



**Final Project Report**

**Advanced Computer Programming**

**Song Recommendation Website**

**Group : 5**

**Instructor : DINH-TRUNG VU**

**2025-06**

# Chapter 1

# Introduction

## Group Information

1. **Group Project Repository**: <https://github.com/wonderhorse90/ACP/tree/main/Final>
2. **Group members**:
   1. Muhammad Luthfi Zafir - 113021203
   2. Zefanya Teguh Foyd Silalahi - 113021217
   3. Joanes Don Bosco Bhramantyo - 113021218 (leader)
   4. Giandra Raihan Nayottama - 113021219

## Overview

In this project, we built a music recommendation web application using advanced features of Python, SQL, and web development libraries. The following tools and technologies were used:

* **Flask** for building the web server and managing user sessions.
* **Jinja2 templating** for dynamic content rendering in HTML.
* **Pandas** for reading and preprocessing CSV data in the ETL process.
* **MySQL Connector** for connecting Python to a remote MySQL database.
* **SQL (CTEs, GROUP\_CONCAT, STRING\_AGG, CONCAT\_WS)** for advanced data aggregation, filtering, and recommendation logic.
* **HTML & CSS (Flexbox, Media Queries)** for building a responsive and user-friendly interface.

Our project successfully:

* Extracted and cleaned music track data from a CSV file.
  + [Raw Data](https://drive.google.com/open?id=1XCHG5q78Yg_62BwjQkfK4jpWr0nm0OOi)
  + [Cleaned Data](https://drive.google.com/open?id=1NjenVKTcCBqjU3atBBrk9Z4aPOcVkSjE)
* Stored it in a structured MySQL database.
* Built a recommendation system that filters songs based on energy levels and genre similarity.
* Allowed users to search, select, and get recommendations in an interactive web interface.

This demonstrates the integration of full-stack development and data filtering logic to deliver a personalized music discovery experience.

# Chapter 2

# Implementation

## 2.1 Class 1: app.py

This Python file contains the Flask web application backend logic for the music recommendation system, including search, selection, and personalized recommendation functionality with session support.

### 2.1.1 Fields

* Flask, Session, mysql.connector: Imported libraries.
* DB\_CONFIG: MySQL connection credentials.
* app.secret\_key: Required for session encryption.
* session['selected\_tracks']: Stores user-selected songs.

### 2.1.2 Methods

* / : Renders index.html
* /recommendations: Displays search form, handles song search logic (using SQL LIKE).
* /add\_track\_direct: Adds a selected track to session.
* /remove\_track/<track\_name>: Removes track from session.
* /clear\_tracks: Clears session selections.
* /show\_recommendations: Generates song recommendations based on average energy and genre.

### 2.1.3 Functions

* SQL query returns search results for matching tracks.
* Selected tracks are stored in session (list).
* Recommendations are filtered:
  + By average ± std. deviation of energy.
  + By partial genre matches.
  + By excluding already selected tracks.

## 2.2 Class 2: index.html

This file is the structure of a simple music recommendation landing page. It uses external CSS styling and Google Fonts, and it includes basic content layout and a navigation button.

### 2.2.1 Fields

* <!DOCTYPE html> – Declares HTML5 document type.
* <html lang="en"> – Root element with English language specification.
* <head> – Contains meta information, fonts, title, and CSS link.
* <body> – Contains the page content:
  + .container-1 – Flex container for text and image.
  + .box-1-text – Contains the heading, paragraph, and button.
  + .box-1-image – Image placeholder (styled via CSS).

### 2.2.2 Methods

* **Page Structure Logic**:
  + Uses semantic elements (<div>, <h1>, <p>, <a>, <button>).
  + Responsive layout through viewport meta tag.
  + Dynamically loads the CSS file via Jinja: {{ url\_for('static', filename = 'homepage.css') }}.

### 2.2.3 Functions

* Get Started button function:
  + HTML <a href="/recommendations"> routes to a music recommendation feature.

## 2.3 Class 3: homepage.css

This file provides layout and visual styling to the homepage. It defines a modern, flexbox-based design with custom typography and button interactivity.

### 2.3.1 Fields

* .container-1 – Flex container set to full viewport height.
* .box-1-text – Text section with padding and vertical centering.
* .box-1-image – Background image with cover and centering.
* .btn – Button styles (padding, border, shadow, hover effects).
* Fonts – Uses "Poppins", sans-serif.

### 2.3.2 Methods

* **Flexbox**:
  + .container-1 splits the page into two responsive halves.
  + .box-1-text and .box-1-image take equal width.
* **Typography**:
  + Consistent use of "Poppins" with varying weights.
  + Large heading (70px), smaller subtext (36px).
* **Button Styling**:
  + Initial .btn class defines look.
  + Padding and border-radius.
  + Soft background and shadow.

### 2.3.3 Functions

* + Adds animation-like feedback through transition and transform.
  + Provides a smooth user experience by styling elements clearly.

## 2.4 Class 4: insert\_tracks.py

This Python script reads a cleaned music dataset from a CSV file, then populates and processes it into two MySQL tables: tracks\_raw and track\_clean. It ensures proper schema, inserts the data, performs cleaning, and transforms it for use in the recommendation system.

### 2.4.1 Fields

* username, password, host, dbname: MySQL connection configuration.
* csv\_path: Path to the cleaned CSV dataset (clean2\_data.csv).
* expected\_cols: Required columns for ingestion from the CSV.
* df: DataFrame for processing and holding the tabular data.

### 2.4.2 Methods

* CSV Loading: Loads CSV file using pandas.read\_csv with comma separator.
* Data Cleaning: Ensures all expected columns are present; fills missing columns with None.
* MySQL Connection: Connects to a PythonAnywhere-hosted MySQL database using mysql.connector.
* Schema Execution: Executes SQL schema from track\_schema.sql to create tracks\_raw table.
* Data Insertion: Inserts all records into tracks\_raw using REPLACE INTO to avoid duplicates.
* Genre Cleaning: Changes all "singer-songwriter" genres to "singer".
* Table Creation (track\_clean): Creates a final table with grouped genres and concatenated artist names.

### 2.4.3 Functions

* **Data Preparation:**
  + Ensures track\_id, track\_name, energy, and artist/genre fields are present.
  + Replaces NaN values with None for SQL compatibility.
* **SQL Logic:**
  + REPLACE INTO: Inserts or updates existing rows with the same track\_id.
  + GROUP\_CONCAT: Merges genre values if multiple entries exist for one track.
  + CONCAT\_WS: Combines all artist fields into a single string.
  + GROUP BY: Groups by unique track info for deduplication.
* **Final Output:**
  + Table track\_clean is ready for querying by the recommendation engine (app.py).
  + Script ends with a confirmation print message.

## 2.5 Class 5: second-page.css

This file defines the visual design and responsive layout for the second page of the music web app. It styles all core components such as the search bar, selected songs list, search results, and recommendations.

### 2.5.1 Fields

* .container-1, .box-1-text, .box-1-image, .box-1-image-content
* .search-container, .search-input, .search-button
* .picked-tracks, .selected-songs-box, .remove-btn, .clear-btn
* .found-songs, .recommendations
* @media screen and (max-width: 768px) for responsive adjustments

### 2.5.2 Methods

* Uses flexbox to organize the page layout into two main sections (text and image).
* Centers text vertically and horizontally inside .box-1-text and .box-1-image-content.
* Styles the search bar with padding, rounded corners, hover effect, and color transition.
* Highlights the selected songs box with scrollable content and styled bullet list.
* Visually separates found songs and recommendations using styled list blocks and color codes.
* Defines hover effects for interactive buttons like .search-button, .remove-btn, and .clear-btn.
* Implements media queries to adjust layout for smaller screens by stacking content vertically and adjusting paddings.

### 2.5.3 Functions

* Provides structured, readable, and aesthetic layout for user interactions.
* Makes the second-page interface responsive and mobile-friendly.
* Adds intuitive visual feedback to actions like searching, selecting, or removing tracks.
* Enhances usability by grouping and styling content into clear, navigable sections.

## 2.6 Class 6: second-page.html

This file defines the HTML structure for the music recommendation system’s interactive page. It contains the search input, selected song list, and dynamically rendered sections for search results and recommendations using Flask and Jinja2.

### 2.6.1 Fields

* HTML meta and title info
* Links to second-page.css and Google Fonts
* <div class="container-1"> for page layout
  + .box-1-text: Main heading, search form, and selected tracks list
  + .box-1-image: Displays search results and recommendations

### 2.6.2 Methods

* Displays the main heading and search form with input and button inside .box-1-text.
* Uses Flask’s url\_for() and Jinja2 to:
  + Submit search form via POST to /recommendations.
  + Show selected tracks using {% if selected %} and {% for %}.
  + Enable removing or clearing selected tracks with dynamic links.
  + Display found songs based on a search term if found\_songs is available.
  + Display song recommendations if tracks data is available.
* Groups all dynamic data rendering inside .box-1-image-content for a clear layout.

### 2.6.3 Functions

* Provides an interactive interface for users to:
  + Search tracks
  + View and manage selected songs
  + Trigger personalized music recommendations
* Dynamically renders song results and recommendations using data passed from Flask backend.
* Ensures a seamless user experience without reloading or redirecting away from the page.

## 

## 2.7 Class 7: track\_recommendation.sql

This SQL script handles the complete pipeline for music track cleaning and recommendation. It cleans the raw dataset, constructs the main track table (track\_clean), prepares temporary tables for selected songs, calculates average energy and standard deviation, and finally recommends similar songs based on energy and genre similarity.

### 2.7.1 Fields

* tracks\_raw: Source table containing original track data.
* track\_clean: Cleaned and normalized table used for recommendations.
* picked\_tracks: Temporary table for holding user-selected songs.
* calc\_track: Temporary table used for computation logic.
* CTEs (energy\_cal, picked\_genre): Common Table Expressions used for calculation and filtering.

### 2.7.2 Methods

* Cleans the track\_genre by converting 'singer-songwriter' to 'singer'.
* Renames original tracks table to tracks\_raw (if not already named).
* Constructs track\_clean using:
  + STRING\_AGG to combine genres into one string.
  + CONCAT\_WS to merge multiple artist columns.
  + GROUP BY on all track identifier fields.
* Renames the artist column from concat\_ws to artists.
* Adds a PRIMARY KEY on track\_id to ensure uniqueness.
* Creates a temporary table picked\_tracks to hold tracks selected by the user.
* Creates another temp table calc\_track by joining selected tracks with track\_clean.
* Uses CTEs:
  + energy\_cal computes average and standard deviation of energy values.
  + picked\_genre extracts unique genres by splitting the combined genre string.
* Performs the final recommendation query:
  + Filters tracks by energy range (within ± std dev).
  + Ensures genre overlap using ILIKE and EXISTS.
  + Excludes already selected tracks.

### 2.7.3 Functions

* Normalizes raw track data into a format optimized for querying and recommendation.
* Calculates statistical boundaries to find songs with similar energy.
* Filters results using genre similarity to refine personalization.
* Uses SQL constructs like temporary tables, WITH clauses, aggregation, and string operations for logic abstraction and performance.

## 2.8 Class 8: track\_schema.sql

This SQL script defines the structure for the tracks\_raw table, which holds the original music track data before cleaning or transformation. It ensures all essential song information is properly typed and stored in a relational database schema.

### 2.7.1 Fields

* track\_id – Unique identifier for each track (primary key, VARCHAR(255)).
* track\_name – Title of the track.
* energy – A floating-point value representing the energy/intensity of the track.
* track\_genre – The genre or category of the track.
* artist1 to artist4 – Names of up to four contributing artists.

### 2.7.2 Methods

* Uses CREATE TABLE IF NOT EXISTS to avoid duplication if the table already exists.
* Defines all columns with sufficient string lengths (VARCHAR(255)).
* Applies FLOAT for the energy attribute to allow decimal values.
* Declares track\_id as the primary key to ensure track uniqueness.

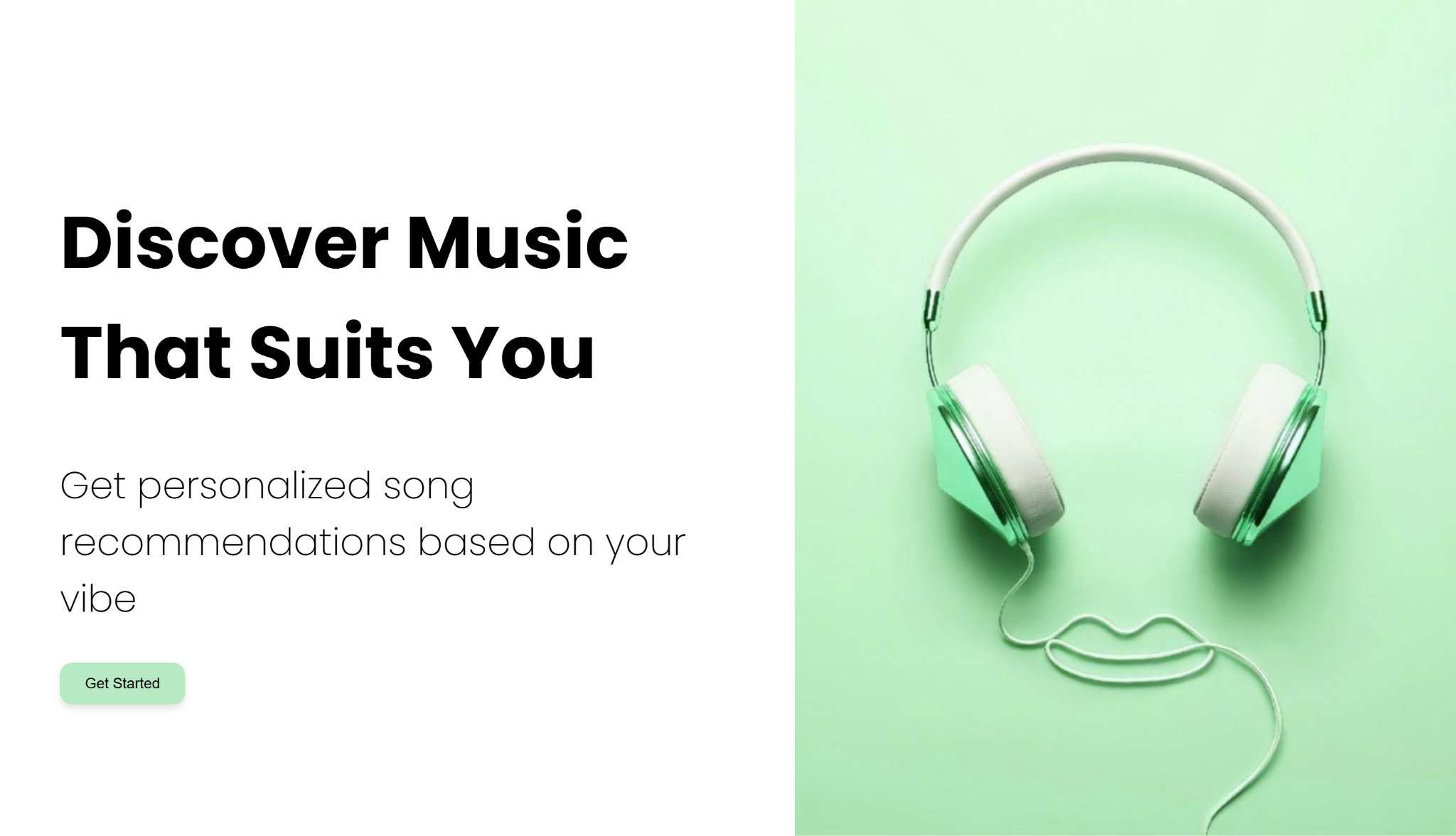
### 2.7.3 Functions

* Establishes a base schema for inserting raw data via ETL scripts (like insert\_tracks.py).
* Enables structured storage of track metadata required for downstream cleaning and recommendation processing.
* Acts as the foundational table used by track\_clean in later transformations.

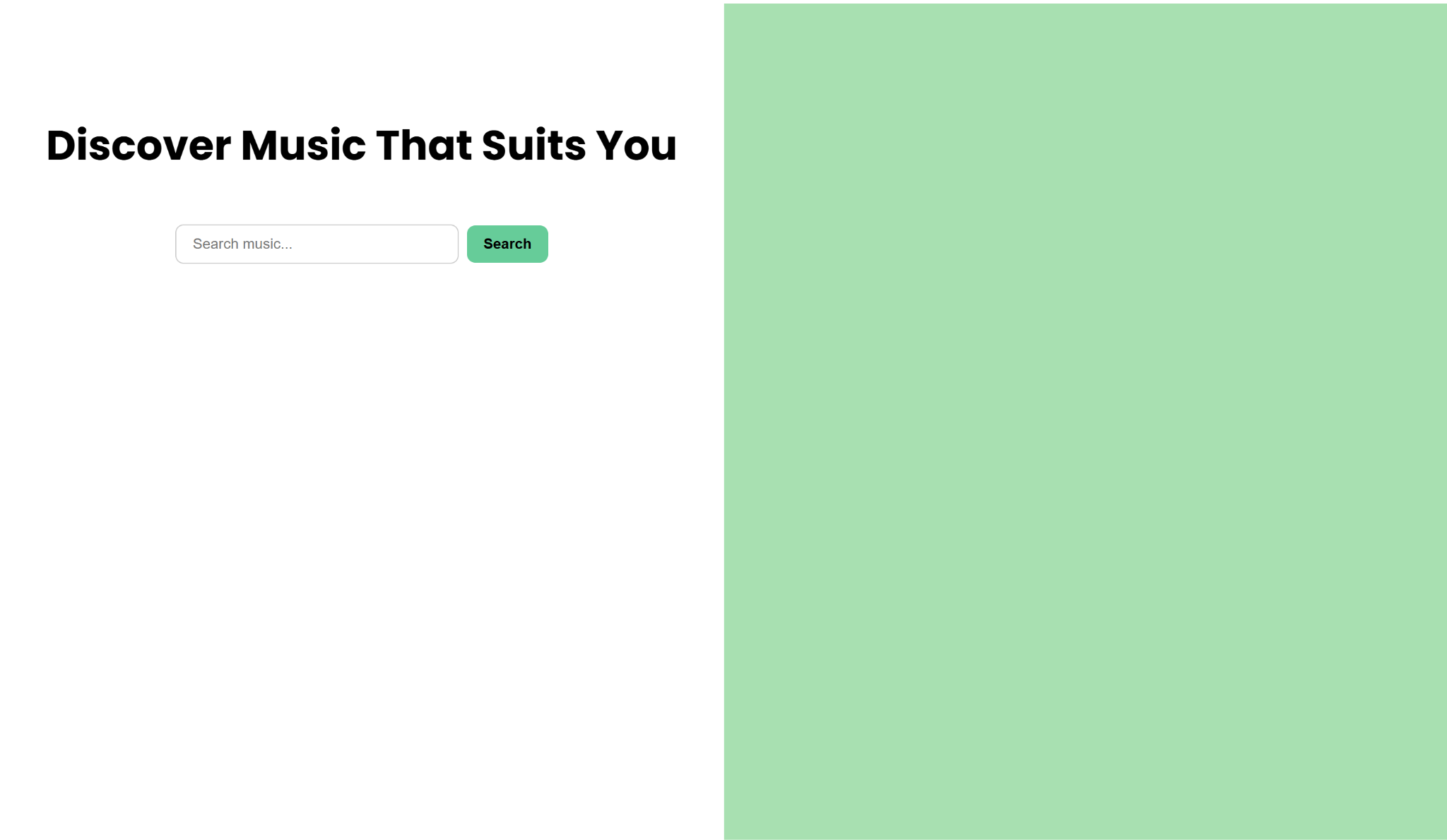
# Chapter 3

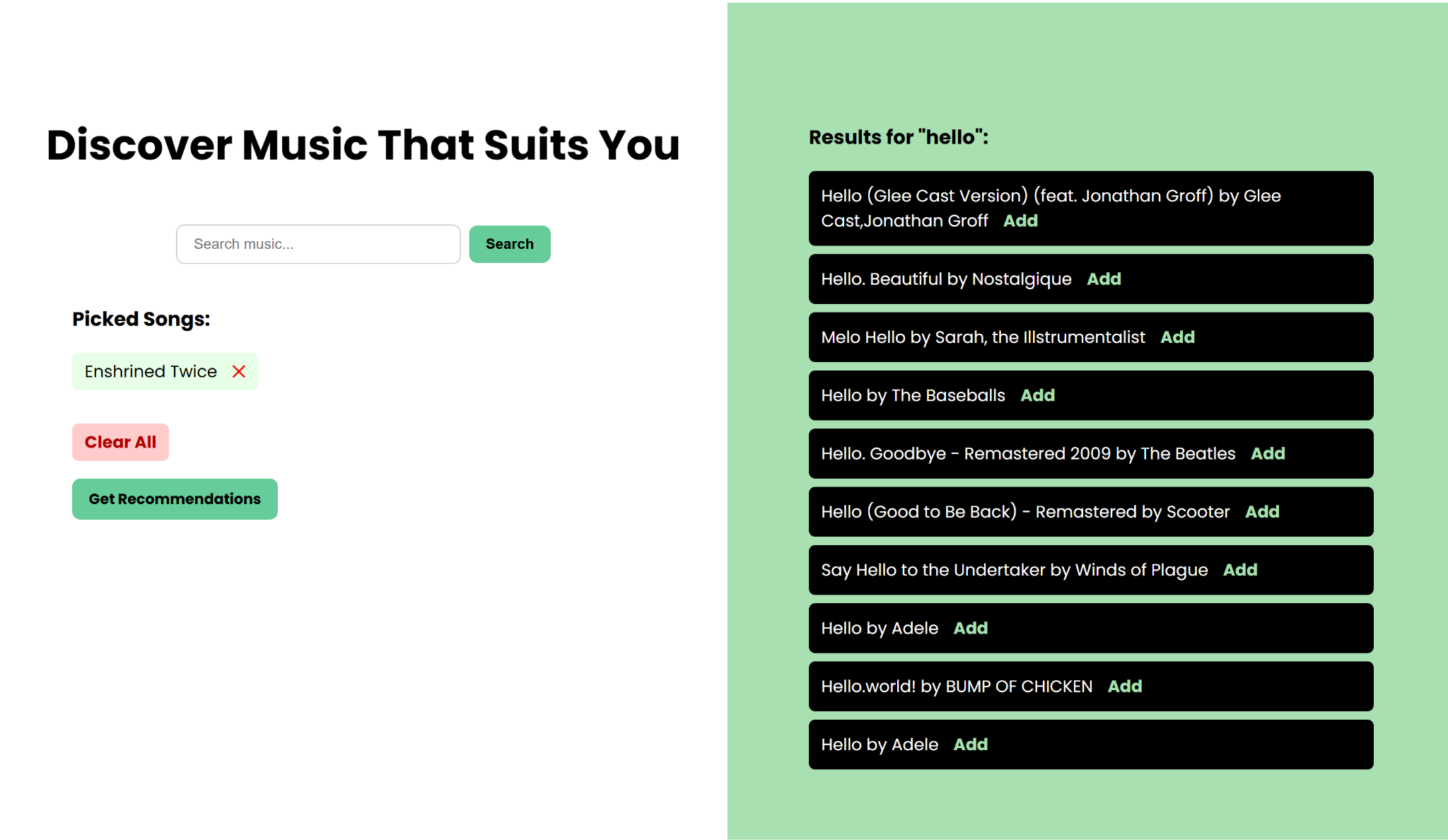
# Results

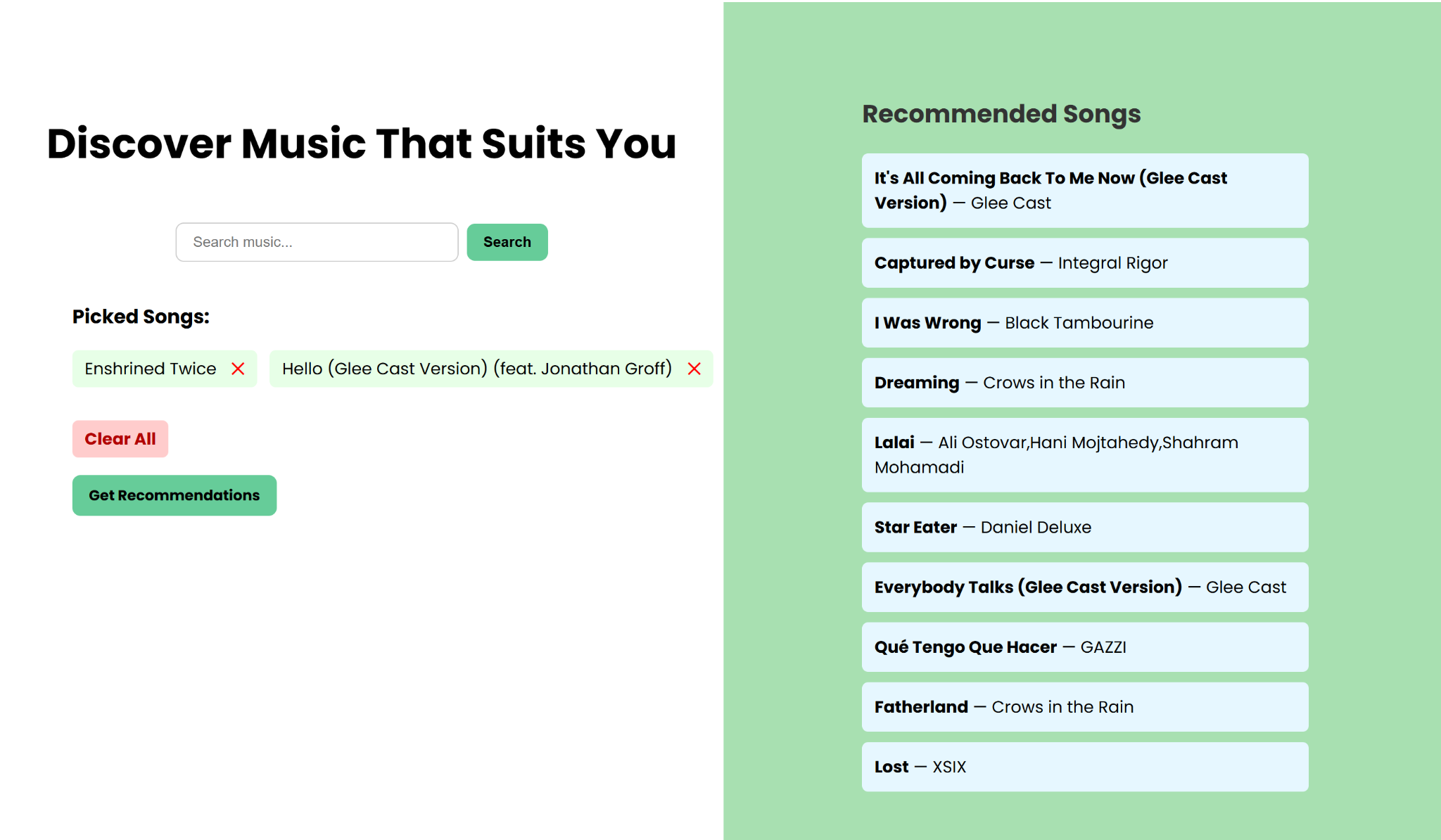
## Home page



## Main page







# Chapter 4

# Conclusions

This project is a full-featured music recommendation web application that integrates frontend, backend, and database components to create a personalized music discovery experience.

The frontend, built using HTML and CSS (index.html, second-page.html, homepage.css, second-page.css), provides users with an intuitive and responsive interface. Users can easily search for songs, select tracks they like, and view tailored recommendations. The layout is clean, mobile-friendly, and styled using Google Fonts and modern CSS techniques like Flexbox and transitions.

The backend, developed using Python and Flask (app.py), manages user interactions and sessions. It processes search requests, stores selected tracks, and generates music recommendations based on energy levels and genre similarity. Each route serves a specific function, such as adding or removing tracks, clearing selections, or displaying recommendations.

The database layer is structured using SQL scripts (track\_schema.sql, track\_recommendation.sql) and powered by MySQL. The schema organizes raw song data, and SQL logic helps clean and transform this data into a usable format (track\_clean). Recommendations are generated by analyzing the average energy of selected songs and matching them with others that have similar energy and genre tags.

To load and prepare the data, the insert\_tracks.py script reads from a CSV file and populates the database. It ensures proper formatting, data cleaning, and table creation, forming the foundation for the recommendation engine.

* All parts of the system are well-integrated:
* The frontend handles user interaction.
* The backend processes data and logic.
* The database stores and filters tracks.
* The ETL script prepares the data pipeline.

Together, they form a smooth and effective recommendation system that helps users discover songs that suit their preferences based on both musical characteristics and mood.