Detailed Conception Document

FFTrack

Team L2B1

Members : Jennifer Zahora, Nicolas Schuldt, Nicolas Fontaine, Ismaël Bouarfa

Supervisors : Hugo Demaret, Ioana Ileana

Université Paris Cité

UFR de Mathématiques et Informatique

– Template par Hugo Demaret 2023-2024

Summary

1 Introduction

This document outlines the architecture, components, and development strategy for our music recognition application, FFTrack. It serves as a comprehensive guide for developers, detailing how each part of the system fits together to achieve accurate and efficient song identification. Our goal is to ensure clarity and facilitate smooth implementation, providing a roadmap from concept to completion.

We will go through the architecture and design methodology of the project, as well as the detailed description of the modules of the application.

2 Design Methodology

The development of our music recognition application is focused on efficiency, accuracy, and scalability. To achieve these objectives, we have adopted a design methodology that combines the object-oriented paradigm with best practices in software engineering and audio processing.

2.1 Object-Oriented Design Paradigm

- **Modularity:** The application is structured into distinct modules, each responsible for a specific aspect of the music recognition process. This modular approach facilitates easier maintenance, testing, and future enhancements.
- **Encapsulation:** By encapsulating data and operations within objects, we ensure a clear separation of concerns and enhance the security and integrity of the application's internal state.
- **Reusability:** Object-oriented design allows us to create reusable components, reducing redundancy and improving efficiency throughout the development process.
- **Extensibility:** The design is intentionally crafted to be extensible, allowing for easy integration of new features or technologies without significant restructuring.

2.2 Music Recognition Algorithm Design

- **Spectrogram Analysis:** The core of our music recognition algorithm is based on spectrogram analysis. By converting audio signals into spectrograms, we can efficiently identify unique features within a song, regardless of recording quality or background noise.
- **Fingerprinting Technique:** We implement an advanced audio fingerprinting technique that captures distinctive patterns in the music. These fingerprints are then used to match query audio against a database of known tracks.
- **Matching Algorithm Optimization:** The matching algorithm is optimized for speed and accuracy, using adapted data structures and search algorithms to quickly find potential matches in the database.
- **Database:** The initial database is based on SQLite, however, following our modularity principle, the database module has been designed to be simply switched for other database choices.
- **Scalability Considerations:** Recognizing the potential for large datasets, the design will include the possibility of being scaled up.

2.3 User Experience Design

- **Simplicity and Accessibility:** The user interface is designed with simplicity and accessibility in mind, ensuring that users of varying technical backgrounds can easily navigate and use the application.

3 System Architecture

The architecture of the FFTrack project is structured around four core components: Audio Processing, Database, Matching, and User Interface (CLI).

- **Audio Processing:** Manages the conversion of audio inputs into a form suitable for analysis.
- **Database:** Stores song metadata, and audio fingerprints.
- **Fingerprint Matching:** Implements algorithms for comparing fingerprints and identifying songs.
- **User Interface (UI):** Provides an interface for user interaction with the system.

Each component is designed to function independently, allowing for easy updates and maintenance. Communication between components is well-defined, ensuring clear data flow and logic.

The following diagram represents the overall view of the system architecture :

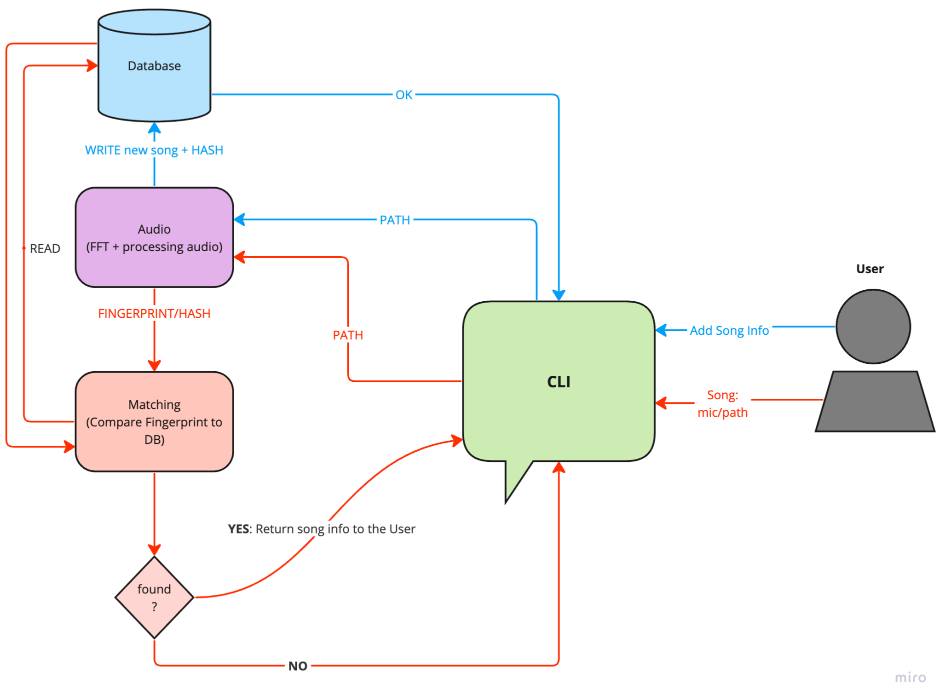


Figure 1: System Architecture

The CLI allows the user to input an audio file or recording. The Audio Processing module takes an audio file as input and generates a unique fingerprint. Thereafter, the Matching module takes the fingerprint as input and uses the Database to search for a song that matches the fingerprint. Then, two cases are possible :

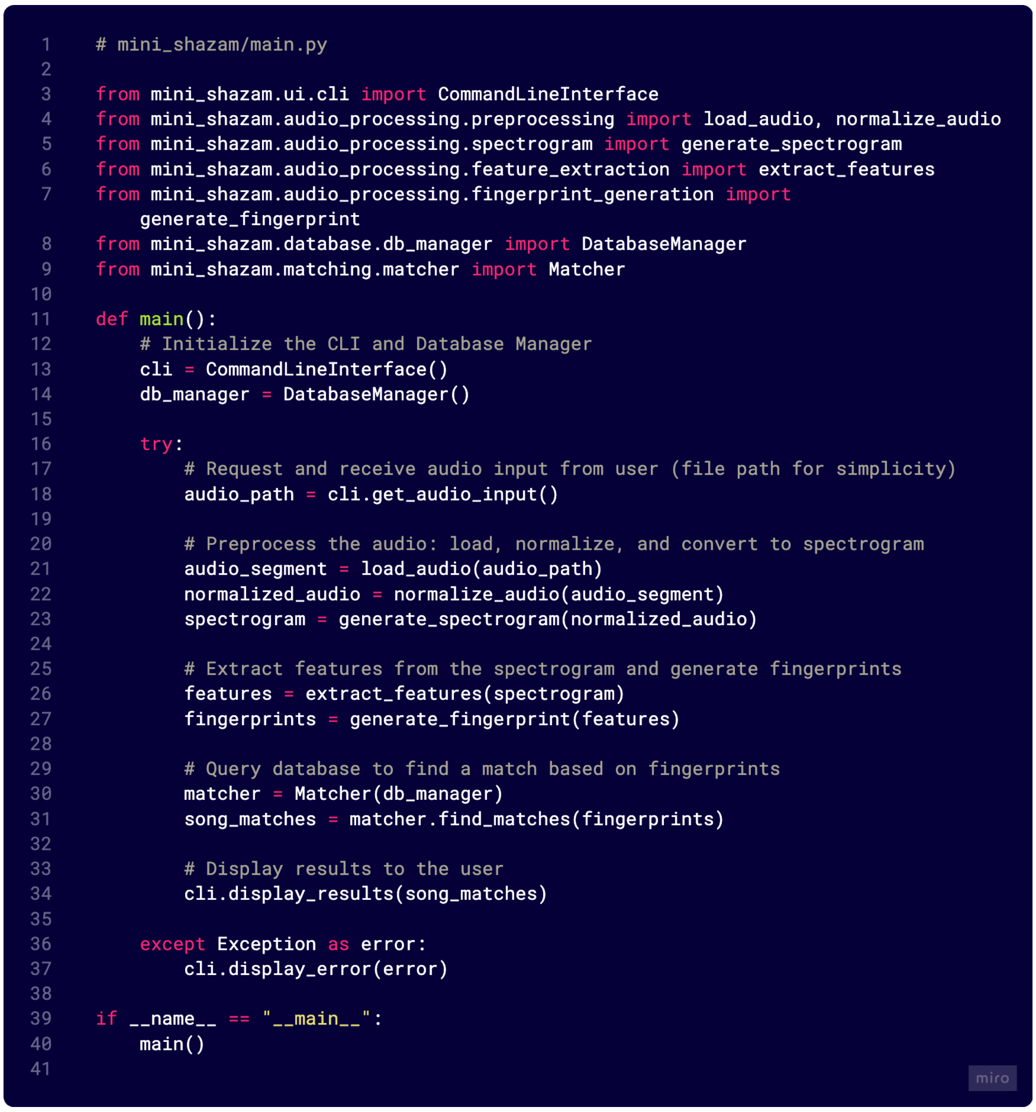
- The song is found in the Database, the output of the Matching module is the song metadata. The CLI retrieves this information and displays it.
- The song is not found in the Database, the Matching module returns nothing. In consequence, the CLI gives the user the choice to add the song information. If so, the new song is added to the Database, together with its fingerprint.

Data Flow: Data moves through the system starting from audio input processing, through fingerprint generation and storage, to song identification. This process involves:

1. Audio input and preprocessing.
2. Feature extraction and fingerprint generation.
3. Fingerprint comparison against the database.
4. Presentation of identification results to the user.

4 Overall System Operation

The `main.py` script within the FFTrack project serves as the entry point for executing the music recognition process. This section provides an overview of how the system operates from receiving user input to displaying the song identification results.



```python
# mini_shazam/main.py

from mini_shazam.ui.cli import CommandLineInterface
from mini_shazam.audio_processing.preprocessing import load_audio, normalize_audio
from mini_shazam.audio_processing.spectrogram import generate_spectrogram
from mini_shazam.audio_processing.feature_extraction import extract_features
from mini_shazam.audio_processing.fingerprint_generation import
    generate_fingerprint
from mini_shazam.database.db_manager import DatabaseManager
from mini_shazam.matching.matcher import Matcher

def main():
    # Initialize the CLI and Database Manager
    cli = CommandLineInterface()
    db_manager = DatabaseManager()

    try:
        # Request and receive audio input from user (file path for simplicity)
        audio_path = cli.get_audio_input()

        # Preprocess the audio: load, normalize, and convert to spectrogram
        audio_segment = load_audio(audio_path)
        normalized_audio = normalize_audio(audio_segment)
        spectrogram = generate_spectrogram(normalized_audio)

        # Extract features from the spectrogram and generate fingerprints
        features = extract_features(spectrogram)
        fingerprints = generate_fingerprint(features)

        # Query database to find a match based on fingerprints
        matcher = Matcher(db_manager)
        song_matches = matcher.find_matches(fingerprints)

        # Display results to the user
        cli.display_results(song_matches)

    except Exception as error:
        cli.display_error(error)

if __name__ == "__main__":
    main()
```

Figure 2: Main Function

1. **User Interface Interaction:** Initialization of the Command Line Interface (CLI) allows for the collection of user input, specifically the path to an audio file.
2. **Audio Preprocessing:** The audio file goes through several preprocessing steps. These include loading the file, normalizing its volume to a standard level, and converting it into a spectrogram-compatible format (wav).
3. **Spectrogram Generation:** The preprocessed audio is transformed into a spectrogram, which visually represents the audio's frequency spectrum over time.
4. **Feature Extraction and Fingerprint Generation:** Key features are extracted from the spectrogram, like peaks, based on which unique fingerprints of the audio are generated. These fingerprints are crucial for the identification process.
5. **Database Querying for Song Match:** The fingerprints are used to query the system's database, through the Database Manager, to find matches with stored songs.
6. **Result Display:** The system, via the CLI, presents the song matches (or identification results) to the user.
7. **Error Handling:** Errors are captured and the CLI informs the user of any issues encountered during execution, ensuring a seamless experience.

5 Detailed Component Breakdown

5.1 Audio Processing

Component Overview

- **Name and type:** AudioProcessing (Class)
- **Purpose:** Processes audio files and then transform them into fingerprints.

Dependencies

- **Uses:** An audio file or a recording.
- **Used by:** The matching module to compare the fingerprints with the ones stored in the database and called by the User interface .
- **Libraries Used:** pyaudio, pydub, wave, numpy.

Interfaces

- **Constructor:**

  Parameters: None.

  Description: Instantiate the AudioProcessing class.

- **record_song:**

  Parameters: None.

  Description: Allow the user to record a song using the pyaudio library.

- **audio_to_wav:**

  Parameters: audio_file_path (str).

  Description: Take the file path of the audio file as the argument, reading it with the pydub library and then create a .wav file that contains the signal of the song and return its path as a string using the wave library.

- **generate_fft:**

  Parameters: audio_samples (np.ndarray)

  Description: Take the file path of the audio file as the argument, reading it with the pydub library and then create a .wav file that contains the signal of the song and return its path as a string using the wave library.

- **load_and_normalize_audio:**

  Parameters: file_path (str)

  Description: Loads an audio file, normalizes its volume, and converts it to a numpy array. Also returns the frame rate of the audio.

- **generate_spectrogram:**

  Parameters: audio_samples (np.ndarray), rate (int)

  Description: Generates a spectrogram from the provided audio samples.

- **find_spectral_peaks:**

  Parameters: spectrogram (np.ndarray), height (float), distance (int)

  Description: Identifies peaks in the spectrogram based on energy and distance criteria.

- **generate_fingerprint:**

  Parameters: peaks (list), fan_value (int)

  Description: Generates fingerprints by pairing peaks and hashing their relationships.

Key data models

- **Attributes/variables**

Fingerprint : Unique identifier used to represent a portion of a song.

FFT : Frequencies and magnitude representing an audio signal.

Core logic

- **Algorithms/processes:**
  - **Spectrogram Generation:**

    Utilizes the Fast Fourier Transform (FFT) to convert audio signals into spectrograms for further analysis.
  - **Peak Finding:**

    Employs signal processing techniques to identify significant peaks within the spectrogram, serving as features for fingerprinting.
  - **FFT:**

    An algorithm that computes the Discrete Fourier Transform (an algorithm that converts a signal into a representation of frequencies) of a sequence by factoring the DFT matrix into a product of sparse factors.
    The FFT is much faster than the DFT in terms of operations ($O(n^2) \rightarrow O(n \log n)$) which is the reason we will be using it.
  - **Possible fingerprinting (indexing) techniques:** will be applied to the identified peaks, creating a unique fingerprint for each audio sample.

    inverted indexing: a data structure that takes a datapoint as an index and directs toward its location. In our case, it is most likely that frequencies are going to be indexed, pointing towards the song in which they appear.

    hashing: a hashing algorithm will be applied to accommodate the utilisation of hash tables or potentially, locality sensitive hashing.

    B-tree: creating indexes using the frequencies, or a combination of frequencies.

Performance and security

- **Security measures:**

  Ensuring that the code can handle exceptions and failure to limit possible bugs.

5.2 Database

Component Overview

- **Name and type:** DatabaseManager (Class)
- **Purpose:** Manages interactions with the database, facilitating operations such as adding and retrieving songs and fingerprints.

Entity-Relation Diagram:

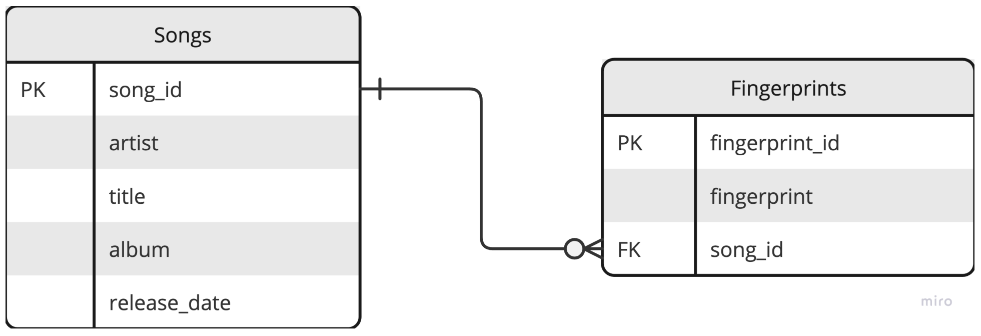


Figure 3: Entity Relation Diagram

Dependencies

- **Uses:** Song, Fingerprint (models), Session (SQLAlchemy session for database interactions)
- **Used by:** Potentially used by audio processing and matching modules for storing and retrieving data.
- **Libraries Used:** The matching module to compare the fingerprints with the ones stored in the database and called by the User interface .

Interfaces

- **Constructor:**

  Parameters: session (optional)

  Description: Initializes DatabaseManager with a session. Creates a new session if none is provided.
- **add_song:**

  Parameters: title (str), artist (str), album (str, optional), release_date (str, optional)

  Description: Adds a new song to the database and returns the song's ID.
- **add_fingerprint:**

  Parameters: song_id (int), fingerprint (str)

  Description: Associates a new fingerprint with a song in the database.
- **get_song_by_id:**

  Parameters: song_id (int)

  Description: Retrieves a song by its ID from the database.

  Returns: Song object.
- **get_all_fingerprints:**

  Parameters: None.

  Description: Retrieves all fingerprints stored in the database.

  Returns: List of all Fingerprint objects in the database.

Key data models

- **Attributes/variables**

  Session : Holds the current database session instance.

Core logic

- **Algorithms/processes:**
  - None.

Performance and security

- **Efficiency Considerations:** Utilises SQLAlchemy's session management and ORM capabilities for efficient database interactions.
- **Security Measures:** Ensures that all data manipulations handle exceptions to prevent partial data corruption.

5.3 Matching

Component Overview

- **Name and type:** Matcher (Class)
- **Purpose:** Identifies potential matches between the fingerprint of the queried audio sample, and existing audio fingerprints within the application' database.

Dependencies

- **Uses:**

  DatabaseManager (Fingerprint model), to retrieve the fingerprints of the songs in the database, and to retrieve the song_id of potential matches

  Query fingerprint, generated by AudioProcessing
- **Used by:** None.

Interfaces

- **Constructor:**

  Parameters: db_manager (class)

  Description: Initialises the matcher with the database manager.
- **compare_fingerprints:**

  Access control: Private

  Parameters: query_fingerprint (fingerprint), database_fingerprint (list of fingerprints)

  Description: Compares the fingerprint of the query with the fingerprints of one song from the database, and assigns a similarity score to it, which represents how accurately the fingerprints resemble each other.

  Returns: The similarity score of the fingerprints.
- **find_matches:**

  Access control: Public

  Parameters: query_fingerprint (fingerprint)

  Attributes: fingerprints (list), similarity_score (int)

  Description: Finds the closest matches of the query fingerprint in the database.

  Returns: A list with the song_id's of the top potential matches

Key data models

- **Attributes/variables**

  db_manager: The database manager object to access the database.

  query_fingerprint: The fingerprint of the queried audio sample.

  fingerprints: List of all the fingerprints of the database.

  database_fingerprint: List of one song's fingerprints.

  similarity_score: A score that determines how similar the query fingerprint is to the fingernt of one song in the database.

  top_matches: List of the top potential matches, has maximum 3/5 songs in it.

Core logic

- **Process of matching:**
  - find all the fingerprints that match the fingerprints of the sample
  - increment the similarity score according to the determined method

  - the song with the highest score is the song we are looking for

- **Possible search algorithms:** will depend on what data structure we decide to use for generating and storing the fingerprints. Each data structure has its own search algorithm.
  - hash tables: hashing the search key (fingerprint datapoint) to determine the bucket, then simply looking up the bucket and returning the data from it. In order to avoid collisions, each bucket will contain either a linked list, or an array of key-value pairs.
  - B-tree: Starting from the root, we traverse down, until we find the node that has the key we are looking for.

Performance and security

- **Security measures:** Ensuring that all data manipulations handle exceptions to avoid bugs.
- **Efficiency considerations:** Implementing efficient data structures and optimisation techniques to handle large-scale fingerprint matching tasks effectively.

5.4 User Interface

Component Overview

- **Name and type:** CommandLineInterface (Class)
- **Purpose:** The CLI is the module with which the user is in direct contact. The class communicates with the Audio Processing to launch the song search, and then displays information on the status of the search and the final result.

Dependencies

- **Uses:**

  Audio Processing, to record a song and get the file path.
- **Used by:** None.

Interfaces

- **Constructor:**

  Parameters: None.

  Description: Instantiate the CommandLineInterface class.
- **get_audio_input:**

  Access control: Public

  Parameters: None.

  Description: Receive the audio input from the user and return the file path.
- **display_results:**

  Access control: Public

  Parameters: song_matches (list)

  Description: Display results to the user.
- **display_error:**

  Access control: Public

  Parameters: error (class)

  Description: Display the error with details if an error has occurred.

Key data models

- **Attributes/variables**

  song_matches: A list of the song ID of the top matches (None if not found).

error: An Exception.

Performance and security

- **Security Measures:**

  Ensuring that all data manipulations handle exceptions to avoid operation failure.

  Explicit and useful error messages in case of operation failure.

  A user manual will be supplied with the application for a better experience.

6 Performance Optimisation

Efficient performance is essential for providing a seamless user experience. To achieve optimal performance, we employ various strategies targeting different aspects of the system, such as minimising latency, reducing computational overhead, and optimising resource utilisation.

6.1 Audio Processing Optimisation

6.1.1 Preprocessing

To enhance the efficiency of feature extraction, we preprocess the query audio file by normalising, turning it into a mono WAV file, and potentially reducing noise and filtering it. These techniques will help extract relevant audio features more accurately.

6.1.2 Real-time processing

We aim to optimise the audio input processing for real-time performance to ensure minimal delay between recording and identification. We aim to implement buffering techniques (e.g. block-based processing) to efficiently handle incoming audio data to minimise latency.

6.2 Matching optimisation

6.2.1 Indexing i.e. fingerprinting

We aim to use an optimal indexing technique to efficiently organise data. We call these indexes fingerprints, and they derive from the FFT of the songs. This allows for faster retrieval and comparison during the identification phase, reducing computational overhead.
Moreover, we aim to utilise memory-efficient data structures to store and manipulate large datasets efficiently, as hashes or trees. These data structures optimise memory usage and access time.

6.2.2 Algorithm optimisation

We will conduct thorough analysis of algorithmic time complexity to identify and optimise performance bottlenecks, i.e. components that might be limiting the capacity of the application, such as inefficient algorithms. By selecting algorithms with lower time complexity, we minimise computational overhead and improve overall responsiveness.

6.2.3 Benchmarking

We will conduct tests to define the performance benchmarks of the application, and will aim to optimise these test by setting stricter and stricter performance targets and benchmarks. These tests will help assess scalability, reliability, latency, and efficiency of the application and its components.

7 Deployment Plan

In this section, we outline the process and strategies for deploying the application, **FFTrack**. This plan includes various stages, from testing and validation to the final delivery of the application.

7.1 Testing and Validation

To ensure that the project meets all requirements, we will conduct extensive testing to optimize the application. As detailed in our User Acceptance Document, a series of unit, integration, functional, and performance tests will be performed to verify and validate the application.

Once the project has passed all necessary tests and meets all validation criteria, a release version of the project will be uploaded to the Subversion server.

7.2 Delivery

The development process and modifications are tracked through Subversion. Furthermore, all documentation related to the project will be uploaded to the Forge's server.

The application and its documentation will then be packaged and uploaded to *PyPi* to facilitate future installations by end-users or developers.

8 Conclusion

In conclusion, this document has provided a comprehensive overview of the design, development, and deployment strategies for the FFTrack application. Through a detailed description of each component, methodologies, and deployment plans, we have laid down a robust foundation for the successful implementation and launch of FFTrack. The adherence to best practices in software development, combined with a flexible and iterative development aproach, ensures that FFTrack is on track with its goals. We are confident that the strategies outlined here will lead to the creation of a reliable, efficient, and user-friendly application.