol and RDF Query Language** (SPARQL). *SPARQL* is a set of specifications that provides languages and protocols for querying and manipulating *RDF graph content* on the Web or in an *RDF store*. The standard includes *SPARQL Query Language and Query Results*, *SPARQL Update*, *SPARQL Protocol*, *SPARQL Graph Store HTTP Protocol*, and various other specifications. [21]

SPARQL Query Language is a query language for querying *RDF data*. It defines four basic query forms:

1. **SELECT query form**, which yields variables associated with a matching query pattern. A *SELECT query form* is made up of a *SELECT* clause (identifying variables to be returned), a *FROM* clause (specifying the data source being queried), and a *WHERE* clause (determining the *triple pattern* to be matched).
2. **CONSTRUCT query form**, which generates a single *RDF graph* defined by a graph template, replacing the variables within the specified graph template.
3. **ASK query form** that checks if a solution exists for the given query pattern.
4. **DESCRIBE query form** that returns a single *RDF graph* describing resource matched by the provided pattern.

[7]**JavaScript Object Notation (JSON)** is a lightweight data-interchange format. **JSON** [access. 2024-04-18]

[8]**Hypertext Markup Language (HTML)** is a standard markup language used to create and design web pages. It uses tags to structure content, defining elements such as headings, paragraphs, links, and images. **HTML — Mozilla** [access. 2024-04-18]

In addition, *SPARQL Query Language* offers **aggregates** such as *GROUP BY* (used to group solutions by a given condition) or *HAVING* (used to filter grouped solution sets), **solution sequence modifiers** such as *ORDER BY* (used to order solutions by a specified rule), *DISTINCT* (used to filter duplicates), *OFFSET* (used to return matched data offset by a specified number of solutions) or *LIMIT* (used to limit the number of returned solutions) and **expressions** such as filters, functions or logical expressions. Moreover, the standard defines the formats in which results can be returned: *XML*, *JSON* or *CSV/TSV*. [22]

SPARQL Update serves as a *modification language* for *RDF graphs*. It employs a syntax based on the *SPARQL Query Language* for *RDF*. Its primary function is to handle triples in an *RDF graph* (such as insert, delete) or to manage *RDF graphs* (such as create, drop, load, clear, and so on). *Graph Update* operations (for managing triples inside an *RDF graph*) include:

1. **INSERT DATA operation** adds the given triples to the given graph. In addition, it should create the destination graph if it does not exist.
2. **DELETE DATA operation** removes the given triples if they are present in the given graph.
3. **DELETE/INSERT operation** removes and inserts the given triples matched by the provided pattern.
4. **LOAD operation** adds triples from a given RDF document to the given graph.
5. **CLEAR operation** removes all triples from the given graph.

Graph Management operations (for managing *RDF graphs*) include the *CREATE operation* (which creates a new *RDF graph*), *DROP operation* (which removes an *RDF graph* together with all its contents), *COPY operation* (which copies contents of one *RDF graph* to another *RDF graph* and overwrites its contents), *MOVE operation* (which moves all contents from one *RDF graph* to another *RDF graph*), and *ADD operation* (which copies contents of one *RDF graph* to another *RDF graph* without overwriting the destination graph's contents). [23]

Lastly, the **SPARQL Protocol** and the **SPARQL Graph Store HTTP Protocol** enable communication with *SPARQL endpoints* over *HTTP*[9], using

[9]**Hypertext Transfer Protocol (HTTP)** is a stateless protocol at the application layer that facilitates the transmission of hypermedia documents like *HTML*. It is used predominantly for interactions between web browsers and web applications. **HTTP — Mozilla** [access. 2024-04-18]

native *HTTP methods* (such as *POST*, *GET*, *PUT*, and *DELETE*), making it easier to integrate *SPARQL services* with web applications and various other applications. [24, 25]

2.5.4 Ontologies

To be able to express the structure of the data in the knowledge graph, the system would need to design and implement an **ontology**. Originally, in *philosophy*, ontology explores the *nature of existence*, asking what types of entities exist. In *computer science*, however, it provides a *structured vocabulary* for defining the meaning of entities in specific domains. In computational terms, an ontology is a formal and explicit specification of shared conceptualization within a domain encoded in a machine-readable format. Key attributes include *formality*, *explicitness*, *consensus*, *conceptuality*, and *domain specificity*, ensuring that the ontology is understandable to humans and usable by machines. In general, an ontology in an information system serves as a *conceptual model of an application domain*, facilitating decision-making through reasoning about domain knowledge.

Ontologies, which are crucial in information systems, differentiate themselves from other conceptual models because of their capacity to access implicit knowledge, acting as storehouses of domain knowledge that are accessed by smart computer systems. In contrast to other models such as *UML class diagrams*[10] or *entity-relationship diagrams*[11], which dictate technical systems during the design phase, ontologies depict the knowledge observed in the domain. The interaction with ontologies also involves reasoning tasks of *verification* and *deduction*.

Verification guarantees the logical coherence and nonexistence of conflicting data, while *deduction* allows drawing conclusions from ontology specifications, accessing implicit knowledge. For instance, deduction can infer supplementary information such as an entity's categorization or attributes beyond what is explicitly stated, thereby enhancing the system's comprehension beyond surface-level information.

In the *Semantic Web*, ontology languages play an important role in providing a standardized framework for *representing knowledge*, *facilitating interoperability between heterogeneous data sources*, and enabling *intelligent information extraction*, *automated reasoning*, and *seamless integration of web resources*. Two of the most used *ontology languages* are **RDF Schema** (RDFS) and **Web Ontology Language** (OWL). [26]

[10]**Unified Modeling Language (UML)** class diagram illustrates the static structure of a system by depicting classes and their relationships, serving as blueprints for object-oriented design. **UML — Miro** [access. 2024-04-21]

[11]**Entity-relationship diagrams** visually represent entity interactions, aiding developers and designers in understanding software relationships and system structures; they are commonly used in database design. **ER Diagram — Miro** [access. 2024-04-21]

RDFS, an extension of *RDF*, serves as a framework for describing resources and providing basic ontological features. *RDFS* allows the *creation of hierarchies of resource classes and properties*, enabling essential features such as *interrelation* and *instantiation*. However, *RDFS* is noted for its limitations, *lacking support for expressing exclusion or negation*, which makes it semantically light. [26]

On the other hand, **OWL** serves as the primary language for the representation of ontologies on the Internet. *OWL* extends *RDFS* by offering more expressive features, including the ability to *construct complex classes* from simpler ones *using logical expressions*, support for *rich axiomatization*, such as *class exclusion*, etc. [26]

2.5.4.1 Music Ontologies

In this subsection several existing music ontologies will be presented.

The Music Ontology

The Music Ontology is a domain-specific ontology used to describe data and relationships in the music domain. It was created in the 2000s and the last revision was made in 2012. The vocabulary contains more than 50 classes and more than 150 different properties.

The Music Ontology enables the representation of information about artists, albums, tracks, genres, events, instruments, performances, musical works, labels, distributors, audio features, and relationships between entities. It enables a detailed description of *musicians* (class **MusicArtist**), *bands* (class **MusicGroup**), composers, singles, *albums* (class **MusicAlbum**), and *tracks* (class **Track**), along with classifications into various *genres* (class **MusicGenre**) and styles. Moreover, it captures metadata about *live performances* (class **Performance**), concerts, *festivals* (class **Festival**), *venues* (class **Show**), and participants, as well as *compositions* (class **Composition**), songs, and pieces of music, including their titles, composers, and associated information. Additionally, it encompasses details about *record labels* (class **Label**), music distributors, and characteristics of music such as tempo, key, and mood. [27]

MusicBrainz Schema

MusicBrainz, previously introduced, offers a comprehensive database and a schema for describing music-related entities. The *MusicBrainz Schema* encompasses a variety of data types essential for the organization and comprehension of musical data. At its core, it includes key entities such as *artists*, *releases*, *tracks*, and *labels*, each characterized by specific attributes like *release date*, *catalog number*, and *genre*. Furthermore, *MusicBrainz* establishes *relationships* among these entities, including *aliases and annotations*, which provide context and links within its database. Additional fundamental terms cover elements like *cover art* and *recordings*, ensuring a thorough depiction of music metadata. Additionally, the *MusicBrainz Schema* incorporates terms for *compilations*, *DJ mixes*, *remixes*, *remasters*, etc., addressing terminology related to *music mixes*. [28]

The **LinkedBrainz** initiative was launched to make *MusicBrainz data* available as *Linked Data*. Typically, Linked Data does not directly utilize the unique identifiers from *MusicBrainz*; instead, it either generates new IDs derived from these identifiers or references them. *LinkedBrainz* functioned as a direct supplier of these data, with the aim of reducing redundancy within Linked Data. Unfortunately, the *LinkedBrainz* project was terminated in 2021, and there has been no subsequent progress. This implies that, at present, the latest *MusicBrainz* data is not available in a *RDF format*. [29]

Schema.org Vocabulary

Schema.org is a collaborative effort aiming to create and maintain structured data schemas for the Internet. Its flexible vocabulary accommodates multiple encoding formats (such as *JSON-LD*, *RDFa*, etc.) and encompasses entities, relationships, and actions, featuring a scalable model. *Schema.org* is used by more than 10 million websites, enabling a variety of applications that enhance user experiences. Initiated by *Google*, *Microsoft*, *Yahoo*, and *Yandex*, its evolution is driven by an open community process through mailing lists and *GitHub*. [30]

As of April 2024, the *Schema.org*'s vocabulary consisted of over 800 different types, over 1400 properties, and 90 enumerations[12], which makes it a relatively large vocabulary. It also provides multiple types to describe entities from various domains: *creative works* (such as art, music, movies, TV series, etc.), *health*, *education*, *people and organizations*, *geography*, etc. The most basic type in the *Schema.org's vocabulary* is **Thing** which can describe **any** entity. It has basic properties such as *name*, *url*, *description*, etc. Every type in the *Schema.org's vocabulary* is derived from the *Thing* type.

To characterize the music domain, various types and their attributes have been established. The *Schema.org's vocabulary* is partially inspired by the previously mentioned ontologies — *The Music Ontology* and *MusicBrainz Schema*.

CreativeWork encompasses a broad range of creative works, such as books, movies, photographs, and software programs. It features attributes like *author*, *description* of the content, *dates* of *creation*, *modification*, *publication*, *genre*, and *language* of the work. Subsequent types are derivatives of the *CreativeWork* type:

[12]from **Organization of Schemas — Schema.org** [access. 2024-04-27]

- **MusicRecording** characterizes individual music recordings, typically single tracks. It extends the *CreativeWork* type by including details on the *artists* involved, the *album* association of the recording, *duration*, and more.
- **MusicComposition** characterizes a musical work. It contains attributes such as *composer*, *lyricist*, *lyrics* of the piece, and *recordings* derived from the musical piece, among others.
- **MusicPlaylist** characterizes a playlist comprising various music tracks. This type details the *total count* of the tracks and the *tracks* themselves.
- **MusicAlbum** builds upon the *MusicPlaylist* type, incorporating the album's *artists*, distinct album *releases*, and two enumerative categories detailing the *album production* and *album release types*:
  - **MusicAlbumProductionType** enumeration categorizes albums according to their content, including types like *studio album*, *compilation*, *live album*, among others.
  - **MusicAlbumReleaseType** enumeration classifies the types of album releases, including categories like *single*, *EP*, and *album*.
- **MusicRelease** type characterizes an individual music album release. It encompasses details such as the *recording label*, the *length* of the album, and the *format of release* (such as CD, vinyl, or digital), among others.

When modeling entities for artists, two types could be used — either more general **Person** type or **MusicGroup** type. *MusicGroup* describes groups of musicians such as bands, orchestras, or choirs, and can also be applied to solo artists. It encompasses details such as the albums of the artists, their musical genre, and their individual tracks.

Wikidata: WikiProject Music

An initiative mentioned previously is *Wikidata*, which uses its own ontology to categorize its entities.

Wikidata supports numerous **WikiProjects**, where groups of contributors work towards enhancing *Wikipedia*. These groups typically concentrate on a particular subject area (such as *WikiProject Mathematics* or *WikiProject India*), a certain section of the encyclopedia (such as *WikiProject Disambiguation*), or a distinct type of activity (such as monitoring newly created pages). [31]

WikiProject Music is a music domain-oriented *WikiProject*. The subset of *Wikidata's ontology* contains some classes and properties to describe entities from the music domain. It provides classes to describe:

- ***Artists***, which include **humans**, **fictional characters**, and **musical ensembles** (musical groups).
- ***Compositions*** are abstract musical works that underpin performances and audio recordings/tracks.
- ***Tracks*** are individual audio and video units, such as CD tracks and music videos.
- ***Releases*** are publications of recorded audio or video, including *studio albums*, *compilation albums*, *EPs*, *mixtapes*, and *singles*. These can include different **versions** or editions.
- ***Labels*** are the brands and companies associated with the releases.

Each class has multiple properties which can be associated with them. In addition, each class can have multiple *external IDs* (such as IDs in different national libraries, music services, digital catalogs, etc.) associated with them. *External IDs* play an important role in the linking of data between different knowledge bases. [32]

DBpedia Ontology

The **DBpedia Ontology** is another crucial ontology to consider. Alongside *Schema.org vocabulary* and *Wikidata Ontology*, the *DBpedia Ontology* serves as a framework for representing data across a broad spectrum of domains. For depicting entities and relationships in music, one can utilize classes like *MusicalWork*, *Album*, *Song*, *MusicalArtist*, *MusicGenre* and others. Many of the *DBpedia* music domain classes correspond to types from *Schema.org vocabulary* or classes from *Wikidata Ontology*. [33]

Other Music Ontologies

- **DOREMUS Ontology**, which includes classes such as *Performed Expression*, which is a particular realization or embodiment of a musical piece, *Performed Work* indicating a specific instance of a musical piece being executed, *Genre* for classifying music according to stylistic or cultural characteristics, *Medium of Performance* indicating the instruments or voices used, *Tempo* determining the rate or speed, *Catalogue Number* and *Opus Number* serving as unique identifiers, and *Performer* denoting the individuals participating in the performance.
- **Audio Features Ontology**, which offers classes and properties to represent various aspects of audio signals. It can describe things such as *Amplitude*, *Beat*, *Pitch*, *Tempo*, etc.

- **Playlist Ontology**, which is a small ontology providing class *Playlist* along with attributes such as *playlist name* and *track list* for expressing relationship between tracks and playlists.

2.5.5 Linked Data and Data Mappings

As Tim Berners-Lee stated in his article about *Linked Data*, the *Semantic Web* is not only about putting the data on the Internet, but also about making links between the data, so a person or a machine can explore the *web of data*. In this way, one can find other related data. As stated at the beginning of this section, the *Semantic Web* uses *RDF* to describe entities and their metadata on the Internet. [34]

Tim Berners-Lee stated four rules for the **Linked Data**:

1. Things should be *identified by an universal IRI*
2. *HTTP URIs* should be used, so the *things can be looked up*
3. Upon looking up the resource identified by an IRI, *some useful information should be provided* (using standards such as *RDF* or *SPARQL*)
4. *Links to other URIs should be included*, so more things could be discovered

Linking could be done using multiple methods, such as using other *URIs*, which point to different resources, or using semantic predicates (for instance, **rdfs:seeAlso** or predicates from **FOAF**[13] ontology). [34]

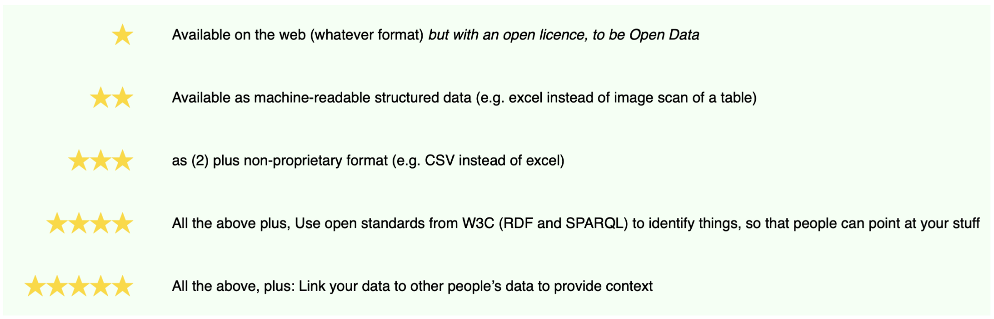
Mapping refers to the creation of a link between two resources that semantically represent the same entity. To create a mapping, the predicate **owl:sameAs** or its equivalent **schema:sameAs** can be used.

In addition, Berners-Lee invents the concept of **Linked Open Data**, which refers to *Linked Data* which is released under an open license (such as *Creative Commons CC-BY*). Furthermore, a star-rating system exists to evaluate the data; 1-3 stars refer to *Open Data*, 4-5 stars refer to *Linked Open Data* (see Figure **2.2**). [34]

2.5.6 RDF Triple Stores

For the storage of the user's knowledge graph, the system will use and **RDF triple store**. An *RDF triple store* is a specialized system designed to handle

[13]**Friend-of-a-Friend (FOAF)** is a project which aims to link people and information using the Interent. It defines *RDF predicates* such as *foaf:knows* (indicating that an individual knows another individual) or *foaf:mbox* (specifies the email address of an individual). **FOAF Vocabulary Specification** [access. 2024-04-21]



★	Available on the web (whatever format) *but with an open licence, to be Open Data*
★★	Available as machine-readable structured data (e.g. excel instead of image scan of a table)
★★★	as (2) plus non-proprietary format (e.g. CSV instead of excel)
★★★★	All the above plus, Use open standards from W3C (RDF and SPARQL) to identify things, so that people can point at your stuff
★★★★★	All the above, plus: Link your data to other people's data to provide context

Figure 2.2 The 5-star data [34]

data in *RDF format*. Like other types of databases, it employs a suitable internal storage structure (such as *native RDF*, *relational*, or *graph*) to facilitate *effective data querying*. Although there are various query languages for *RDF data*, *SPARQL* has emerged as the standard widely adopted by *RDF triple stores*. Additionally, some *RDF triple stores* feature *inferencing capabilities*, which allows the logical derivation of information not explicitly present in the data, using *RDFS and OWL ontologies*. Most *RDF triple stores* are available in an open-source version, including **Blazegraph DB**, **Apache Jena Fuseki**, and **OpenLink Virtuoso**. [35]

2.5.7 RDF Validation

Given that *RDF* operates without a fixed schema, it allows any property to hold values of various types, ranging from simple string literals to more elaborate structures. A validation tool must satisfy various *requirements*, including *value constraints*, *cardinality constraints*, and *predicate constraints* such as defined *property ranges*, *logical operators*, and *negation*. Several methods can be implemented to maintain the consistency and quality of the data within the knowledge graph. Approaches that meet all the criteria for validation methods employ what are known as **shapes**. A *shape* specifies the constraints that a particular *node* (a *RDF triple*) must meet to be considered valid. These *shapes* can impose constraints on the entire *node* or just a *specific property*. [36, 37]

Shapes Constraint Language (SHACL) is one of the approaches that can be used for the *RDF validation*. **NodeShapes** act as a templates for a node within a *data graph*, outlining the necessary conditions and constraints the node needs to meet. **PropertyShapes** outline specific constraints related to a node's property, specifying rules about the property's values, cardinality, and other aspects. A *Focus Node* is a particular node in the data graph that is assessed against the constraints set forth in the *NodeShape* and its related *PropertyShapes*, verifying if the data graph adheres to the defined shapes and constraints. Each *NodeShape* can have defined **targets**: *classes* (*sh:targetClass*), *specific nodes* (*sh:targetNode*), *subjects or objects of the spe-*

cific predicate (*sh:targetSubjectOf*, *sh:targetObjectsOf*). *PropertyShapes* use **property paths** to target specific properties. In *SHACL* there are several *constraint component types* defined: [38]

- **Value Type Constraint Components** — restrictions based on the value's type (*sh:class* limits the value's class, *sh:datatype* specifies the value's datatype, sh:nodeKind defines the node type (*IRI*, blank node, literal, or their combination))
- **Cardinality Constraint Components** — limitations on the quantity of values (*sh:minCount* determines the minimum number and **sh:maxCount** determines the maximum number of values)
- **Value Range Constraint Components** — limitations on the value's range (*sh:minExclusive* indicates $<$, *sh:minInclusive* indicates $\leq$, *sh:maxExclusive* indicates $>$, and *sh:maxInclusive* indicates $\geq$)
- **String-based Constraint Components** — constraints on string values, including length constraints (*sh:minLength*, *sh:maxLength*), pattern constraints (*sh:pattern* for matching specific patterns), and language constraints (*sh:languageIn* for specifying permissible language tags for strings and *sh:uniqueLang* ensuring a unique string for each language tag)
- **Property Pair Constraint Components** — constraints concerning relationships between properties, (*sh:equals* indicates that linked properties must share identical values, *sh:disjoint* indicates that linked properties must possess distinct values, *sh:lessThan* and *sh:lessThanOrEquals* indicate that one of the linked values should be lesser (or equal to) than the other value)
- **Logical Constraint Components** — constraints applying logic to properties, *sh:not* serves to negate an expression or acts as a logical $\neg$ operator, *sh:and* enables conjunction or a logical $\wedge$ operator, *sh:or* enables disjunction or a logical $\vee$ operator, and *sh:xone* dictates that each *value node* must match precisely one of the specified *shapes*
- **Shape-based Constraint Components** — components that define constraints allowing for the specification of intricate conditions through the validation of value nodes against designated shapes
- **Other Constraint Components** — constraint components which cannot be assigned to any specific group, *sh:closed* and *sh:ignoredProperties* enable the specification of properties exempt from validation, while **sh:hasValue** and **sh:in** assess if the examined value matches a specific value or belongs to a predefined list of values

- **SPARQL-based Constraint Components** — additionally, *SHACL* supports validation through advanced *SPARQL queries* based on *SELECT* or *ASK queries*

Other shape-based approaches include **Shape Expressions (ShEx)** and **Resource Shapes (ReSh)**. [36]

2.5.8 MP3 Metadata

Even before music metadata services existed, it was possible to embed metadata in *MP3 files* using the so-called *MP3 tags*. *MP3 tags* are vital for storing metadata such as title, artist, album, and track number. Despite the comprehensive nature of *MP3 standards*, they lack explicit guidelines for tag formats or a uniform metadata container. Over time, tag formats such as *ID3v1* (which later became *ID3v2*) and *APEv2* have naturally become the industry's de facto accepted standards. These tags, usually found at the start or end of *MP3 files*, are clearly separate from the actual audio data with which they are associated. *MP3 decoders* have the ability to extract information from these tags or disregard them as *non-MP3 data*. [39, 40]

Label	ID3 Tag	Value
Track title	**TIT2**	Back To The Radio
Track number	**TRCK**	2 / 13
Disk or media number	**TPOS**	1 / 1
Artist	**TPE1**	Porridge Radio
Artists	**TPE2**	Porridge Radio
Album artist		Porridge Radio
Release year		2022
Album	**TALB**	Waterslide, Diving Board, Ladder To The Sky
Release date	**TDRC**	2022-05-20
ISRC	**TSRC**	US38W2145002
Copyright	**TCOP**	2022 Secretly Canadian

Table 2.1 Example of *MP3 metadata* of a single track in **ID3 format**

Chapter 3

Analysis and Design

To allow users to create their own music knowledge graphs, the creation of the **Music KG system** is proposed. The **Music KG system** will accommodate the following tasks:

- *upload* media files from users' devices and *parse and edit metadata*,
- *connect to music services* to obtain metadata which they are offering,
- *add local tracks* to the user's knowledge graph,
- *add tracks and playlists* from the connected streaming services,
- *browse user's knowledge graph* using *SPARQL endpoint* or using *GUI*,[1]
- *link* users' music metadata *with existing music knowledge bases*

This chapter will present the analysis of requirements, including outputs such as use-case scenarios, functional, and non-functional requirements. Following this, the system's design will be showcased, detailing the individual components and elements and their integration within the system.

3.1 Requirements Analysis

3.1.1 Target Group

To accurately outline the needs of a software application, it is crucial to consider its usual user base. The typical user of the **Music KG system** is

[1]**User interface (UI)** serves as the medium for interaction and communication between humans and computers within a device. It encompasses elements such as display screens, keyboards, a mouse, and the overall desktop interface. In addition, it represents the method by which users interact with applications or websites. **Graphical user interface (GUI)** is a visual type of a *UI*. **User Interface — TechTarget** [access. 2024-04-28]

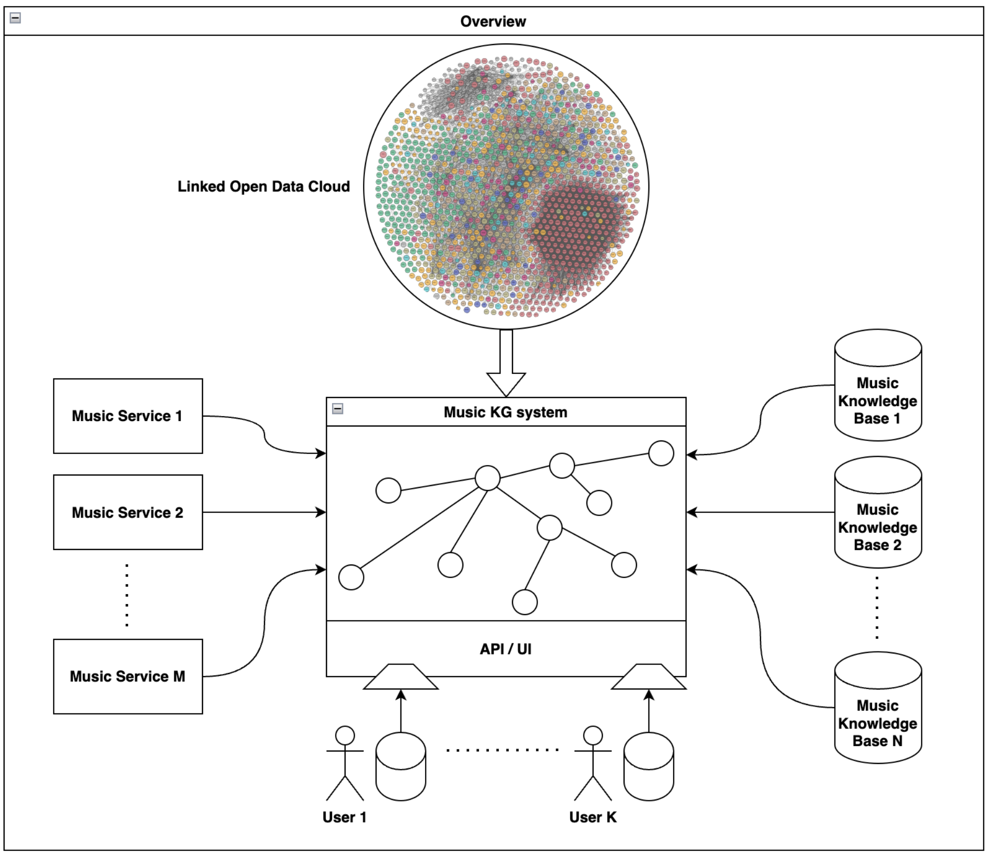


Figure 3.1 General overview of the **Music KG system**. Source: Linked Open Data cloud image

someone who has a deep love for music, enjoys categorizing their preferred music according to various classifications, and has a desire to expand their understanding of their musical preferences. With that in mind, the identification and formulation of use case scenarios and user requirements for the system was realized. Documenting both use case scenarios as well as user requirements helps keep track of what the desired functionality should be, as well as creating testing scenarios. There are, in total, 21 *use case scenarios*, 32 *functional requirements* and 3 *non-functional requirements*.

3.1.2 Use Case Scenarios

3.1.2.1 User and User Profiles Use Case Scenarios

- **UC–1 — Register user:** Allows an unregistered user to create a new account.

  1. The system will open the registration form.

2. The user fills in the form and submits it.
3. If the received email address is already assigned to another user, the scenario fails; otherwise, the system creates a new user and shows the user the success message.

- **UC–2 — Login registered user:** Allows a registered user to log into the system.

  Preconditions: The user is registered.

  1. The system will open the log-in form.
  2. The user fills in the form and submits it.
  3. If the received email address does not exist in the system or the combination of received email address and password does not match the combination stored in the system's database, the scenario fails; otherwise, the system logs in the user and shows them the success message.

- **UC–3 — Create user profile:** Allows a logged-in user to create a user profile.

  Preconditions: The user is logged-in.

  1. The system will check if the user has not created their user profile yet, if not, the scenario ends; otherwise, it will open the *Create Profile* form.
  2. The user fills in the form and submits it.
  3. The system creates a new profile for the user and shows the user the success message.

- **UC–4 — Display user profile:** Allows a logged-in user to display their user profile.

  Preconditions: The user is logged-in.

  1. The system will check if the user has created their user profile; if not, the system lets the user create a new user profile instead (see **UC-3**); otherwise, it will show the logged-in user's profile.

3.1.2.2 Media Files Use Case Scenarios

- **UC–5 — Upload media file:** Allows the user to upload media files in order to add new music tracks to their music library.

  Preconditions: The user is logged in.

  1. The system will open the dialog for uploading the file.
  2. The user chooses the file or multiple files to upload.

3. The system will parse the MP3 metadata from the uploaded files and fill in the form with the obtained values.

- **UC–6 — Edit uploaded media file:** Allows the user to edit the uploaded file metadata.

  Preconditions: The user is logged in. At least one uploaded file.

  1. The user selects the file they wish to edit from the list of uploaded files
  2. The system will open the form for editing parsed metadata
  3. The user clicks on the button to unlock the form for editing
  4. The system unlocks the form for editing
  5. The user edits the values and clicks on the button to saves the changes
  6. The system saves the changes and locks the form for editing

- **UC–7 — Assign Spotify IDs to uploaded media file:** Allows the user to search and assign *Spotify IDs* to uploaded file metadata.

  Preconditions: The user is logged in. At least one uploaded file; selected file and unlocked for editing, selected field to assign *Spotify ID* to.

  1. The user clicks on the button to assign the *Spotify ID* to the selected field
  2. The system opens dialog with the*Spotify search*
  3. The user selects search criteria and submits
  4. The system searches *Spotify* based on the provided criteria and shows the user retrieved results
  5. The user selects retrieved ID to assign to the field in the form
  6. The system assigns the selected *Spotify ID* to the selected form field

- **UC–8 — Add new music tracks from uploaded media files:** Allows the user to add new music tracks from uploaded media files to their music library.

  Preconditions: The user is logged in. At least one uploaded file.

  1. The user clicks on the button to add new tracks to their music library
  2. The system checks if the filled data are valid, if they are invalid, the scenario ends and system will show the failure message; otherwise the system creates a new music track in the system for each uploaded file and shows the success message

3.1.2.3 Music Library Management Use Case Scenarios

- **UC–9 — Browse added albums:** Allows the user to browse albums in their music library.

  Preconditions: The user is logged in.

  1. The user clicks on the button to show albums
  2. The system retrieve albums from the user's music library and displays them

- **UC–10 — Show album detail:** Allows the user to access the selected album's detail.

  Preconditions: The user is logged in. At least one album is in the music library.

  1. The user clicks on the selected *album ID*
  2. The system retrieve information about the selected album and shows it to the user

- **UC–11 — Browse added artists:** Allows the user to browse artists in their music library.

  Preconditions: The user is logged in.

  1. The user clicks on the button to show artists
  2. The system retrieve artists from the user's music library and displays them

- **UC–12 — Show artist detail:** Allows the user to access the selected artist's detail.

  Preconditions: The user is logged in. At least one artist is in the music library.

  1. The user clicks on the selected *artist ID*
  2. The system retrieve information about the selected artist and shows it to the user

- **UC–13 — Browse added playlists:** Allows the user to browse playlists in their music library.

  Preconditions: The user is logged in.

  1. The user clicks on the button to show playlists
  2. The system retrieve playlists from the user's music library and displays them

- **UC–14 — Show playlist detail:** Allows the user to access the selected playlist's detail.

  Preconditions: The user is logged in. At least one playlist is in the music library.

  1. The user clicks on the selected *playlist ID*
  2. The system retrieve information about the selected playlist and shows it to the user

- **UC–15 — Browse added tracks:** Allows the user to browse tracks in their music library.

  Preconditions: The user is logged in.

  1. The user clicks on the button to show tracks
  2. The system retrieve tracks from the user's music library and displays them

- **UC–16 — Show track detail:** Allows the user to access the selected track's detail.

  Preconditions: The user is logged in. At least one track is in the music library.

  1. The user clicks on the selected *track ID*
  2. The system retrieve information about the selected track and shows it to the user

3.1.2.4 Spotify Integration Use Case Scenarios

- **UC–17 — Authorize Spotify account:** Allows the user to authorize their *Spotify account* to enable information retrieval such as obtaining *recently played tracks* and accessing *user's playlists*.

  Preconditions: The user is logged in and has a *Spotify account*.

  1. The user clicks on the button to authorize the system to use their *Spotify account*
  2. The system redirects user to *Spotify authorization*
  3. The user authorize the system to use their *Spotify account*
  4. Upon successful authorization, the system shows the user the menu with further *Spotify actions*

- **UC–18 — Obtain recently played tracks on Spotify:** Allows an authorized user to access their *Spotify library* and get their recently played tracks.

  Preconditions: The user is logged in and has authorized the application to use their *Spotify account.*

  1. The user clicks on the button to get the recently played tracks on *Spotify*
  2. The system obtains the recently played tracks from *Spotify* and shows them to user

- **UC–19 — Obtain user's Spotify playlists:** Allows an authorized user to access their *Spotify library* and get their saved playlists.

  Preconditions: The user is logged in and has authorized the application to use their *Spotify account.*

  1. The user clicks on the button to get their playlists on *Spotify*
  2. The system obtains the user's playlists from *Spotify* and shows them to user

- **UC–20 — Add music track from Spotify to user's music library:** Allows an authorized user to add a music track from *Spotify* into the music library.

  Preconditions: The user is logged in and has authorized the application to use their *Spotify account.*

  1. The user opens *Spotify track detail* and clicks the button to add the track into their music library
  2. The system creates a new track in user's library; if the request is successful, it will show the user the success message, otherwise, it will show the user the failure message

- **UC–21 — Add Spotify playlist to user's music library:** Allows an authorized user to add a music playlist from *Spotify* into the music library.

  Preconditions: The user is logged in and has authorized the application to use their *Spotify account.*

  1. The user opens *Spotify playlist detail* and clicks the button to add the playlist into their music library
  2. The system creates a new playlist (including creating tracks contained in the playlist) in user's library; if the request is successful, it will show the user the success message, otherwise, it will show the user the failure message

3.1.3 Functional Requirements

- **FR–1 — Registration Form:** The system shall provide a form accessible to unregistered users for registration. The form shall include fields for the user to input email address and password. The system shall verify that all required fields are filled out before allowing submission. Additionally, during registration, the system shall check the uniqueness of the email address provided by the user.

- **FR–2 — User Account Creation:** Upon successful validation of the registration form and the uniqueness of the email address, the system shall create a new user account during registration. The system shall store the user's registration information securely in the database.

- **FR–3 — Registration Success/Error Message:** After successfully creating the user account during registration, the system shall display a success message to the user confirming their registration. In case of any errors during the registration process, the system shall display an appropriate error message.

- **FR–4 — Login Form:** The system shall provide a form accessible to registered users for logging in. The form shall include fields for the user to input their email address and password. Upon submission of the login form, the system shall authenticate the user's credentials against the information stored in the system's database. If the email address provided by the user does not exist in the system or if the combination of the email address and password does not match the stored credentials, the system shall prevent the login and display an appropriate error message.

- **FR–5 — Successful Login:** If the credentials provided are valid, the system shall log in the user and grant access to their account. The system shall display a success message confirming the user's login.

- **FR–6 — Profile Creation Check:** Upon logging in, the system shall check if the user has already created their user profile. If the user has not yet created their profile, the system shall allow them to proceed with profile creation.

- **FR–7 — Profile Creation Form:** The system shall provide a form for creating a user profile accessible to logged-in users who have not yet created their profile. The form shall include a field for inputting the logged-in user's name.

- **FR–8 — User Profile Creation:** Upon submission of the profile creation form, the system shall create a new profile for the user. The system shall store the profile information securely in the database.

- **FR–9 — Profile Creation Success Message:** After successfully creating the user profile, the system shall display a success message to the user.
- **FR–10 — Display User Profile:** If the user has created their profile, the system shall display the user profile information. The system shall provide options for the user to edit their profile or perform other profile-related actions.
- **FR–11 — Upload Dialog and File Selection:** The system shall provide a dialog for uploading media files accessible to logged-in users, allowing them to choose one or multiple files to upload.
- **FR–12 — MP3 Metadata Parsing:** When uploading the file, the system shall parse the MP3 metadata of the uploaded files and fill in the form with the values obtained.
- **FR–13 — File Selection for Editing:** The user shall be able to select the file they wish to edit from the list of uploaded files.
- **FR–14 — Edit Form:** The system shall provide a form for editing parsed metadata accessible to logged-in users with at least one uploaded file.
- **FR–15 — Form Editing and Locking:** The user shall be able to edit the values in the form and save the changes. Upon saving changes, the system shall lock the form for editing.
- **FR–16 — Spotify Search Dialog:** The system shall provide a dialog with *Spotify search* functionality. The user shall be able to select search criteria and submit them to the system. The system shall search *Spotify* based on the provided criteria and display the retrieved results.
- **FR–17 — Spotify ID Selection and Assignment:** The user shall be able to select a retrieved *Spotify ID* from the search results and assign it to the selected field in the form.
- **FR–18 — New Track Creation:** The user shall be able to add new tracks to their music library. The system shall verify that the filled data are valid. If the data are invalid, the system shall show a failure message. If the data filled are valid, the system shall create a new music track in the system for each uploaded file and show a success message.
- **FR–19 — Browse Albums from Music Library:** The system shall retrieve albums from the user's music library and display them upon browsing.
- **FR–20 — Display Album from Music Library:** The system shall retrieve and display information about the selected album.

- **FR–21 — Browse Artists from Music Library:** The system shall retrieve artists from the user's music library and display them upon browsing.
- **FR–22 — Display Artist from Music Library:** The system shall retrieve and display information upon selecting the artist.
- **FR–23 — Browse Playlists from Music Library:** The system shall retrieve playlists from the user's music library and display them upon browsing.
- **FR–24 — Display Playlist from Music Library:** The system shall retrieve and display information upon selecting the playlist.
- **FR–25 — Track Retrieval and Display:** The system shall retrieve tracks from the user's music library and display them upon browsing.
- **FR–26 — Track Information Retrieval and Display:** The system shall retrieve and display information upon selecting the track.
- **FR–27 — Spotify Account Authorization:** The system shall allow the user to authorize the application to use their *Spotify account*.
- **FR–28 — Menu Display Upon Authorization:** Upon successful authorization, the system shall display a menu with further *Spotify actions — recently played tracks* and *user's playlists*.
- **FR–29 — Retrieve Recently Played Tracks:** The system shall obtain the user's recently played tracks from *Spotify* and display them.
- **FR-30 — Retrieve User's Playlists:** The system shall obtain the user's playlists from *Spotify* and display them.
- **FR–31 — Add Music Track from Spotify to User's Library:** Upon selecting a track, the system shall create a new track in the user's library. If successful, it shall display a success message; otherwise, it shall display a failure message.
- **FR–32 — Add Spotify Playlist to User's Library:** Upon selecting a playlist, the system shall create a new playlist (including creating tracks contained in the playlist) in the user's library. If successful, it shall display a success message; otherwise, it shall display a failure message.

3.1.4 Non-functional Requirements

- **NFR–1 — Responsive User Interaction:** The system shall respond to user interactions, such as browsing albums or retrieving tracks, within 2 seconds under normal operating conditions.

- **NFR–2 — Intuitive User Interface:** The user interface shall adhere to established design principles and provide intuitive navigation and interaction patterns to minimize the usage learning curve.
- **NFR–3 — Error Handling:** Error handling mechanisms shall be implemented to gracefully handle unexpected failures and provide informative error messages to users.

3.2 System Design

The following section highlights the process and conclusions from designing the system considering use case scenarios and functional and non-functional requirements.

3.2.1 System Architecture

The **Music KG system** consists of four integral parts: the *back-end server*, the *RDF store*, the *database for user credentials* and the *front-end client*. Figure **3.2** shows parts of the system and the relations between them.

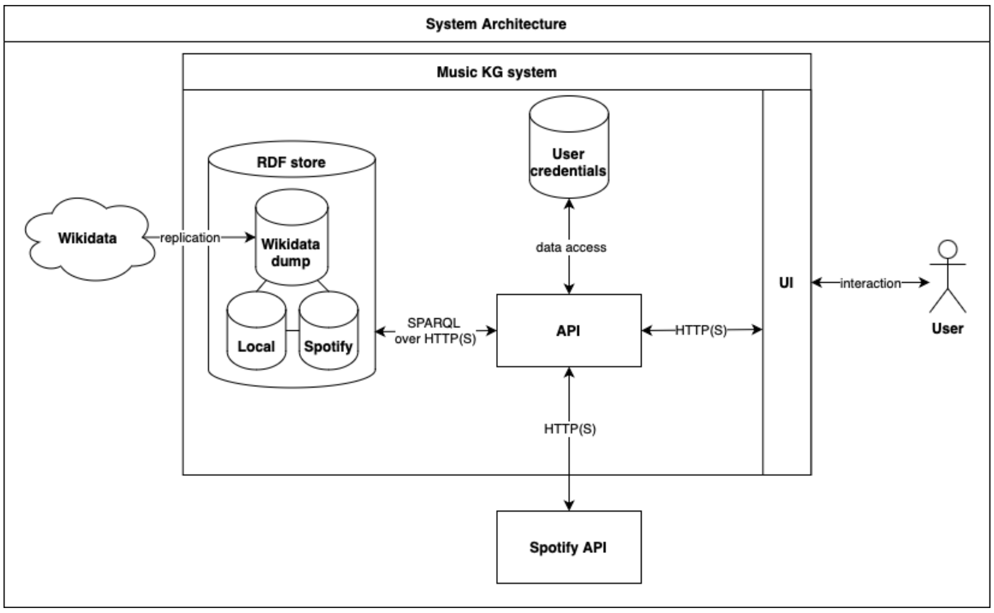


Figure 3.2 The **Music KG system** architecture

3.2.2 Back-end Server

The backbone of the **Music KG system** is its **back-end server**. It provides support for any operation required from the system, whether it is *user authen-*

tication, *music library management*, *retrieving information from Spotify*, etc. The detailed architecture can be seen in Figure **3.3**.

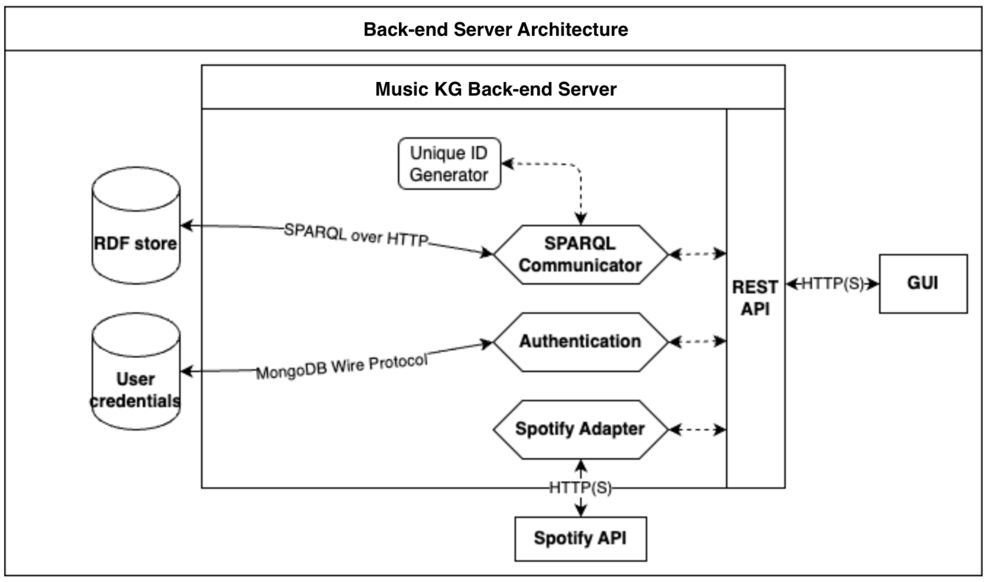


Figure 3.3 The architecture of the **Back-end Server**

For that purpose, three main modules are created:

1. **Authentication module** that mainly focuses on *registering new users*, *providing an access token* for the user which logs in, and *authenticating incoming requests* to the *API*.
2. **SPARQL Communicator module** that mainly focuses on *communication with the RDF store*, *manipulating the data* inside the RDF store, including providing methods to translate *CRUD-L*[2] operations into *SPARQL queries* and retrieving information about the entities inside of the *RDF store*.
3. **Spotify Adapter module** that mainly focuses on *obtaining various metadata* from *Spotify API* such as information about albums, artists, playlists and tracks, including user's saved playlists or their recently played tracks. To facilitate this, the module also provides an authorization method that enables the user to grant permission to the *API* to retrieve and access their personal data from *Spotify*.

[2]**Create, Read, Update, Delete (CRUD)** encapsulates the fundamental operations of data management which are crucial for interacting with preserved data in databases or software applications. *CRUD-L* refers to the four operations mentioned above with the **List (L)** operation added, which returns all entities from some given collection. **CRUD — Mozilla** [access. 2024-04-22]

Technology Stack

The *back-end server* uses the **Node.js Express** framework and is written in **TypeScript**.

Node.js is a flexible open source *JavaScript* runtime that works on various platforms, utilizing the *V8 JavaScript engine* from *Google Chrome*. Its architecture, based on a *single process and asynchronous I/O operations*, achieves *high performance* by eliminating the need to create new threads for each request and avoiding code execution blockage. Consequently, *Node.js* can efficiently manage many simultaneous connections without the difficulties associated with thread management. In addition, it provides an easy transition for front-end developers, allowing them to program in *JavaScript* for both the client and the server sides. [41]

Node.js Express is a minimal and flexible *Node.js* web application framework offering a comprehensive suite of functionalities for both web and mobile platforms. *Express* delivers a streamlined layer of essential web application capabilities, encompassing *middleware* and *HTTP utility methods*. [42]

The *back-end server* also uses one custom middleware: **cors middleware**, which provides configuration and setup for *CORS*[3].

Both frameworks also support development in **TypeScript**, enhancing the developer experience with its *extended syntax* and features. One of its main objectives is to *identify errors* in an early stage of development, simplifying the process of immediate error correction. As *TypeScript* is compiled into *JavaScript* during the build process, it does not impose restrictions on its use in application development. [43]

For requests to external services such as *music services' APIs* or *the RDF store*, the *back-end server* uses **axios**, a promise-driven *HTTP client* compatible with both *Node.js servers* and *browsers*. A key feature is its ability to operate on both platforms using the same code-base. It provides an *API* for *making HTTP requests*, *intercepting outgoing requests and incoming responses*, *handling errors*, etc. [44]

Authentication and Authorization

Users will be able to register and log in to access the resources in the *RDF store* and access the data retrieved from *Spotify*. The *back-end server* will be able to process requests for *registering a new account*, *logging in to an existing*

[3]**Cross-Origin Resource Sharing (CORS)** is a protocol using *HTTP* headers that allows servers to define which foreign origins may access their content. This includes a preflight check by the browser to verify if the server allows the specific request, including the methods and headers of *HTTP*. **CORS — Mozilla** [access. 2024-04-28]

account, *providing* a time-bounded *access token*, and will be able to *verify the legitimacy of the provided access token*. Verified users will be able to access the resources and manage the data in the *RDF store*.

External Integration

The *back-end server* will expose endpoints through a **RESTful**[4] API that client applications could use to communicate with the server.

The back-end server will communicate with *Spotify API* using methods defined in the **Spotify Web API SDK**, which provides interfaces, types, and methods to connect to *Spotify API*. SDK also offers different methods for authorization based on the context in which the *API* is accessed. Given that the server also requires interaction with user data, it must secure authorization to access this information on the *Spotify platform*.

Moreover, the *back-end server* will communicate with the *RDF store* using **SPARQL Protocol**. *SPARQL Protocol* enables communication using *HTTP* making it easier for servers to communicate with the *RDF store* directly. Using *SPARQL Protocol* enables the execution of SPARQL queries via the *POST method* by embedding a *SPARQL request* as a query parameter *?query* in the request. The results can be obtained in the formats *JSON*, *XML*, or *CSV*.

To be able to create *SPARQL queries* from *JSON objects* (from incoming requests), the *back-end server* uses the **SPARQL.js** library, which offers a *SPARQL generator* and a *SPARQL parser* for transforming *JSON objects* into *SPARQL queries* and vice versa. Since manipulating a *JSON object* is much easier than manipulating strings in *TypeScript* applications, the *SPARQL.js* library enables a simple, straightforward and effective way to create *SPARQL queries*.

3.2.3 RDF Store

A crucial aspect of developing a knowledge base is the database that stores the data. Given that the data are in the *RDF format*, it is advisable to opt for a specialized database, specifically an **RDF store**. Details about *RDF stores* have been previously discussed in Section **2.5.6**.

[4]**Representational State Transfer (REST)** is an architectural style used in *API* implementation. A *RESTful API* is an interface that supports stateless interactions, allows data caching between requests, maintains a consistent interface, and enables resources to be transmitted in a standardized format, among other features. **What is a REST API?** [access. 2024-04-29]

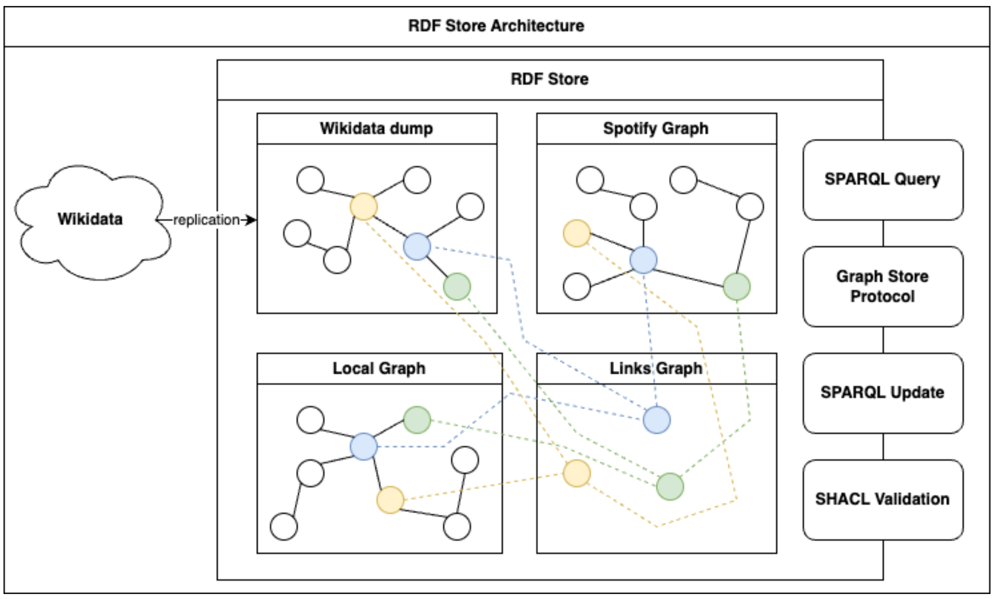


Figure 3.4 The architecture of the **RDF Store**

Technology Stack

Apache Jena Fuseki was selected for this task due to its ease of setup and its prior use in the *Semantic Web course*, making it a clear choice. **Apache Jena Fuseki** is an *RDF store* and *SPARQL server* within the *Apache Jena* framework. **Apache Jena** is a free and open source *Java* framework for building *Semantic Web* and *Linked Data* applications. *Fuseki* includes a user interface and offers extensive configurabilty, providing numerous services for integration and utilization within the *RDF store*. Four services will be used in the **Music KG RDF Store**:

- **SPARQL Query service**, which provides *SPARQL endpoint* and support for *SPARQL queries* (refer to Section **2.5.3**)
- **Graph Store Protocol** service, which provides support for graph manipulation such as creating graphs, editing graphs, etc. (refer to Section **2.5.3**)
- **SPARQL Update** service, which provides support for data manipulation using *HTTP POST* method by sending *SPARQL query* as a query paramter (refer to Section **2.5.3**)
- **SHACL Validation** service, which verifies the compliance of graph data with specific rules outlined by *SHACL Shapes* (refer to Section **2.5.7**)

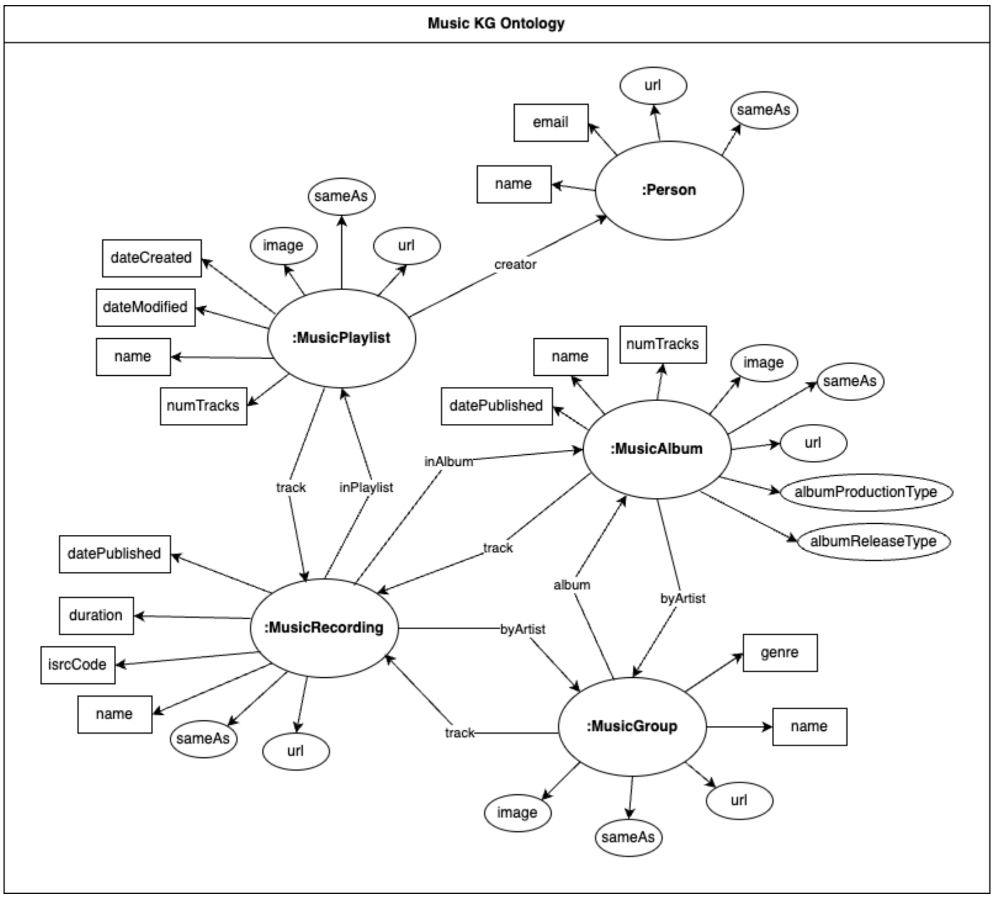


Figure 3.5 Music KG ontology. All types and predicates are from **schema.org**.

Music Ontology

For the *RDF store*, the following **music domain-oriented ontology** will be considered; it is based on *Schema.org vocabulary* (see Section **2.5.4**).

- **MusicRecording** class describes a *single recording*, a *track* or a *song*. It includes the following properties:
  - *byArtist* — an artist or artists that performed the recording
  - *datePublished* — the initial release date of the recording, typically aligning with the album's release date
  - *duration* — the duration of the recording in *ISO 8601 format*.[5] Example of a duration written in *ISO 8601 format* is e.g: *P3Y6M4DT12H30M5S*

[5]Maintained by the *International Organization for Standardization* (ISO), **ISO 8601** is an international standard covering the worldwide exchange and communication of date and time-related data. **ISO 8601 — Wikipedia** [access. 2024-04-29]

which corresponds to: 3 years, 6 months, 4 days, 12 hours, 30 minutes and 5 seconds

  - *inAlbum* — the album to which the recording belongs
  - *inPlaylist* — a playlist or playlists to which the recording belongs
  - *isrcCode* — the *ISRC code*[6] for the recording
  - *name* — the name of the recording
  - *sameAs* — the link to a different resource which identifies the same recording
  - *url* — the external *URL* bound with the recording

- **MusicPlaylist** class describes a playlist which consist of several music tracks. It includes the following properties:

  - *creator* — a person or people which created the playlist
  - *dateCreated* — the date and time of the creation of the playlist in the RDF store
  - *dateModified* — the date and time of the latest modification of the playlist in the RDF store
  - *image* — the image bound with the playlist
  - *name* — the name of the playlist
  - *numTracks* - the number of the tracks in the playlist
  - *sameAs* — the link to a different resource which identifies the same playlist
  - *track* — a track or a list of tracks which belong to the playlist
  - *url* — the external *URL* bound with the playlist

- **MusicAlbum** class describes an album or a collection of music tracks. It contains the following properties:

  - *albumProductionType* — a type of production of the album, it can be one of the following types: *CompilationAlbum*, *DJMixAlbum*, *DemoAlbum*, *LiveAlbum*, *MixtapeAlbum*, *RemixAlbum*, *SoundtrackAlbum*, *SpokenWordAlbum*, *StudioAlbum*
  - *albumReleaseType* — a type of release of the album, it can be one of the following values: *AlbumRelease*, *BroadcastRelease*, *EPRelease*, *SingleRelease*
  - *byArtist* — an artist or artists that performed the album

[6]**International Standard Recording Code (ISRC)** is a unique and permanent identifier for sound recordings and music videos. It helps distinguish between different recordings and with the management of content rights when the recording is used on different platforms. **Home — International Standard Recording Code** [access. 2024-04-29]

  - *datePublished* — the initial release date of the album
  - *image* — the image bound with the album
  - *name* — the name of the album
  - *numTracks* — the number of the tracks in the album
  - *sameAs* — the link to a different resource which identifies the same album
  - *track* — a track or a list of tracks which belong to the album
  - *url* — the external *URL* bound with the album

- **MusicGroup** class describes a music artist, whether it is a band or a solo musician. It contains the following properties:
  - *album* — an album or albums that were recorded by the artist
  - *genre* — a genre or genres associated with the artist
  - *image* — the image bound with the artist
  - *name* — the name of the artist
  - *sameAs* — the link to a different resource which identifies the same artist
  - *track* - a track or a list of tracks which were recorded by the artist
  - *url* — the external *URL* bound with the artist
- **Person** class describes either an user of the **Music KG system** or a creator of a playlist (e.g. *Spotify user*). It includes the following properties:
  - *email* — the email address of the person
  - *name* — the name of the person
  - *sameAs* — the link to a different resource which identifies the same person
  - *url* — the external *URL* bound with the person

3.2.4 User Credentials Database

The **User Credentials Database** holds information such as user emails and hashed passwords. Given the straightforward nature of the storage model, a simpler database suffices. **MongoDB** was selected for this task. **MongoDB**, a widely-used, open-source, *NoSQL database*, manages data in a flexible *JSON-like structure* called *BSON*. Renowned for its high performance, scalability, and adaptability, it is ideal for any application from small projects to extensive enterprise applications. The document-oriented design of *MongoDB* simplifies data modeling, reflecting the object structure in the application code. With its dynamic schema system, developers can rapidly evolve and respond to new requirements without losing efficiency. Moreover, *MongoDB* boasts powerful search features, including comprehensive indexing and an advanced aggregation framework, which improve data access and analysis. An additional robust capability is its cloud-based database **Atlas**, utilized for this project. Setting up and establishing a connection through a *connection string* or any of the various available client libraries is straightforward. [45]

3.2.5 Front-end Client

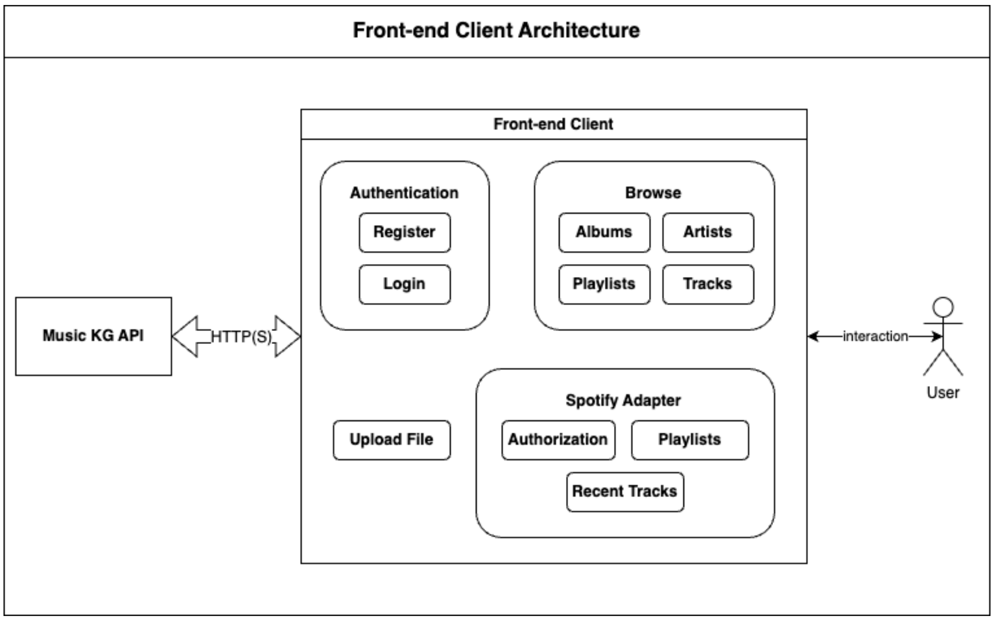


Figure 3.6 The architecture of the **Front-end Client**

The point of the main user interaction with the **Music KG system** is its **front-end client**. It provides a user interface for performing various tasks in the **Music KG system**. It enables users to register and log in, authorize their *Spotify account*, retrieve data from their *Spotify account*, or browse their knowledge graph. The detailed architecture can be seen in Figure **3.6**. For that purpose, four main modules are created:

1. **Authentication module** containing two sub-modules for user registration and user logging in. Both of the sub-modules feature a form in which the user can fill in their email and password and register or log in and receive the access token, respectively.

2. **Browse module**, comprising four distinct sub-modules for data access within the knowledge graph. This module can fetch and exhibit four unique entities: *MusicAlbum* (**Albums** sub-module), *MusicGroup* (**Artists** sub-module), *MusicPlaylist* (**Playlists** sub-module), and *MusicRecording* (**Tracks**

sub-module). Each displayed resource includes all its attributes in an easily readable format and includes links to additional resources in the knowledge graph and to external platforms such as *Spotify* or *Wikidata*.

3. **Upload File module** allows users to upload media files from their devices. The system then processes the embedded metadata to extract details such as *artists*, *album*, *track duration*, *release date*, among others. It also includes a form for modifying the extracted metadata, enabling users to verify and edit the information before adding it to their knowledge graph.

4. **Spotify Adapter module** allows users to authenticate their *Spotify account* and acquire a *Spotify access token*. This token is used by the *back-end server* to fetch user data from the *Spotify API*. The client manages two types of data from the *Spotify API*: *user's playlists* and *recently played tracks*. To manage these, two sub-modules are implemented: the **Playlists** sub-module and the **Recent Tracks** sub-module. The **Playlists** sub-module is responsible for fetching playlists and presenting them in a list format. It allows for an in-depth view of each playlist, including specifics about every track. The **Recent Tracks** sub-module gathers information on the user's latest played tracks along with their details. Both sub-modules enable the addition of tracks into the user's knowledge graph. When a new track is added to the knowledge graph, the system also obtains and incorporates related artist and album information into the knowledge graph.

Technology Stack

The *front-end client* uses the **React** *UI library* and is written in **TypeScript**. The application is build using **Vite** and for *CSS*[7] styling uses **Tailwind** library.

React is a framework consisting of user interface components that simplify the creation of interactive user interfaces. *React* employs components to construct web pages that produce a *DOM*[8] reflecting the component's current state. These components adapt to changes in data and update the relevant sections of the tree structure within the *DOM*. It employs *JSX*, a syntax extension for *JavaScript*, enabling the incorporation of *HTML-like elements* into *JavaScript* files. The preferred method for creating components is using *JSX*. In addition to *React*, various libraries associated with *React* are utilized. These include **React Router DOM** for managing dynamic routing across components, **Re-**

[7]**Cascading Style Sheets (CSS)** allow a developer to define styling rules that are used when when building a web application to be applied to individual *HTML template elements*. **What is CSS? — Mozilla** [access. 2024-04-30]

[8]**Document Object Model (DOM)** represents a web page as a tree structure and allows manipulation of its structure, content and styles using scripts and programming languages. **Document Object Model — Introduction — Mozilla** [access. 2024-04-30]

act Hook Form to simplify form handling and validation, and **React Icons** for incorporating icons. [46]

Vite serves as a bundler for front-end applications. Its name (*vite*, translating to ’fast’ in French) reflects its goal to enhance the development process for contemporary web projects by making it quicker and more efficient. This is achieved through the implementation of advanced features like *Hot Module Replacements*, which allow for immediate and accurate updates without the need to refresh the page or disrupt the application’s state. Additionally, it offers extensive configurability and includes support for testing integration. [47]

Tailwind CSS is a *utility-first CSS framework* designed to simplify web development through a robust collection of ready-to-use, atomic utility classes. Differing from conventional *CSS frameworks*, *Tailwind* avoids using *pre-designed components* and instead provides *basic utility classes* that apply styling directly to *HTML* elements. This method allows developers to quickly develop *unique designs* without the need for *custom CSS*, enhancing uniformity and scalability within projects. *Tailwind* enables developers to effectively construct responsive designs, craft complex interfaces, and uphold a tidy, manageable code-base, thus becoming a favored option in contemporary web development. [48]

The **Music Metadata Browser** library is utilized to parse uploaded media files. This library provides a metadata parser compatible with most of the audio formats and tag headers. Parsing is executed directly in the browser, eliminating the need to transmit the media files to a *back-end API*. [49]

External Integration

The *front-end client* will communicate with the *back-end API* using *HTTP(S)* and would require the use of the access token provided by *API*.

Chapter 4
Implementation

Before initiating the implementation phase, it was necessary to consider the structure of the code-base. Initially, the two applications were developed in *distinct repositories*, which subsequently led to issues since both required the same interfaces and data types, resulting in code redundancies. During the initial development phase, the applications were consolidated into one single *monorepo*, significantly enhancing the efficiency of the development process. Early on, various integration experiments were conducted, including *Spotify authorization* (referencing tutorials in **Spotify SDK's GitHub repository**), setting up the *RDF store* with *Apache Jena Fuseki*, executing *HTTP requests for SPARQL queries* and *uploading files* to the browser client.

In this chapter, the implementation strategy and the definitive choices made during the implementation of the **Music KG system** will be discussed.

4.1 Code Organization and Structure

The entire code-base is structured within a centralized **monorepo**. A *monorepo* refers to a single repository that houses several distinct projects linked by clear relationships. The advantages include maintaining *consistent library package versions* throughout the monorepo to aid integration, typically offering tools to *develop new applications or libraries with minimal extra work*, ensuring *atomic commits* throughout the repository — which *reduces the likelihood of errors*, and promoting *uniformity within the code of a monorepo.* [50]

For monorepo management, **Nx** is used. *Nx* is a build tool that helps to create applications and libraries using a wide range of commands called *recipes*. It offers *recipes* to create *Node.js servers*, *React applications*, *data type libraries*, etc. [51]

Typically, an *Nx monorepo* includes several applications and libraries. Libraries may consist of shared components or shared data types, essentially any code that can be reused across different applications. In the **Music KG system** there exist two applications: **api** (the *back-end server*) and **dashboard** (the *front-end client*); and two libraries: **data** (for sharing *data types* between the *back-end server* and the *front-end client*) and **sparql-data** (including data types and models to express *SPARQL* and *RDF* related data). The relationships between applications and libraries can be seen in Figure **4.1**.

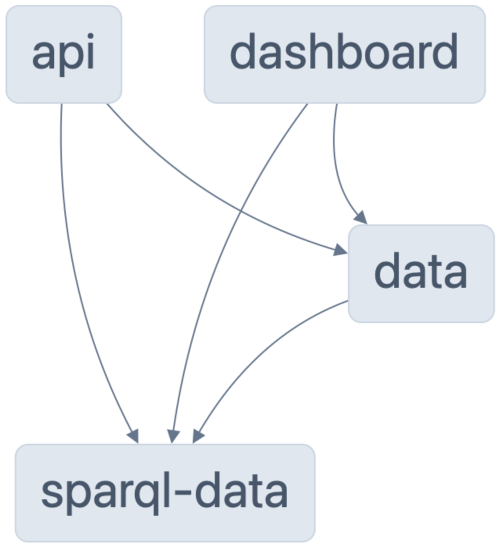


Figure 4.1 Relationships between applications and libraries in the **Music KG Nx monorepo**

Because both applications utilize frameworks or libraries (*Node.js Express* and *React*) that are *unopinionated*, implying that there is no prescribed folder or organizational structure for the projects to adhere to, developers are free to create and employ the structure they find most suitable. For each application, an inspired code-base structure was created. They will be described in detail later in the relevant sections of this chapter (see Section **4.2** and Section **4.3**).

4.2 Back-end Server

4.2.1 Code-base Structure

The code-base structure of the *back-end server* is heavily inspired by ***A future-proof Node.js express file/folder structure*** by *Codemzy*.

- **api** — root folder for the *API routes*
  - **auth** — folder for the *Authentication module*

  - **sparql** — folder for the *SPARQL Communicator module*
  - **spotify** — folder for the *Spotify Adapter module*
- **database** — folder for the *User Credentials database* integration
- **utils** — folder for shared utility functions
- **main.ts** — the mounting point of the application; handling configuration of the *server's port*, *CORS*, *connection to the database* and *routing*
- **routes.ts** — the routing configuration for the top-level routing

The sub-folders of the **api** folder correspond to their related routes (e.g. *api/auth* becomes `<BASE_URL>/api/auth`). Each sub-folder contains the same structure for handling the incoming requests:

- **constants** — stores constants relevant to the module
- **features** — includes functions invoked by handlers, such as those encapsulating *business logic*
- **handlers** — includes functions triggered upon matching specific routes
- **helpers** — includes auxiliary functions that assist a handler's feature
- **middleware** — includes middleware that can be integrated into a processing pipeline for specific routes
- **models** — stores models like datatypes or interfaces relevant to the module
- **routes** — the configuration for routing within the module
- **utils** — includes utility functions relevant to the module

Each category could be either a *single file* or a *folder that holds multiple files*. Starting as a single file, reaching more than one hundred lines, it could be split into multiple files to improve readability.

4.2.2 Authentication Module

The **Authentication module** implements *UC–1 — Register user*, *UC–2 — Login registered user* and implements support functions for *UC–3 — Create user profile* and *UC–4 — Display user profile* by providing information about the current logged-in user based on the received access token.

The **Authentication module** features three endpoints:

- `GET /api/auth/current` — retrieves the information about the current logged-in user from the received access token

- `POST /api/auth/login` — provides an access token upon successful log in (received user credentials match the ones stored in the database)
- `POST /api/auth/register` — receives email and password and if the email does not already exist in the database, creates a new user in the system

Registering a new user

The system queries the *user credentials database* if there is already some user with the provided *email*. If not, a new user is created. To ensure safe storage of passwords provided, the received password is encrypted using the **scrypt** function of the *Node's crypto* library. The *scrypt* function receives three parameters: password, salt (a random string), and length of the key (the system uses 64). For every user, the system generates a different salt, which is stored alongside the hashed password in the database. The entry in the database is e.g:

```
{
    email: abc@example.com,
    hash: <salt>:<hashed_password>
}
```

where `hashed_password` is password hashed using the *scrypt* function.

Logging in the register user

The system queries the *user credentials database* to find out if the user exists in the database and then checks if the provided password matches the hashed password stored in the database. If the user is successfully logged in, the server creates an access token with which the client can authorize any subsequent request. For authorization tokens, the system employs **JSON Web Tokens (JWT)**, a widely recognized standard for securely transmitting claims between two parties.

Get Current User

The system decodes and parses the received access token and returns the current user's *email address* and their *ID* in the *user credentials database*.

4.2.3 SPARQL Communicator Module

The **SPARQL Communicator module** implements support functions for the following use case scenarios: *UC–3 — Create user profile*, *UC–4 — Display user profile*, *UC–8 — Add new music tracks from uploaded media files*, *UC–9 — Browse added albums*, *UC–10 — Show album detail*, *UC–11 — Browse added artists*, *UC–12 — Show artist detail*, *UC–13 — Browse added playlists*, *UC–14 - Show playlist detail*, *UC–15 — Browse added tracks*, *UC–16 — Show* *track detail*, *UC–20 — Add music track from Spotify to user's music library* and *UC–21 — Add Spotify playlist to user's music library*.

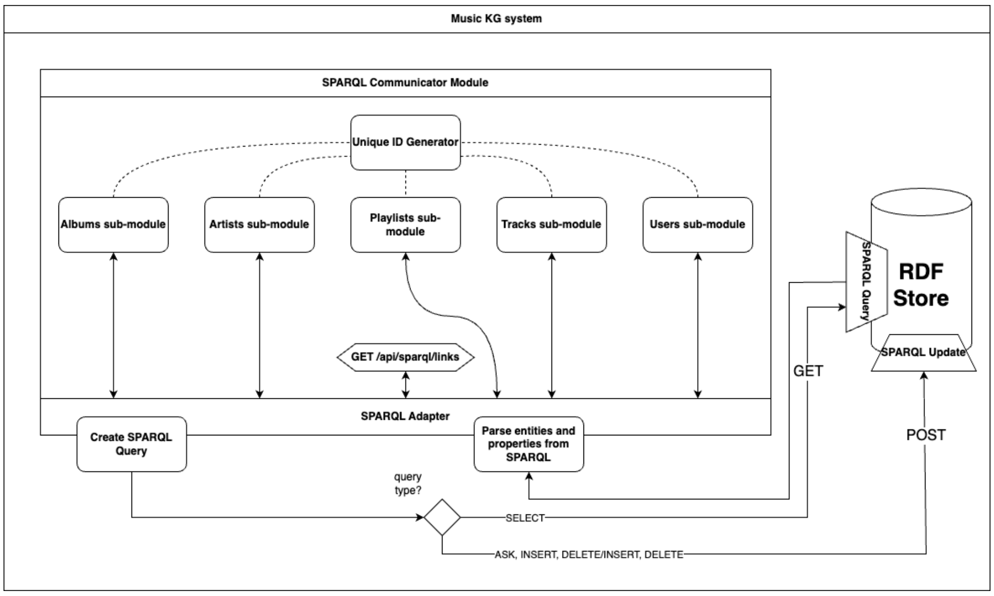


Figure 4.2 The overview of the **SPARQL Communicator** module in the back-end server

The most complex module in the entire **Music KG system** is the **SPARQL Communicator module**. As shown in Figure **4.2**, it contains five sub-modules (*Albums*, *Artists*, *Playlists*, *Tracks* and *Users*), one top-level endpoint providing *sameAs links* from the *Links graph*, the *SPARQL Adapter* (containing functions, models and interfaces for creating various *SPARQL queries* and parsing the results received from the *RDF store*) and the *unique ID generator*.

The most important component of the module is the **SPARQL Adapter**, located in the *helpers* directory, which acts as a bridge between the *back-end server* and the *RDF store*. The capabilities of the *SPARQL Adapter* are categorized into four main groups based on their functionality:

1. *SPARQL Query* formulation
2. *SPARQL Results* interpretation
3. functions dealing with *triples*
4. miscellaneous functions (like prefix replacement, etc.)

For bounding the variables to be used in *SPARQL results interpretation*, the variable names: `?subject` (for subjects), `?predicate` (for predicates) and `?object` (for objects) are used.

SPARQL Query Formulation

The first group of *SPARQL Adapter functions* is *SPARQL Query Formulation functions.* The system offers 11 different *SPARQL queries.* The following section will briefly present all of them.

SELECT queries

SELECT queries return data bound by given variables in the query pattern. *SPARQL Adapter* can create 5 different *SELECT queries*:

- **createGetQuery function** creates a *SELECT query* to find all triples for one given subject (`resource`) from the given graph (`graph`). The created query is as follows:

```
SELECT ?predicate ?object
FROM <graph> WHERE {
    <resource> ?predicate ?object.
}
```

- **createGetByExternalUrlQuery function** creates a *SELECT query* to find all triples for a subject with the given external URL (`external_url`) from the given graph (`graph`). The created query is as follows:

```
PREFIX schema: <https://schema.org/>
SELECT ?subject ?predicate ?object
FROM <graph> WHERE {
    ?subject schema:url <external_url>;
        ?predicate ?object.
}
```

- **createGetAllQuery function** creates a *SELECT query* to find all entities of the given class (`class`) from the given graph (`graph`). The created query is as follows:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns
   #>
SELECT ?subject
FROM <graph> WHERE {
    ?subject rdf:type <class>.
}
```

- **createGetFromWikidataQuery function** creates a *SELECT query* to find all triples for the given subject by the *Spotify ID* (`spotify_entity_id`) which is further specified by the *Wikidata predicate* (`wikidata_prop` — it can be one of those values: *P1902*: Spotify artist ID, *P2205*: Spotify album ID or *P2207*: Spotify track ID) from the *Wikidata graph* (`wikidata_graph`). The created query is as follows:

```
PREFIX wdt: <http://www.wikidata.org/prop/direct/>
SELECT ?subject ?predicate ?object
FROM <wikidata_graph> WHERE {
    ?subject wdt:<wikidata_prop> <spotify_entity_id>;
        ?predicate ?object.
}
```

- **createGetLinksQuery function** creates a *SELECT query* to find all linked entities of the given subject (`resource`) from the links graph (`links_graph`). The created query is as follows:

```
PREFIX schema: <https://schema.org/>
SELECT ?subject
FROM <links_graph> WHERE {
    {
        <resource> schema:sameAs ?subject.
    }
    UNION
    {
        ?subject schema:sameAs <resource>.
    }
}
```

ASK queries

ASK queries return true if the given pattern is matched or false, if it is not matched. *SPARQL Adapter* can create 2 different *ASK queries*:

- **createExistsByEntityIdQuery function** creates an *ASK query* to find out whether there is an entity with the given ID present in the *RDF store*, it is used when the unique ID generator checks for the non-existence of the newly generated ID in the given graph. The created *ASK query* is as follows (`graph` is the given graph and `id` is the given id):

```
ASK WHERE {
    GRAPH <graph> {
        <graph>/<id> ?predicate ?object.
    }
}
```

- **createExistsByExternalIdQuery function** creates an *ASK query* to find out whether an entity with some external URL (`external_url`) is present in the given graph (`graph`) in the *RDF store*. The created *ASK query* is as follows:

```
PREFIX schema: <https://schema.org/>
ASK WHERE {
    GRAPH <graph> {
        ?subject schema:url <external_url> .
    }
}
```

SPARQL Update queries

SPARQL Update queries are used for inserting or deleting the data from the given graph. *SPARQL Adapter* can create 3 different *SPARQL Update queries*:

- **createDeleteQuery function** creates a *DELETE query* to remove all triples associated with the given subject (`resource`) from the given graph (`graph`) in the *RDF store*. The created *DELETE query* is as follows:

```
DELETE WHERE {
    GRAPH <graph> {
        <resource> ?predicate ?object.
    }
}
```

- **createInsertQuery function** creates an *INSERT DATA query* to add a new resource with some properties to the given graph. Since each of the classes would have different properties added upon inserting, examples will be provided later when describing the five entity sub-modules (see section **Music KG Entities Sub-modules**).
- **createUpdateQuery function** creates a *DELETE/INSERT* query to firstly remove the matched triples (based on the given predicates to update) from the graph and after adding the updated triples. Since each of the classes would have different properties updated, examples will be provided later when describing the five entity sub-modules (see section **Music KG Entities Sub-modules**).

SPARQL Results Interpretation

Another group of useful functions is SPARQL Results Interpretation. These functions parse the received response from the *RDF store's SPARQL endpoint* and create *JSON objects* which can be later utilized on the *back-end server*. The *SPARQL Communicator module* contains 3 functions that transform responses from the *SPARQL endpoint* to *JSON objects*. The response has the following structure:

```
{
    "head": {
        "vars": [...] // an array of bound variables
    },
    "results": {
        "bindings": [...] // an array of bindings - the
           values bound to specific variables
    }
}
```

Since there are only 3 defined variables that can be bound in *Music KG system* (`?subject`, `?predicate` and `?object`), the property `head` is unnecessary and can be omitted. The resulting object adheres to the following interfaces:

```
type SparqlResults = {
  bindings: SparqlBinding[];
};

type SparqlBinding = {
  subject?: SparqlSubject;
  predicate?: SparqlPredicate;
  object?: SparqlObject;
};

type SparqlSubject = {
  type: 'uri';
  value: string;
};

type SparqlPredicate = {
  type: 'uri';
  value: string;
};

type SparqlObject = {
  type: 'uri' | 'literal';
  datatype?: string;
  value: string;
};
```

From the listing provided, it is evident that only *URIs* can serve as subjects and predicates, while objects can be *URIs* or *literals*, which complies with the *RDF specification*. Parsing is done by one of the following functions, based on the context in which the data are required.

- **getPropertiesFromBindings function** is used for extracting properties from the *SPARQL Query Results* and creating objects according to a clearly defined interface within the **Music KG system**. It can extract the properties from the given bindings for the given set of *RDF predicates*. It is done by parsing and aggregating the extracted values into one single *JSON object* representing some entity from the **Music KG system**. If there are multiple bindings for a single predicate, function instead of single value creates an array of values.

- **getEntityIdsFromBindings function** is used when retrieving information about all entities of some given type (album, artist, playlist, track or user). Function parses the bindings and returns only IDs in the context of the given graph.
- **getLinksFromBindings function** is used when retrieving *sameAs links* from the *Links graph* for the given entity. Function parses the bindings and returns *URIs* which represent the given entity in different contexts.

Triples Helper Functions

There are also two functions which create the so-called *Triple objects* which are used in either the *createInsertQuery function* or the *createUpdateQuery function*. A *Triple object* is a *JSON object* used by *SPARQL.js library* when generating *SPARQL queries* from *JSON objects*. It contains three properties: `subject`, `predicate`, and `object`, which correspond to triples of the *RDF specification*. In this context, it remains the case that only *URIs* are allowed as subjects and predicates, whereas objects may be *URIs* or *literals*.

- **getTriplesForComplexPredicate function** handles the creation of triples, where the predicate requires non-literal object (such as *schema:album*, *schema:byArtist*, *schema:sameAs*, *schema:url*, etc.).
- **getSecondaryEntityTriples function** is a complex function which handles the creation of the so-called **secondary entities** and their linking to the **primary entity**, i.e. the one which is currently being created. A *secondary entity* refers to any entity that must be recognized by a *URI*. If such a *secondary entity* is not already present in the given graph, it will be created based on the information provided at the time of creation, with the *schema:name* property being mandatory for all entities. The process of entity creation is detailed later in the section titled **Creating an Entity**.

Unique ID Generator

To ensure data consistency and to prevent data overwriting, a **unique ID generator** was implemented for the resources in the *RDF store*. It is used by all five above-mentioned sub-modules. The system employs *UUIDs*[1] for

[1]**Universally Unique Identifier (UUID)** is a tag that uniquely distinguishes a resource from others of the same type. *UUIDs* are 128-bit numbers typically shown as a 36-character

identification. Each time, a fresh *UUID* is generated and the system, through an *ASK query* checks if this *UUID* has already been allocated to any entity within the specified graph. If not assigned, this identifier is then applied to the new entity. In contrast, if the *UUID* is already in use, the system will produce a new *UUID* and continue this verification cycle until an unassigned *UUID* is found. Some examples of the generated **IDs/URIs** are:

```
<http://localhost:3030/music-kg/local/4bf56e7e-0226-417f-
    a462-aec4f5bb03a7>
<http://localhost:3030/music-kg/local/39bfa22a-1b98-4607-
    ba9e-80a4d345223f>
<http://localhost:3030/music-kg/spotify/f2d20fbb-1271-4
    dce-b147-68106d5640ac>
<http://localhost:3030/music-kg/spotify/363a6ef5-8b2c
    -4603-83f4-fd5aedc67f28>
```

Music KG Entities Sub-modules

Five entity sub-modules (**Albums**, **Artists**, **Playlists**, **Tracks**, and **Users**) provide *CRUD-L* methods to manage entities in the *RDF store*. Each sub-module offers six distinct endpoints, except for the *Users sub-module*, which does not offer an endpoint for finding a user based on their external URL. Each endpoint is only available to authorized users (with a valid access token) and has a required query parameter `origin`, which can be `local` or `spotify`, based on the origin of the music metadata. The *Users sub-module* does not use the `origin` parameter, because it uses its own *Users graph*. Endpoints which accept `POST` or `PUT` methods also check for a non-empty request body. In each endpoint presented below, `entityModule` stands for one of these values: `albums`, `artists`, `playlists`, `tracks` or `users`. The path parameter `entityId` is replaced by a specific ID.

- `GET /api/{entityModule}` — *Retrieves a list of all entities* of the given entity class. The returned list contains IDs of the entities present in the `origin` graph. For retrieval, a *SELECT query* produced by the *createGetAllQuery function* is used.

- `POST /api/{entityModule}` — *Creates a new entity* of the given entity class in the *RDF store*. Will be explained later in this subsection.

- `GET /api/{entityModule}/find` — Accepts an additional query parameter `spotifyUrl`. If the system finds any entity in the `origin` graph, which is associated with the provided URL, *it retrieves all properties associated with this entity*. The search is performed using a *SELECT query* obtained from the *createExistsByExternalIdQuery function*.

string, formatted as five hexadecimal strings divided by hyphens. **UUID** — **Mozilla** [access. 2024-05-01]

- `DELETE /api/{entityModule}/{entityId}` — *Removes all properties associated with the given entity* from the `origin` graph using a *DELETE query* obtained from the *createDeleteQuery function.*
- `GET /api/{entityModule}/{entityId}` — *Retrieve all properties associated with the given entity* from the `origin` graph using a *SELECT query* obtained from the *createGetQuery function.*
- `PUT /api/{entityModule}/{entityId}` — *Updates an existing entity* of the given entity class in the *RDF store.* Two modes of updating are available: `append` and `replace`. In the `append` mode, the system fetches the properties of the specified existing entity that need updating and attempts to merge them with the properties provided in the update request. In the `replace` mode, the properties chosen for the update are removed and substituted with the new properties from the update request. Will be explained later in this subsection.

Creating an Entity

Creating an entity in the system is specified through multiple interfaces. Although each *entity class* possesses unique attributes, there are common properties among them. Each instance of any *entity class* has `name` and a single or multiple `externalUrls`. On top of that, each *entity class* has its specific properties, which can be either required or optional during their creation. For every instance of any *entity class*, the parameter `name` is required. Entities of the *entity class MusicRecording* have also required parameters `album` and `artists` upon creation. The following listing shows interfaces used in the system:

```
type CreateAlbumRequest = {
  name: string;
  albumProductionType?: MusicAlbumProductionType;
  albumReleaseType?: MusicAlbumReleaseType;
  artists?: EntityData | EntityData[];
  datePublished?: string;
  externalUrls?: ExternalUrls;
  imageUrl?: string;
  numTracks?: number;
  tracks?: EntityData | EntityData[];
};

type CreateArtistRequest = {
  name: string;
  albums?: EntityData | EntityData[];
  externalUrls?: ExternalUrls;
  genres?: string | string[];
  imageUrl?: string;
  tracks?: EntityData | EntityData[];
};

type CreatePlaylistRequest = {
  name: string;
  creators?: EntityData | EntityData[];
  description?: string;
  externalUrls?: ExternalUrls;
  imageUrl?: string;
  numTracks?: number;
  tracks?: EntityData | EntityData[];
};

type CreateRecordingRequest = {
  album: EntityData;
  artists: EntityData | EntityData[];
  name: string;
  externalUrls?: ExternalUrls;
  datePublished?: string;
  duration?: number;
  isrc?: string;
};
```

For some of the properties, for example, *artists*, *tracks*, *genres*, etc. multiple values can be used when creating an instance of the given entity class. The system can receive a single value or an array of values.

The `EntityData` type is used to handle properties where for the object of the corresponding triple, there should be an *URI* — i.e. a *secondary entity*. The `EntityData` type contains: `type` — a type of the *secondary entity* (one of these values: `album`, `artist`, `playlist`, `track` or `user`), `name` — a name of the *secondary entity*, `externalUrls` - a single or multiple External URLs for the *secondary entity* and `id` — the ID of the *secondary entity* already present in the `origin` graph.

```
type EntityData = {
  type: EntityType;
  externalUrls?: ExternalUrls;
  id?: EntityId;
  name?: string;
};
```

The `EntityData` type is crucial for the *getSecondaryEntityTriples function*, which handles the creation and association of *secondary entities* with the *primary entity* (the one being created) (for more information, see sub-section **Triples Helper Functions**). If the parameter `id` is present, the system converts it into *URI* and assigns it to the *primary entity*. If the `id` is not available, but `EntityData` contains any `externalUrl`, the system tries to find an entity associated with the given external URL, and if it is found, it is assigned to the *primary entity*. If not and there is a `name` parameter available for the *secondary entity*, the system will create a new entity for the given *external URL* and the *name* and assign it to the *primary entity*. If only the parameter `name` is available, the system will create a new entity with the provided *name* and assign it to the *primary entity*.

Property	RDF predicate	Entities
album	*schema:inAlbum*	tracks
albumProductionType	*schema:albumProductionType*	albums
albumReleaseType	*schema:albumReleaseType*	albums
albums	*schema:album*	artists
artists	*schema:byArtist*	albums, tracks
creators	*schema:creator*	playlists
dateCreated	*schema:dateCreated*	playlists
dateModified	*schema:dateModified*	playlists
datePublished	*schema:datePublished*	albums, tracks
description	*schema:description*	playlists
duration	*schema:duration*	tracks
externalUrls	*schema:url*	albums, artists, playlists, tracks, users
email	*schema:email*	users
genres	*schema:genre*	artists
imageUrl	*schema:image*	albums, artists, playlists
isrc	*schema:isrcCode*	tracks
name	*schema:name*	albums, artists, playlists, tracks, users
numTracks	*schema:numTracks*	albums, playlists
tracks	*schema:track*	albums, artists, playlists

Table 4.1 Mapping properties to RDF predicates

For each property of the interfaces presented, there exists a bidirectional mapping between the *property* and its *RDF predicate* shown in Table **4.1**. Following the association of each property of the request body with its corresponding *RDF predicate* and the generation of *RDF triples*, the system, through the *createInsertQuery function*, creates an *INSERT DATA query*. For example, the following **request body** for creating a new *MusicRecording*:

```
{
  "album": {
    "name": "Breakfast for Pathetics",
    "externalUrls": { "spotify": "https://open.spotify.
       com/album/4cr6QE3ff10cnK8W1AWZYo" },
    "type": "album"
  },
  "artists": {
    "name": "Tired Lion",
    "externalUrls": { "spotify": "https://open.spotify.
       com/artist/5Vf0Z6jyM0Gr07Gf8irDMt" },
    "type": "artist"
  },
  "datePublished": "2020-11-20",
  "duration": 170817,
  "isrc": "AUUM72000417",
  "name": "Diet Sick",
  "externalUrls": { "spotify": "https://open.spotify.com/
     track/0YWvsVz551MpqvqsbfDRJc" }
}
```

results in this **INSERT DATA query**, where `graph` is the *URI* of the `origin` graph, `resource_uri` is the *URI* of the *primary entity*, `album_uri` is the URI of the *secondary entity* ***album*** and `artist_uri` is the *URI* of the *secondary entity* ***artist***:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX schema: <https://schema.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
INSERT DATA {
  GRAPH <graph> {
    <resource_uri> rdf:type schema:MusicRecording.
    <resource_uri> schema:name "Diet Sick".
    <resource_uri> schema:inAlbum <album_uri>.
    <resource_uri> schema:byArtist <artist_uri>.
    <resource_uri> schema:datePublished "2020-11-20"^^xsd
       :date.
    <resource_uri> schema:duration "PT2M50S"^^xsd:
       duration.
    <resource_uri> schema:isrcCode "AUUM72000417".
    <resource_uri> schema:url <https://open.spotify.com/
       track/0YWvsVz551MpqvqsbfDRJc>.
  }
}
```

After the successful creation of the entity, the system attempts to form links between the same resources in the other graphs in the *RDF store.* For example, when creating an entity in the `local` graph, the system will also check the existence of the same entity in the `spotify` and *Wikidata dump* graphs. Using an external URL such as *Spotify URL*, the system is able to find the entities in the other graphs and then create *sameAs links* in the *Links graph.*

Updating an Entity

The process of updating an entity is very similar to the process of creating one. The interfaces which are used for the updating are almost the same, the only difference being that all of the properties are optional. First, the system checks if the updated entity exists in the system. There are two update modes available: `append` and `replace`. In the `append` mode, for properties where multiple values are allowed and are present in the request body, the system retrieves them from the `origin` graph and merges them with the properties from the request body. In the `replace` mode, the system ignores the properties present in the `origin` graph and considers only the properties from the request body. After that, the updated properties are mapped to their *RDF predicate* counterparts and the triples are created. Finally, the system, using the *createUpdateQuery function*, creates a *DELETE/INSERT query* to update the entity. Working with the example *MusicRecording* from before, the following **update request body**:

```
{
  "name": "Diet Sick.",
  "datePublished": "2022-12-05"
}
```

results in the following **DELETE/INSERT query**, where `graph` is the URI of the `origin` graph, and `resource_uri` is the URI of the *updated entity*:

```
PREFIX schema: <https://schema.org/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
WITH <graph>
DELETE { <resource_uri> ?predicate ?object. }
INSERT {
    <resource_uri> schema:datePublished "2022-12-05"^^xsd
       :date;
        schema:name "Diet Sick.".
}
WHERE {
  OPTIONAL {
    <resource_uri> ?predicate ?object.
    FILTER(!BOUND(?predicate) || !BOUND(?object) || (?
       predicate = schema:datePublished) || (?predicate =
        schema:name))
  }
}
```

The `FILTER` clause in the *DELETE/INSERT query* ensures that *only the triples with the given predicates* are updated.

Similarly as in the previous case with the entity creation, after updating the entity, the system checks if any *sameAs links* can be created between the updated entity and some entity in the other graphs. For more information, see section **Creating an Entity**.

Get Links Endpoint

The endpoint `GET /api/sparql/links` returns an array of *sameAs links* for the *Links graph* associated with the given entity (specified by a query parameter `id`) from the given graph (specified by a query parameter `origin`).

4.2.4 Spotify Adapter Module

The **Spotify Adapter module** is used for obtaining user data and music metadata from *Spotify*, using the **Spotify Web API**. The **Spotify Adapter module** implements *UC–7 — Assign Spotify IDs to uploaded media file*, *UC–17 — Authorize Spotify account*, *UC–18 — Obtain recently played tracks on Spotify* and *UC–19 — Obtain user's Spotify playlists*.

To use the *Spotify Web API*, it is necessary to first set up a *Spotify account* and **creating an application** on the *Spotify Developer dashboard*. Upon creating the application, it will be assigned a *Client ID* and *Client Secret*, which serve as identifiers for the *Spotify application*. There are **four different methods**, which can be used to authorize the application to use the *Spotify Web API*: *Authorization code*, *Authorization code with PKCE extension*, *Client credentials*, or *Implicit grant*. The *Spotify Web API* documentation provides a detailed examination of the advantages and disadvantages of each method.

Since the system must work with both user data and music metadata provided by *Spotify*, multiple authorization strategies must be used. To fetch music metadata, including details on albums, artists, tracks, or public playlists, the *Client credentials* authorization method is sufficient. This method employs *Client ID* and *Client Secret*, and can be implemented on the server-side. In contrast, accessing user data (such as *recently played tracks* or *user's playlists*) requires a unique access token, which can only be acquired through a web browser application or a mobile application using the *Authorization code with PKCE extension* method. The acquired access token can be supplied to the *back-end server*, allowing it to manage the user data on behalf of the user. To be able to provide the access token to the *back-end server*, a custom *HTTP header* `music-kg-spotify-authorization` was created. Applications communicating with the **Music KG back-end server's API** are required to attach this header to every request targeting endpoints that handle user data from *Spotify*.

The **Spotify Adapter module** serves seven endpoints:

- `GET /api/spotify/albums/{albumId}` — retrieves metadata for the specified *Spotify album*, covering the album's artists, title, track count as well as tracks, artwork, release date, *Spotify link*, the recording label, and the release type of the album
- `GET /api/spotify/artists/{artistId}` — retrieves metadata for the specified *Spotify artist*, covering the artist's albums, genres, name, artwork, and *Spotify link*
- `GET /api/spotify/playlists` — retrieves the current authorized user's playlists (using the custom *HTTP header*)

- `GET /api/spotify/playlists/{playlistId}` — retrieves metadata for the specified *Spotify playlist*, covering its creator, name, title, description, overall track count, artwork, *Spotify link*, and tracks (including just IDs or full track details)
- `GET /api/spotify/recently-played` — retrieves last 50 recently played tracks for the current authorized user (using the custom *HTTP header*)
- `GET /api/spotify/search` — leverages *Spotify Search*, it is mainly used for retrieving *Spotify IDs* for assigning to the metadata obtained through uploading a file
- `GET /api/spotify/tracks/{trackId}` — retrieves metadata for the specified *Spotify track*, covering the track's name, *ISRC code*, artists, album, duration in milliseconds, and *Spotify link*

4.2.5 Error Handling

If any request fails, an error handling mechanism using the **try..catch** clause is implemented. Upon catching an exception or an error, the server returns the error message as well as the related status code: *400* for any client's bad request, *401* if there is no access token present in the request, *403* if the access token was altered in any way or it cannot be decoded and parsed, *404* if the requested resource is not found, and *500* for any failure caused at the *back-end server*.

4.3 Front-end Client

4.3.1 Code-base Structure

The code-base structure is inspired by ***How I structure my React projects*** by *Lars Wächter*.

- **components** — includes reusable *UI components* utilized across the application
  - **alert** — includes reusable *Alert* components - like *Error Alert*, *Form Error Alert*, *Info Alert*, *Success Alert*, and *Warning Alert*
  - **icons** — holds icons and icon wrappers
  - **layout** — holds layout components such as *Navbar* and *Navigation Menus*
- **contexts** — contexts facilitate data transmission across the component hierarchy, eliminating the need for manual prop propagation at each layer. In the client application, a context is utilized to store the current user's information.

- **hooks** — hooks enable the extraction of reusable logic from components, like handling an access token
- **models** — includes datatypes that specify the configuration of data objects in the application, like enums for routing, *API URLs*, and more.
- **pages** — represent screens accessible to users. Each page is linked to a distinct route within the application and includes several components.
  - **auth** — folder for the *Authentication module*
  - **browse** — folder for the *Browse module*
  - **spotify** — folder for the *Spotify Adapter module*
  - **upload-file** — folder for the *Upload File module*
  - **user** — page for managing user profiles, including creation and display
  - **home.tsx** — home page
- **services** — includes utilities for communication with the *back-end server's API* such as an *axios HTTP client*, tools for inputting music metadata, and more.
- **utils** — includes helper functions to convert *ISO 8601* durations into milliseconds and to format durations from milliseconds into hours, minutes, and seconds
- **main.tsx** — the main entry point of the *front-end client* which renders the root component and sets up the application routing
- **routes.tsx** — specifies the routing setup for the application. It includes a collection of routes linked to their respective pages.
- **styles.css** — encompasses all the *CSS rules* applicable across the entire application. Given that the *front-end client* employs the *Tailwind CSS* library, this global style-sheet holds all the styles.

4.3.2 Authentication Module

The **Authentication module** implements *UC–1 — Register user* and *UC–2 — Login registered user* by providing the registration and log-in forms; and *UC–3 — Create user profile* and *UC–4 — Display user profile*. The module contains three pages — **Register User page**, **Login User page** and **User Profile page**.

Both pages have an identical structure — a simple form containing two fields: *email* and *password*. The **email form field** utilizes a standard *HTML email text input*, automatically verifying the correct email format. The **password**

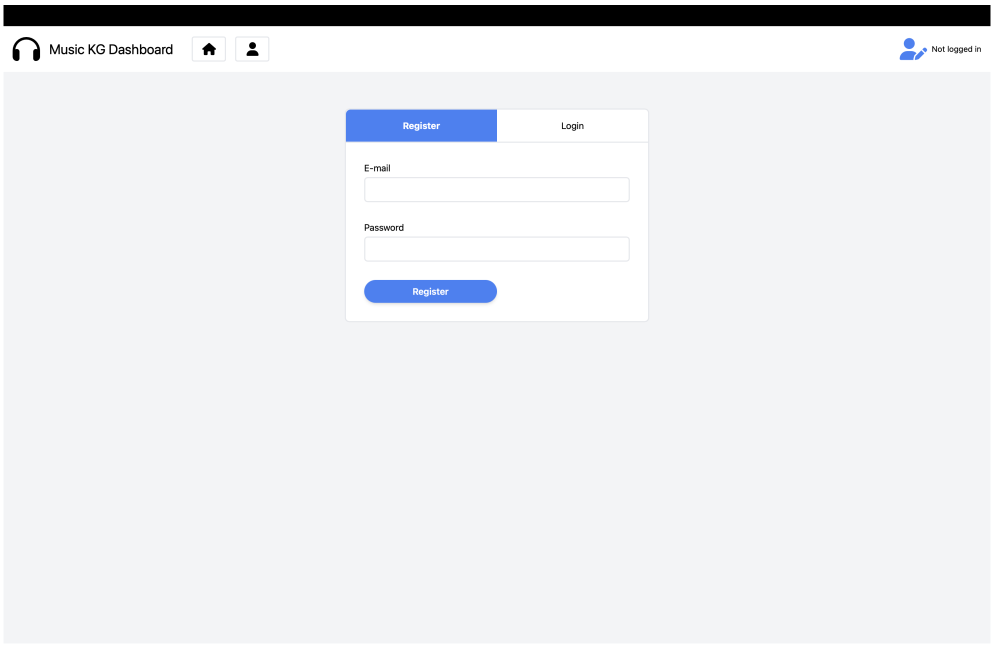
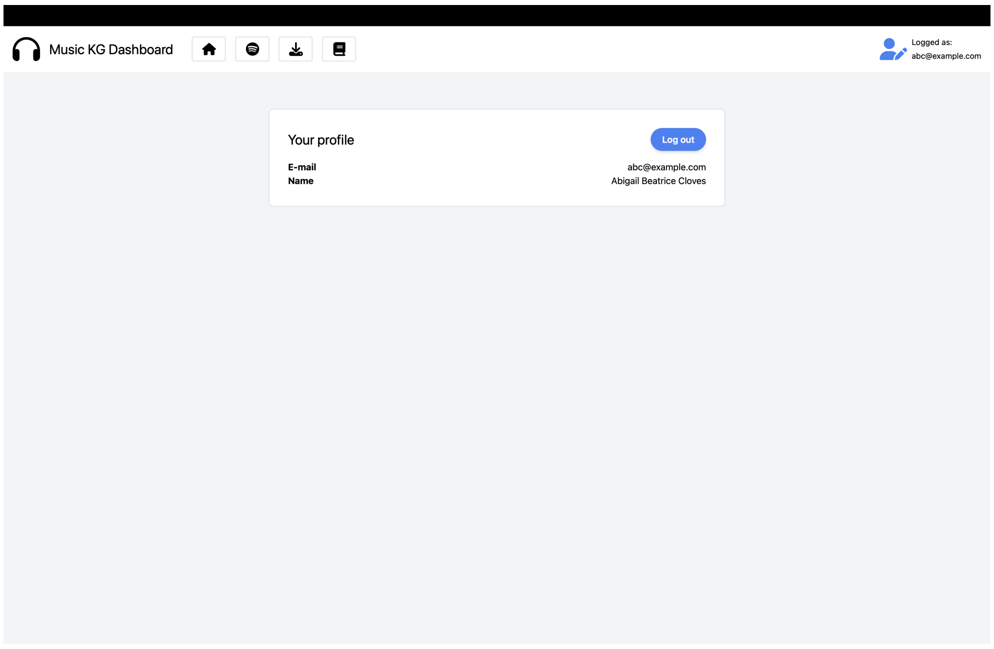


Figure 4.3 The *Register User* page in the **front-end client**



Your profile

E-mail	abc@example.com
Name	Abigail Beatrice Cloves

Figure 4.4 The *User Profile* page in the **front-end client**

form field uses a standard *HTML password text input*, concealing the text as it is entered.

After submitting a valid email address with a password to the system, it is sent to the **Music KG API**. If the *front-end client* receives a successful response, it shows a success alert with text. Upon receiving the error response, it displays an error alert with the error message. After successful log-in, the user is redirected to the **User Profile page**.

On the **User Profile page** system checks whether the user already exists in the **Music KG system** (i.e., that the user is present in the *Users graph* in the RDF store). If the user already exists, the system will show their profile, which currently contains their email address and name (see Figure **4.4**).

If the user does not yet exist, the system asks them to create a profile in the **Music KG system**. After the user submits their name to the *API*, the system creates the user profile in the *RDF store*. The example of the user in the *RDF store* may look like this:

```
PREFIX schema: <https://schema.org/>

<http://localhost:3030/music-kg/users/65
   e86c494668ff47d678eb94>
        a              schema:Person ;
        schema:email   "abc@example.com" ;
        schema:name    "Abigail Beatrice Cloves" .
```

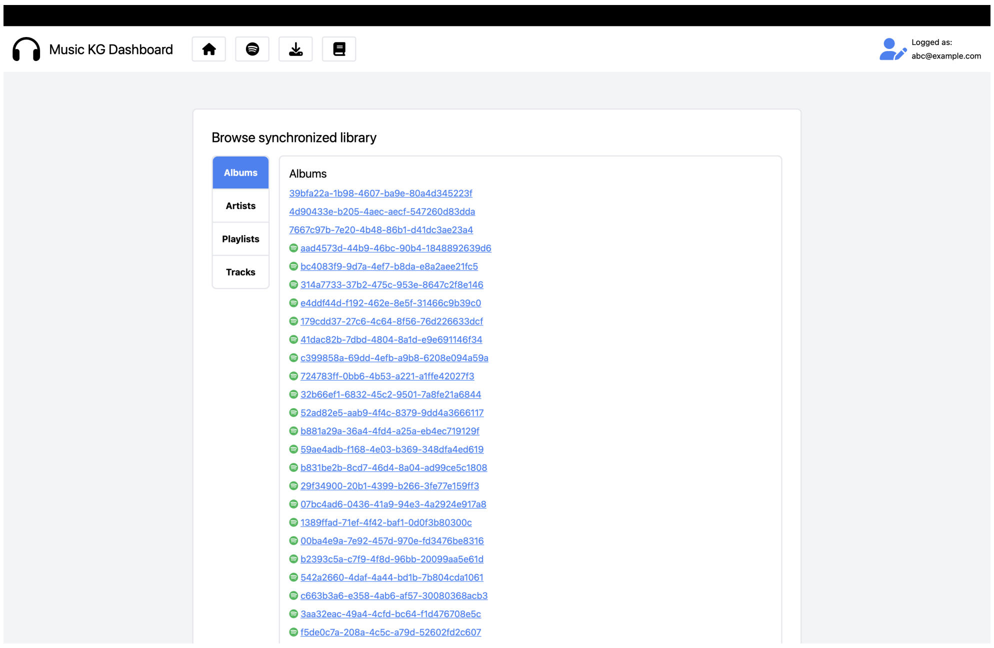
4.3.3 Browse Module

The **Browse module** implements the following use case scenarios: *UC–9 — Browse added albums*, *UC–10 — Show album detail*, *UC–11 — Browse added artists*, *UC–12 — Show artist detail*, *UC–13 — Browse added playlists*, *UC–14 - Show playlist detail*, *UC–15 — Browse added tracks*, and *UC–16 — Show track detail*.

Once authenticated, the user has the ability to browse the music metadata added to the *RDF store* through the browsing feature. Currently, users can browse in two modes: *list* and *detail*, across four different types of entities.

In the **list mode**, the system fetches the IDs of the chosen entity type (such as the album in Figure **4.5**) from the *RDF store* through an *API*. The system identifies the source of the entity (whether it is local or from *Spotify*, which is shown by the *Spotify icon* adjacent to the entity ID). All entity ID are clickable links which will open the details of the selected entity.

In the **detail mode**, the system will retrieve all properties associated with the selected entity. Each entity detail has different properties that they are showing (if they are available):


Music KG Dashboard

Logged as:
abc@example.com

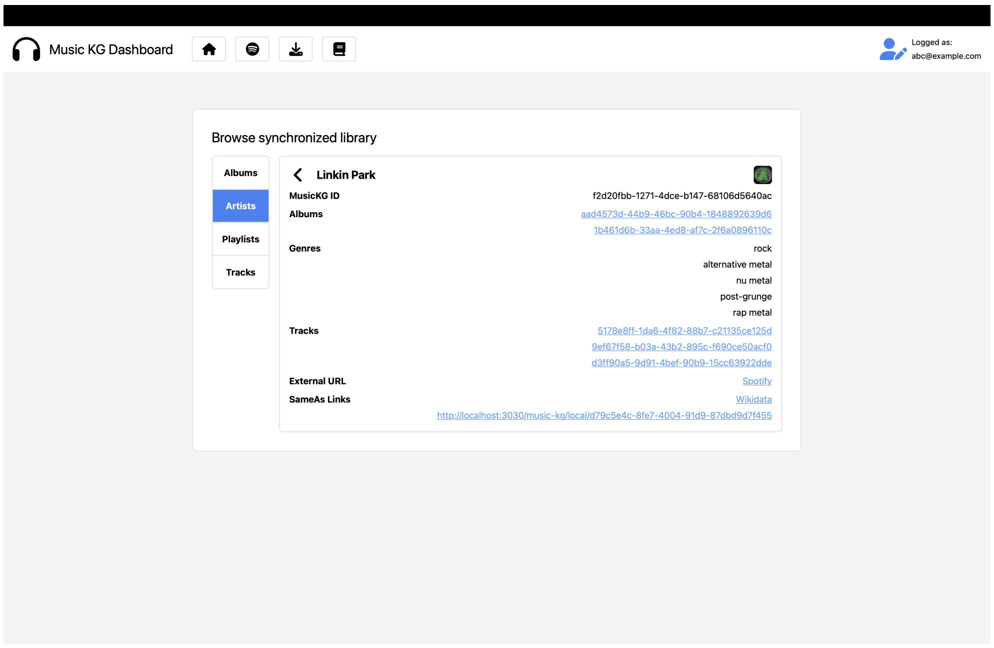
Browse synchronized library

Albums
Artists
Playlists
Tracks

Albums

39bfa22a-1b98-4607-ba9e-80a4d345223f
4d90433e-b205-4aec-aecf-547260d83dda
7667c97b-7e20-4b48-86b1-d41dc3ae23a4
● aad4573d-44b9-46bc-90b4-1848892639d6
● bc4083f9-9d7a-4ef7-b8da-e8a2aee21fc5
● 314a7733-37b2-475c-953e-8647c2f8e146
● e4ddf44d-f192-462e-8e5f-31466c9b39c0
● 179cdd37-27c6-4c64-8f56-76d226633dcf
● 41dac82b-7dbd-4804-8a1d-e9e691146f34
● c399858a-69dd-4efb-a9b8-6208e094a59a
● 724783ff-0bb6-4b53-a221-a1ffe42027f3
● 32b66ef1-6832-45c2-9501-7a8fe21a6844
● 52ad82e5-aab9-4f4c-8379-9dd4a3666117
● b881a29a-36a4-4fd4-a25a-eb4ec719129f
● 59ae4adb-f168-4e03-b369-348dfa4ed619
● b831be2b-8cd7-46d4-8a04-ad99ce5c1808
● 29f34900-20b1-4399-b266-3fe77e159ff3
● 07bc4ad6-0436-41a9-94e3-4a2924e917a8
● 1389ffad-71ef-4f42-baf1-0d0f3b80300c
● 00ba4e9a-7e92-457d-970e-fd3476be8316
● b2393c5a-c7f9-4f8d-96bb-20099aa5e61d
● 542a2660-4daf-4a44-bd1b-7b804cda1061
● c663b3a6-e358-4ab6-af57-30080368acb3
● 3aa32eac-49a4-4cfd-bc64-f1d476708e5c
● f5de0c7a-208a-4c5c-a79d-52602fd2c607

Figure 4.5 The *Albums* page in the **front-end client**


Music KG Dashboard

Logged as:
abc@example.com

Browse synchronized library

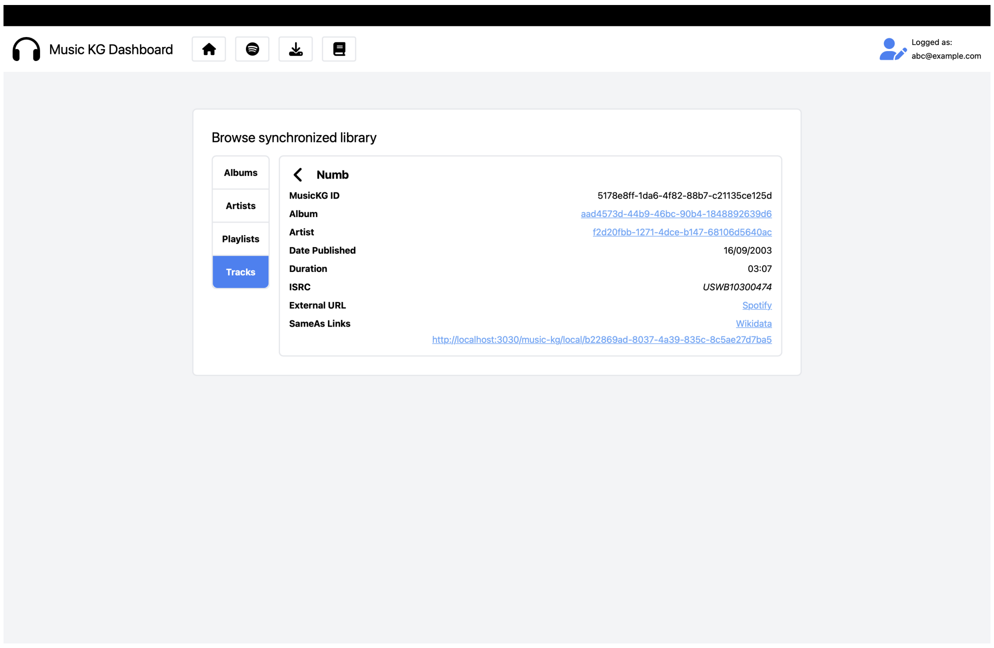
Albums
Artists
Playlists
Tracks

Linkin Park

MusicKG ID	f2d20fbb-1271-4dce-b147-68106d5640ac
Albums	aad4573d-44b9-46bc-90b4-1848892639d6 1b461d6b-33aa-4ed8-af7c-2f6a0896110c
Genres	rock alternative metal nu metal post-grunge rap metal
Tracks	5178e8ff-1da6-4f82-88b7-c21135ce125d 9ef67f58-b03a-43b2-895c-f690ce50acf0 d3ff90a5-9d91-4bef-90b9-15cc63922dde
External URL	Spotify
SameAs Links	Wikidata http://localhost:3030/music-kg/local/d79c5e4c-8fe7-4004-91d9-87dbd9d7f455

Figure 4.6 The *Artist Detail* page in the **front-end client**

- **Album detail** — album production type, album release type, artist (only ID, which is clickable link), date of publishing, number of tracks, and list of tracks (only IDs, which are clickable links)



Browse synchronized library

Numb

MusicKG ID	5178e8ff-1da6-4f82-88b7-c21135ce125d
Album	aad4573d-44b9-46bc-90b4-1848892639d6
Artist	f2d20fbb-1271-4dce-b147-68106d5640ac
Date Published	16/09/2003
Duration	03:07
ISRC	*USWB10300474*
External URL	Spotify
SameAs Links	Wikidata
	http://localhost:3030/music-kg/local/b22869ad-8037-4a39-835c-8c5ae27d7ba5

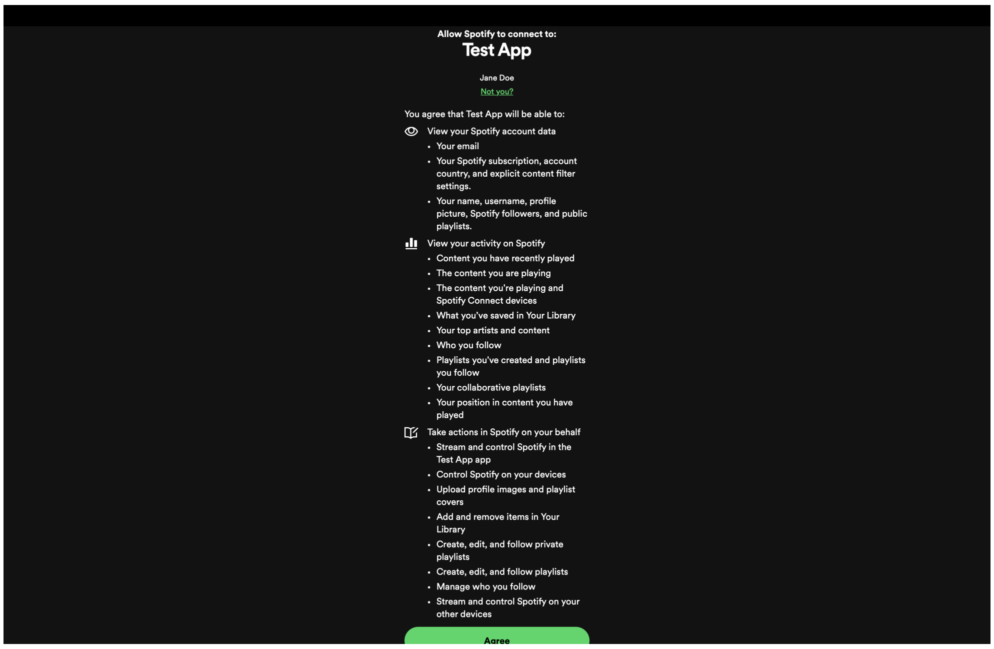
Figure 4.7 The *Track Detail* page in the **front-end client**

- **Artist detail** (seen in Figure **4.6**) — list of albums (only IDs, which are clickable links), list of genres, and list of tracks (only IDs, which are clickable links)
- **Playlist detail** — date of creation, date of modification, number of tracks, and list of tracks (only IDs, which are clickable links)
- **Track detail** (seen in Figure **4.7**) — album and artist(s) (only IDs, which are clickable links), date of publishing, duration, and *ISRC*

On top of that, each entity detail contain its *ID* within the **Music KG system**, *external URLs*, which lead to external services (such as *Spotify*), and *sameAs links* from the *Links graph*, either from the **Music KG system** or from the external knowledge bases (such as *Wikidata*).

4.3.4 Spotify Adapter Module

The **Spotify Adapter module** implements the following use case scenarios: *UC–17 — Authorize Spotify account*, *UC–18 — Obtain recently played tracks on Spotify*, *UC–19 — Obtain user's Spotify playlists*, *UC–20 — Add music track from Spotify to user's music library* and *UC–21 — Add Spotify playlist to user's music library*. The **Spotify Adapter** includes the *Spotify account authorization* feature, enabling users to authenticate their accounts, thus permitting the **Music KG system** to access their *Spotify data*. Once the account is authenticated, users have two options to incorporate data from the *Spotify API* into their knowledge graph: selecting from their *recently played tracks* or their *saved playlists*.



Allow Spotify to connect to:

Test App

Jane Doe

Not you?

You agree that Test App will be able to:

View your Spotify account data
- Your email
- Your Spotify subscription, account country, and explicit content filter settings.
- Your name, username, profile picture, Spotify followers, and public playlists.

View your activity on Spotify
- Content you have recently played
- The content you are playing
- The content you're playing and Spotify Connect devices
- What you've saved in Your Library
- Your top artists and content
- Who you follow
- Playlists you've created and playlists you follow
- Your collaborative playlists
- Your position in content you have played

Take actions in Spotify on your behalf
- Stream and control Spotify in the Test App app
- Control Spotify on your devices
- Upload profile images and playlist covers
- Add and remove items in Your Library
- Create, edit, and follow private playlists
- Create, edit, and follow playlists
- Manage who you follow
- Stream and control Spotify on your other devices

Figure 4.8 The *Spotify authorization* page

Access to the **Spotify Adapter module** requires user authentication. Once authenticated, the module can be reached through the navigation bar links. The **Authentication page** features only the *Authorize button* that initializes the *Spotify authorization* (see Figure **4.8**). After the user allows the application to access their data and is redirected back to the *front-end client*, they need to confirm the authorization by clicking the *Finish Authorization button*. This process saves the access token needed for interactions with the *Spotify API*, applying predefined **user scopes**. For simplicity, *all user scopes* are used in the configuration. The *Spotify access token* is sent to the **Music KG API** with every request to access *Spotify user data* using a custom `music-kg-spotify-authorization` *HTTP header*.

After the authorization is completed, the user can choose between either *Latest tracks* or *Playlists* to retrieve the data. The **Latest tracks** sub-module features a component to retrieve up to 50 latest tracks which user have listened to on *Spotify*. Users can then choose tracks from the list to open more information about them. Additional information include *duration*, *track number* in the track's album, *ISRC*, and *URI* within the *Spotify service*, as well as information about the track's album such as: *album name*, *album's artists*, *album type*, *total number of tracks*, and *album's release date*. The **Spotify**

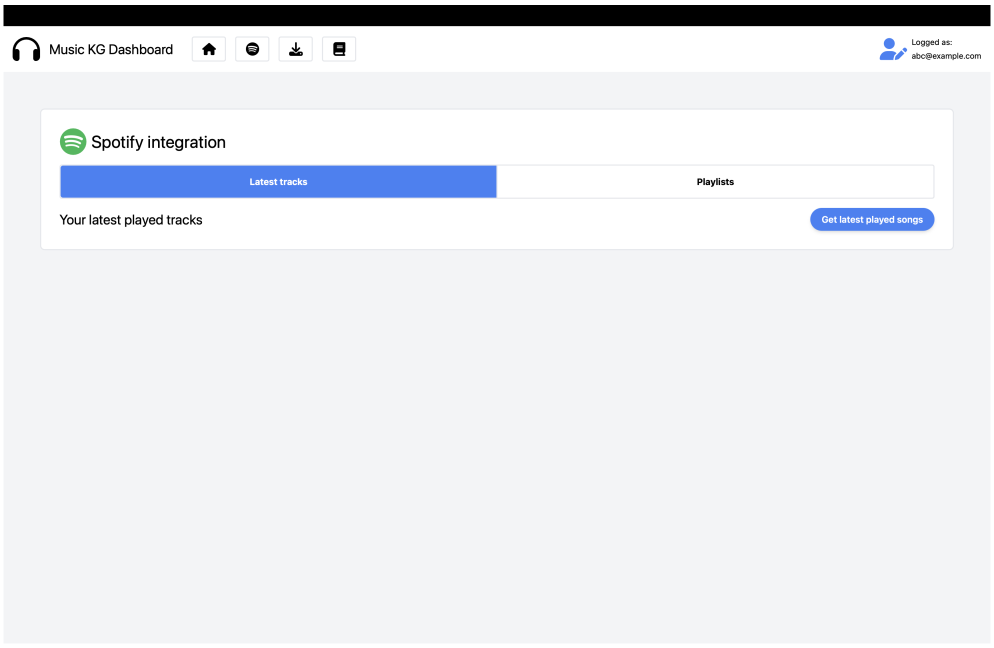
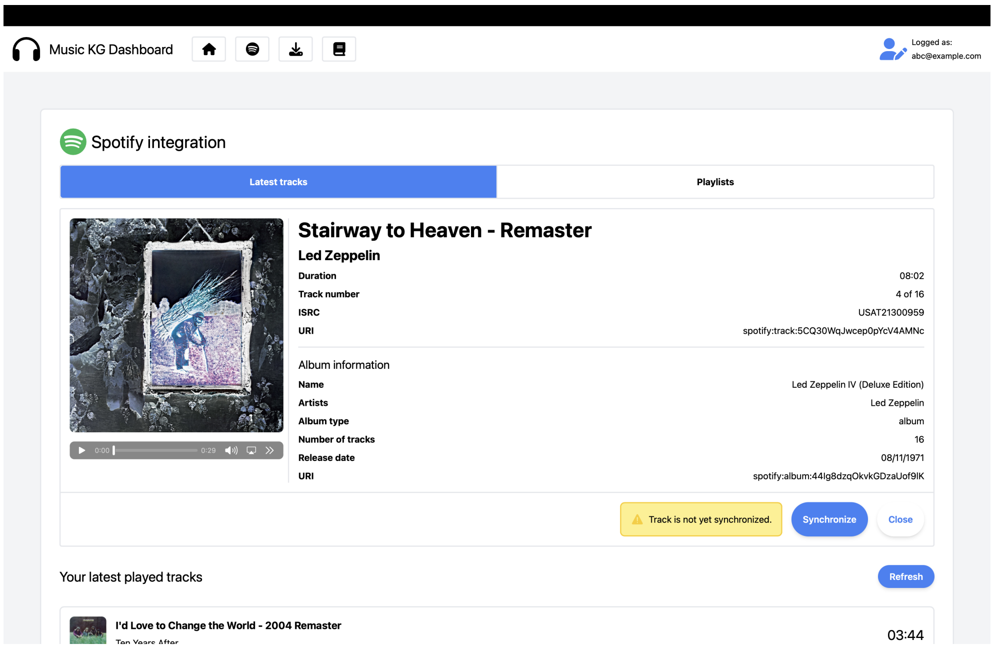


Figure 4.9 The *Spotify Adapter* main page in the **front-end client**



Stairway to Heaven - Remaster

Led Zeppelin

Duration	08:02
Track number	4 of 16
ISRC	USAT21300959
URI	spotify:track:5CQ30WqJwcep0pYcV4AMNc

Album information

Name	Led Zeppelin IV (Deluxe Edition)
Artists	Led Zeppelin
Album type	album
Number of tracks	16
Release date	08/11/1971
URI	spotify:album:44Ig8dzqOkvkGDzaUof9lK

Track is not yet synchronized.

I'd Love to Change the World - 2004 Remaster 03:44

Ten Years After

Figure 4.10 The *Spotify Track Detail* in the **Latest tracks sub-module**

Track Detail component (shown in Figure **4.10**) also shows the *artwork of album* and, if available, the *30-second preview* audio. To add the chosen track to the user's music knowledge graph, the *Synchronize button* is utilized.

This action initiates a service that generates a new track using the data from *Spotify* for inclusion in the *RDF store*. Additionally, the system connects the track to the relevant artist(s) and album, creating or updating records for the artist or album as necessary. This process allows for the storage of *all available information* with a single click.

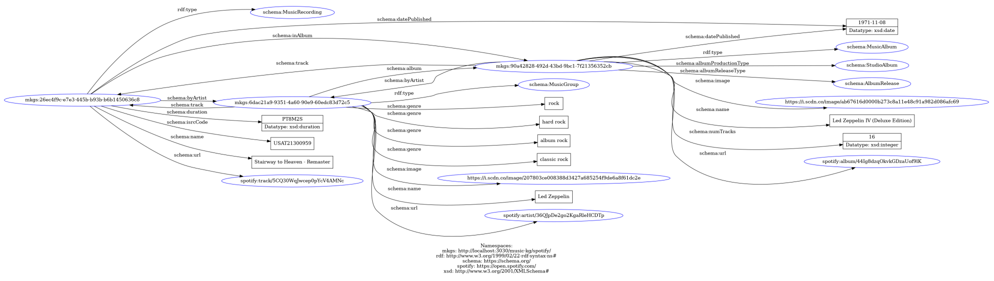

rdf:type
schema:MusicRecording
schema:datePublished
schema:inAlbum
schema:track
schema:album
schema:byArtist
rdf:type
mkgs:90a42828-492d-43bd-9bc1-7f21356352cb
1971-11-08
Datatype: xsd:date
schema:datePublished
rdf:type
schema:MusicAlbum
schema:albumProductionType
schema:StudioAlbum
schema:albumReleaseType
schema:AlbumRelease
schema:image
https://i.scdn.co/image/ab67616d0000b273c8a11e48c91a982d086afc69
schema:name
Led Zeppelin IV (Deluxe Edition)
schema:numTracks
16
Datatype: xsd:integer
schema:url
spotify:album/44Ig8dzqOkvkGDzaUof9lK
mkgs:26ec4f9c-e7e3-445b-b93b-b6b1450636c8
schema:byArtist
schema:track
schema:duration
schema:isrcCode
schema:name
schema:url
mkgs:6dac21a9-9351-4a60-90e9-60edc83d72c5
schema:MusicGroup
PT8M2S
Datatype: xsd:duration
USAT21300959
Stairway to Heaven - Remaster
spotify:track/5CQ30WqJwcep0pYcV4AMNc
schema:genre
rock
schema:genre
hard rock
schema:genre
album rock
schema:genre
classic rock
schema:image
https://i.scdn.co/image/207803ce008388d3427a685254f9de6a8f61dc2e
schema:name
Led Zeppelin
schema:url
spotify:artist/36QJpDe2go2KgaRleHCDTp
Namespaces:
mkgs: http://localhost:3030/music-kg/spotify/
rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns#
schema: https://schema.org/
spotify: https://open.spotify.com/
xsd: http://www.w3.org/2001/XMLSchema#

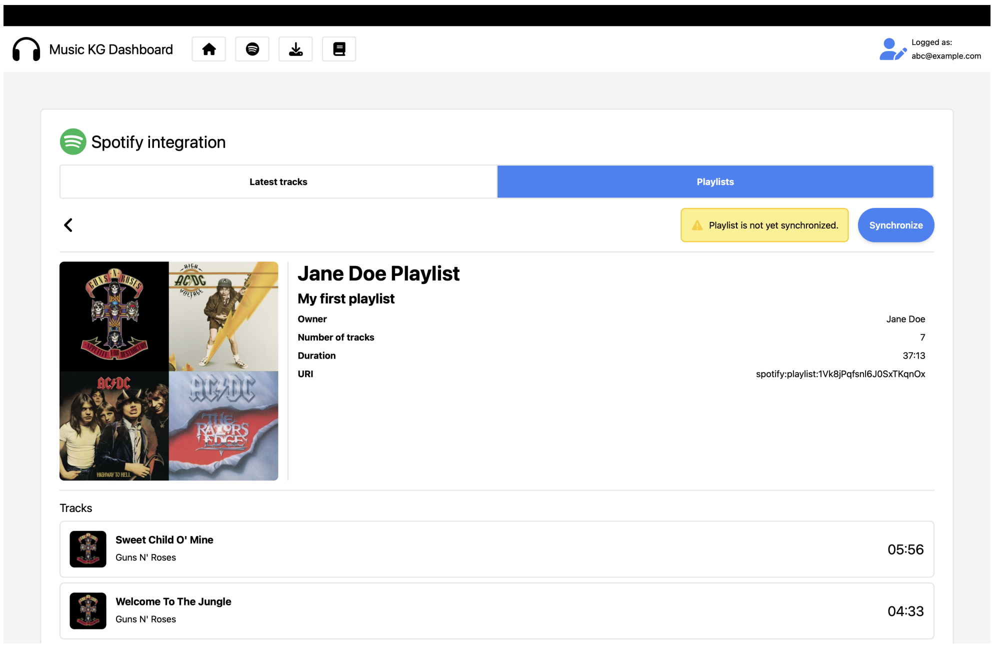
Figure 4.11 The *RDF graph* visualization after inserting one track from *Spotify*

For example, when adding a track to the user's knowledge graph, multiple HTTP requests are done. The resulting *RDF graph* can be seen in Figure **4.11**. The system:

1. creates a new track and, if they are not already present (verified by searching for the entities using the given *Spotify URL*), it also creates new entries for the album and artists (only including their *Spotify URL* and *name*)
2. searches for an already existing or newly created album within the user's knowledge graph and updates it with the gathered information, and also associating the *album* with the *newly created track*
3. searches for an artist, whether newly created or already existing in the user's knowledge graph, gathers additional details like the artist's genres and artwork, and updates the existing record. This includes associating the *artist* with both the *newly created track* and the *album*.

The **Playlists sub-module** gathers and shows the user's playlists, including the count of tracks in each. When a user selects a playlist, the **Spotify Playlist Detail component** (referenced in Figure **4.12**) appears. This component resembles the **Spotify Track Detail component** but offers unique details like the *playlist's owner*, the *track count*, and the *total duration of the playlist*. The interface also lists the tracks in the playlist. Choosing a track will launch the **Spotify Track Detail component** for that specific track, which is also part of this sub-module.

The process of adding a playlist to the user's knowledge graph mirrors the method used for a single track, with the initial step of inserting the playlist followed by applying the previously described steps to each individual track



Music KG Dashboard

Logged as: abc@example.com

Spotify integration

Latest tracks | Playlists

Playlist is not yet synchronized. Synchronize

Jane Doe Playlist

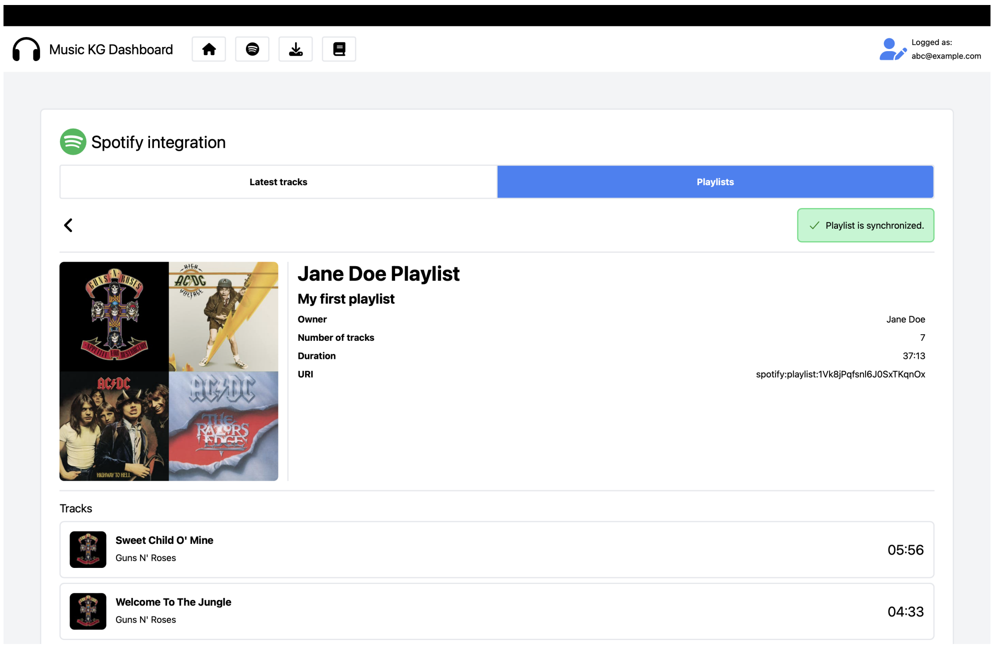
My first playlist

Owner	Jane Doe
Number of tracks	7
Duration	37:13
URI	spotify:playlist:1Vk8jPqfsnl6J0SxTKqnOx

Tracks

Sweet Child O' Mine Guns N' Roses	05:56
Welcome To The Jungle Guns N' Roses	04:33

Figure 4.12 The *Spotify Playlist Detail* component in the **Playlists** sub-module



Music KG Dashboard

Logged as: abc@example.com

Spotify integration

Latest tracks | Playlists

Playlist is synchronized.

Jane Doe Playlist

My first playlist

Owner	Jane Doe
Number of tracks	7
Duration	37:13
URI	spotify:playlist:1Vk8jPqfsnl6J0SxTKqnOx

Tracks

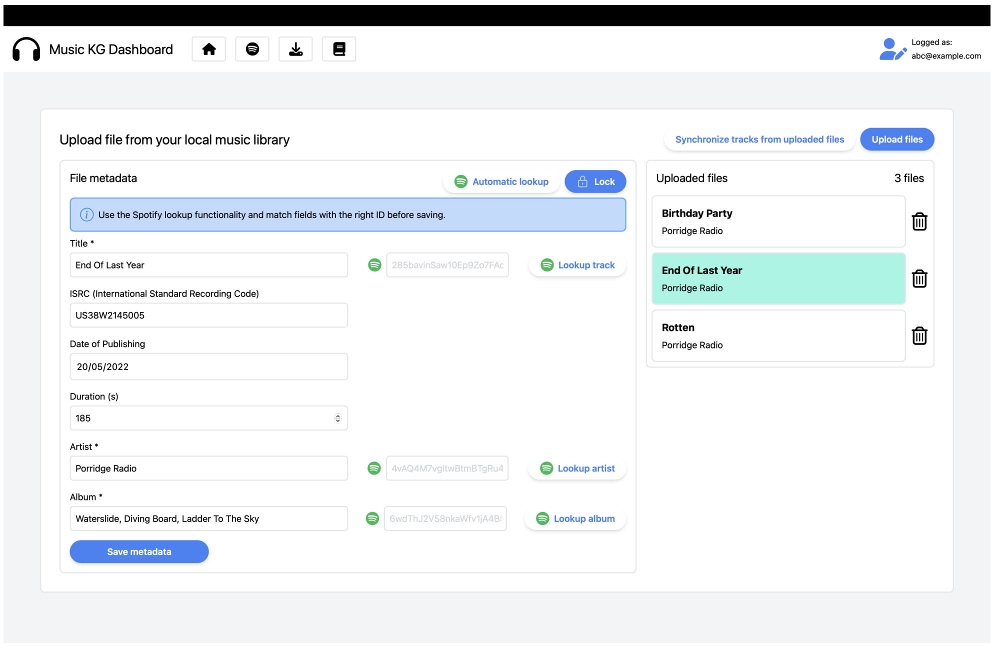
Sweet Child O' Mine Guns N' Roses	05:56
Welcome To The Jungle Guns N' Roses	04:33

Figure 4.13 The *Spotify Playlist Detail* component showing a recently added playlist

within that playlist. Once the playlist is incorporated into the knowledge graph, the *Alert component* adjacent to the *Synchronize button* updates to reflect that the playlist has been added (see Figure **4.13**).

4.3.5 Upload File Module

The last module featured in the *front-end client* is the **Upload File module**. The *Upload File module* implements the following use case scenarios: *UC–5 — Upload media file*, *UC–6 — Edit uploaded media file*, *UC–7 — Assign Spotify IDs to uploaded media file*, and *UC–8 — Add new music tracks from uploaded media files*.



Upload file from your local music library

File metadata

Use the Spotify lookup functionality and match fields with the right ID before saving.

Title *
End Of Last Year
285bavinSaw10Ep9Zo7FAc

ISRC (International Standard Recording Code)
US38W2145005

Date of Publishing
20/05/2022

Duration (s)
185

Artist *
Porridge Radio
4vAQ4M7vgltwBtmBTgRu4

Album *
Waterslide, Diving Board, Ladder To The Sky
6wdThJ2V58nkaWfv1jA4B!

Uploaded files 3 files

Birthday Party
Porridge Radio

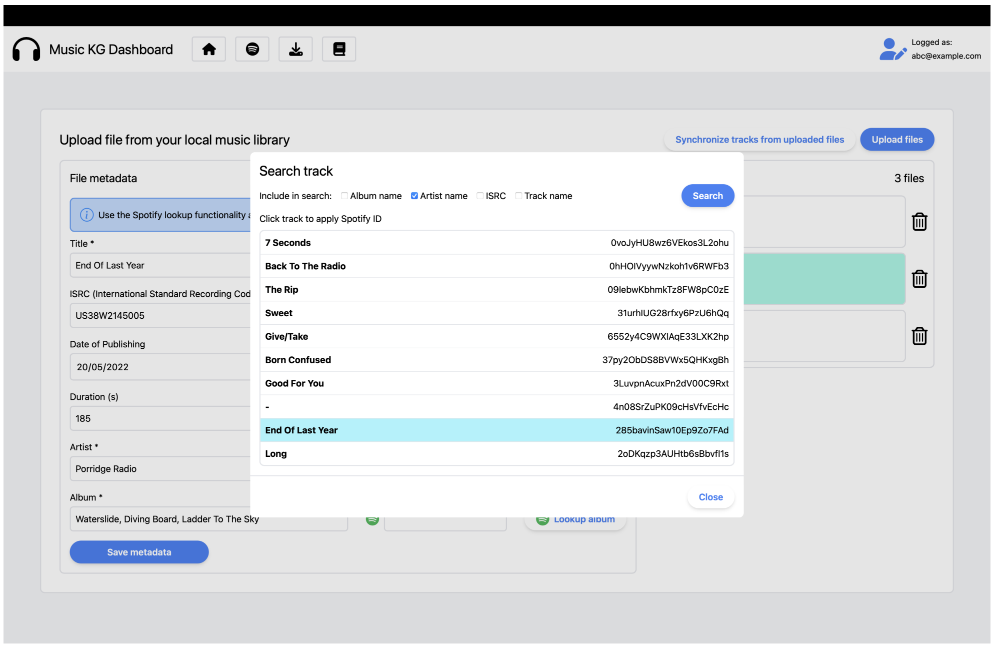
End Of Last Year
Porridge Radio

Rotten
Porridge Radio

Figure 4.14 The *Upload File* page in the **front-end client**

Like the modules mentioned above, the **Upload File module** also requires user authentication. This module consists of a single page, the **Upload File page** (shown in Figure **4.14**). On this page, users have the capability to upload one or several media files, which the system will then process. Following this, users can modify the processed metadata and incorporate these tracks into their knowledge graph.

By clicking the *Upload files button*, users can select a single file or multiple files using the native *HTML file input*. Once the files are uploaded and processed, they are included in the list of processed files. The available *MP3 metadata tags* are shown in Table **2.1** or are available on the **music-metadata-browser GitHub repository** website. Users have the option to select metadata for a file to edit. When a processed file is selected, an editing form appears in a locked state. To modify the form, users must first unlock it by pressing the *Edit button*. This feature prevents accidental modifications of the metadata. The editing form includes mandatory fields like *track title*, *artist name*, and *album name*, as well as optional fields such as *ISRC*, *date of publishing*, and *duration*. Should the system successfully extract values during file upload, these are automatically populated in the form.


Music KG Dashboard
Logged as:
abc@example.com
Upload file from your local music library
Synchronize tracks from uploaded files
Upload files
File metadata
3 files
Use the Spotify lookup functionality
Search
Lookup album
Save metadata
Close

Title *
End Of Last Year

ISRC (International Standard Recording Cod
US38W2145005

Date of Publishing
20/05/2022

Duration (s)
185

Artist *
Porridge Radio

Album *
Waterslide, Diving Board, Ladder To The Sky

Search track

Include in search: ☐ Album name [✓ Artist name] ☐ ISRC ☐ Track name

Click track to apply Spotify ID

7 Seconds	0voJyHU8wz6VEkos3L2ohu
Back To The Radio	0hHOIVyywNzkoh1v6RWFb3
The Rip	09lebwKbhmkTz8FW8pC0zE
Sweet	31urhIUG28rfxy6PzU6hQq
Give/Take	6552y4C9WXlAqE33LXK2hp
Born Confused	37py2ObDS8BVWx5QHKxgBh
Good For You	3LuvpnAcuxPn2dV00C9Rxt
-	4n08SrZuPK09cHsVfvEcHc
End Of Last Year	285bavinSaw10Ep9Zo7FAd
Long	2oDKqzp3AUHtb6sBbvfl1s

Figure 4.15 The *Search* component on the **Upload File** page

To establish connections between different entities in the **Music KG system**, it is essential to use a specific ID for linking. Given that associated graphs such as the *Spotify graph* and the *Wikidata dump graph* utilize *Spotify IDs*, the **Upload File** page includes a feature to access the *Spotify API* and align the extracted metadata with the information from *Spotify*. During this process, users must select the appropriate *Spotify ID* to ensure accurate linking and prevent data errors. To facilitate this, the **Search component** is available (shown in Figure **4.15**). It enables searching the *Spotify API* through the **Music KG API**. It offers various search parameters for each type of entity (referencing the **Spotify API search parameters**): *artist name* (applicable to artists, albums, and tracks), *album name* (for albums and tracks), *track name*, and *ISRC* (exclusively for tracks). After the user controls the parsed metadata, they can add the tracks to their knowledge graph. The procedure is explained in the **Spotify Adapter module**.

4.3.6 Error Handling

Similarly to the *back-end server*, the *front-end client* utilizes an error handling strategy with the **try..catch** clause. When an exception or error is captured, the application displays an error alert along with the message. This approach guarantees that users are kept aware of the application's status and understand the reasons behind the request's failure.

4.4 Data Type Libraries

There are two supporting *data type libraries* in the **Music KG system**, which define data types and utility functions utilized by both the *back-end server* and the *front-end client.*

1. **data** library — includes data types for *back-end server endpoints* (types for request bodies and response bodies), primarily to avoid redundant definitions required by both applications

2. **sparql-data** library — includes data types for parsing *SPARQL Query Results* (e.g., extracting properties from *bindings*), and provides wrappers for *SPARQL.js* datatypes like *IriTerm*, *LiteralTerm*, *VariableTerm*, and *BlankTerm.* It also offers ontology and vocabulary wrappers for *RDFS*, *Schema.org*, *XML Schema datatypes*, and *Wikidata* in *TypeScript* to improve the developer experience.

4.5 RDF Store

The user knowledge graphs are preserved within the *RDF store*, utilizing **Apache Jena Fuseki**. This software is provided as an Out-of-the-Box solution, which inherently includes extensive pre-configurations. **Apache Jena Fuseki** is equipped with a **SPARQL endpoint** for handling *SPARQL* queries, a **SPARQL Update service** that manages data manipulation queries such as *INSERT DATA*, *DELETE*, and *DELETE/INSERT* using the *SPARQL Update protocol*, and a **Graph Store Protocol service** that facilitates the creation, modification, and removal of named graphs.

In the **Music KG system**, four distinct named graphs are present. These include the **Local graph** and the **Spotify graph**, which contains user-provided data; the **Wikidata dump graph** (refer to Section **4.6**), which stores music-related information from *Wikidata knowledge base*; and the **Links graph**, which contains *sameAs links* connecting entities across the aforementioned graphs. The unnamed (default) graph remains unused.

The interaction with the RDF store occurs by sending *HTTP(S) POST* or *GET* request to the primary endpoint `<BASE_URL>/music-kg`. The request includes a *SPARQL* query to run in the *RDF store*. Detailed explanations of the *SPARQL* queries available in the **Music KG system** can be found in Section **4.2**. Alternatively, direct interaction with the *RDF store* can be achieved through the *SPARQL endpoint*.

Some examples of the *RDF data* from the user's knowledge graph stored in the **RDF store**:
From the **Spotify graph**:

```
PREFIX mkgs:    <http://localhost:3030/music-kg/spotify/>
PREFIX schema:  <https://schema.org/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

mkgs:aad4573d-44b9-46bc-90b4-1848892639d6
    a                            schema:MusicAlbum;
    schema:albumProductionType   schema:StudioAlbum;
    schema:albumReleaseType      schema:AlbumRelease;
    schema:byArtist              mkgs:f2d20fbb-1271-4dce-
       b147-68106d5640ac;
    schema:datePublished         "2003-09-16"^^xsd:date;
    schema:image                 <https://i.scdn.co/image/
       ab67616d0000b2735f1f51d14e8bea89484ecd1b>;
    schema:name                  "Meteora";
    schema:numTracks             13;
    schema:track                 mkgs:5178e8ff-1da6-4f82
       -88b7-c21135ce125d;
    schema:url                   <https://open.spotify.com
       /album/4Gfnly5CzMJQqkUFfoHaP3> .

mkgs:f2d20fbb-1271-4dce-b147-68106d5640ac
    a               schema:MusicGroup;
    schema:album    mkgs:aad4573d-44b9-46bc-90b4
       -1848892639d6 , mkgs:1b461d6b-33aa-4ed8-af7c-2
       f6a0896110c;
    schema:genre    "rock" , "alternative metal" , "nu
       metal" , "post-grunge" , "rap metal";
    schema:image    <https://i.scdn.co/image/
       ab6761610000e5eb84a0dd74f21e8acce6a9fd49>;
    schema:name     "Linkin Park";
    schema:track    mkgs:5178e8ff-1da6-4f82-88b7-
       c21135ce125d , mkgs:9ef67f58-b03a-43b2-895c-
       f690ce50acf0 , mkgs:d3ff90a5-9d91-4bef-90b9-15
       cc63922dde;
    schema:url      <https://open.spotify.com/artist/6
       XyY86QOPPrYVGvF9ch6wz> .

mkgs:5178e8ff-1da6-4f82-88b7-c21135ce125d
    a                      schema:MusicRecording;
    schema:byArtist        mkgs:f2d20fbb-1271-4dce-b147
```

```
        -68106d5640ac;
    schema:datePublished  "2003-09-16"^^xsd:date;
    schema:duration       "PT3M7S"^^xsd:duration;
    schema:inAlbum        mkgs:aad4573d-44b9-46bc-90b4
        -1848892639d6;
    schema:isrcCode       "USWB10300474";
    schema:name           "Numb";
    schema:url            <https://open.spotify.com/track
        /2nLtzopw4rPReszdYBJU6h> .
```

From the **Local graph**:

```
PREFIX mkgl:    <http://localhost:3030/music-kg/local/>
PREFIX schema:  <https://schema.org/>
PREFIX xsd:     <http://www.w3.org/2001/XMLSchema#>

mkgl:39bfa22a-1b98-4607-ba9e-80a4d345223f
    a            schema:MusicAlbum;
    schema:name  "Waterslide, Diving Board, Ladder To The
         Sky";
    schema:url   <https://open.spotify.com/album/6
        wdThJ2V58nkaWfv1jA4B5> .

mkgl:16fe8760-cecc-4350-bb37-25cdf1702939
    a            schema:MusicGroup;
    schema:name  "Porridge Radio";
    schema:url   <https://open.spotify.com/artist/4
        vAQ4M7vgItwBtmBTgRu48> .

mkgl:4bf56e7e-0226-417f-a462-aec4f5bb03a7
    a                     schema:MusicRecording;
    schema:byArtist       mkgl:16fe8760-cecc-4350-bb37-25
        cdf1702939;
    schema:datePublished  "2022-05-20"^^xsd:date;
    schema:duration       "PT3M7S"^^xsd:duration;
    schema:inAlbum        mkgl:39bfa22a-1b98-4607-ba9e-80
        a4d345223f;
    schema:isrcCode       "US38W2145002";
    schema:name           "Back To The Radio";
    schema:url            <https://open.spotify.com/track
        /0hHOIVyywNzkoh1v6RWFb3> .
```

From the **Links graph**:

```
PREFIX mkgl:   <http://localhost:3030/music-kg/local/>
PREFIX mkgs:   <http://localhost:3030/music-kg/spotify/>
PREFIX schema: <https://schema.org/>
PREFIX wd:     <http://www.wikidata.org/entity/>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

mkgs:aad4573d-44b9-46bc-90b4-1848892639d6
```

```
    schema:sameAs  mkgl:4d90433e-b205-4aec-aecf-547260
       d83dda .

mkgs:f2d20fbb-1271-4dce-b147-68106d5640ac
    schema:sameAs  wd:Q261 , mkgl:d79c5e4c-8fe7-4004-91d9
       -87dbd9d7f455 .

mkgs:5178e8ff-1da6-4f82-88b7-c21135ce125d
    schema:sameAs  wd:Q19973 , mkgl:b22869ad-8037-4a39
       -835c-8c5ae27d7ba5 .

mkgs:363a6ef5-8b2c-4603-83f4-fd5aedc67f28
    schema:sameAs  mkgl:74710489-2e7f-48f4-b6b3-7544
       f8e1dd97 .

mkgs:009b870d-82e0-405c-a100-dcf01045cde8
    schema:sameAs  wd:Q104667947 .
```

4.5.1 SHACL Shapes

To ensure that the *RDF data* in the *RDF store* align with a specified schema, a set of **SHACL shapes** were developed. These shapes adhere to the ontology outlined in Section **3.2.3**. In total, 28 *property shapes* (including 12 *value type constraint components*, 10 *cardinality constraint components*, 1 *value range constraint component*, 2 *string-based constraint components*, 1 *property pair constraint component* and 2 *other constraint components* (using *sh:in*)) and 6 *node shapes* were developed. For each presented example, the following **prefixes** are considered:

```
@prefix mkgsh: <http://localhost:3030/music-kg/shacl/> .
@prefix schema: <https://schema.org/> .
@prefix sh: <http://www.w3.org/ns/shacl#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
```

Value Type Constraint Components

The developed **value type constraint component** shapes validate the *correct class usage for objects* or the *correct datatype usage for properties*. For example, the first shape checks that every resource which is found in the triples where the predicate equals *schema:album* as an object is an *IRI* and that it has the *schema:MusicAlbum class*. The second shape checks that dates of publishing have the *xsd:date* datatype:

```
mkgsh:AlbumClassPropertyShape a sh:PropertyShape ;
    sh:path schema:album ;
    sh:class schema:MusicAlbum ;
    sh:nodeKind sh:IRI ;
    sh:name "album class"@en ;
    sh:message "album must be an instance of the
       MusicAlbum class."@en ;
    sh:severity sh:Violation .

mkgsh:DatePublishedDatatypePropertyShape a sh:
   PropertyShape ;
    sh:path schema:datePublished ;
    sh:datatype xsd:date ;
    sh:name "datePublished datatype"@en ;
    sh:message "datePublished should use datatype xsd:
       date."@en ;
    sh:severity sh:Warning .
```

Cardinality Constraint Components

The developed **cardinality constraint component** shapes verify the appropriate count of specified predicates (for example, each entity must have precisely one *name*, each track should belong to only one *album*, each playlist should have just one *description*, each album must feature only one *production type* and a single *release type*, and every track and album should have a unique *date of publishing*). The following example shows the first mentioned shape:

```
mkgsh:NameQuantityPropertyShape a sh:PropertyShape ;
    sh:path schema:name ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
    sh:maxCount 1 ;
    sh:name "name quantity"@en ;
    sh:message "Entity must have exactly one name."@en ;
    sh:severity sh:Violation .
```

Value Range Constraint Components

The only **value range constraint component** shape that was developed verifies that the *number of tracks* in any album or playlist is a positive integer.

```
mkgsh:NumTracksPropertyShape a sh:PropertyShape ;
    sh:path schema:numTracks ;
    sh:minExclusive 0 ;
    sh:datatype xsd:integer ;
    sh:name "numTracks validation"@en ;
    sh:message "numTracks must be positive integer."@en ;
    sh:severity sh:Violation .
```

String-based Constraint Components

The developed **string-based constraint component** shapes offer **pattern validations** for *emails* and *ISRC codes*.

```
mkgsh:EmailPatternPropertyShape a sh:PropertyShape ;
    sh:path schema:email ;
    sh:datatype xsd:string ;
    sh:pattern "^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\\.[A-Za
       -z]{2,}$" ;
    sh:name "email pattern"@en ;
    sh:message "Invalid email format."@en ;
    sh:severity sh:Violation .

mkgsh:ISRCCodePatternPropertyShape a sh:PropertyShape ;
    sh:path schema:isrcCode ;
    sh:datatype xsd:string ;
    sh:pattern "^[A-Z]{2}[A-Z0-9]{3}\\d{7}$" ;
    sh:name "isrcCode pattern"@en ;
    sh:message "Invalid ISRC code format."@en ;
    sh:severity sh:Violation .
```

Property Pair Constraint Components

The only **property pair constraint component** shape that was developed checks whether the *date of creation* of the playlist is *earlier or the same* as the *date of its modification*.

```
mkgsh:
   DateCreatedLessThanOrEqualsDateModifiedPropertyShape a
    sh:PropertyShape ;
    sh:path schema:dateCreated ;
    sh:lessThanOrEquals schema:dateModified ;
    sh:name "dateCreated-dateModified chronological
       sequence"@en ;
    sh:message "dateModified is earlier than dateCreated"
       @en ;
    sh:severity sh:Violation .
```

Other Constraint Components

The **other constraint components** developed are designed to verify whether the property adheres to an allowed value of an enumerated type. Within the **Music KG system**, both the *production type* and *release type* of the albums are derived from an enumeration type. The following shapes verifies that the album's production type is one of the allowed values:

```
mkgsh:AlbumProductionTypeEnumValuePropertyShape a sh:
   PropertyShape ;
```

```
    sh:path schema:albumProductionType ;
    sh:in (
        schema:CompilationAlbum
        schema:DJMixAlbum
        schema:DemoAlbum
        schema:LiveAlbum
        schema:MixtapeAlbum
        schema:RemixAlbum
        schema:SoundtrackAlbum
        schema:SpokenWordAlbum
        schema:StudioAlbum
    ) ;
    sh:name "albumProductionType enum value"@en ;
    sh:message "albumProductionType must be one of these
       values: (schema:CompilationAlbum, schema:
       DJMixAlbum, schema:DemoAlbum, schema:LiveAlbum,
       schema:MixtapeAlbum, schema:RemixAlbum, schema:
       SoundtrackAlbum, schema:SpokenWordAlbum, or schema
       :StudioAlbum)."@en ;
    sh:severity sh:Violation .
```

Node Shapes

The developed **node shapes** incorporate property shapes to provide different validations for the whole targeted class. The following example presents a node shape that validates entities of the *schema:MusicRecording class*. It checks:

- whether the entity's *artists and album belong to the correct class*,
- whether *all properties have the correct cardinalities*,
- and whether the provided *ISRC follows the right pattern*

```
mkgsh:MusicRecordingShape a sh:NodeShape ;
    sh:targetClass schema:MusicRecording ;
    sh:property
        mkgsh:ByArtistClassPropertyShape,
        mkgsh:DatePublishedDatatypePropertyShape,
        mkgsh:DatePublishedQuantityPropertyShape,
        mkgsh:DurationDatatypePropertyShape,
        mkgsh:DurationQuantityPropertyShape,
        mkgsh:InAlbumClassPropertyShape,
        mkgsh:InAlbumQuantityPropertyShape,
        mkgsh:ISRCCodePatternPropertyShape,
        mkgsh:ISRCCodeQuantityPropertyShape .
```

To trigger the *RDF validation*, an *HTTP POST request* should be made, including the *Shapes Graph* in *Turtle format*. The **curl command** to retrieve a validation report from the *SHACL service* is structured as follows (using `?graph=union` to perform the validation on all graphs within the *RDF store*):

```
curl -XPOST --data-binary shacl.ttl  \
 --header 'Content-type: text/turtle' \
 'http://localhost:3030/music-kg/shacl?graph=union'
```

4.6 Wikidata Dump

To effectively manage large datasets and knowledge graphs, utilizing pre-loaded data is advantageous as it avoids the need for repeated service queries every time data access is required. **Data dumps**, which are pre-loaded files, may contain complete or partial datasets. Typically, data providers release these dumps at regular or sporadic intervals. These can then be downloaded and used without burdening the system with excessive queries.

Wikidata offers their knowledge base as a **data dump** in multiple formats — *JSON*, *XML* and *RDF*. Since it contains over 1.4 billion triples, it is almost impossible to work with the whole data dump at once. Unfortunately, smaller data dumps (e.g. based on some topic or some domain) are not available. To facilitate the use of the *Wikidata's music-knowledge base*, the creation of a specialized data dump containing solely music-related information was considered. Various methods for this were explored.

1. **Downloading the whole data dump and filtering the music-related data** — the first question is: *How to choose the correct data to be included in the custom specialized dump?* Insights into this can be discovered within the **WikiProject: Music** segment of Wikidata's ontology, specifically through the use of relevant classes and properties associated with music. Various entity classes detail musical-related entities: **musical ensemble (*Q2088357*)** — 113669 unique entities, **musician (*Q639669*)** — 384665 unique entities, **audio track (*Q7302866*)** — 28493 unique entities, **musical release (*Q2031291*)** — 3886 unique entities, and **album (*Q482994*)** — 296487 unique entities. The entity counts are current as of April 4, 2024. This would involve processing more than 827 thousand entities in some manner. The second question is: *Once the data is selected, how can we filter the data dump to include only the desired data?* The compressed data dump can be filtered using a combination of **zcat** and **grep**, or simply using **zgrep**. However, constructing a pattern that efficiently and accurately captures the required entities is nearly unfeasible. Alternatively, a custom program, such as a *Python script*, could be utilized to sequentially read and filter the data dump, although this method also suffers from inefficiency.

2. **Accessing data through the SPARQL endpoint** — An alternative method involves using the **Wikidata's SPARQL endpoint** to access the required information. It is possible to formulate a *SPARQL CONSTRUCT query* to gather specific triples concerning the targeted entities. However, the reliability of the *Wikidata's SPARQL endpoint* fluctuates significantly: at times, data are retrieved almost instantly, while at other times, the process may fail due to timeouts. The task of fetching millions, or even tens or hundreds of millions, of *music-related triples* can require a considerable and unpredictable amount of time.

3. **Direct Access to Entity Triples** — Directly access the triples for any entity in the *Wikidata knowledge base* through the **Special:EntityData endpoint**. Data is available in several formats, including *JSON*, *RD-F/XML*, *N-Triples*, and *Turtle*. The *Special:EntityData endpoint* ensures comprehensive data provision by including all triples and extensive additional information for the specified entity.

To simplify the creation process and focus on data that can be connected via their *Spotify IDs*, it was determined to only include music-related entities featuring any of the *Spotify* properties (**Spotify artist ID (*P1902*)**, **Spotify track ID (*P2207*)**, or **Spotify album ID (*P2205*)**). Ultimately, a hybrid of the *second* and *third* approaches was adopted. Initially, the relevant entity IDs were retrieved through the *Wikidata's SPARQL endpoint*.

```
SELECT DISTINCT ?item WHERE {
  {
    ?item p:P1902/ps:P1902 _:anyValueP1902.
  }
  UNION
  {
    ?item p:P2205/ps:P2205 _:anyValueP2205.
  }
  UNION
  {
    ?item p:P2207/ps:P2207 _:anyValueP2207.
  }
}
```

Using the *SELECT query* shown above, 96926 unique IDs were collected. It should be noted that this represents just a fraction of the music-related data available in the *Wikidata's knowledge base*. Subsequently, the *Special:EntityData endpoint* was used to get all triples with respect to the chosen entities. Then these triples were merged into a single file, creating a custom data dump. Due to numerous duplicates resulting from this merging process, a simple bash script was utilized to eliminate these duplicates. Ultimately, the process yielded more than **35 million** unique triples.

4.7 User Credentials Database

The *MongoDB* database, though the smallest, is an essential component of the system. This database is used for **holding user credentials**. Currently, the system lacks designated *private* and *public* sections (where users can choose which segments of their knowledge graph are accessible to all), but such features may be added later. As a result, a user authentication system is already in place, ensuring availability should the feature be introduced later. The database operates on a free shared cluster within the **MongoDB Atlas cloud** environment. To connect to the database, the system uses a native *Node.js* driver along with the connection string.

Chapter 5
Validation and Evaluation

5.1 Software Evaluation

After developing the **front-end client**, a short user evaluation was performed. The main focus of the evaluation was the fulfillment of the functional and non-functional requirements set in Section **3.1.3** and Section **3.1.4**. The test was performed by a few voluntary users, who wanted to participate in the test. They were asked to perform tasks reflecting use case scenarios presented in Section **3.1.2**. Each test session took between 15 and 30 minutes. Two testing environments were tried during the tests: testing on the device, where the system was mostly developed on; and setting up the environment on the user's device. Both environments were prepared for the user before the test.

Initially, users were required to sign up for a new account and log into the system. Subsequently, they needed to set up their profiles. Then they were instructed to upload a file from a given selection. The next step involved editing the parsed metadata and attempting to link *Spotify IDs*. The users were then directed to incorporate this parsed data into their knowledge graphs. Following this, they were to authenticate a provided *Spotify account*. Users were also asked to select a recently played track and incorporate it into the knowledge graph, followed by selecting a playlist to add. Towards the end, they were to navigate the knowledge graph via the *GUI*, reviewing the data they had previously entered. Finally, users were encouraged to explore the application freely to understand its capabilities. Usability feedback was collected throughout these activities.

Table **5.1** shows the failed requirements and only partially successful requirements and the steps taken to address them. The requirements not shown in this table were fulfilled successfully.

ID	Status	Notes	Resolution
FR-3	**fail**	the success message is missing	**fixed**
FR-5	**fail**	the success message is missing	**will not fix, user is redirected after a successful log-in**
FR-12	**success**	the application handles the exception when the user uploads an unsupported file (such as an image), but does not show the user any message	**fixed**
FR-15	**success**	the form does not reset completely when choosing a different track	**fixed**
FR-16	**success**	if there is only one search criterion option available, it should be selected by default	**will not be implemented**
FR-27	**success**	the Music KG API does not handle Spotify authorization rejection correctly	**fixed**
NFR-1	**success**	loaders should be added to notify the user that there is something happening in the background	**implemented**
NFR-3	**success**	error/success messages should be displayed longer	**fixed**

Table 5.1 User evaluation results

5.2 SHACL Evaluation

After user evaluation, the added data was validated using the *SHACL shapes* presented in Section **4.5.1**. The evaluation is done using the following *HTTP POST request*, which was first presented in the same section.

```
curl -XPOST --data-binary shacl.ttl  \
 --header 'Content-type: text/turtle' \
 'http://localhost:3030/music-kg/shacl?graph=union'
```

Calling a *SHACL validation* endpoint returns a *SHACL validation report*. A **SHACL validation report** represents the outcome of the validation procedure, detailing both the conformance and all associated validation results. The evaluation revealed that datatype mismatches exist in the *schema:datePublished* property for certain entities. Further investigations revealed that the *Spotify API* provides only the year of publication as the release date for certain entities. This leads to a *data type mismatch* when these dates are entered into the *RDF store*. As users cannot modify the data retrieved from the *Spotify API* prior to adding them to the *RDF store*, they lack the capability to avoid mismatch of data types. A potential solution could be to allow users to *change the data before storing them in the RDF store*, or to have the **Music KG API** address the mismatches in data types by retrieving the complete release date from an alternative service or adding an arbitrary day and month to meet the *xsd:date* requirements.

Short excerpt from the *SHACL validation report*:

```
PREFIX mkgs:   <http://localhost:3030/music-kg/spotify/>
PREFIX mkgsh:  <http://localhost:3030/music-kg/shacl/>
PREFIX rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-
   ns#>
PREFIX schema: <https://schema.org/>
PREFIX sh:     <http://www.w3.org/ns/shacl#>
PREFIX xsd:    <http://www.w3.org/2001/XMLSchema#>

[ rdf:type     sh:ValidationReport;
  sh:conforms  false;
  sh:result [ rdf:type sh:ValidationResult;
              sh:focusNode mkgs:724783ff-0bb6-4b53-a221-
                 a1ffe42027f3;
              sh:resultMessage "datePublished should use
                 datatype xsd:date."@en;
              sh:resultPath schema:datePublished;
              sh:resultSeverity sh:Warning;
              sh:sourceConstraintComponent sh:
                 DatatypeConstraintComponent;
              sh:sourceShape mkgsh:
                 DatePublishedDatatypePropertyShape;
              sh:value "2008"^^xsd:date
            ];
  sh:result [ rdf:type sh:ValidationResult;
              sh:focusNode mkgs:40b863bb-a5aa-4c17-9d5f-
                 b6a176d861c3;
              sh:resultMessage "datePublished should use
                 datatype xsd:date."@en;
              sh:resultPath schema:datePublished;
              sh:resultSeverity sh:Warning;
              sh:sourceConstraintComponent sh:
                 DatatypeConstraintComponent;
              sh:sourceShape mkgsh:
                 DatePublishedDatatypePropertyShape;
              sh:value "1984"^^xsd:date
            ]
] .
```

All other entities were valid, and no other shapes were violated, ensuring that the added data are mostly consistent.

5.3 Discussion and Future Work

The **Music KG system** offers a platform for the creation, modification, removal, and utilization of *RDF data* through a *RESTful API*. This allows even applications unfamiliar with *RDF* to engage with the system. As the data are already in *RDF format*, they are ready for connection to the *Linked Open Data Cloud*, thus enriching the current knowledge base. Using a service like *Wikidata*, which provides connections to various music-related services, users can link their data to multiple other platforms, helping to further integrate existing *RDF data*. Furthermore, the **Music KG's RDF store** includes a *SPARQL endpoint* for querying and accessing the stored data. The system is also equipped with a *simple GUI*, which provides the viewing of the data in an interpreted way. This *GUI* can also interact with the *Spotify API* via the **Music KG API** to fetch user-specific data, such as playlists or recently played songs. These data can be easily incorporated into the user's music knowledge graph with just a few clicks, simplifying the creation of new music-related *RDF data*.

The designed *API* facilitates *easy addition of various music-related services*, enhancing the diversity of sources for user data integration. A key enhancement would be the capability for users to *modify RDF data directly within the GUI*, making data management more straightforward. Currently, the system is unable to *distinguish between private and public segments of the user's music libraries*; however, this feature might be implemented in the future to allow users more privacy control. Furthermore, users would have the *ability to move their playlists between different services through music library synchronization*, giving them complete autonomy over their data irrespective of the service provider. These are just a few of the potential future enhancements and ideas.

Chapter 6

Conclusion

With the increasing popularity of multimedia streaming services, users may face challenges with data control, backup, and integration across platforms. This includes concerns about losing their music library and playlists if a service is shut down, as well as difficulties managing duplicate or disorganized content when using multiple platforms. Furthermore, it could be beneficial to integrate music libraries with knowledge bases such as *Wikidata* for richer music information. The *objective* of this thesis was to *design and implement* a system that allows users to *organize their music collections*, *add data from multiple sources*, and *link them to existing knowledge bases.*

To achieve this goal, the **Music KG system** was developed. This system comprises four key components: the **back-end server**, which facilitates integration among the *Spotify API*, *users' music libraries on their devices*, and *Wikidata data dump*, all feeding into the dedicated music knowledge graph; the **front-end client**, which allows users to view or add data to their music knowledge graph; the **RDF store**, which maintains the music knowledge graphs; and the **database** that stores users' credentials.

Initially, an analysis of the background on knowledge graphs, music services, community initiatives, and their applications in the field was conducted. Following this, key technologies were explored and described, such as *Semantic Web*, *RDF*, *SPARQL*, *Music Ontologies*, *Linked Data*, and *RDF validation.* Subsequently, an analysis of the requirements was performed, and the system was developed to fulfill these specifications. The details of the system's implementation were then elaborated, showcasing examples of the integrated data. Additionally, a *Wikidata dump* was generated, encompassing more than 90 thousand unique music-related entities.

Through the **Music KG system** the users can *authorize* their *Spotify accounts* to gain access to their *playlists* and *recently played tracks* and add music-related data to their knowledge graph. The system also enables users to *upload media files* to obtain metadata of tracks from their offline music libraries. Furthermore, the system is able to *link the added data* to the created

Wikidata dump to gain access to one of the most extensive knowledge bases available.

Lastly, a *software validation* was performed and evaluated against the set requirements, proving that the developed system met the expectations and the set goals of this thesis. In addition, several future enhancements and directions were suggested.

Appendix A
Manual

A.1 Running a project

The compressed archive `music-kg-develop.zip` enclosed with the thesis contains the source code repository for the implemented system. The same repository can be cloned from **GitHub**.

It is possible to get the environment files for both applications to run from **Google Drive**. It must be accessed using a Google account from the ČVUT FIT group.

After retrieving the environment files, they have to be placed in the following directories:

```
.env -> /apps/api
.env.local -> /apps/dashboard
```

There are two ways to run applications. Using either `npm` scripts or using `Docker` images.

Using npm scripts — Make sure that *Node.js* is installed. For the development, *Node.js v20.12.2* was used. To set the repository up, run these commands:

```
npm install -g pnpm@8
pnpm install
```

After successful installation, the API and the front-end client can be started. The following command will start the API development server, which will listen on port 8000.

```
npx nx run api:serve
```

To start the React FE development server, run the following command. The GUI will be available at `http://localhost:4200`.

```
npx nx run dashboard:serve --host 0.0.0.0
```

Using Docker images — There are Dockerfiles for both applications to create Docker images. The created images will then be used in a Docker compose script, which effectively runs the applications. To proceed, call these commands from the root directory of the repository:

```
docker build --file=apps/dashboard/Dockerfile  -t music-
    kg-dashboard .
docker build --file=apps/api/Dockerfile  -t music-kg-api
    .
docker compose -f docker-compose.yaml up
```

A.1.1 RDF store

1. Use `./tools/download-fuseki.sh` to download Apache Jena Fuseki
2. Navigate to `tools/fuseki` folder
3. Run `./fuseki-server` to start a Fuseki server
4. Open `http://localhost:3030` to access Fuseki GUI
5. Create a new dataset called `music-kg` and choose *Persistent (TDB2)* as *Dataset type*

A SHACL validation service is not enabled by default. To enable it, open the configuration file `tools/fuseki/run/configuration/music-kg.ttl` and add the following triple to it:

```
:service_tdb_all fuseki:endpoint [ fuseki:name "shacl";
                                   fuseki:operation
                                      fuseki:shacl
                                 ] .
```

For more information on SHACL validation in Fuseki, refer to **Fuseki Documentation**.

SHACL shapes for the project are available in `tools/shacl.ttl`.

A.1.2 Testing

For testing purposes, it is possible to create a new user account within the system or use the existing one. For passwords to the **Music KG system** or Spotify testing account, refer to **Google Drive**. It must be accessed using a Google account from the ČVUT FIT group.

A.2 Wikidata dump

The created Wikidata dump can be obtained from **Google Drive** (around 400MB). It must be accessed using a Google account from the ČVUT FIT group.

To use it with the **Music KG system**, it is possible to insert it into Fuseki RDF store using its **GUI**. In the *Dataset graph name* form field input put:

```
http://localhost:3030/music-kg/wikidata-dump
```

Then, unzip the retrieved data dump (around 5.3GB) and upload it to the RDF store by selecting the uncompressed file and choosing *upload now* in the *actions* column.

Bibliography

1. *The American Heritage Dictionary entry: Music* [https://ahdictionary.com/word/search.html?q=Music]. [N.d.]. [access. 2024-04-14].
2. *music - Wikitionary, the free dictionary* [https://en.wiktionary.org/wiki/music]. [N.d.]. [access. 2024-04-14].
3. *What is a knowledge graph?* [https://www.ibm.com/topics/knowledge-graph]. [N.d.]. [access. 2024-04-15].
4. *What is a Knowledge Graph?* [https://www.ontotext.com/knowledgehub/fundamentals/what-is-a-knowledge-graph]. [N.d.]. [access. 2024-04-15].
5. FENSEL, Dieter; ŞIMŞEK, Umutcan; ANGELE, Kevin; HUAMAN, Elwin; KÄRLE, Elias; PANASIUK, Oleksandra; TOMA, Ioan; UMBRICH, Jürgen; WAHLER, Alexander. Why We Need Knowledge Graphs: Applications. In: *Knowledge Graphs: Methodology, Tools and Selected Use Cases*. Cham: Springer International Publishing, 2020, pp. 95–112. ISBN 978-3-030-37439-6. Available from DOI: 10.1007/978-3-030-37439-6_4.
6. EHRLINGER, Lisa; WÖSS, Wolfram. Towards a Definition of Knowledge Graphs. In: *International Conference on Semantic Systems*. 2016. Available also from: https://ceur-ws.org/Vol-1695/paper4.pdf.
7. *Wikidata:Introduction - Wikidata* [https://www.wikidata.org/wiki/Wikidata:Introduction]. [N.d.]. [access. 2024-04-17].
8. *About DBpedia - DBpedia Association* [https://www.dbpedia.org/about]. [N.d.]. [access. 2024-04-17].
9. *Knowledge Graphs - DBpedia Association* [https://www.dbpedia.org/resources/knowledge-graphs]. [N.d.]. [access. 2024-04-17].
10. *About - MusicBrainz* [https://musicbrainz.org/doc/About]. [N.d.]. [access. 2024-04-17].
11. *About Last.fm - Last.fm* [https://www.last.fm/about]. [N.d.]. [access. 2024-04-17].

12. *API Docs - Last.fm* [`https://www.last.fm/api`]. [N.d.]. [access. 2024-04-17].

13. *About Spotify - Spotify* [`https://newsroom.spotify.com/company-info`]. [N.d.]. [access. 2024-04-17].

14. *Web API - Spotify* [`https://developer.spotify.com/documentation/web-api`]. [N.d.]. [access. 2024-04-17].

15. *About - Bandcamp* [`https://bandcamp.com/about`]. [N.d.]. [access. 2024-04-27].

16. ORAMAS, Sergio; OSTUNI, Vito Claudio; NOIA, Tommaso Di; SERRA, Xavier; SCIASCIO, Eugenio Di. Sound and Music Recommendation with Knowledge Graphs. *ACM Trans. Intell. Syst. Technol.* 2016, vol. 8, no. 2. ISSN 2157-6904. Available from DOI: `10.1145/2926718`.

17. BERTRAM, Niels; DUNKEL, Jürgen; HERMOSO, Ramón. I am all EARS: Using open data and knowledge graph embeddings for music recommendations. *Expert Systems with Applications*. 2023, vol. 229, p. 120347. ISSN 0957-4174. Available from DOI: `https://doi.org/10.1016/j.eswa.2023.120347`.

18. BERNERS-LEE, Tim. *Semantic Web Road map* [`https://www.w3.org/DesignIssues/Semantic.html`]. [N.d.]. [access. 2024-04-18].

19. *RDF 1.1 Primer* [`https://www.w3.org/TR/rdf11-primer`]. [N.d.]. [access. 2024-04-18].

20. *IETF RFC 3987* [`https://www.ietf.org/rfc/rfc3987.txt`]. [N.d.]. [access. 2024-04-18].

21. *SPARQL 1.1 Overview* [`https://www.w3.org/TR/sparql11-overview`]. [N.d.]. [access. 2024-04-18].

22. *SPARQL 1.1 Query Language* [`https://www.w3.org/TR/sparql11-query`]. [N.d.]. [access. 2024-04-18].

23. *SPARQL 1.1 Update* [`https://www.w3.org/TR/2013/REC-sparql11-update-20130321`]. [N.d.]. [access. 2024-04-18].

24. *SPARQL 1.1 Protocol* [`https://www.w3.org/TR/2013/REC-sparql11-protocol-20130321`]. [N.d.]. [access. 2024-04-18].

25. *SPARQL 1.1 Graph Store HTTP Protocol* [`https://www.w3.org/TR/2013/REC-sparql11-http-rdf-update-20130321`]. [N.d.]. [access. 2024-04-18].

26. GRIMM, Stephan; ABECKER, Andreas; VÖLKER, Johanna; STUDER, Rudi. Ontologies and the Semantic Web. In: *Handbook of Semantic Web Technologies*. Ed. by DOMINGUE, John; FENSEL, Dieter; HENDLER, James A. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 507–579. ISBN 978-3-540-92913-0. Available from DOI: `10.1007/978-3-540-92913-0_13`.

27. *Specification - The Music Ontology* [`http://musicontology.com/specification`]. [N.d.]. [access. 2024-04-27].
28. *Terminology - MusicBrainz* [`https://musicbrainz.org/doc/Terminology`]. [N.d.]. [access. 2024-04-28].
29. *LinkedBrainz - MusicBrainz* [`https://musicbrainz.org/doc/LinkedBrainz`]. [N.d.]. [access. 2024-04-28].
30. *Schema.org* [`https://schema.org`]. [N.d.]. [access. 2024-04-27].
31. *WikiProject - Wikipedia* [`https://en.wikipedia.org/wiki/Wikipedia:WikiProject`]. [N.d.]. [access. 2024-04-28].
32. *WikiProject Music - Wikidata* [`https://www.wikidata.org/wiki/Wikidata:WikiProject_Music`]. [N.d.]. [access. 2024-04-28].
33. *Ontology Classes - DBpedia* [`https://dief.tools.dbpedia.org/server/ontology/classes`]. [N.d.]. [access. 2024-04-30].
34. BERNERS-LEE, Tim. *Linked Data* [`https://www.w3.org/DesignIssues/LinkedData.html`]. [N.d.]. [access. 2024-04-21].
35. HOSE, Katja; SCHENKEL, Ralf. RDF Stores. In: *Encyclopedia of Database Systems*. Ed. by LIU, Ling; ÖZSU, M. Tamer. New York, NY: Springer New York, 2018, pp. 3100–3106. ISBN 978-1-4614-8265-9. Available from DOI: `10.1007/978-1-4614-8265-9_80676`.
36. TOMASZUK, Dominik. RDF Validation: A Brief Survey. In: KOZIELSKI, Stanisław; MROZEK, Dariusz; KASPROWSKI, Paweł; MAŁYSIAK-MROZEK, Bożena; KOSTRZEWA, Daniel (eds.). *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation.* Cham: Springer International Publishing, 2017, pp. 344–355. ISBN 978-3-319-58274-0.
37. LABRA GAYO, Jose Emilio; PRUD'HOMMEAUX, Eric; BONEVA, Iovka; KONTOKOSTAS, Dimitris. *Validating RDF Data.* Vol. 7. Morgan & Claypool Publishers LLC, 2017. Synthesis Lectures on the Semantic Web: Theory and Technology, no. 1. Available from DOI: `10.2200/s00786ed1v01y201707wbe016`.
38. *Shapes Constraint Language (SHACL)* [`https://www.w3.org/TR/shacl/`]. [N.d.]. [access. 2024-05-03].
39. *History ID3.org - Web Archive* [`https://web.archive.org/web/20101224080318/http://id3.org/History`]. [N.d.]. [access. 2024-04-18].
40. *APEv2 Specification - Hydrogenaudio Knowledgebase* [`https://wiki.hydrogenaud.io/index.php?title=APEv2_specification`]. [N.d.]. [access. 2024-04-18].
41. *Introduction to Node.js* [`https://nodejs.org/en/learn/getting-started/introduction-to-nodejs`]. [N.d.]. [access. 2024-04-28].

42. *Express - Node.js web application framework* [https://expressjs.com]. [N.d.]. [access. 2024-04-28].
43. *TypeScript* [https://www.typescriptlang.org]. [N.d.]. [access. 2024-04-28].
44. *Getting Started - Axios* [https://axios-http.com/docs/intro]. [N.d.]. [access. 2024-04-28].
45. *MongoDB* [https://www.mongodb.com]. [N.d.]. [access. 2024-04-29].
46. *React* [https://react.dev]. [N.d.]. [access. 2024-04-30].
47. *Vite* [https://vitejs.dev]. [N.d.]. [access. 2024-04-30].
48. *Tailwind CSS* [https://tailwindcss.com/docs/installation]. [N.d.]. [access. 2024-05-01].
49. *Music Metadata Browser - GitHub* [https://github.com/Borewit/music-metadata-browser]. [N.d.]. [access. 2024-05-01].
50. *Monorepo Explained* [https://monorepo.tools]. [N.d.]. [access. 2024-05-01].
51. *Intro to Nx* [https://nx.dev/getting-started/intro]. [N.d.]. [access. 2024-05-01].

Contents of the Attachment

`music-kg-develop.zip`	source code of the implementation
`thesis-latex.zip`	source code of the thesis in the LaTeX format
`thesis.pdf`	thesis text in the PDF format