МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»
Кафедра інформаційних систем та мереж

ЗВІТ

про виконання лабораторної роботи № 7
«Робота з API та веб-сервісами»
з дисципліни "Спеціалізовані мови програмування”

Виконала:
ст. гр. ІТ-32,
Троцько О. М.
Прийняв:
Щербак С. С.

ЛЬВІВ – 2023

Мета: створення консольного об’єктно - орієнтованого додатка з використанням API.

План роботи

Завдання 1: Вибір провайдера API

Виберіть надійний API, який надає через HTTP необхідні дані для віддаленого зберігання, вивантаження або реалізуйте свій. Для прикладу це може бути jsonplaceholder.org

Завдання 2: Інтеграція API

Виберіть бібліотеку для роботи з API та обробки HTTP запитів (для прикладу це може бути бібліотека Requests). Інтегруйте обраний API в ваш консольний додаток на Python. Ознайомтеся з документацією API та налаштуйте необхідний API-ключ чи облікові дані.

Завдання 3: Введення користувача

Розробіть користувальницький інтерфейс, який дозволяє користувачам візуалізувати всі доступні дані в табличному вигляді та у вигляді списку. Реалізуйте механізм для збору та перевірки введеного даних користувачем.

Завдання 4: Розбір введення користувача

Створіть розбірник для видобування та інтерпретації виразів користувача на основі регулярних виразів, наприклад, для візуалізації дат, телефонів, тощо. Переконайтеся, що розбірник обробляє різні формати введення та надає зворотний зв'язок про помилки.

Завдання 5: Відображення результатів

Реалізуйте логіку для візуалізації даних через API в консолі. Обробляйте відповіді API для отримання даних у вигляді таблиць, списків. Заголовки таблиць, списків мають виділяться кольором та шрифтом, які задається користувачем

Завдання 6: Збереження даних

Реалізуйте можливості збереження даних у чіткому та читабельному форматі JSON, CSV та TXT

Завдання 7: Обробка помилок

Розробіть надійний механізм обробки помилок для керування помилками API, некоректним введенням користувача та іншими можливими проблемами. Надавайте інформативні повідомлення про помилки.

Завдання 8: Ведення історії обчислень

Включіть функцію, яка реєструє запити користувача, включаючи введені запити та відповідні результати. Дозвольте користувачам переглядати та рецензувати історію своїх запитів.

Завдання 9: Юніт-тести

Напишіть юніт-тести для перевірки функціональності вашого додатку. Тестуйте різні операції, граничні випадки та сценарії помилок.

Код програми:

```
# auth.py
"""
Authentication Module

This module manages the authentication process to obtain access tokens from the authentication server.
It provides functions to retrieve tokens and generate authorization headers required for API requests.
"""
import os
import base64
import json

from dotenv import load_dotenv
from requests import post
from shared.settings import get_lab_settings

settings = get_lab_settings("lab7")
URLS = settings["urls"]
TOKEN_URL = URLS["token_url"]
```

```
load_dotenv()

CLIENT_ID = os.getenv("CLIENT_ID")
CLIENT_SECRET = os.getenv("CLIENT_SECRET")

def get_token():
    """
    Retrieves an access token from the authentication server.

    Returns:
        str: The access token.
    """
    auth_string = CLIENT_ID + ":" + CLIENT_SECRET
    auth_bytes = auth_string.encode("utf-8")
    auth_base64 = str(base64.b64encode(auth_bytes), "utf-8")

    headers = {
        "Authorization": "Basic " + auth_base64,
        "Content-Type": "application/x-www-form-urlencoded"
    }

    data = {"grant_type":"client_credentials"}

    result = post(TOKEN_URL, headers=headers, data=data)
    json_result = json.loads(result.content)
    token = json_result["access_token"]

    return token


def get_auth_header(token):
    """
    Generates the authorization header using the provided access token.

    Args:
        token (str): The access token.

    Returns:
        dict: The authorization header.
    """
```

```
    return { "Authorization": "Bearer " + token }
```

 # album.py

```
"""
Spotify Album Module

This module provides a class for representing albums and methods to interact with the Spotify API
to retrieve album information. It includes functionality to initialize an Album object, search for
an album by name, and obtain details such as the album's release date, artist information, and
Spotify link.
"""

from shared.settings import get_lab_settings
from classes.lab7.api_classes.api_error_handling.api_error_handling import APIError, APIRequest
from classes.lab7.auth.auth import get_auth_header, get_token

settings = get_lab_settings("lab7")
BASE_URL = settings["urls"]["base_url"]

class Album:
    """
    Represents an album.

    Attributes:
        id (str): The ID of the album.
        album_name (str): The name of the album.
           artist (dict): The artist of the album, containing the ID, name, and Spotify link.
        release_date (str): The release date of the album.
        spotify_link (str): The Spotify link of the album.
    """

    # def __init__(self, name):
    #     """
    #     Initializes an Album object.
```

```
#     Args:
#         name (str): The name of the album.
#     """
#     self.id = None
#     self.album_name = None
#     self.artist = None
#     self.release_date = None
#     self.spotify_link = None
#     self.init_album(name)

def __init__(self):
    """
    Initializes an empty Album object.
    """
    self.id = None
    self.album_name = None
    self.artist = None
    self.release_date = None
    self.spotify_link = None

def __str__(self):
    """
    Returns a formatted JSON representation of the album.

    Returns:
        str: The formatted JSON representation of the album.
    """
    return str(self.get_album_formatted_json())

# def get_album_json_from_api(self, album_name):
#     """
#     Retrieves the album JSON data from the API.

#     Args:
#         album_name (str): The name of the album.

#     Returns:
#         dict: The album JSON data.
#     """
#     try:
```

```
    #         token = get_token()
    #         headers = get_auth_header(token)
    #         url = BASE_URL + "search"
    #         query = f"?q={album_name}&type=album&limit=1"

    #         query_url = url + query
    #         result = get(query_url, headers=headers)
    #         json_result = json.loads(result.content)["albums"]["items"]
    #         if not json_result:
    #             print("No album with this name exists...")
    #             return None
    #         return json_result[0]

    #     except exceptions.RequestException as exception:
    #         print(f"Error making API request: {exception}")
    #         return None

    #     except json.JSONDecodeError as exception:
    #         print(f"Error decoding JSON response: {exception}")
    #         return None

    #     except KeyError as exception:
    #         print(f"Unexpected response format: {exception}")
    #         return None

    #     except Exception as exception:
    #         print(f"An unexpected error occurred: {exception}")
    #         return None

    def get_album_json_from_api(self, album_name):
        """
        Retrieves the JSON data of an album from the API based on the album name.

        Args:
            album_name (str): The name of the album to search for.

        Returns:
             dict: The JSON data of the album, or None if no album with the given
name exists.
```

```
        Raises:
            APIError: If there is an error while making the API request.
        """

        api_request = APIRequest(BASE_URL)
        try:
            token = get_token()
            headers = get_auth_header(token)
                     album_data  =  api_request.make_request("search",  params={"q":
album_name, "type": "album", "limit": 1}, headers=headers)
            if not album_data:
                print("No album with this name exists...")
                return None
            return album_data["albums"]["items"][0]
        except APIError as api_error:
            print(f"API Error: {api_error.message}")

    def init_album(self, name):
        """
        Initializes the album object with the given name.

        Args:
            name (str): The name of the album.
        """
        album_json = self.get_album_json_from_api(name)

        if album_json is None:
            print("The object was not created")
            return
        self.set_values(album_json)

    def set_values(self, album_json):
        """
        Sets the values of the album object based on the album JSON data.

        Args:
            album_json (dict): The album JSON data.
        """
        self.id = album_json["id"]
        self.album_name = album_json["name"]
```

```
        self.release_date = album_json["release_date"]
        artist_id = album_json["artists"][0]["id"]
        artist_name = album_json["artists"][0]["name"]
        artist_link = album_json["artists"][0]["external_urls"]["spotify"]
            self.artist = {"id": artist_id, "name": artist_name, "spotify_link":
artist_link}
        self.spotify_link = album_json["external_urls"]["spotify"]

    def get_album_formatted_json(self):
        """
        Returns a formatted dictionary representation of the album.

        Returns:
            dict: The formatted dictionary representation of the album.
        """
        return {
            'id': self.id,
            'album_name': self.album_name,
            'album_artist': self.artist,
            'release_date': self.release_date,
            'spotify_link': self.spotify_link
        }
```

 # artist.py

```
"""
Artist Module

This module defines the Artist class for retrieving artist information from an API.
"""
from shared.settings import get_lab_settings
from     classes.lab7.api_classes.api_error_handling.api_error_handling     import
APIError, APIRequest
from classes.lab7.auth.auth import get_auth_header, get_token

settings = get_lab_settings("lab7")
BASE_URL = settings["urls"]["base_url"]

class Artist:
    """
```

```
     Represents an artist and provides methods for retrieving artist information
from an API.
    Attributes:
        id (str): The ID of the artist.
        artist_name (str): The name of the artist.
        spotify_link (str): The Spotify link of the artist.
    """

    def __init__(self):
        """
        Initializes an instance of the Artist class.
        """
        self.id = None
        self.artist_name = None
        self.spotify_link = None

    def __str__(self):
        """
        Returns a string representation of the Artist object.
        """
        return str(self.get_artist_formatted_json())

    # def get_artist_json_from_api(self, artist_name):
    #     """
    #     Retrieves the JSON data for an artist from the API.

    #     Parameters:
    #     - artist_name (str): The name of the artist.

    #     Returns:
     #       - dict: The JSON data for the artist, or None if the artist does not
exist.
    #     """
    #     try:
    #         token = get_token()
    #         headers = get_auth_header(token)
    #         url = BASE_URL + "search"
    #         query = f"?q={artist_name}&type=artist&limit=1"

    #         query_url = url + query
```

```
    #         result = get(query_url, headers=headers)
    #         json_result = json.loads(result.content)["artists"]["items"]
    #         if not json_result:
    #             print("No artist with this name exists...")
    #             return None
    #         return json_result[0]

    #     except exceptions.RequestException as exeption:
    #         print(f"Error making API request: {exeption}")
    #         return None

    #     except json.JSONDecodeError as exeption:
    #         print(f"Error decoding JSON response: {exeption}")
    #         return None

    #     except KeyError as exeption:
    #         print(f"Unexpected response format: {exeption}")
    #         return None

    #     except Exception as exeption:
    #         print(f"An unexpected error occurred: {exeption}")
    #         return None

    def get_artist_json_from_api(self, artist_name):
        """
        Retrieves the JSON data for an artist from the API.

        Parameters:
        - artist_name (str): The name of the artist.

        Returns:
        - dict: The JSON data for the artist, or None if the artist does not exist.
        """
        api_request = APIRequest(BASE_URL)
        try:
            token = get_token()
            headers = get_auth_header(token)
                    artist_data  =  api_request.make_request("search",  params={"q":
artist_name, "type": "artist", "limit": 1}, headers=headers)
            return artist_data["artists"]["items"][0]
```

```
    except APIError as api_error:
        print(f"API Error: {api_error.message}")
        return None

def init_artist(self, name):
    """
    Initializes the Artist object with the data for the specified artist.

    Parameters:
    - name (str): The name of the artist.
    """
    artist_json = self.get_artist_json_from_api(name)

    if artist_json is None:
        print("The object was not created")
        return

    self.set_values(artist_json)


def set_values(self, artist_json):
    """
    Sets the values of the Artist object using the provided artist JSON data.

    Parameters:
    - artist_json (dict): The JSON data for the artist.
    """
    self.id = artist_json["id"]
    self.artist_name = artist_json["name"]
    self.spotify_link = artist_json["external_urls"]["spotify"]

def get_artist_formatted_json(self):
    """
    Returns a formatted dictionary representation of the Artist object.

    Returns:
    - dict: The formatted dictionary representation of the Artist object.
    """
    return {
        'id': self.id,
```

```
            'artist_name': self.artist_name,
            'spotify_link': self.spotify_link
        }
```

 # track.py

```
"""
Track Module

This module defines the Track class, which represents a track object and provides
methods for retrieving
track information from the Spotify API.
"""

from shared.settings import get_lab_settings
from     classes.lab7.api_classes.api_error_handling.api_error_handling     import
APIError, APIRequest
from classes.lab7.auth.auth import get_auth_header, get_token

settings = get_lab_settings("lab7")
BASE_URL = settings["urls"]["base_url"]

class Track:
    """
    Represents a track object.

    Attributes:
        id (str): The ID of the track.
        track_name (str): The name of the track.
        artist (dict): The information about the artist of the track.
        album (dict): The information about the album of the track.
        spotify_link (str): The Spotify link of the track.
    """

    # def __init__(self, name):
    #     """
    #     Initializes a Track object with the given name.

    #     Args:
    #         name (str): The name of the track.
    #     """
```

```
#     self.id = None
#     self.track_name = None
#     self.artist = None
#     self.album = None
#     self.spotify_link = None
#     self.init_track(name)

def __init__(self):
    """
    Initializes an empty Track object.
    """
    self.id = None
    self.track_name = None
    self.artist = None
    self.album = None
    self.spotify_link = None

def __str__(self):
    """
    Returns a formatted JSON string representation of the Track object.

    Returns:
        str: The formatted JSON string representation of the Track object.
    """
    return str(self.get_track_formatted_json())

# def get_track_json_from_api(self, token, track_name):
#     """
#     Retrieves the track JSON data from the API.

#     Args:
#         token (str): The access token for the API.
#         track_name (str): The name of the track.

#     Returns:
#         dict: The track JSON data.
#     """
#     try:
#         token = get_token()
#         headers = get_auth_header(token)
```

```
#         url = BASE_URL + "search"
#         query = f"?q={track_name}&type=track&limit=1"

#         query_url = url + query
#         result = get(query_url, headers=headers)
#         json_result = json.loads(result.content)["tracks"]["items"]
#         if not json_result:
#             print("No track with this name exists...")
#             return None
#         return json_result[0]
#     except exceptions.RequestException as exception:
#         print(f"Error making API request: {exception}")
#         return None

#     except json.JSONDecodeError as exception:
#         print(f"Error decoding JSON response: {exception}")
#         return None

#     except KeyError as exception:
#         print(f"Unexpected response format: {exception}")
#         return None

#     except Exception as exception:
#         print(f"An unexpected error occurred: {exception}")
#         return None


def get_track_json_from_api(self, track_name):
    """
    Retrieves the JSON data for a track from the API.

    Args:
        track_name (str): The name of the track.

    Returns:
        dict: The JSON data for the track, or None if the track does not exist.

    Raises:
        APIError: If there is an error with the API request.
    """
```

```
        api_request = APIRequest(BASE_URL)
        try:
            token = get_token()
            headers = get_auth_header(token)
                     track_data  =  api_request.make_request("search",  params={"q":
track_name, "type": "track", "limit": 1}, headers=headers)
            if not track_data:
                print("No track with this name exists...")
                return None
            return track_data["tracks"]["items"][0]
        except APIError as api_error:
            print(f"API Error: {api_error.message}")


    def init_track(self, name):
        """
        Initializes the Track object with the given name.

        Args:
            name (str): The name of the track.
        """
        track_json = self.get_track_json_from_api(name)

        if track_json is None:
            print("The object was not created")
            return

        self.set_values(track_json)

    def set_values(self, track_json):
        """
        Sets the values of the Track object based on the track JSON data.

        Args:
            track_json (dict): The track JSON data.
        """
        self.id = track_json["id"]
        self.track_name = track_json["name"]
        artist_id = track_json["artists"][0]["id"]
        artist_name = track_json["artists"][0]["name"]
```

```
        artist_link = track_json["artists"][0]["external_urls"]["spotify"]
            self.artist = {"id": artist_id, "name": artist_name, "spotify_link":
artist_link}
        album_id = track_json["album"]["id"]
        album_name = track_json["album"]["name"]
        album_link = track_json["album"]["external_urls"]["spotify"]
             self.album =   {"id": album_id, "name": album_name, "spotify_link":
album_link}
        self.spotify_link = track_json["external_urls"]["spotify"]


    def get_track_formatted_json(self):
        """
        Returns the formatted JSON representation of the Track object.

        Returns:
            dict: The formatted JSON representation of the Track object.
        """
        return {
            'id': self.id,
            'track_name': self.track_name,
            'artist': self.artist,
            'album': self.album,
            'spotify_link': self.spotify_link
        }
```

 # data_by_artist.py

```
"""
Spotify Data by Artist Module

This module provides a class for retrieving data related to a specific artist from
the Spotify API.
It includes functionality to initialize an instance of the DataByArtist class,
search for an artist,
retrieve albums and top tracks by the artist, and format the obtained data.
"""
import json
from requests import get, exceptions

from classes.lab7.auth.auth import get_auth_header, get_token
```

```
from classes.lab7.api_classes.artist import Artist
from classes.lab7.api_classes.album import Album
from classes.lab7.api_classes.track import Track
from shared.settings import get_lab_settings

settings = get_lab_settings("lab7")
BASE_URL = settings["urls"]["base_url"]

class DataByArtist(Artist):
    """
    A class that represents data retrieval for a specific artist.

    Attributes:
        data (list): A list to store the retrieved data.
    """

    def __init__(self):
        """
        Initializes an instance of the DataByArtist class.
        """
        self.data = []
        super().__init__()

    def init_artist(self, name):
        """
        Initializes the artist by name.

        Args:
            name (str): The name of the artist.

        Returns:
            bool: True if the artist is successfully initialized, False otherwise.
        """
        return super().init_artist(name)

    def get_albums_by_artist_json_from_api(self):
        """
        Retrieves the albums by the artist from the API.

        Returns:
```

```
            list: A list of albums in JSON format.
        """
        try:
            token = get_token()
            headers = get_auth_header(token)
            url = BASE_URL + f"artists/{self.id}/albums"
            result = get(url, headers=headers)
            json_result = json.loads(result.content)["items"]
            if not json_result:
                print("No top tracks found for this artist.")
                return None
            return json_result

        except exceptions.RequestException as exeption:
            print(f"Error making API request: {exeption}")
            return None

        except json.JSONDecodeError as exeption:
            print(f"Error decoding JSON response: {exeption}")
            return None

        except KeyError as exeption:
            print(f"Unexpected response format: {exeption}")
            return None

        except Exception as exeption:
            print(f"An unexpected error occurred: {exeption}")
            return None

    def get_albums_formatted_json(self):
        """
        Retrieves the albums by the artist in a formatted JSON format.

        Returns:
            list: A list of albums in a formatted JSON format.
        """
        json_albums = self.get_albums_by_artist_json_from_api()
        data = []
        for json_album in json_albums:
            album = Album()
```

```
            album.set_values(json_album)
            data.append(album.get_album_formatted_json())
        return data

    def get_top_tracks_by_artist_json_from_api(self):
        """
        Retrieves the top tracks by the artist from the API.

        Returns:
            list: A list of top tracks in JSON format.
        """
        try:
            token = get_token()
            headers = get_auth_header(token)
            url = BASE_URL + f"artists/{self.id}/top-tracks?country=UA"
            result = get(url, headers=headers)
            json_result = json.loads(result.content)["tracks"]

            if not json_result:
                print("No top tracks found for this artist.")
                return None
            return json_result

        except exceptions.RequestException as exeption:
            print(f"Error making API request: {exeption}")
            return None

        except json.JSONDecodeError as exeption:
            print(f"Error decoding JSON response: {exeption}")
            return None

        except KeyError as exeption:
            print(f"Unexpected response format: {exeption}")
            return None

        except Exception as exeption:
            print(f"An unexpected error occurred: {exeption}")
            return None

    def get_tracks_formatted_json(self):
```

```
        """
        Retrieves the top tracks by the artist in a formatted JSON format.

        Returns:
            list: A list of top tracks in a formatted JSON format.
        """
        json_tracks = self.get_top_tracks_by_artist_json_from_api()
        data = []

        for json_track in json_tracks:
            track = Track()
            track.set_values(json_track)
            data.append(track.get_track_formatted_json())
        return data
```

 # recommendation.py

```
"""
Spotify Recommendation Module

This module provides a class for retrieving track recommendations from the Spotify API.
It includes functionality to initialize a Recommendation object, retrieve and format track recommendations,
and formulate API requests based on seed artists, genres, and tracks.
"""
import json
from requests import get, exceptions

from classes.lab7.auth.auth import get_auth_header, get_token
from classes.lab7.api_classes.artist import Artist
from classes.lab7.api_classes.track import Track

class Recommendation():
    """
     Represents a recommendation object that retrieves track recommendations from the Spotify API.

    Attributes:
        limit (int): The maximum number of track recommendations to retrieve.
        seed_artists (list): A list of seed artist names.
```

```
        seed_genres (list): A list of seed genre names.
        seed_tracks (list): A list of seed track names.
    """

           def   __init__(self,   limit=5,   seed_artists=None,   seed_genres=None,
seed_tracks=None):
        """
        Initialize a Recommendation object.

        Args:
             limit (int): The maximum number of track recommendations to retrieve.
Default is 5.
            seed_artists (list): A list of seed artist names. Default is None.
            seed_genres (list): A list of seed genre names. Default is None.
            seed_tracks (list): A list of seed track names. Default is None.
        """
        self.limit = limit
        self.seed_artists = seed_artists
        self.seed_genres = seed_genres
        self.seed_tracks = seed_tracks

    def get_track_recommendation_json_from_api(self):
        """
        Retrieve track recommendation JSON from the Spotify API.

        Returns:
            list: A list of track recommendation JSON objects.
        """
        try:
            url = self.form_url()
            token = get_token()
            headers = get_auth_header(token)
            result = get(url, headers=headers)
            json_result = json.loads(result.content)["tracks"]
            if not json_result:
                print("No track recommendation...")
                return None
            return json_result
        except exceptions.RequestException as exception:
            print(f"Error making API request: {exception}")
```

```
            return None

        except json.JSONDecodeError as exception:
            print(f"Error decoding JSON response: {exception}")
            return None

        except KeyError as exception:
            print(f"Unexpected response format: {exception}")
            return None

        except Exception as exception:
            print(f"An unexpected error occurred: {exception}")
            return None


    def get_track_recommendation_formatted_json(self):
        """
        Retrieve formatted track recommendation JSON from the Spotify API.

        Returns:
            list: A list of formatted track recommendation JSON objects.
        """
        json_track_recommendation = self.get_track_recommendation_json_from_api()
        data = []

        for json_track in json_track_recommendation:
            track = Track()
            track.set_values(json_track)
            data.append(track.get_track_formatted_json())

        return data


    def get_seed_artists_id(self):
        """
        Get the IDs of the seed artists.

        Returns:
            list: A list of seed artist IDs.
        """
```

```
    artists_ids = []
    for artist in self.seed_artists:
        obj = Artist()
        obj.init_artist(artist)
        if obj.id is not None:
            artists_ids.append(obj.id)

    return artists_ids


def get_seed_tracks_id(self):
    """
    Get the IDs of the seed tracks.

    Returns:
        list: A list of seed track IDs.
    """
    tracks_ids = []
    for track in self.seed_tracks:
        obj = Track()
        obj.init_track(track)
        if obj.id is not None:
            tracks_ids.append(obj.id)

    return tracks_ids

def form_url(self):
    """
    Formulate the URL for the Spotify API request.

    Returns:
        str: The URL for the API request.
    """
    url = f"https://api.spotify.com/v1/recommendations?limit={self.limit}"
    if self.seed_artists:
        url += "&seed_artists="
        artists_ids = self.get_seed_artists_id()
        url += self.add_item_to_url(artists_ids)
    if self.seed_genres:
        url += "&seed_genres="
```

```
            url += self.add_item_to_url(self.seed_genres)
        if self.seed_tracks:
            url += "&seed_tracks="
            tracks_id = self.get_seed_tracks_id()
            url += self.add_item_to_url(tracks_id)
        return url

    def add_item_to_url(self, items):
        """
        Add items to the URL.

        Args:
            items (list): A list of items to add to the URL.

        Returns:
            str: The URL with the added items.
        """
        url_part = ""
        if items is None:
            return url_part

        for idx in range(len(items)):
            if idx == len(items) - 1:
                url_part +=  str(items[idx])
            else:
                url_part +=  str(items[idx]) + "%2C"

        return url_part
```

 # data_from_console.py

```
"""
Input Handling Module

This module provides functions for getting user input related to object names,
colors, and recommendations.
"""
import re
from shared.input_handler import InputHandler

def get_name(obj):
```

```
    """
    Get the name of an object from the user.

    Parameters:
    obj (str): The name of the object.

    Returns:
    str: The name entered by the user.
    """
    obj = InputHandler().get_str_input(f"Enter {obj} name")
    return obj

def get_color():
    """
    Get a color from the user.

    Returns:
    str: The color entered by the user.
    """
    list_of_colors = ['RED', 'GREEN', 'YELLOW', 'BLUE', 'MAGENTA', 'CYAN', 'WHITE']
    print("Available colors: red, green, yellow, blue, magenta, cyan, white")
     input_color = InputHandler().get_one_of_list_input_ignore_case("Enter color",
list_of_colors)
    color_name = input_color.upper()
    return color_name

def get_user_input_recommendations():
    """
    Get user input for recommendations.

    Returns:
       dict: A dictionary containing user input for genre, artist, and track
recommendations.
    """
            user_input = InputHandler().get_str_input("Enter parameters for
recommendations\ne.g. genre=pop, rock; track=blinding lights; artist=the weeknd,
metallica\n")
    pattern = re.compile(r'\b(genre|artist|track)\s*=\s*([^;]+)(?:;|$)')

    user_recommendations = {'genre': [], 'artist': [], 'track': []}
```

```
    matches = pattern.finditer(user_input)

    if not matches:
        print("No matches")
        return None

    for match in matches:
        category, values = match.groups()
               user_recommendations[category].extend([value.strip()  for  value  in
values.split(',')])

    return user_recommendations
```

 # data_saver.py

```
"""
Data Saving Module

This module provides a class, DataSaver, with methods to save data to different
file formats such as JSON, CSV, and TXT.
"""
import json
import csv

from shared.file_handler import FileHandler
from shared.settings import get_lab_settings

settings = get_lab_settings("lab7")
HISTORY_FILE_PATH = settings["history_file_path"]
JSON_FILE_PATH = settings["json_file_path"]
CSV_FILE_PATH = settings["csv_file_path"]
TXT_FILE_PATH = settings["txt_file_path"]

class DataSaver:
    """
    A class that provides methods to save data to different file formats.

    Attributes:
    - data: The data to be saved.
    """
```

```
def __init__(self, data):
    """
    Initializes a DataSaver object.

    Parameters:
    - data: The data to be saved.
    """
    self.data = data

def save_to_json(self):
    """
    Saves the data to a JSON file.
    """
    json_file = FileHandler(JSON_FILE_PATH)
    json_file.write_to_file(json.dumps(self.data, indent=2))

def save_to_csv(self):
    """
    Saves the data to a CSV file.
    """
    if isinstance(self.data, dict):
        self.data = [self.data]

    flat_list = [self.flatten_json(item) for item in self.data]
    fieldnames = flat_list[0].keys() if flat_list else []

    with open(CSV_FILE_PATH, 'w', newline='', encoding='utf-8') as csv_file:
        writer = csv.DictWriter(csv_file, fieldnames=fieldnames)
        writer.writeheader()
        writer.writerows(flat_list)

def save_to_txt(self):
    """
    Saves the data to a TXT file.
    """
    txt_file = FileHandler(TXT_FILE_PATH)
    txt_file.write_to_file("")

    if isinstance(self.data, dict):
```

```
            self.data = [self.data]

        flat_list = [self.flatten_json(item) for item in self.data]
        for item in flat_list:
            for key, value in item.items():
                txt_file.append_to_file(f"{key}: {value}\n")
            txt_file.append_to_file('\n')

    def flatten_json(self, data, parent_key='', sep='.'):
        """
        Flattens a nested JSON object.

        Parameters:
        - data: The JSON object to be flattened.
        - parent_key: The parent key for the current level of the JSON object.
        - sep: The separator to be used between keys.

        Returns:
        - flat_data: The flattened JSON object.
        """
        flat_data = {}
        for key, value in data.items():
            new_key = f"{parent_key}{sep}{key}" if parent_key else key
            if isinstance(value, dict):
                flat_data.update(self.flatten_json(value, new_key, sep=sep))
            else:
                flat_data[new_key] = value
        return flat_data
```

 # data_visualization.py

```
"""
Data Visualization Module

A module that defines the DataVisualization class for visualizing data as a table
or a list.
Contains the flatten_json_data function for flattening a nested JSON object.
"""
from tabulate import tabulate
from colorama import Fore, Style
from shared.settings import get_lab_settings
```

```
from .data_from_console import get_color

settings = get_lab_settings("lab7")
DEFAULT_DATA_VISUALIZATION_SETTINGS = settings["data_visualization_settings"]
DEFAULT_TABLE_FORMAT = DEFAULT_DATA_VISUALIZATION_SETTINGS["table"]
DEFAULT_COLOR = DEFAULT_DATA_VISUALIZATION_SETTINGS["color"]

class DataVisualization:
    """
    A class that provides methods for visualizing data as a table or a list.
    """

    def __init__(self):
        """
        Initializes an instance of the DataVisualization class.
        """
        self.data = None
        self.color = DEFAULT_COLOR

    def set_data(self, data):
        """
        Set the data to be visualized.

        Parameters:
        - data: The data to be visualized.
        """
        self.data = data

    def visualize_as_table(self):
        """
        Visualize the data as a table.
        """
        if isinstance(self.data, dict):
            self.data = [self.data]

        flat_list = [self.flatten_json_data(item) for item in self.data]
        headers = flat_list[0].keys()

        color = getattr(Fore, self.color, None)
```

```
            colored_headers = [f"{color}{header}{Style.RESET_ALL}" for header in
headers]
        max_col_width = 90 // len(headers)

        table = []
        for item in flat_list:
            table.append(item.values())

            print(tabulate(table, colored_headers, tablefmt=DEFAULT_TABLE_FORMAT,
maxcolwidths=max_col_width))


    def visualize_as_list(self):
        """
        Visualize the data as a list.
        """
        if isinstance(self.data, dict):
            self.data = [self.data]

        color = getattr(Fore, self.color, None)
        flat_list = [self.flatten_json_data(item) for item in self.data]

        for i, item in enumerate(flat_list):
            line = f"{i+1}. "
            for j, key in enumerate(item):
                if j == 0:
                    print(f"{line}{color}{key}{Style.RESET_ALL} - {item.get(key)}")
                else:
                    spaces = len(line)*" "
                                print(f"{spaces}{color}{key}{Style.RESET_ALL} -
{item.get(key)}")
            print()

    def settings(self):
        """
        Set the color for data visualization.
        """
        color = get_color()
        self.color = color
        print(self.color)
```

```
    def view_settings(self):
        """
        View the current color setting for data visualization.
        """
        print("Color:", self.color)

    def flatten_json_data(self, data, parent_key='', sep='.'):
        """
        Flatten a nested JSON object.

        Parameters:
        - data: The JSON object to be flattened.
           - parent_key: The parent key of the current level of the JSON object
(default: '').
        - sep: The separator to be used between keys (default: '.').

        Returns:
        - flat_data: The flattened JSON object.
        """
        flat_data = {}
        for key, value in data.items():
            new_key = f"{parent_key}{sep}{key}" if parent_key else key
            if isinstance(value, dict):
                flat_data.update(self.flatten_json_data(value, new_key, sep=sep))
            else:
                flat_data[new_key] = value
        return flat_data
```

 # api_menu.py

```
"""
Spotify API Menu Module

This module provides a command-line interface for interacting with the Spotify API.
It includes a menu system with options to search for artists, tracks, and albums,
retrieve artist-related information, get recommendations, manage user history,
and save or print the obtained data in different formats.
"""
from UI.menu import Menu
from UI.menu_item import Item
```

```
from shared.history import History
from shared.settings import get_lab_settings
from classes.lab7.api_classes.artist import Artist
from classes.lab7.api_classes.album import Album
from classes.lab7.api_classes.track import Track
from classes.lab7.api_classes.data_by_artist import DataByArtist
from classes.lab7.data_manupulation.data_saver import DataSaver
from classes.lab7.data_manupulation.data_visualization import DataVisualization
from classes.lab7.data_manupulation.data_from_console import get_name, get_user_input_recommendations
from classes.lab7.api_classes.recommendation import Recommendation
import classes.lab7.tests.main as tests

settings = get_lab_settings("lab7")
HISTORY_FILE_PATH = settings["history_file_path"]

class APIMenu:
    """
    A class representing the API Menu for Spotify API.

    Attributes:
    - data: The data obtained from API calls.
    - history: An instance of the History class to manage user history.
     - data_visualization: An instance of the DataVisualization class to visualize data.
    """

    def __init__(self):
        """
        Initializes an instance of APIMenu.

        This method sets up the initial state of the object, including:
        - Setting `data` attribute to None.
          - Initializing the `history` attribute with an instance of the History class,
          loading historical data from the specified file path.
         - Initializing the `data_visualization` attribute with an instance of the DataVisualization class.

        Parameters:
```

```
        None

        Returns:
        None
        """
        self.data = None
        self.history = History(HISTORY_FILE_PATH)
        self.data_visualization = DataVisualization()

    def menu(self):
        """
          Displays the Spotify API menu and allows the user to navigate through
different options.
        """
        menu = Menu("\nSpotify API Menu")
        menu.set_color("green")
        menu.add_item(Item("1", "Search Menu", self.search_menu))
        menu.add_item(Item("2", "Artist Menu", self.player_menu))
        menu.add_item(Item("3", "History Menu", self.history_menu))
        menu.add_item(Item("4", "Tests", tests.__main__))
        menu.add_item(Item("0", "Exit", ))
        menu.run()

    def history_menu(self):
        """
        Displays a menu for interacting with the history feature.

        This method creates a menu with options to view and clear history.
         The user's choice triggers corresponding actions and updates the program
state.

        Parameters:
        None

        Returns:
        None
        """
        history_menu = Menu("\nHistory Menu")
        history_menu.set_color("green")
```

```
                                history_menu.add_item(Item("1",    "View    History",
self.history.print_history))
                                history_menu.add_item(Item("2",    "Clear    History",
self.history.clear_history))
        history_menu.add_item(Item("0", "Back"))
        history_menu.run()

    def search_menu(self):
        """
        Displays a menu for searching artists, tracks, and albums.

        This method creates a menu with options to search for artists, tracks, and
albums.
          User choices trigger specific search functions and update the program
state.

        Parameters:
        None

        Returns:
        None
        """
        search_menu = Menu("\nSearch Menu")
        search_menu.set_color("green")
        search_menu.add_item(Item("1", "Search Artist", self.__search_artist))
        search_menu.add_item(Item("2", "Search Track", self.__search_track))
        search_menu.add_item(Item("3", "Search Album", self.__search_album))
        search_menu.add_item(Item("0", "Back"))
        search_menu.run()

    def __search_artist(self):
        """
          Searches for an artist and updates the program state with the artist's
information.

          This method prompts the user to enter an artist's name and retrieves
information about the artist.
        The artist's data is formatted and stored, and the action is logged in the
history.
```

```
        Parameters:
        None

        Returns:
        None
        """
        artist_name = get_name("artist")
        artist = Artist()
        artist.init_artist(artist_name)
        self.data = artist.get_artist_formatted_json()
        self.history.add_event(f"Get Artist {artist_name}")
        self.choose_menu()

    def __search_track(self):
        """
           Searches for a track and updates the program state with the track's information.

            This method prompts the user to enter a track's name and retrieves information about the track.
         The track's data is formatted and stored, and the action is logged in the history.

        Parameters:
        None

        Returns:
        None
        """
        track_name = get_name("track")
        track = Track()
        track.init_track(track_name)
        self.data = track.get_track_formatted_json()
        self.history.add_event(f"Get Track {track_name}")
        self.choose_menu()

    def __search_album(self):
        """
           Searches for an album and updates the program state with the album's information.
```

```
            This method prompts the user to enter an album's name and retrieves
information about the album.
         The album's data is formatted and stored, and the action is logged in the
history.

        Parameters:
        None

        Returns:
        None
        """
        album_name = get_name("album")
        album = Album()
        album.init_album(album_name)
        self.data = album.get_album_formatted_json()
        self.history.add_event(f"Get Album {album_name}")
        self.choose_menu()

    def player_menu(self):
        """
        Displays a menu for interacting with artist-related features.

          This method creates a menu with options to get an artist's top tracks,
albums, and recommendations.
          User choices trigger specific actions related to artists and update the
program state.

        Parameters:
        None

        Returns:
        None
        """
        player_menu = Menu("\nArtist Menu")
        player_menu.set_color("green")
                    player_menu.add_item(Item("1",   "Get   Artist's   Top   Tracks",
self.__get_artist_top_tracks))
                        player_menu.add_item(Item("2",   "Get   Artist's   Albums",
self.__get_artist_albums))
```

```
                        player_menu.add_item(Item("3",   "Get   Recommendations",
self.__get_recommendations))
        player_menu.add_item(Item("0", "Back"))
        player_menu.run()

    def __get_recommendations(self):
        """
        Retrieves track recommendations based on user input and updates the program
state.

         This method prompts the user for input, generates track recommendations,
and stores the recommendations.
        The action is logged in the history.

        Parameters:
        None

        Returns:
        None
        """
        user_input = get_user_input_recommendations()
                 user_rec  =  Recommendation(seed_artists=user_input.get("artist"),
seed_genres=user_input.get("genre"), seed_tracks=user_input.get("track"))
        self.data = user_rec.get_track_recommendation_formatted_json()
        self.history.add_event("Get recommendations")
        self.choose_menu()

    def __get_artist_top_tracks(self):
        """
        Retrieves an artist's top tracks and updates the program state.

         This method prompts the user to enter an artist's name, retrieves the top
tracks,
        and stores the tracks' data. The action is logged in the history.

        Parameters:
        None

        Returns:
        None
```

```
        """
        artist_name = get_name("artist")
        artist = DataByArtist()
        artist.init_artist(artist_name)
        self.data = artist.get_tracks_formatted_json()
        self.history.add_event(f"Get {artist_name} Top Tracks")
        self.choose_menu()

    def __get_artist_albums(self):
        """
        Retrieves an artist's albums and updates the program state.

          This method prompts the user to enter an artist's name, retrieves the albums,
        and stores the albums' data. The action is logged in the history.

        Parameters:
        None

        Returns:
        None
        """
        artist_name = get_name("artist")
        artist = DataByArtist()
        artist.init_artist(artist_name)
        self.data = artist.get_albums_formatted_json()
        self.history.add_event(f"Get {artist_name} Albums")
        self.choose_menu()

    def choose_menu(self):
        """
        Displays a menu for choosing between printing, saving, or going back.

         This method creates a menu with options to print, save, or go back to the previous menu.
        User choices trigger specific actions and update the program state.

        Parameters:
        None
```

```
        Returns:
        None
        """
        choose_menu = Menu("\nPrint or Save")
        choose_menu.set_color("green")
        choose_menu.add_item(Item("1", "Print", self.print_menu))
        choose_menu.add_item(Item("2", "Save", self.save_menu))
        choose_menu.add_item(Item("0", "Back"))
        choose_menu.run()

    def save_menu(self):
        """
        Displays a menu for saving data in different formats.

         This method creates a menu with options to save data in JSON, CSV, or TXT
formats.
        User choices trigger specific saving actions.

        Parameters:
        None

        Returns:
        None
        """
        save_menu = Menu("\nSave Menu")
        save_menu.set_color("green")
        save_menu.add_item(Item("1", "Save JSON", self.__save_json))
        save_menu.add_item(Item("2", "Save CSV", self.__save_csv))
        save_menu.add_item(Item("3", "Save TXT", self.__save_txt))
        save_menu.add_item(Item("0", "Back", self.choose_menu))
        save_menu.run()

    def __save_json(self):
        """
        Saves the current data to a JSON file.

        This method initializes a DataSaver object with the current data and
        triggers the save_to_json method to save the data in JSON format.

        Parameters:
```

```
        None

        Returns:
        None
        """
        data_saver = DataSaver(self.data)
        data_saver.save_to_json()

    def __save_csv(self):
        """
        Saves the current data to a CSV file.

        This method initializes a DataSaver object with the current data and
        triggers the save_to_csv method to save the data in CSV format.

        Parameters:
        None

        Returns:
        None
        """
        data_saver = DataSaver(self.data)
        data_saver.save_to_csv()

    def __save_txt(self):
        """
        Saves the current data to a TXT file.

        This method initializes a DataSaver object with the current data and
        triggers the save_to_txt method to save the data in TXT format.

        Parameters:
        None

        Returns:
        None
        """
        data_saver = DataSaver(self.data)
        data_saver.save_to_txt()
```

```
    def print_menu(self):
        """
        Displays a menu for printing data.

         This method creates a menu with options to print data as a table, list,
view settings, or change color.
        User choices trigger specific print or settings actions.

        Parameters:
        None

        Returns:
        None
        """
        print_menu = Menu("\nPrint Menu")
        print_menu.set_color("green")
        print_menu.add_item(Item("1", "Print Table", self.__print_table))
        print_menu.add_item(Item("2", "Print List", self.__print_list))
        print_menu.add_item(Item("3", "Settings", self.settings_menu))
        print_menu.add_item(Item("0", "Back", self.choose_menu))
        print_menu.run()

    def __print_table(self):
        """
        Prints the current data as a table.

            This method sets the data for visualization, and then triggers the
visualization
        of the data in table format using the DataVisualization class.

        Parameters:
        None

        Returns:
        None
        """
        self.data_visualization.set_data(self.data)
        self.data_visualization.visualize_as_table()

    def __print_list(self):
```

```
        """
        Prints the current data as a list.

            This method sets the data for visualization, and then triggers the
visualization
        of the data in list format using the DataVisualization class.

        Parameters:
        None

        Returns:
        None
        """
        self.data_visualization.set_data(self.data)
        self.data_visualization.visualize_as_list()

    def settings_menu(self):
        """
        Displays a menu for managing visualization settings.

        This method creates a menu with options to view settings, change color, or
go back.
        User choices trigger specific settings actions.

        Parameters:
        None

        Returns:
        None
        """
        settings_menu = Menu("\nSettings Menu")
        settings_menu.set_color("green")
        settings_menu.add_item(Item("1", "View Settings", self.print_settings))
        settings_menu.add_item(Item("2", "Change Color", self.change_color))
        settings_menu.add_item(Item("0", "Back", self.print_menu))
        settings_menu.run()

    def print_settings(self):
        """
        Displays the current application settings.
```

```
        This method retrieves and displays the current settings for visualization,
        allowing users to view and adjust the application's display settings.

        Parameters:
        None

        Returns:
        None
        """
        self.data_visualization.set_data(self.data)
        self.data_visualization.view_settings()

    def change_color(self):
        """
        Allows the user to change the color settings of the application.

        This method prompts the user to input new color preferences, which are then
applied
        to the visualization components of the application.

        Parameters:
        None

        Returns:
        None
        """
        self.data_visualization.set_data(self.data)
        self.data_visualization.settings()
```

 # runner.py

```
"""
Module: run_api_menu

Module provides a simple script to run the API Menu for Lab 7.
"""
from classes.lab7.api_menu.api_menu import APIMenu

def run():
    """
```

```
    Initializes and runs the API Menu.
    """
    api_menu = APIMenu()
    api_menu.menu()
```

GitHub Repository: https://github.com/trolchiha/SPL-labs.git

Висновок: під час виконання лабораторної роботи навчилася створювати проект, який надає досвід роботи з API, дизайном користувацького інтерфейсу, валідацією введення, обробкою помилок та тестування.