**Code structure of our music recommender system:**

**1)Requirements.txt file:**

It is used to specify the project's dependencies. It lists all the Python packages and their respective versions that are required for the project to run correctly. This file is commonly used in Python projects to manage and install dependencies using package managers like pip.

**2)CSV handler:**

The CSV handler is responsible for dealing with data stored in CSV (Comma-Separated Values) files, and we’ll use it because our source code is in a CSV format.

So the CSV handler is used for reading music-related data from our CSV files,including information about songs, artists, genres, and other relevant details and turning this data into a pandas dataframe.

DataFrames provide a structured and tabular representation of the data, making it easy to organize, explore, and manipulate the data effectively.

import pandas as pd

def read\_csv\_to\_dataframe(csv\_file\_path):

    try:

        df = pd.read\_csv(csv\_file\_path)

        return df

    except FileNotFoundError:

        print(f"File not found at path: {csv\_file\_path}")

        return None

    except Exception as e:

        print(f"An error occurred: {str(e)}")

        return None

csv\_file\_path = 'track.csv'

df = read\_csv\_to\_dataframe(csv\_file\_path)

if df is not None:

    print(df.head())

**3)data handler:**

As we proceed, we’re going to have to be working with databases(specially PostgeSQL) and other different sources, we need a module to handle data from different sources based on the specific data type.

We’ll be using the Pandas library for handling CSV and Excel data. We’ll also use **psycopg2** for PostgreSQL database handling and other modules related to error handling and data logging.

This is an idea of how our code is going to look like:

import pandas as pd

import psycopg2

from lookups import FileType, ErrorHandling

from data\_logging import return\_error\_log

def extract\_data(data\_type, data\_path):

    dataframe = None

    try:

        if data\_type == FileType.CSV:

            dataframe = pd.read\_csv(data\_path)

        elif data\_type == FileType.EXCEL:

            dataframe = pd.read\_excel(data\_path)

        elif data\_type == FileType.PostgreSQL:

            pass

        elif data\_type == FileType.mongoDB:

            pass

    except Exception as e:

        suffix = ErrorHandling.extract\_data\_info\_df.value

        prefix = str(e)

        return\_error\_log(suffix, prefix)

    finally:

        return dataframe

def list\_all\_files(files\_directory):

    pass

**4)data Logging Module:**

In the context of a music recommender system, logging can be crucial for several reasons:

* **Debugging and Troubleshooting:**
  + Logging debug messages helps developers identify issues, trace program flow, and debug the system during development.
* **Error Handling and Monitoring:**
  + Logging errors and related information allows the system to record and report errors, enabling proactive monitoring and troubleshooting.
* **User and System Activity Tracking:**
  + Logging can be used to track user interactions, system events, and user preferences, aiding in improving recommendations and user experience.
* import logging
* # Assuming you have appropriate loggers set up
* info\_logger = logging.getLogger('info\_logger')
* debug\_logger = logging.getLogger('debug\_logger')
* error\_logger = logging.getLogger('error\_logger')
* def return\_info\_log(suffix, prefix):
* # Log information messages
* info\_message = f"{prefix} - {suffix}"
* info\_logger.info(info\_message)
* def return\_debug\_log(log\_message):
* # Log debug messages
* debug\_logger.debug(log\_message)
* def return\_error\_log(suffix, prefix):
* # Log error messages
* error\_message = f"{prefix} - {suffix}"
* error\_logger.error(error\_message)

1. The return\_info\_log function logs information messages using the info\_logger.
2. The return\_debug\_log function logs debug messages using the debug\_logger.
3. The return\_error\_log function logs error messages using the error\_logger. We construct an error message by combining the prefix and suffix.

**5)Lookups file**

In a music recommender system, a lookup file, or a lookup table, is a data structure used to store predefined values or mappings that are used within the system. It consists of a set of key-value pairs or codes that represent certain attributes, categories, or classifications related to music or the functioning of our recommender system. Reasons we need it:

1. **Handling Error Codes or Statuses:**
   * Error codes or status indicators might be used for error handling or tracking the state of certain processes. A lookup file can map these codes to meaningful messages, aiding in debugging and troubleshooting.
2. **Localization and Multilingual Support:**
   * If the recommender system is used in multiple regions or languages, a lookup file can store translations or localized versions of certain labels, enhancing user experience.
3. **Configuration and Parameters:**
   * Lookup files can also store configuration parameters, thresholds, or other settings that can be easily accessed and modified without changing the actual codebase.
4. **Data Integrity and Consistency:**
   * By centralizing common values in a lookup file, we ensure consistency and avoid discrepancies in data representation across different parts of the system.

We’ll use it mostly to store the filetypes we’ll be dealing with and to display error messages relevant to our system's data extraction or other processes.

from enum import Enum

class FileType(Enum):

    CSV = ".csv"

    EXCEL = ".xls"

    TEXT = ".txt"

    PostgreSQL = "postgreSQL"

    mongoDB = "mongo"

    XML = ".xml"  # XML file type

    JSON = ".json"  # JSON file type

class ErrorHandling(Enum):

    EXTRACT\_DATA\_INFO\_DF = "extract\_data\_info\_df"

    INVALID\_FILE\_FORMAT = "invalid\_file\_format"

    DATABASE\_CONNECTION\_ERROR = "database\_connection\_error"

Here extract\_data\_info\_df: Represents an error or issue related to extracting data

**6)Prehook:**

It’s a pre-processing hook script that is executed before certain operations or processes within the system. It's a common practice to use pre-hooks to perform necessary setup, validations, or transformations before executing the main functionality.

Here's a general outline of what our prehook.py will contain:

1. Setup and Configuration:
   * Initialize configurations, environment variables, or any necessary resources needed for the system to function properly.
2. Input Validation:
   * Validate input parameters, data formats, or any incoming data to ensure they meet the expected criteria before proceeding.
3. Data Preprocessing:
   * Preprocess data if required, such as data cleaning, normalization, or transformation, to prepare it for further processing.
4. Dependency Checks:
   * Verify and ensure that all required dependencies, libraries, or services are available and properly configured.
5. **Connection Initialization:**
   * **Initialize connections to databases, APIs, or any external services that the system interacts with.**

Here’s a code overview:

import logging

# Initialize logging

logging.basicConfig(level=logging.INFO)

logger = logging.getLogger(\_\_name\_\_)

def initialize\_system():

    # Setup and configuration

    initialize\_configurations()

    initialize\_dependencies()

    initialize\_database\_connection()

    # Add more setup tasks as needed

def initialize\_configurations():

    # Initialize configurations based on environment variables or config files

    # Example: Load configurations from a file

    # config.load\_from\_file('config.json')

    pass

def initialize\_dependencies():

    # Verify and initialize dependencies

    # Example: Check if required libraries are installed

    # assert check\_dependency('pandas'), "Dependency 'pandas' not found"

    pass

def initialize\_database\_connection():

    # Initialize database connection

    # Example: Connect to the database

    # db.connect('database\_url', 'username', 'password')

    pass

if \_\_name\_\_ == "\_\_main\_\_":

    initialize\_system()

**7)hook.py file**

In a data engineering project, a **hook.py** file is likely used to define hooks or functions that provide specific functionalities related to data processing, data transformations, and potentially connecting to data sources or targets. These hooks can be used to modularize and organize data engineering tasks, making the codebase more maintainable and easier to extend.

Here's what **hook.py** might be used for in a data engineering-focused music recommender system project, and what it could contain:

**Usage of hook.py:**

1. **Data Processing Functions:**
   * Define functions to process data in various ways, such as cleaning, aggregating, or transforming music-related data.
2. **Data Ingestion and Extraction:**
   * Define functions to extract data from different sources (e.g., databases, files) and ingest it into a format suitable for the recommender system.
3. **Data Loading:**
   * Define functions to load processed data into the desired storage or database.
4. **Data Transformation:**
   * Define functions for transforming raw data into features that can be used for modeling in the recommender system.
5. **Data Quality Checks:**
   * Implement functions to perform data quality checks and ensure the integrity and consistency of the data being used.
6. **Integration with ETL Pipelines:**
   * Define functions to integrate with Extract, Transform, Load (ETL) pipelines, allowing smooth data movement and processing.

Here’s what it might look like:

import pandas as pd

def process\_music\_data(data):

    # Perform data processing tasks specific to music data

    # Example: Cleaning, feature extraction, etc.

    processed\_data = data.dropna()  # Simple example: Drop rows with missing values

    return processed\_data

def extract\_data\_from\_database(database\_url, query):

    # Extract music-related data from a database

    # Example: Use pandas to read data from a SQL database

    connection = pd.read\_sql\_query(query, database\_url)

    return connection

def load\_data\_to\_storage(data, storage\_path):

    # Load processed data into storage

    # Example: Save the data as a CSV file

    data.to\_csv(storage\_path, index=False)

def transform\_music\_data(raw\_data):

    # Transform raw data into features suitable for modeling

    # Example: Extract relevant features from raw data

    transformed\_data = raw\_data[['feature1', 'feature2', 'feature3']]

    return transformed\_data

**8)misc handler**

In a music recommender, these functions might be used to work with enums that represent sources or types of data. For instance, the enum could represent various types of music sources (e.g., Spotify, YouTube, Local files) or data formats (e.g., CSV, JSON, XML) relevant to the data engineering tasks. These functions could be used to extract and print information about these sources or data formats for further processing or handling in the data engineering pipeline.

**9)posthook**  
we use it for

1. **Result Logging and Analysis:**
   * After generating recommendations or processing data, the posthook logs the results, analyze the recommendations, and store insights for future improvements.
2. **Updating User History:**
   * After generating recommendations, the posthook updates the user's listening history or preferences based on the recommendations provided.
3. **Data Quality Checks:**
   * Conduct post-processing data quality checks to ensure that the generated recommendations meet certain quality standards or criteria.
4. **Caching or Storage:**
   * Cache or store the generated recommendations in a persistent storage for faster access in subsequent requests.
5. **Logging Metrics:**
   * Log various metrics related to the recommendation process, such as processing time, memory usage, or other performance metrics.