ОЛИМПИАДА ШКОЛЬНИКОВ «ШАГ В БУДУЩЕЕ»

НАУЧНО-ОБРАЗОВАТЕЛЬНОЕ СОРЕВНОВАНИЕ

«ШАГ В БУДУЩЕЕ, МОСКВА»

18695

регистрационный номер

Факультет ИУ «Информатика и системы управления»

Кафедра ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

СОЗДАНИЕ МУЗЫКАЛЬНОГО МОБИЛЬНОГО ПРИЛОЖЕНИЯ С СИСТЕМОЙ РЕКОМЕНДАЦИЙ И РАСПОЗНАВАНИЕМ МУЗЫКИ

Авторы: Кулаков Никита Михайлович

Наумов Марк Алексеевич

ГБОУ г. Москвы «Бауманская инженерная школа №1580
11 класс

Научный руководитель: Митрофанов Михаил Сергеевич

ГБОУ г. Москвы «Бауманская инженерная школа №1580

_____

Москва, 2023

Создание музыкального мобильного приложения с системой рекомендаций и распознаванием музыки

Аннотация

Цель работы - создать музыкальный стриминговый сервис: базу данных, методы взаимодействия FrontEnd и BackEnd частей приложения, макет приложения, описать взаимодействия его компонентов, способы посылки сообщений. Предмет работы - создание музыкального стримингового сервиса.

Актуальность работы - на данный момент большое количество зарубежных мобильных приложений для прослушивания музыки были вынуждены прекратить свою работу на территорию РФ, что означает формирование большого количества пользователей, не определившихся с выбором новой площадки для прослушивания музыки. Им можно предоставить возможность перехода на стриминговую платформу собственного производства, а также привлечь внимание к своему приложению пользователей других площадок.

Методологическая основа:

Теоретическая:

1. Анализ литературы
2. Аналогия
3. Классификация
4. Конкретизация
5. Моделирование
6. Синтез
7. Формализация

Практическая:

1. Описание
2. Практическое моделирование
3. Тестирование
4. Анкетирование

Срок работы: декабрь-март 2023 года

Практическая значимость: созданное приложение может быть использовано большим кругом населения для прослушивания музыки

Инструменты создания: Средой разработки был выбран Visual Studio Code как наиболее удобная в использовании среда, используемая в сфере

создания приложений и сайтов. Веб-сервис GitHub служил хостингом для совместной разработки проекта.

FrontEnd: Инструменты: JavaScript, React Native EXPO, JSX, различные библиотеки для компонентов.

BackEnd:

1. Python
2. Дочерние библиотеки Python:
    - Tortoise ORM
    - FastAPI
    - Pydantic
    - NumPy
    - Librosa
    - Pandas
    - Scikit-learn
3. СУБД (MySQL):
    - MySQL Workbench 8.0 CE
    - MySQL Server 8.0
4. Docker (MySQL-image, Adminer-image)

Содержание

Введение

Музыка – неотъемлемый атрибут жизни многих людей. Через нее люди могут выразить те эмоции, которые они не могут выразить ни в каком другом виде деятельности. Многие люди не могут представить себе жизнь без музыки. Также в настоящее время с трудом можно найти человека, у которого нет мобильного телефона с доступом в интернет.

Учитывая то, какими стремительными темпами развивается индустрия разработки мобильных приложений, требуется создать приложение для онлайн-прослушивания музыки.

Каждый человек обладает индивидуальным музыкальным вкусом, а музыкальная библиотека с каждым годом музыкальная библиотека пополняется все новыми и новыми альбомами, песнями, жанрами. Для того, чтобы удовлетворить среднестатистического человека, необходимо создать алгоритм, который будет подбирать музыку, похожую по стилистике и жанру на музыку, которую слушает пользователь, открывая ему, новых, неизвестных ему исполнителей. Данная особенность будет полезна как для пользователей, которые слушают нравящуюся музыку, так и для малоизвестных исполнителей, чьи треки могут обрести популярность.

Довольно часто возникает необходимость в том, чтобы узнать название песни по ее звучанию, так как не всегда пользователь может знать имя исполнителя или название трека. Для решения данной проблемы требуется создать алгоритм, который по аудиодорожке трека сможет находить похожие музыкальные композиции.

При регистрации в приложении должен собираться минимальный набор персональных данных пользователя (имя, адрес электронной почты) для исключения утечки важных для пользователя данных.

Как итог, приложение должно удовлетворять следующим требованиям:

1. Обширная библиотека с композициями различных жанров

2. Удобный, понятный среднестатистическому пользователю интерфейс
3. Поддержка популярных мобильных платформ (IOS, Android)
4. Возможность поиска похожих песен по аудиодорожке
5. Адаптивная система рекомендаций
6. Безопасность данных пользователя

Цель работы - создание музыкального стримингового сервиса, который будет включать в себя алгоритм распознавания музыки и систему рекомендаций, базы данных, методов взаимодействия клиентской и серверной части приложения. Предмет работы - создание музыкального стримингового сервиса.

Актуальность: на данный момент большое количество зарубежных мобильных приложений для прослушивания музыки были вынуждены прекратить свою работу на территорию РФ, что означает формирования большой ниши, в которой расположены пользователи, не определившиеся с выбором новой площадки для прослушивания музыки. Эту нишу можно заполнить приложением собственного производства, а также привлечь внимание к своему приложению пользователей других площадок.

Аппаратная часть проекта включает в себя:

- *Серверная часть* – реализована с использованием языка программирования Python, библиотек:
    - FastAPI и Pydantic (обработка запросов клиентской части приложения)
    - Tortoise ORM (работа с базой данных MySQL)
    - NumPy, Librosa, Pandas, Scikit-learn (создание системы рекомендаций)

    - Docker (создание MySQL, FastAPI контейнеров для предотвращения непредвиденных конфликтов между частями серверной части, обеспечения безопасности данных)
- *Клиентская часть* – реализована с использованием языка программирования JavaScript и фреймворков:
    - React Native
    - EXPO AV
    - MobX
    - Различные внутренние библиотеки

Методы алгоритмической и программной реализации

Методологическую основу разработанных и программно-реализованных алгоритмов составляет принятие и обработка запросов клиента на серверной части с последующим внесением изменений в базу данных или получением данных из нее. Запросы можно условно разделить на 2 типа: извлечение данных (get) и внесение данных (post, put, delete). ***Средство программной реализации*** – язык программирования Python с применением объектно-ориентированной методологии. В качестве стека разработки были использованы следующие сервисы и среды:

- Visual Studio code как основная среда разработки ввиду своей адаптивности и возможности работать с большинством типов файлов, а также удобной работой с системой контроля версий git. Это обуславливается наличием большого количества расширений разной направленности в приложении.
- Pycharm как среда для написания алгоритма распознавания песен.
- Docker desktop как приложение для удобной визуализации и контролем за контейнерами (в случае проекта – MySQL, FastAPI, Adminer).

- Swagger UI как набор инструментов, встроенный в FastAPI для удобной работой с запросами и их тестированием.

Для создания схемы базы данных была выбрана библиотека для Python – Tortoise ORM с последующей миграцией в СУБД MySQL. Это было сделано по ряду причин:

1. Объектно-ориентированность библиотеки
2. Простые в реализации миграции
3. Удобные в использовании обращения к полям базы данных
4. Тесное взаимодействие с библиотекой обработки запросов FastAPI
5. Возможность асинхронной работы

При создании схемы базы данных были описаны следующие таблицы:

1. User – таблица с данными пользователей с ключом на Library.id пользователя
2. Library – таблица с данными о библиотеке пользователя (Tracks, Albums, Playlists, Artists, Genres)
3. Track – таблица с данными о песнях
4. Playlist – таблица с данными о плейлистах
5. Genre – таблица с данными о жанрах
6. Album – таблица с данными об альбомах
7. Artist – таблица с данными об исполнителях

В результате формирования Many-To-Many зависимостей между таблицами было образовано еще несколько побочных таблиц.

После создания схемы базы данных я произвожу миграцию схемы в MySQL базу данных с помощью метода register_tortoise.

Программную структуру приложения можно разделить на несколько этапов:

1. Получение запроса от клиента (в зависимости от типа запроса могут задействоваться различные средства обработки запроса)
    - Get запрос – получение идентификационных данных запроса (зачастую – id) для получения определенных полей из базы данных
    - Post запрос – получение данных от клиента в виде JSON или FileUpload для формирования новых полей в базе данных
    - Put запрос – получение данных от клиента в виде JSON или FileUpload для внесения изменений в существующие поля в базе данных
    - Delete запрос – получение идентификационных данных запроса (зачастую – id) для удаления существующих полей в базе данных
2. Обработка запроса - форматирование полученных от клиента данных с помощью методов для последующей работы с ними
3. Формирование ответа на запрос:
    - Post, put, delete запросы – ответ в виде HTTP кода выполнения запроса
    - Get запрос – ответ в виде JSON или FileRequest

Рассмотрим каждый из этапов программной структуры приложения подробно и опишем принципы их работы.

Этап 1. Получение запроса от клиента

Получение запроса от клиента приложения является первым этапом в обработке запросов и работы приложения в целом. В клиентской части запрос с использованием функции fetch выглядит следующим образом:

1. Указание URL-адреса, на который посылается запрос

2. Указание метода запроса (POST, GET, PUT, DELETE)
3. Указание Content-type
4. Ввод тела запроса (JSON, File или HTML-form)

В серверной части запрос выполняется асинхронно с помощью декоратора с указанием названия переменной класса RouterAPI и метода запроса (POST, GET, PUT, DELETE), URL-адреса, на который будет производиться запрос; после вводится асинхронная функция, в параметрах которой указываются данные, получаемые в запросе. Как дополнительные параметры запроса указываются:

- Response_class – указание класса ответа запроса
- Response_model – указание модели ответа запроса в виде Pydantic модели
- Status_code – указание кода, возвращаемого при успешном выполнении запроса

В зависимости от метода запроса, формируются различные ответы (JSON, FileRequest или Status-code).

Этап 2. Обработка запроса

Для обработки запросов используются следующие инструменты:

- Pydantic для создания моделей тела запроса и тела ответа
- Tortoise для взаимодействия с полями базы данных
- Hashlib (в частности, алгоритм шифрования pbkdf2_hmac) для шифрования отдельных данных пользователя

В обобщении принцип обработки запроса можно разделить на несколько этапов:

- Для GET запроса:
    - Получение тела запроса (зачастую – идентификационные данные)

    - Обращение к базе данных, их получение
    - Структуризация полученных из базы данных полей, формирование тела ответа запроса
- Для POST, PUT запросов:
    - Получение тела запроса (данные для внесения в базу данных)
    - Предобработка данных (например, хеширование, формализация)
    - Обращение к базе данных, добавление новых полей или изменение уже существующих
- Для DELETE запроса:
    - Получение тела запроса (зачастую – идентификационные данные)
    - Обращение к базе данных, удаление нужных полей

Этап 3. Формирование ответа на запрос

В качестве ответа на запрос могут выступать различные структуры данных в зависимости от метода запроса:

- JSON – используется в качестве тела ответа запроса для метода GET. Также применяется для формирования сложных ответов в методах POST, PUT, DELETE. Представляет из себя структуру в виде словаря с данными типа ключ-значение.
- FileReponse – применяется в качестве тела ответа запроса для метода GET. Представляет из себя файл, содержащий метадату, с именем файла и расширением.
- Status_code – применяется в качестве тела ответа запроса для методов POST, PUT, DELETE. Является числовым значением, значение которого определяется таблицей значений status_code HTTP протокола безопасности.

Файловая структура приложения

Для описания функционала серверной части приложения была использования следующая файловая структура:

- /data
- /src
    - /app
        - base
            - service_base.py
        - library
            - models.py
            - <any model>
                - schemas.py
                - service.py
                - <any model>_endpoint.py
        - user
            - models.py
            - schemas.py
            - service.py
            - user_endpoint.py
        - router.py
        - router_schemas.py
        - router_service.py
    - /config
        - config.py
- main.py
- docker-compose.yml

Где

- data – директория хранения файлов пользователя и приложения (изображения, аудиофайлы)
- service_base.py – файл с базовыми типами данных и функциями обработки запросов
- <any model> - определенная модель таблицы из списка (с. 9) за исключением таблицы User
- schemas.py – файл с описанием моделей тела запроса и ответа на запрос
- service.py – файл с описанием методов обработки запросов

- <any model>_endpoint.py – файл с описанием запросов к определенной модели базы данных
- router.py, router_schemas.py, router_service.py – файлы для обработки глобальных запросов (получение всех данных для отображения определенной страницы)
- config.py – файл с описанием сервера подключения базы данных, списка моделей приложения, списка солей шифрования с использованием динамической соли
- docker-compose.yml – файл с описанием контейнеров, использующихся для работы сервера

Принцип работы системы рекомендаций

Для реализации алгоритма музыкальных рекомендаций была использована модель KNN (k-nearent neighbors), в качестве ***исходных данных*** был выбран датасет сервиса Spotify, в качестве алгоритма – KNeighborsClassifier библиотеки sklearn.

Датасет представляет из себя файл формата .csv с определенным количеством песен (для демонстрации работы алгоритма было взято порядка 100 песен)

Этапы работы модели можно описать следующим образом:

1. Чтение датасетов (data.csv, data_by_genres.csv, data_by_year.csv) с помощью метода read_csv библиотеки pandas
2. Нормализация данных (в случае взятого за основу датасета – данные нормализованы изначально)
3. Формирование кластеров жанров на основе числовых параметров (valence, acousticness, danceability, energy, liveness, instrumentalness)
4. Формирование кластеров песен на основе числовых параметров, указанных выше

5. Сопоставление кластеров жанров и песен
6. Формирование списка песен, похожих на данную либо на список данных песен

Результаты программной реализации серверной части

Выполнение этапов обработки запросов выполняется в строго указанной последовательности.

Для создания архитектуры серверной части приложения была выбрана концепция построения модульной архитектуры приложения, что обеспечивает:

- Масштабируемость
- Ремонтопригодность
- Заменимость модулей
- Возможность тестирования
- Переиспользование
- Сопровождение

Приложение построено с применением объектно-ориентированного стиля программирования с соблюдением всех основных парадигм стиля ООП для языка динамической типизации (инкапсуляция, полиморфизм, наследование, сокрытие, посылка сообщений)

Все компоненты приложения описаны в виде классов (модели тел запросов и ответов, классы обработки запросов, модели таблиц базы данных)

В результате программной разработки создана серверная часть приложения-стримингового сервиса музыки, состоящее из классов моделей тел запросов и ответов на запросы, классов методов обработки запросов, классов моделей таблиц базы данных, класса обучения алгоритма рекомендаций, сторонних вспомогательных структур.

Создание UI приложения

Создание UI приложения можно разделить на несколько частей:

1. Выбор инструментов для разработки
2. Создание дизайна
3. Реализация

Выбор инструментов

React Native был выбран по нескольким причинам:

1) он имеет структуру, схожую с React, который используется для написания сайтов, что упрощает разработку для человека, который уже знаком с React.
2) React Native поддерживает кроссплатформенную разработку для IOS и Android, что ускоряет процесс разработки.
3) React Native имеет обширную документацию, а так же большое количество внутренних библиотек, так же облегчающих разработку

Создание Дизайна

Дизайн создавался в приложении Figma (рис. 29), являющееся одним из лучших на данный момент приложений в сфере дизайна.

Параллельно с созданием дизайна, регулярно проводились опросы, в ходе которых выяснялось, чего конкретно пользователям не хватает в музыкальных приложениях, которыми они пользуются сейчас. На основе этого велась доработка дизайна.

Реализация UI

Реализовано приложение на языке Java Script в среде разработки Visual Studio Code (рис. 30). Для хранения глобальных состояний и переменных, таких как очередь плеера, использовался MobX(Рис. 31). MobX - это

библиотека, которая делает управление состоянием простым и масштабируемым за счёт функционально-реактивного программирования (TFRP). Вся остальная верстка заключается в тщательном изучении документации, верстки по макету и доработок дизайна.

Алгоритм распознавания музыки

- Что делает алгоритм

Пользователь открывает в приложении запись звука, записывает звук и он передается в алгоритм. Там создается аудио отпечаток (далее fingerprint) песни, который сравнивается с такими же fingerprint песен в базе данных.

- Как делается fingerptrint:

1. Преобразование аудиофайла в файл .wav формата
2. Поиск частот сигналов при помощи *дискретного преобразования Фурье* (STFT).
3. Создание спектрограмма звука на основе преобразования (рис. 32)
4. Поиск пиковых значений спектрограммы, которые создают массив точек-пиков(рис. 33)
5. Сравнение созданных пиков частот с уже имеющимися.
6. Нахождение искомого сигнала путем поиска наибольшего количества совпадений

- Что можно улучшить

В будущем можно улучшить алгоритм с помощью составления векторов из пар точек, что даст более точную оценку пиков.

Использование результатов

Реализованный программный продукт позволяет хранить данные о большом количестве песен, жанров, исполнителей, альбомов, плейлистов, обрабатывать запросы большого количества пользователей (ввиду полной асинхронной работы), формирования плейлиста песен, рекомендованных пользователю на основе понравившихся ему песен.

При дальнейшей работе над программной структурой приложения предполагается добавление следующих нововведений:

- Улучшение алгоритма рекомендаций музыки с учетом недавно прослушанных пользователем песен и популярности отдельно взятой песни
- Возможность написания комментариев под песней
- Введение дневных плейлистов и плейлистов с песнями разных жанров
- Возможность добавления пользователем собственных песен

Разработанное приложение может найти практическое применение в жизни каждого человека как сервис для прослушивания музыки. Более того, в долгосрочной перспективе приложение предполагается как конкурент многим сервисам прослушивания музыки.

Предполагается использование проекта в индивидуальных целях, а также для сбора информации о песнях, пользующихся популярностью, и формирования базы данных для дальнейшего ее анализа зависимости популярности песни, основываясь на многих ее параметрах. После добавления нововведений, описанных выше, предполагается превращение приложения из простого стримингового сервиса в свободную для музыкального творчества площадку с возможностью продвижения собственного творчества.

Срок завершения проекта и перехода от тестового образца к полностью рабочему, способному выполнять поставленные задачи с нужной

эффективностью и качеством, варьируется в районе 6 месяцев. При таких условиях проект будет представлять интерес для коммерческих предприятий, а также будет возможность обретения лицензии на вещание песен, защищенных авторским правом, монетизации проекта путем введения платной подписки.

Для доведения приложения до полноценно рабочего, необходимо реализовать написанные выше алгоритмы и структуры, а также получить большое количество данных о музыкальных предпочтениях пользователей разных социальных групп, что позволит построить более эффективный алгоритм подбора музыки.

На данный момент большое количество зарубежных мобильных приложений для прослушивания музыки были вынуждены прекратить свою работу на территорию РФ, что означает формирование большого количества пользователей, не определившихся с выбором новой площадки для прослушивания музыки. Им можно предоставить возможность перехода на стриминговую платформу собственного производства, а также привлечь внимание к своему приложению пользователей других площадок.

С финансовой точки зрения, затраты на поддержание работоспособности приложения обуславливаются стоимостью хостинга серверной части, в перспективе – оплатой лицензии на стриминг песен, защищенных авторским правом.

Список литературы

- Stephen Shum - The Basics of audio fingerprinting
- Shakeel Zafar, Imran Fareed, Muhammad Majid - Speech Quality Assessment using Mel Frequency Spectrograms of Speech Signals
- Avery Li-Chun Wang - An Industrial-Strength Audio Search Algorithm
- https://pandas.pydata.org/docs
- https://datatofish.com/
- https://www.geeksforgeeks.org/
- https://www.kaggle.com/
- https://habr.com/
- https://tonais.ru/library/algoritm-knn-scikit-learn-python
- https://librosa.org/doc/
- https://stackoverflow.com/
- https://towardsdatascience.com/
- https://tortoise.github.io/
- https://proglib.io/
- https://fastapi.tiangolo.com/
- https://docs.docker.com/
- https://dev.mysql.com/doc/
- https://docs.expo.dev/
- https://mobx.js.org/README.html
- https://reactnative.dev/docs/getting-started
- https://habr.com/ru/company/yandex/blog/181219/
- https://github.com/Aaron-AALG/SpectroMap

Приложение

```python
class User(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=63)
    login = fields.CharField(max_length=127,
                             unique=True)
    email = fields.CharField(max_length=255,
                             unique=True)
    hashed_password = fields.CharField(max_length=1000)
    registration_date = fields.CharField(max_length=30)

    picture_file_path = fields.CharField(max_length=1000, default='data/default_image.png')

    class PydanticMeta:
        exclude = ['hashed_password']

    class Meta:
        table = 'User'
```

Рис. 1 Модель таблицы User

```python
class Library(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)

    user: fields.ForeignKeyRelation['User'] = fields.ForeignKeyField('models.User',
                                                                     related_name='User_id')

    tracks: fields.ManyToManyRelation['Track'] = fields.ManyToManyField('models.Track',
                                                                         related_name='libraries',
                                                                         through='Library_tracks')
    artists: fields.ManyToManyRelation['Artist'] = fields.ManyToManyField('models.Artist',
                                                                           related_name='libraries',
                                                                           through='Library_artists')
    albums: fields.ManyToManyRelation['Album'] = fields.ManyToManyField('models.Album',
                                                                         related_name='libraries',
                                                                         through='Library_albums')
    playlists: fields.ManyToManyRelation['Playlist'] = fields.ManyToManyField('models.Playlist',
                                                                               related_name='libraries',
                                                                               through='Library_playlists')
    genres: fields.ManyToManyRelation['Genre'] = fields.ManyToManyField('models.Genre',
                                                                         related_name='libraries',
                                                                         through='Libraries_genres')
```

Рис. 2 Модель таблицы Library

```python
class Track(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=255,
                            not_null=True)
    duration_s = fields.IntField(default=None)
    track_file_path = fields.CharField(max_length=1000,
                                       not_null=True,
                                       default='')

    picture_file_path = fields.CharField(max_length=1000,
                                       not_null=True, default='data/default_image.png')

    libraries: fields.ManyToManyRelation['Library']
    artists: fields.ManyToManyRelation['Artist'] = fields.ManyToManyField('models.Artist',
                                                                          related_name='tracks',
                                                                          through='Tracks_artists')
    playlists: fields.ManyToManyRelation['Playlist'] = fields.ManyToManyField('models.Playlist',
                                                                              related_name='tracks',
                                                                              through='Tracks_playlists')

    album: fields.ForeignKeyNullableRelation['Album'] = fields.ForeignKeyField('models.Album',
                                                                              related_name='Album_id',
                                                                              null=True)
    genre: fields.ForeignKeyRelation['Genre'] = fields.ForeignKeyField('models.Genre',
                                                                       related_name='Album_id')
```

Рис. 3 Модель таблицы Track

```python
class Playlist(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=63,
                            not_null=True)
    description = fields.TextField()
    release_date = fields.CharField(max_length=30)

    picture_file_path = fields.CharField(max_length=1000,
                                         not_null=True,
                                         default='data/default_image.png')

    libraries: fields.ManyToManyRelation['Library']
    tracks: fields.ManyToManyRelation['Track']
    genres: fields.ManyToManyRelation['Genre'] = fields.ManyToManyField('models.Genre',
                                                                        related_name='playlists',
                                                                        through='Playlists_genres')

    creator: fields.ForeignKeyRelation['User'] = fields.ForeignKeyField('models.User',
                                                                        related_name='playlist_creator_id')

    class PydanticMeta:
        allow_cycles = False

    class Meta:
        table = 'Playlist'
```

Рис. 4 Модель таблицы Library

```python
class Genre(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=63,
                            unique=True,
                            not_null=True)
    alt_name = fields.CharField(max_length=63,
                                unique=True)
    description = fields.TextField()

    libraries: fields.ManyToManyRelation['Library']
    artists: fields.ManyToManyRelation['Artist']
    albums: fields.ManyToManyRelation['Album']
    playlists: fields.ManyToManyRelation['Playlist']

    class PydanticMeta:
        allow_cycles = False

    class Meta:
        table = 'Genre'
```

Рис. 5 Модель таблицы Genre

```python
class Album(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=255,
                            not_null=True)
    description = fields.TextField()
    release_date = fields.CharField(max_length=127)

    picture_file_path = fields.CharField(max_length=1000,
                                         not_null=True,
                                         default='data/default_image.png')

    libraries: fields.ManyToManyRelation['Library']
    artists: fields.ManyToManyRelation['Artist']
    genres: fields.ManyToManyRelation['Genre'] = fields.ManyToManyField('models.Genre',
                                                                        related_name='albums',
                                                                        through='Albums_genres')

    class PydanticMeta:
        allow_cycles = False

    class Meta:
        table = 'Album'
```

Рис. 6 Модель таблицы Album

```python
class Artist(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    name = fields.CharField(max_length=63,
                            unique=True,
                            not_null=True)
    registration_date = fields.CharField(max_length=30)

    picture_file_path = fields.CharField(max_length=1000,
                                         not_null=True,
                                         default='data/default_image.png')

    libraries: fields.ManyToManyRelation['Library']
    tracks: fields.ManyToManyRelation['Track']
    genres: fields.ManyToManyRelation['Genre'] = fields.ManyToManyField('models.Genre',
                                                                        related_name='artists',
                                                                        through='Artists_genres')
    albums: fields.ManyToManyRelation['Album'] = fields.ManyToManyField('models.Album',
                                                                        related_name='artists',
                                                                        through='Artists_albums')

    class PydanticMeta:
        allow_cycles = False

    class Meta:
        table = 'Artist'
```

Рис. 7 Модель таблицы Artist

```
+--------------------+
| Tables_in_testing  |
+--------------------+
| Album              |
| Albums_genres      |
| Artist             |
| Artists_albums     |
| Artists_genres     |
| Comment            |
| Genre              |
| Libraries_genres   |
| Library            |
| Library_albums     |
| Library_artists    |
| Library_playlists  |
| Library_tracks     |
| Playlist           |
| Playlists_genres   |
| Track              |
| Track_params       |
| Tracks_artists     |
| Tracks_playlists   |
| User               |
+--------------------+
20 rows in set (0.00 sec)
```

Рис. 8 Список таблиц после совершения миграции, включая промежуточные таблицы

```python
@user_router.get('/get_picture/{user_id}', response_class=FileResponse)
async def user_get_picture(
    user_id: int
):
    return await service.user_s.get_image(id=user_id)
```

Рис. 9 Метод запроса Get на примере получения изображения пользователя

```python
@user_router.post('/create', status_code=status.HTTP_200_OK)
async def user_create(
    user: schemas.User_create_request = Depends(schemas.User_create_request.as_form),
    artists: List[int] = Depends(artists_checker),
    genres: List[int] = Depends(genres_checker),
    picture_file: UploadFile = File(...)
):
    return await service.user_s.create(user, artists, genres, picture_file)
```

Рис. 10 Метод запроса Post на примере запроса создания поля User

```python
@user_router.put('/change/data/{user_id}', response_model=schemas.User_change_picture_response)
async def user_change_picture(
    user_id: int = Depends(schemas.User_change_picture.as_form),
    new_picture_file: UploadFile = File(...),
):
    return await service.user_s.change_picture(user_id, new_picture_file)
```

Рис. 11 Метод запроса Put на примере изменения изображения пользователя

```python
@user_router.delete('/delete/data/{user_id}', status_code=204)
async def user_delete_picture(
    user_id: int = Depends(schemas.User_change_picture.as_form)
):
    return await service.user_s.delete_picture(user_id)
```

Рис. 12 Метод запроса Delete на примере удаления изображения пользователя

```python
async def upload_file(self, dir: str, file: UploadFile = File(...)):
    try:
        file_directory = f'data/{dir}'
        if not os.path.exists(file_directory):
            os.makedirs(file_directory)
        file.filename = f'{str(uuid.uuid4())}.{file.filename.split(".")[1]}'
        f = await run_in_threadpool(open, f'{file_directory}/{file.filename}', 'wb')
        await run_in_threadpool(shutil.copyfileobj, file.file, f)
    except Exception():
        return {'file_path': 'NULL'}
    finally:
        if 'f' in locals(): await run_in_threadpool(f.close)
        await file.close()

    return {'file_path': f'{file_directory}/{file.filename}'}

async def get_image(self, **kwargs):
    obj = await self.model.get(**kwargs)
    if obj.picture_file_path != 'data/default_image.png':
        return FileResponse(obj.picture_file_path,
                            media_type=f'image/{obj.picture_file_path.split(".")[1]}',
                            filename=f'{obj.picture_file_path.split("/")[1]}_{obj.id}.{obj.picture_file_path.split(".")[-1]}')
    else:
        return FileResponse(obj.picture_file_path,
                            media_type=f'image/png',
                            filename='none_picture.png')

async def create(self, schema, *args, **kwargs) -> Optional[Create_schema_type]:
    obj = await self.model.create(**schema.dict(exclude_unset=True), **kwargs)
    return await self.get_schema.from_tortoise_orm(obj)

async def update(self, schema, **kwargs) -> Optional[Update_schema_type]:
    await self.model.filter(**kwargs).update(**schema.dict(exclude_unset=True))
    return await self.get_schema.from_queryset_single(self.model.get(**kwargs))

async def delete(self, **kwargs):
    obj = await self.model.filter(**kwargs).delete()
    if not obj:
        raise HTTPException(status_code=404, detail='Object does not exist')
```

Рис. 13 Методы обработки запросов на примере базовых методов обработки запросов класса Service_base

Рис. 14 Визуализация запросов для полей таблицы Artist в Swagger UI

Рис. 15 Визуализация запросов для полей таблицы Album в Swagger UI

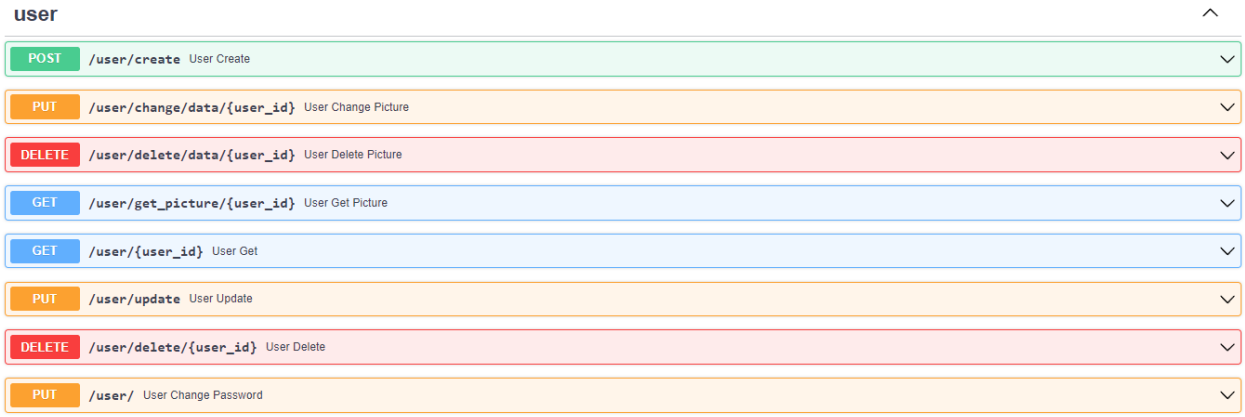

Рис. 16 Визуализация запросов для полей таблицы User в Swagger UI

Рис. 17 Визуализация запросов для полей таблицы Playlist в Swagger UI

Рис. 18 Визуализация запросов для полей таблицы Track в Swagger UI

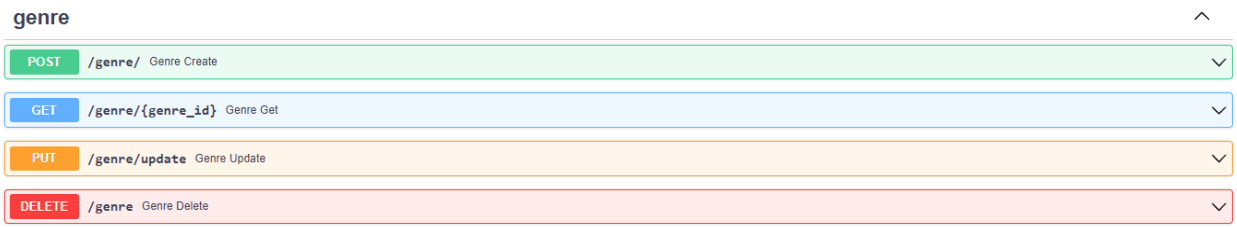

Рис. 19 Визуализация запросов для полей таблицы Genre в Swagger UI

Рис. 20 Визуализация запросов получения страниц приложения в Swagger UI

```python
class Track(BaseModel):
    id: int
    name: str
    duration_s: int
    track_file_path: str
    picture_file_path: str

    @classmethod
    def as_form(cls,
                id: int = Form(...),
                name: str = Form(...),
                duration_s: int = Form(...),
                track_file_path: str = Form(...),
                picture_file_path: str = Form(...)
    ):
        return cls(id=id,
                   name=name,
                   duration_s=duration_s,
                   track_file_path=track_file_path,
                   picture_file_path=picture_file_path)

    class Config:
        orm_mode=True
```

Рис. 21 Пример описания модели тела запроса на примере тела запроса создания поля таблицы Track

```python
class Track_params(BaseModel):
    valence: Optional[float] = None
    acousticness: Optional[float] = None
    danceability: Optional[float] = None
    energy: Optional[float] = None
    explicit: Optional[bool] = None
    instrumentalness: Optional[float] = None
    liveness: Optional[float] = None
    loudness: Optional[float] = None
    speechiness: Optional[float] = None
    tempo: Optional[float] = None

    @classmethod
    def as_form(cls,
                valence: float = Form(None),
                acousticness: float = Form(None),
                danceability: float = Form(None),
                energy: float = Form(None),
                explicit: bool = Form(None),
                instrumentalness: float = Form(None),
                liveness: float = Form(None),
                loudness: float = Form(None),
                speechiness: float = Form(None),
                tempo: float = Form(None)):
            return cls(valence=valence,
                       acousticness=acousticness,
                       danceability=danceability,
                       energy=energy,
                       explicit=explicit,
                       instrumentalness=instrumentalness,
                       liveness=liveness,
                       loudness=loudness,
                       speechiness=speechiness,
                       tempo=tempo)

    class Config:
        orm_mode=True
```

Рис. 22 Модель тела запроса создания поля таблицы Track_params

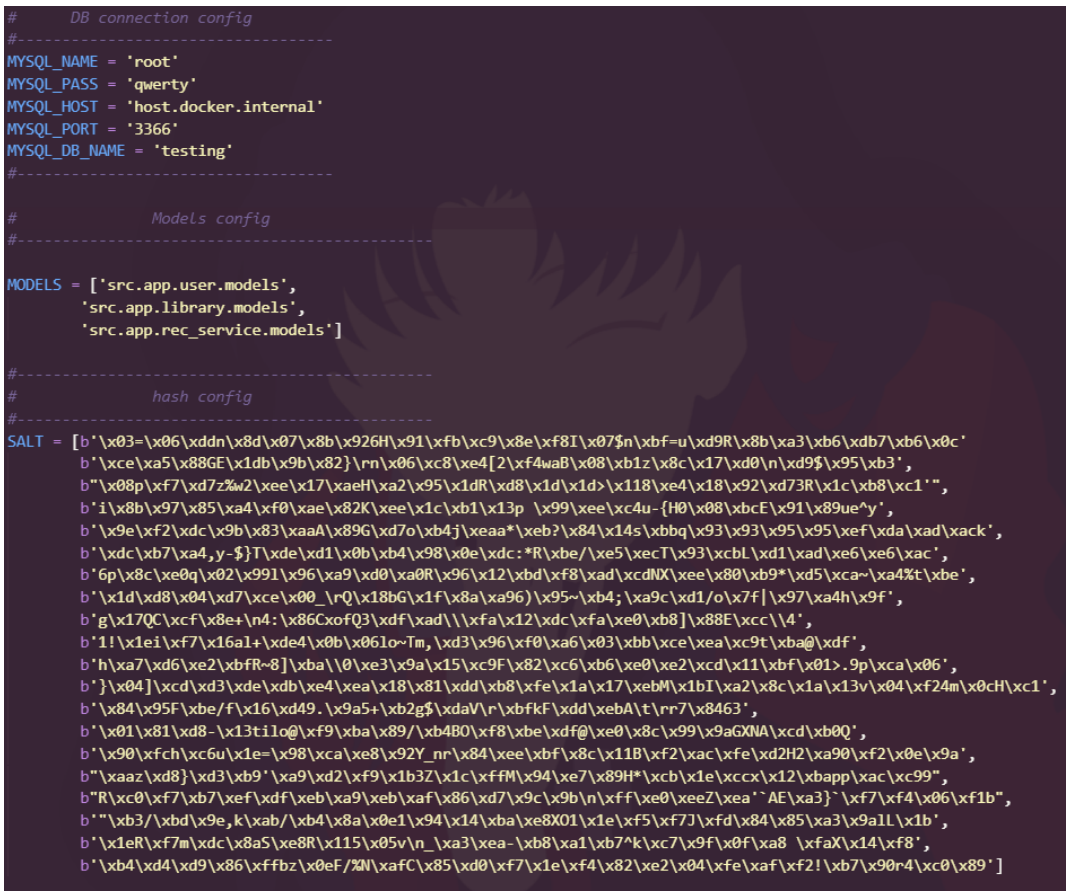

Рис. 23 Содержимое файла config.py

```yaml
version: '3.1'

services:
  db:
    container_name: "db"
    image: mysql:5.7
    restart: always
    environment:
      MYSQL_DATABASE: 'main_db'
      MYSQL_USER: 'cherry4xo'
      MYSQL_PASSWORD: 'qwerty'
      MYSQL_ROOT_PASSWORD: 'qwerty'
    ports:
      - '3366:3306'
    volumes:
      - /var/lib/mysql

  adminer:
    image: adminer
    restart: always
    ports:
      - 8080:8080
```

Рис. 24 Содержимое файла docker-compose.py

```python
api_router.include_router(album_router, prefix='/album', tags=['album'])
api_router.include_router(artist_router, prefix='/artist', tags=['artist'])
api_router.include_router(genre_router, prefix='/genre', tags=['genre'])
api_router.include_router(playlist_router, prefix='/playlist', tags=['playlist'])
api_router.include_router(track_router, prefix='/track', tags=['track'])
api_router.include_router(library_router, prefix='/library', tags=['library'])
api_router.include_router(user_router, prefix='/user', tags=['user'])
api_router.include_router(comment_router, prefix='/comment', tags=['comment'])
api_router.include_router(page_router)
```

Рис. 25 Включение роутеров с запросами таблиц в единый роутер приложения

```python
app = FastAPI(
    title="BMSTU project API",
    description='Author - Kulakov Nikita (AKA cherry4xo)',
    version='0.1.0'
)

app.include_router(router.api_router)

register_tortoise(
    app,
    db_url=f'mysql://{config.MYSQL_NAME}:{config.MYSQL_PASS}@{config.MYSQL_HOST}:{config.MYSQL_PORT}/{config.MYSQL_DB_NAME}',
    modules={'models': config.MODELS},
    generate_schemas=True,
    add_exception_handlers=True
)
```

Рис. 26 Содержимое файла main.py, где производится миграция базы данных и инициализация API

```python
class Track_params(Model):
    id = fields.IntField(pk=True,
                         unique=True,
                         not_null=True,
                         auto_increment=True)
    track_id: fields.ForeignKeyRelation['Track'] = fields.ForeignKeyField('models.Track', related_name='Track_id')
    valence = fields.FloatField()
    acousticness = fields.FloatField()
    danceability = fields.FloatField()
    energy = fields.FloatField()
    explicit = fields.BooleanField()
    instrumentalness = fields.FloatField()
    liveness = fields.FloatField()
    loudness = fields.FloatField()
    speechiness = fields.FloatField()
    tempo = fields.FloatField()

    class PydanticMeta:
        allow_cycles=False

    class Meta:
        table='Track_params'
```

Рис. 27 Модель таблицы Track_params с данными для алгоритма рекомендаций

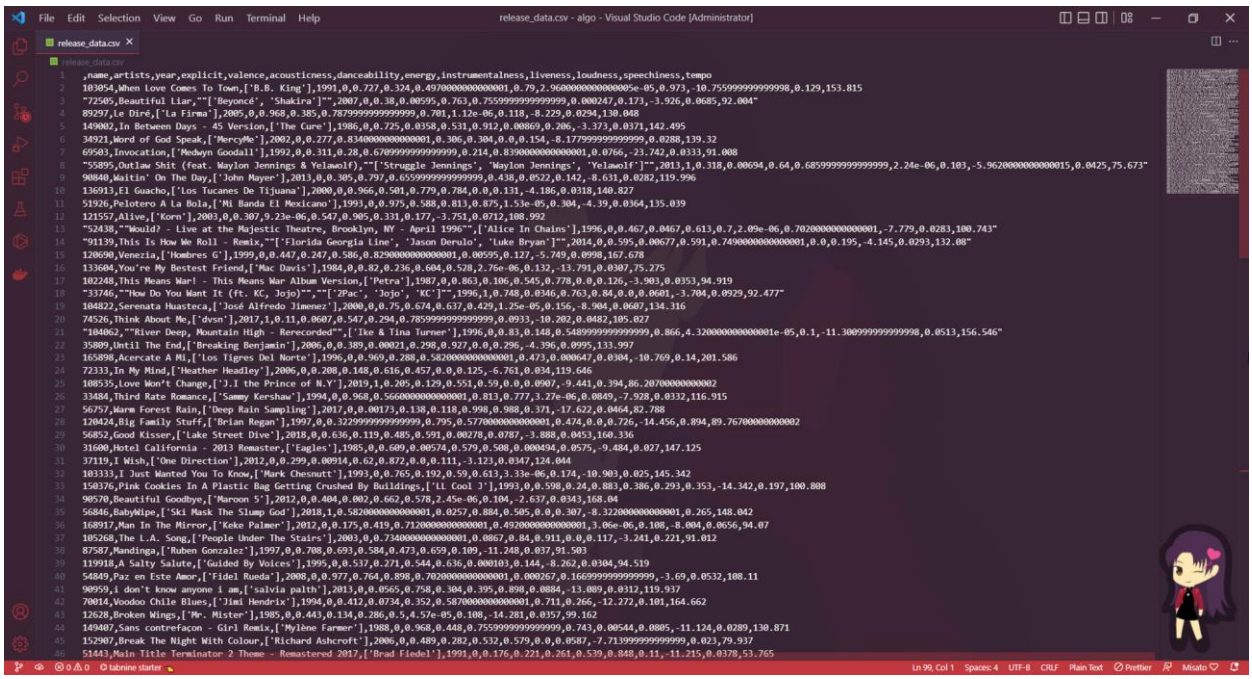

Рис. 28 Файл data.csv с демонстрацией структуры датасета

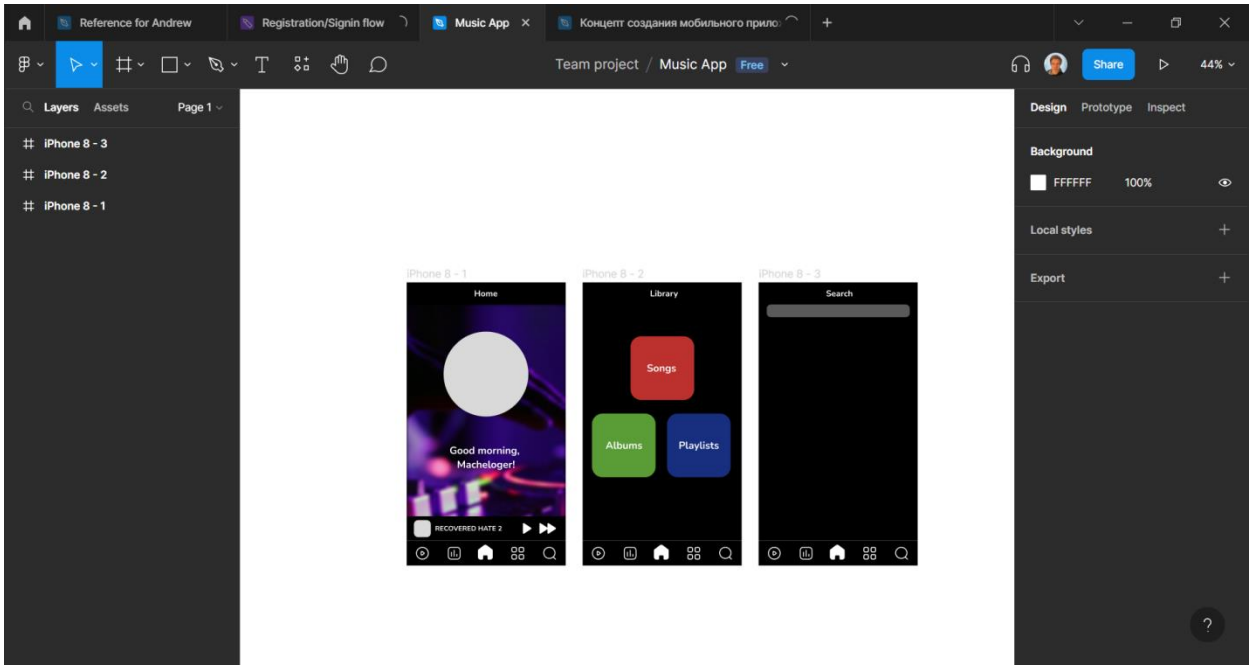

Рис. 28 Создание макета в сервисе Figma

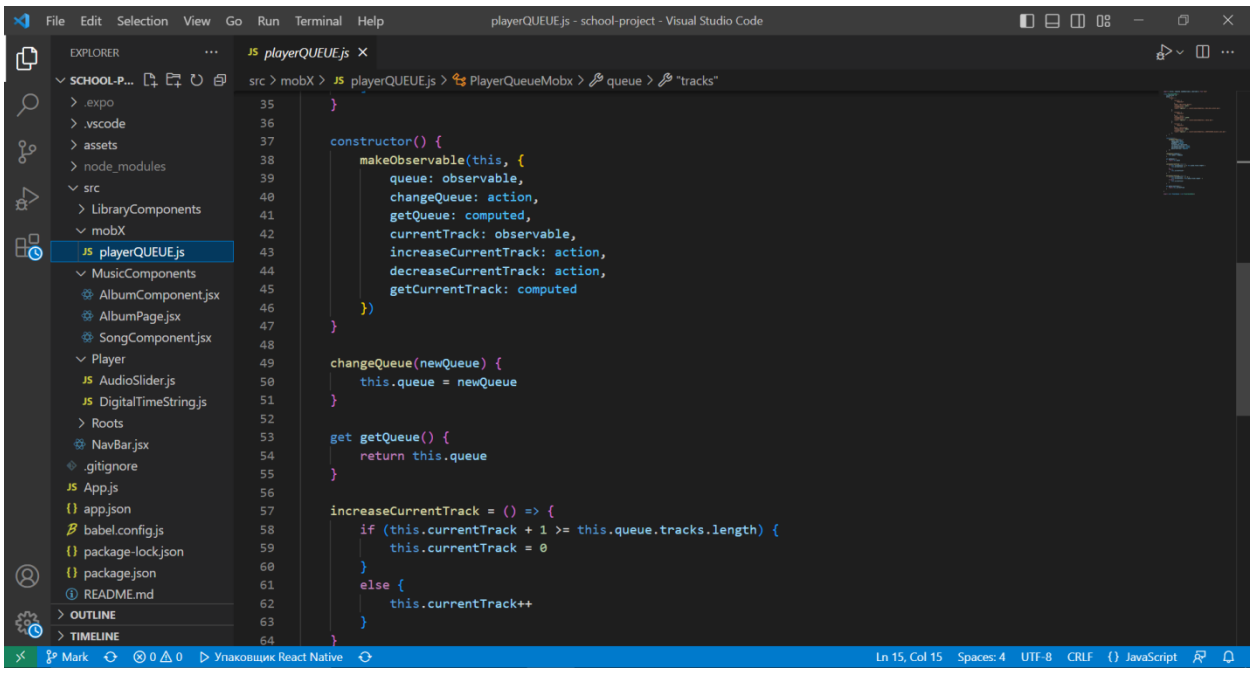

Рис. 30 Использование Visual Studio в создании UI

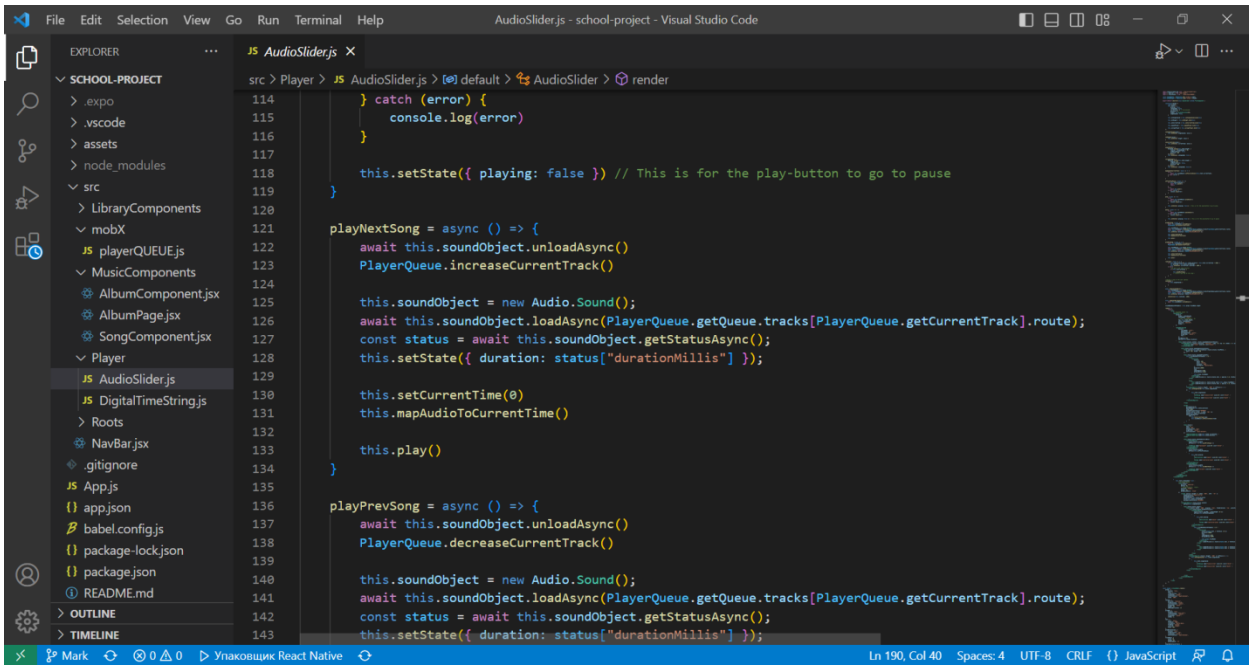

Рис. 31 Использование MobX в качестве хранилища глобальных состояний

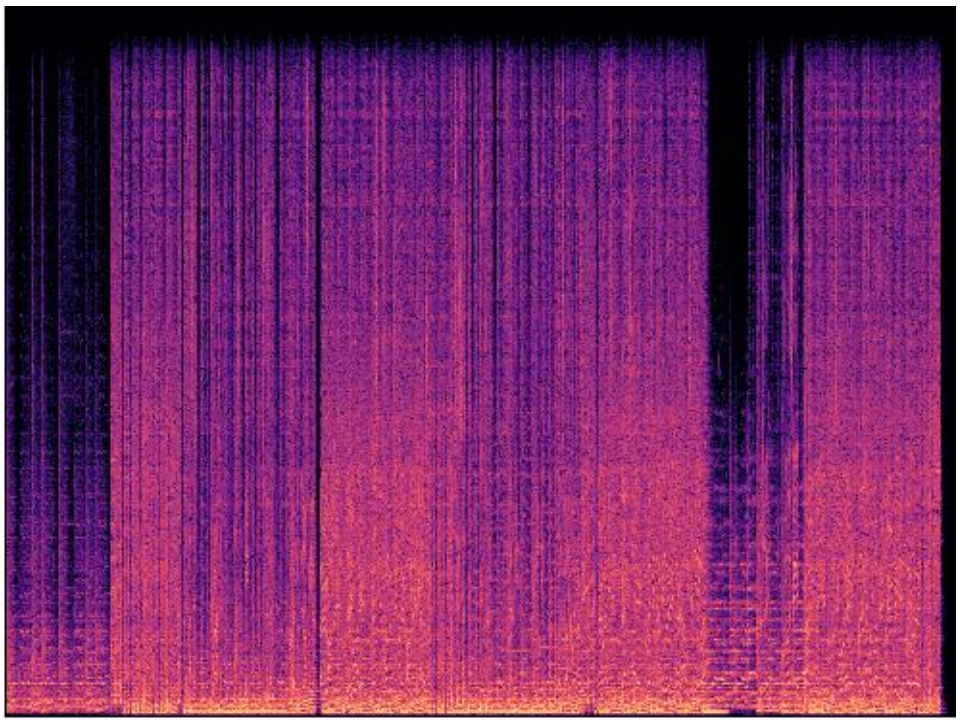

Рис. 32 Мел-Спектрограмма песни Rammstein - Adieu

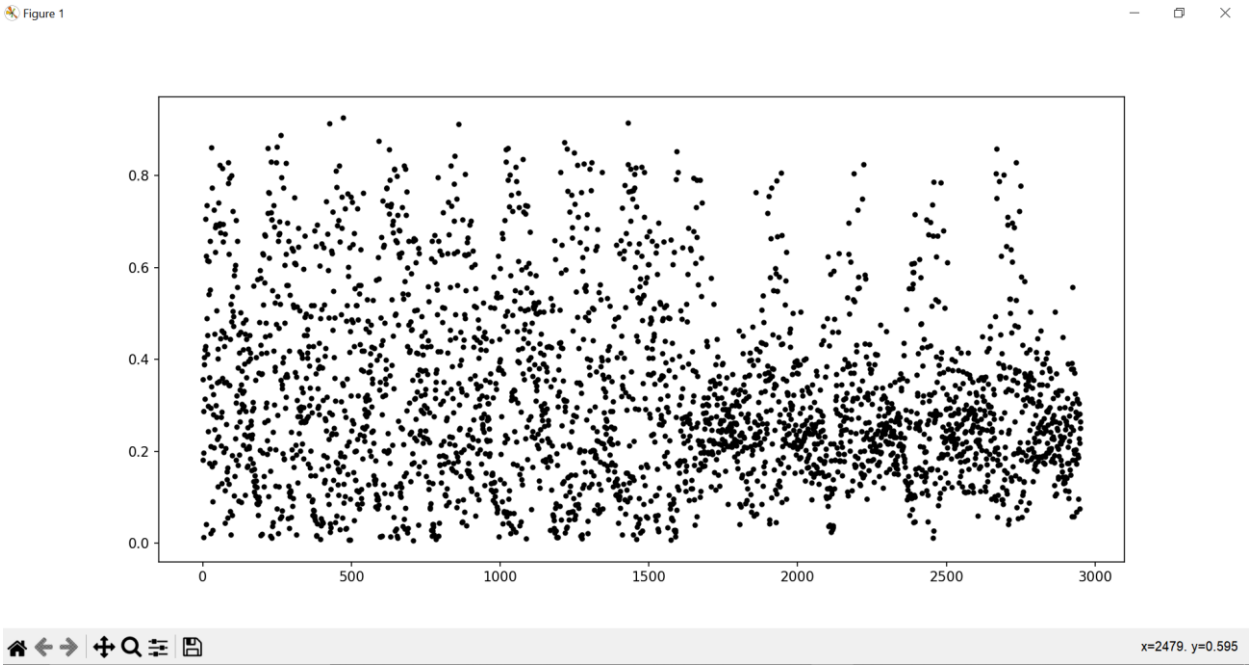

Рис. 33 Массив пиков в 15-ти секундном отрывке песни