МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ  
Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ   
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий

Кафедра информационных систем и технологий

Специальность 1-40 05 01 «Информационные системы и технологии»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА**

по дисциплине «Базы данных»

Тема: «Реализация базы данных для музыкальной площадки с использованием технологии применения мультимедийных типов данных»

**Исполнитель**

студент 2 курса 2 группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. В. Коренчук

подпись, дата

**Руководитель**

ассистент \_\_\_\_\_\_\_\_\_\_\_\_ А.В. Харланович

(должность, уч. звание) (подпись, дата)

Допущен(а) к защите \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_

дата, подпись

Курсовой проект защищен с оценкой

Руководитель \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.В. Харланович

подпись дата инициалы и фамилия

Минск 2023

**Содержание**

[Введение 5](#_Toc135051384)

[1 Анализ требований к программному средству 6](#_Toc135051385)

[1.1 Аналитический обзор аналогов 6](#_Toc135051386)

[1.1.1 Аналог Spotify 6](#_Toc135051387)

[1.1.2 Аналог Apple Music 7](#_Toc135051388)

[1.2 Разработка функциональных требований, определение вариантов использования 8](#_Toc135051391)

[1.3 Вывод 10](#_Toc135051393)

[2 Разработка архитектуры проекта 11](#_Toc135051394)

[2.1 Обобщенная структура управлением приложения 11](#_Toc135051395)

[2.2 Диаграммы UML, взаимосвязь всех компонентов. 11](#_Toc135051396)

[2.3 Описание информационных объектов 12](#_Toc135051397)

[2.4 Вывод 13](#_Toc135051398)

[3 Разработка модели базы данных 14](#_Toc135051399)

[3.1 Создание необходимых объектов 14](#_Toc135051400)

[3.1.1 Представления базы данных 15](#_Toc135051401)

[3.1.2 Индексы базы данных 15](#_Toc135051402)

[3.1.3 Триггеры базы данных 16](#_Toc135051403)

[3.2 Описание используемой технологии 17](#_Toc135051404)

[3.3 Вывод 18](#_Toc135051405)

[4 Установка, настройка и использование PosgtreSQL 14.5 20](#_Toc135051406)

[4.1 Установка PostgreSQL 20](#_Toc135051407)

[4.2 Создание таблиц 20](#_Toc135051408)

[4.3 Создание ролей для разграничения 21](#_Toc135051409)

[4.4 Создание пакетов процедур для базы данных 23](#_Toc135051410)

[4.4.1 Выборка данных из таблиц 24](#_Toc135051411)

[4.4.2 Выборка данных по поисковому запросу 25](#_Toc135051412)

[4.4.3 Заполнение таблиц 100 000 строк 27](#_Toc135051413)

[4.4.4 Добавление данных в таблицы 27](#_Toc135051414)

[4.4.5 Удаление данных в таблицы 29](#_Toc135051415)

[4.4.6 Изменение данных в таблицы 29](#_Toc135051416)

[4.4.7 Дополнительные функции 30](#_Toc135051417)

[4.5 Описание процедур экспорта и импорта 32](#_Toc135051418)

[4.6 Вывод 35](#_Toc135051419)

[5 Тестирование 36](#_Toc135051420)

[5.1 Тестирование производительности базы данных 36](#_Toc135051421)

[5.2 Вывод 37](#_Toc135051422)

[6 Руководство по использованию программного средства 38](#_Toc135051423)

[6.1 Руководство пользователя 38](#_Toc135051424)

[6.2 Установка приложения 43](#_Toc135051425)

[6.3 Вывод 43](#_Toc135051426)

[Заключение 44](#_Toc135051427)

[Список литературных источников 45](#_Toc135051428)

[Приложение А 46](#_Toc135051429)



# Введение

Цель данной работы заключается в создании реляционной базы данных для музыкальной платформы, которая обеспечивает пользователя доступом к имеющимся на ней музыкальным композициям. В рамках работы также необходимо разработать соответствующее приложение, которое будет демонстрировать функциональность базы данных и обеспечивать клиентов доступом к музыкальным трекам.

База данных - это организованное собрание данных, которое обычно хранится в электронном виде в компьютерной системе. БД используются для хранения, организации и управления большим объемом структурированных и неструктурированных данных. Реляционная база данных является наиболее распространенной формой организации данных, в которой данные представлены в виде таблиц, состоящих из строк и столбцов, где каждый столбец представляет атрибут, а каждая строка представляет кортеж или запись. В данной работе для управления базой данных была выбрана СУБД Postgres SQL, поскольку эта система обладает высокой надежностью и производительностью, что позволяет обеспечить эффективное хранение, обработку и управление музыкальными данными.

Также необходимо разработать приложение для демонстрации функциональности базы данных и взаимодействия с ней. Приложение было реализовано с использованием языка программирования Node.js и фреймворка React с TypeScript.

Для гарантированной безопасности пользователей приложения в курсовой работе применяется метод шифрования паролей перед их сохранением в базу данных. Также для обеспечения функциональности приложения используются мультимедийные форматы данных при сохранении аудио и картинок.

Основные требования к приложению:

* Реализация ролей администратора и пользователя;
* поиск аудиозаписей по альбому, исполнителю или жанру;
* загрузка аудиозаписей на платформу пользователем;
* взаимодействие с базой данных при помощи хранимых процедур и функций.

В пояснительной записке содержится информация о сопоставимых продуктах, структуре и реализации проекта, а также инструкции по использованию приложения.

1. Анализ требований к программному средству

## Аналитический обзор аналогов

Музыка - это одна из самых популярных и распространенных форм искусства, которая имеет давнюю историю и оказывает значительное влияние на культуру и общество в целом. В настоящее время музыкальные платформы являются важной частью музыкальной индустрии, обеспечивая людям доступ к огромной библиотеке музыкальных произведений различных жанров и эпох.

Одним из основных преимуществ музыкальных платформ является возможность слушать музыку в любое время и в любом месте с помощью смартфона, компьютера или другого устройства с доступом в Интернет. Это делает музыкальные платформы очень популярными среди миллионов пользователей по всему миру.

### 1.1.1 Аналог Spotify

Spotify [1] - это популярная музыкальная платформа, которая позволяет пользователям стримить музыку онлайн, создавать плейлисты и делиться ими с друзьями, слушать радио, а также находить новую музыку, основываясь на своих предпочтениях. Пример интерфейса данного сервиса представлен на рисунке 1.1.

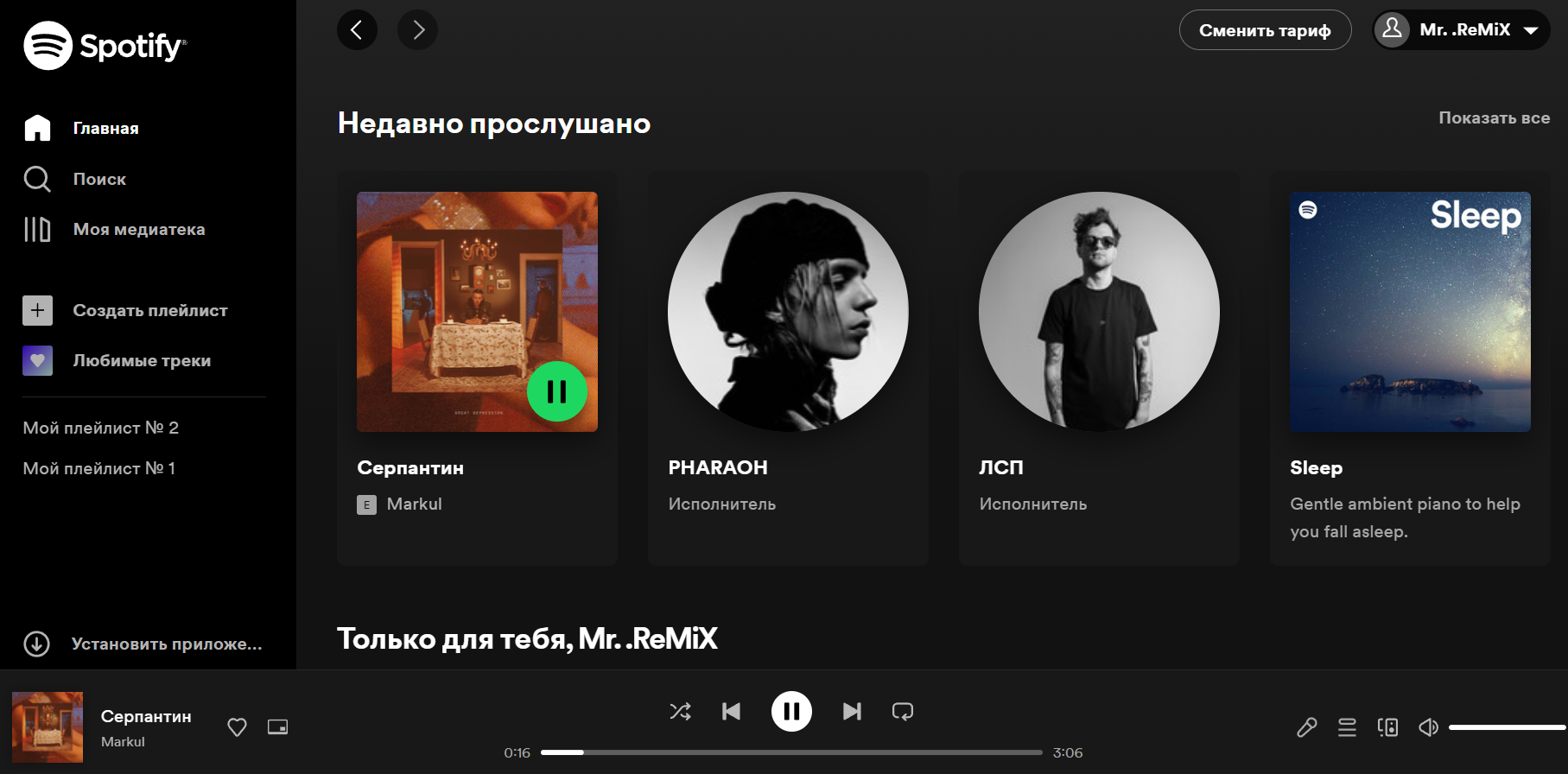


Рисунок 1.1 – Интерфейс сервиса Spotify

Одной из основных функций Spotify является поиск музыки. Пользователи могут искать музыку по альбому, исполнителю, жанру или песне. Кроме того, Spotify предлагает персонализированные рекомендации в соответствии с предпочтениями пользователя, а также плейлисты, созданные другими пользователями и кураторами платформы.

Пользователи могут создавать свои собственные плейлисты и делиться ими с друзьями или сохранять музыку для офлайн-воспроизведения. Spotify также позволяет пользователям слушать радио и подкасты, включая оригинальный контент, созданный самой платформой.

Другая функция Spotify - это возможность использовать платформу как социальную сеть, где пользователи могут подписываться на друг друга, просматривать их плейлисты и рекомендации, а также обмениваться сообщениями.

Spotify также предлагает два варианта подписки: бесплатную и платную. Бесплатная версия содержит рекламу и ограничения в использовании, в то время как платная версия позволяет получить неограниченный доступ к музыке, отсутствие рекламы и другие функции.

Spotify использует несколько баз данных в своей архитектуре, включая Cassandra и Apache Kafka. Cassandra - это высокомасштабируемая NoSQL база данных, которая используется для хранения метаданных треков, включая информацию об альбомах, исполнителях, жанрах и т.д. Cassandra также используется для хранения информации о пользовательских аккаунтах и настроек. Apache Kafka - это распределенная платформа потоковой обработки данных, которая используется для передачи и обработки потоковых данных между различными компонентами Spotify, такими как служба рекомендаций, система поиска и т.д.

В общем, Spotify использует масштабируемые и высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему музыкальных данных и обработки потоковых данных для предоставления персонализированных рекомендаций и других функций платформы.

### 1.1.2 Аналог Apple Music

Apple Music [2] - это популярная музыкальная платформа, которая позволяет пользователям стримить музыку онлайн. Apple Music позволяет пользователям прослушивать музыку на различных устройствах, а также создавать собственные плейлисты и подборки.

Apple Music предлагает пользователю возможность доступа к более чем 75 миллионам песен, а также к подкастам и аудиокнигам. Сервис также предлагает персонализированные рекомендации, основанные на предпочтениях пользователя.

Пользователи могут создавать свои плейлисты и делиться ими с другими пользователями. Apple Music также предлагает радиостанции, созданные известными диджеями и музыкантами, а также живые трансляции концертов и мероприятий.

Интерфейс Apple Music схож с интерфейсом Spotify и предлагает удобный поиск по альбомам, песням и исполнителям. Также сервис позволяет скачивать музыку для офлайн прослушивания. Пример интерфейса данного сервиса представлен на рисунке 1.2.

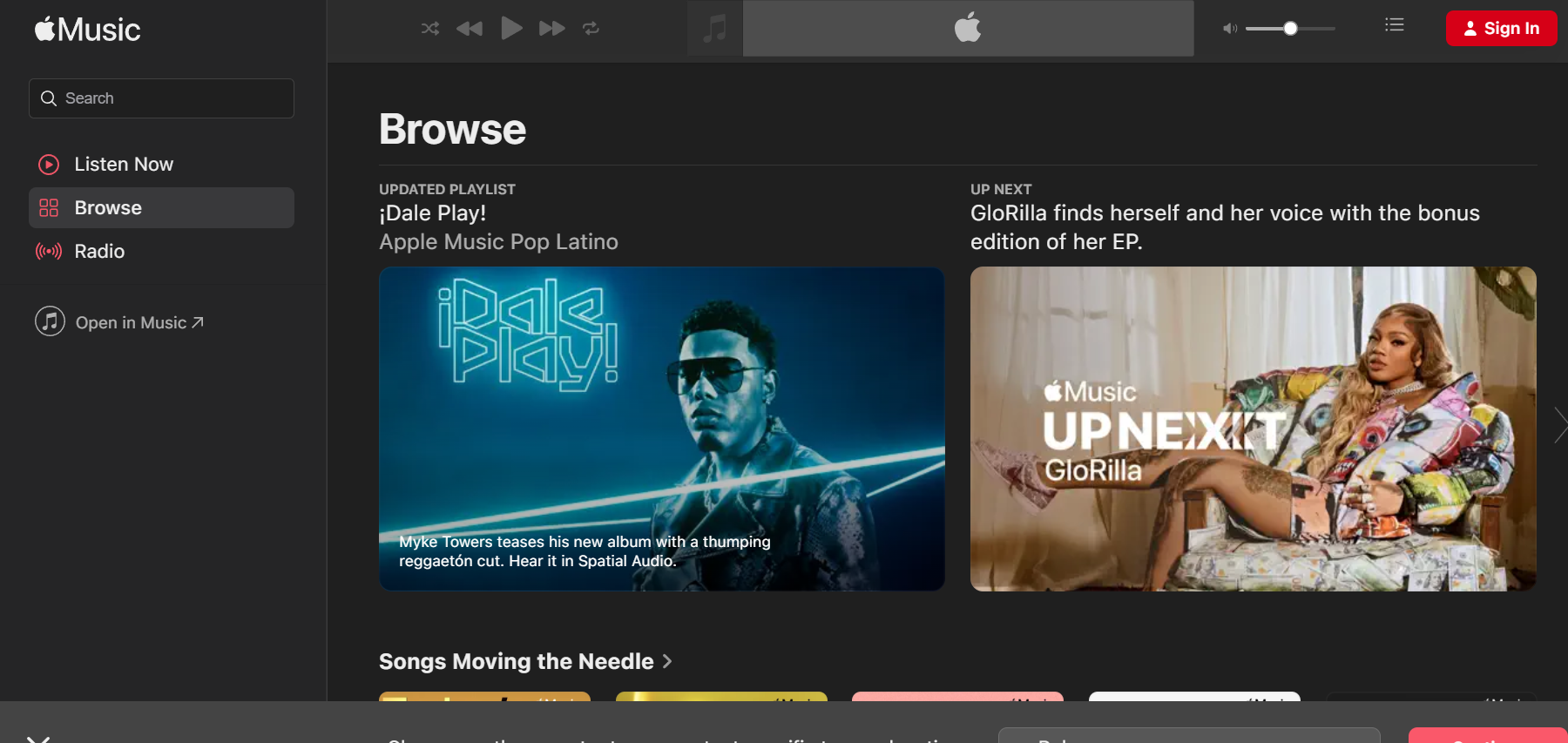


Рисунок 1.2 – Интерфейс сервиса Apple Music

## Apple Music использует несколько баз данных в своей архитектуре, включая MySQL и Cassandra. MySQL - это реляционная база данных, которая используется для хранения пользовательской информации, такой как логины и пароли, а также информации о подписке и оплате. Cassandra также используется для хранения метаданных треков, включая информацию об альбомах, исполнителях, жанрах и т.д., а также для предоставления быстрого доступа к этой информации пользователям. Кроме того, Apple Music использует инфраструктуру Amazon Web Services для обеспечения высокой доступности и масштабируемости своей платформы.

## В целом, Apple Music также использует масштабируемые и высокопроизводительные базы данных для обеспечения быстрого доступа к большому объему музыкальных данных и управления информацией о пользователях и их подписках.

## 1.2 Разработка функциональных требований, определение вариантов использования

Функциональные требования базы данных определяют, как база данных должна обрабатывать данные и предоставлять пользовательскому интерфейсу необходимую функциональность. Это может включать в себя описание того, как данные должны храниться и организовываться, как происходит поиск и выборка данных, каким образом обновляются данные и какие механизмы используются для защиты данных. Кроме того, функциональные требования могут определять интеграцию базы данных с другими системами и программами. Например, для музыкальной площадки функциональные требования могут включать в себя функции для хранения информации о музыкальных треках и пользователях, поиска музыки по категориям и критериям, создания и управления плейлистами, а также функции для оценки и прослушивания музыки.

Помимо функциональных требований, важно также определить роли пользователей и их варианты использования системы. Варианты использования описывают, как пользователи будут взаимодействовать с системой в зависимости от своих ролей. Это помогает определить, какие функции должны быть доступны для каждой роли, какие данные должны быть доступны для каждой роли, а также как должна быть организована навигация в системе. Варианты использования обычно представляются в виде UML диаграмм, которые позволяют наглядно отобразить взаимодействие между пользователями и системой.

Роли пользователя — это набор прав, которые пользователь может получить в системе. В зависимости от роли пользователя, он может иметь доступ к различным функциям системы. В данном проекте роли пользователей будут следующими:

* Guest.
* User.
* Manager.

На основе предоставленного списка ролей необходимо построить варианты использование. Варианты использование изображена на рисунке 1.3.

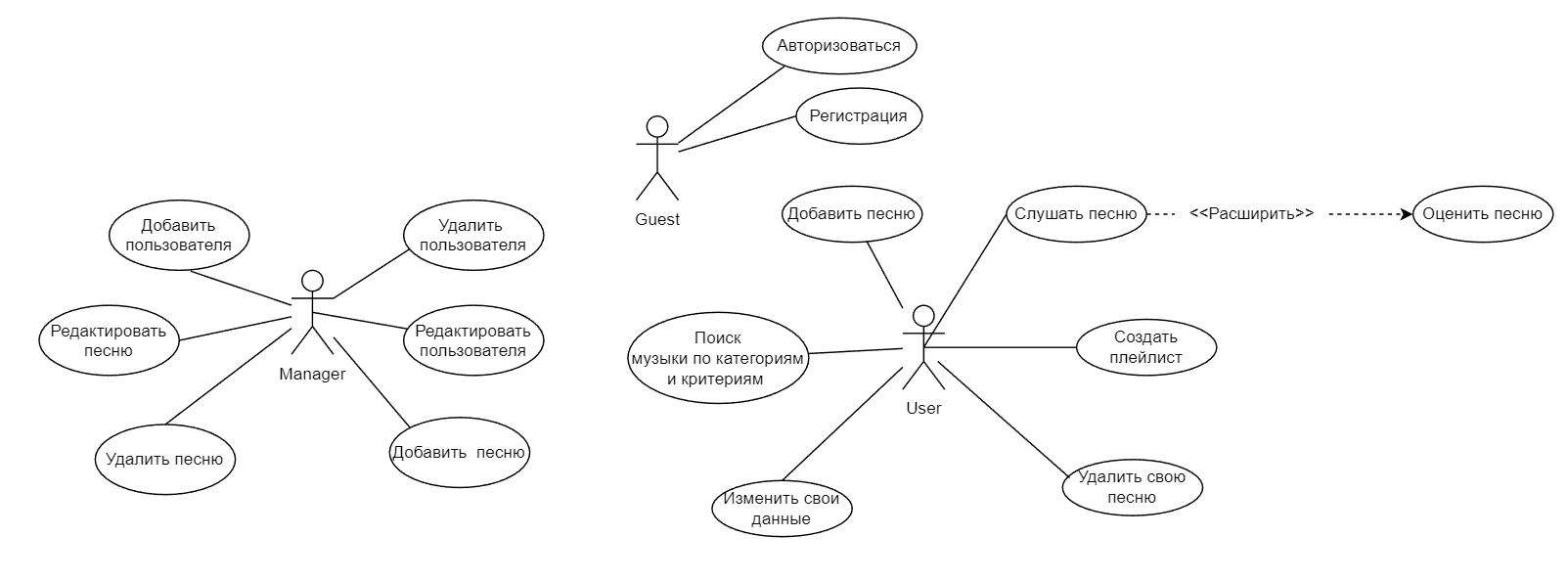


Рисунок 1.3 – UML диаграмма вариантов использования

В начале работы с приложением пользователь является гостем. Ему будет доступна только просмотр всей информации без возможности слушать музыку. После регистрации пользователь становится пользователем User.

Роль User получает возможность слушать музыку, добавлять понравившиеся треки в свой персональный плейлист, оценивать музыкальные треки и осуществлять поиск музыки по категориям и критериям. Кроме того, User может добавлять в свою библиотеку любимые треки в соответствии с настроением или жанром.

Роль Manager заключается в управлении площадкой и мониторинге действий пользователей. Администратор имеет доступ к просмотру информации о всех пользователях, в том числе их действиях на площадке. При необходимости администратор может редактировать информацию о треках и пользователях, например, изменять категории и теги у треков, а также изменять личную информацию о пользователях.

## В данном разделе были определены роли пользователей и разработаны варианты использования системы в зависимости от этих ролей. Гость может только просматривать информацию, пользователь получает доступ к функциям системы, таким как прослушивание музыки, добавление треков в плейлист и поиск музыки, а администратор имеет права на управление площадкой и редактирование информации о пользователях и треках.

## 1.3 Вывод

Итого, был проведен аналитический обзор аналогов музыкальных платформ и сервисов, которые уже существуют на рынке. Этот обзор позволил определить основные характеристики и функциональные возможности, которые необходимо предусмотреть в разрабатываемой системе. Также были определены функциональные требования базы данных, а также роли пользователей и варианты использования системы в зависимости от этих ролей. Была разработана UML-диаграмма, на которой отображены основные функции, которые доступны для каждой из ролей пользователей.

1. Разработка архитектуры проекта

## 2.1 Обобщенная структура управлением приложения

Для обеспечения управления приложением с использованием базы данных необходимо разработать удобный и интуитивно понятный интерфейс, который позволит пользователю взаимодействовать с базой данных и эффективно управлять данными. Это может включать в себя разработку оптимизированных запросов для вставки, обновления и удаления данных, а также разработку механизмов для извлечения и обработки информации из базы данных.

В функциональность приложения для прослушивания музыки может входить возможность быстрого и удобного поиска музыкальных композиций, исполнителей и альбомов по различным параметрам, таким как жанр, год выпуска, рейтинг и т.д.

## 2.2 Диаграммы UML, взаимосвязь всех компонентов.

Диаграмма базы данных таблиц (Database Table Diagram) - это визуальное представление структуры базы данных и отношений между таблицами, которые хранятся в этой базе данных. Диаграмма базы данных будет представлена на рисунке 2.1.

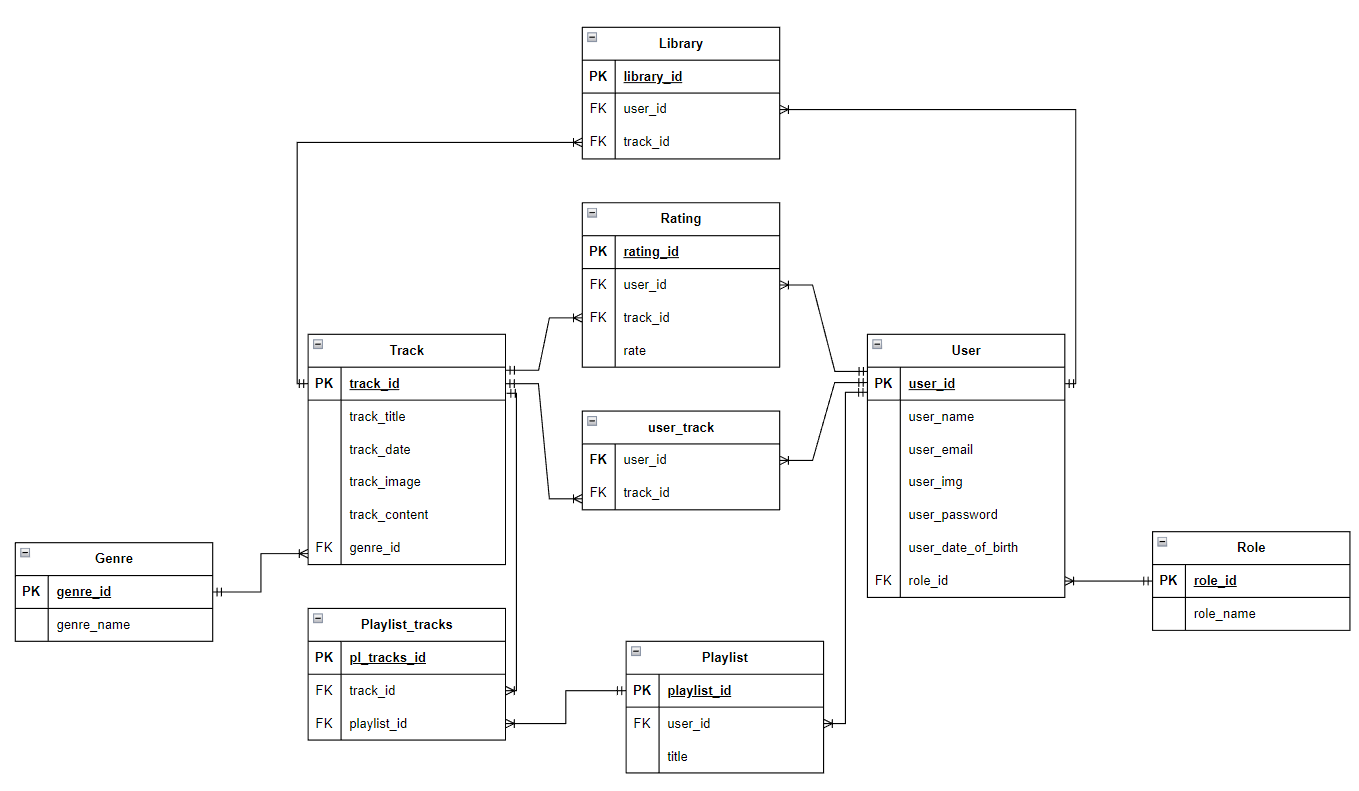


Рисунок 2.1 – Диаграмма базы данных

Таким образом, диаграмма показывает связи между таблицами и полями, а также отношения между ними, такие как связи "один-ко-многим", "многие-ко-многим". Например, таблица Users связана с таблицами Rating, Playlist и Library\_user через внешние ключи user\_id. Также видно, что таблицы Track, Rating, Playlist\_tracks и Library\_track связаны с таблицей Users, а таблицы User\_track, Track и Genre связаны друг с другом через внешние ключи.

## 2.3 Описание информационных объектов

Для реализации базы данных было разработано 9 таблиц. В структуру схемы базы данных для проекта входят следующие таблицы: Role, Users, Genre, User\_track, Track, Rating, Playlist, Playlist\_tracks, Library\_track. Ниже будет описание про каждую из них более подробно.

Таблица Role представляет собой перечень ролей, которые могут иметь пользователи. Имеющиеся поля: role\_id (идентификатор роли, типа SERIAL, первичный ключ), role\_name (текстовое название роли, тип VARCHAR(255)).

Таблица Users содержит информацию о пользователях. Имеющиеся поля: user\_id (идентификатор пользователя, типа SERIAL, первичный ключ), user\_name (имя пользователя, тип VARCHAR(255)), user\_img (изображение пользователя, тип BYTEA), user\_email (адрес электронной почты пользователя, тип VARCHAR(255)), user\_password (пароль пользователя, тип VARCHAR(255)), user\_date\_of\_birth (дата рождения пользователя, тип DATE), user\_role\_id (идентификатор роли пользователя, тип INTEGER, внешний ключ ссылающийся на role\_id в таблице Role).

Таблица Genre содержит информацию о жанрах музыки. Имеющиеся поля: genre\_id (идентификатор жанра, типа SERIAL, первичный ключ), genre\_name (название жанра, тип VARCHAR(255)).

Таблица User\_track содержит информацию об исполнителях. Имеющиеся поля: track\_id (идентификатор трека, тип INTEGER, внешний ключ ссылающийся на track\_id в таблице Track), user\_id (идентификатор пользователя-владельца, тип INTEGER, внешний ключ ссылающийся на user\_id в таблице Users).

Таблица Track содержит информацию о треках. Имеющиеся поля: track\_id (идентификатор трека, типа SERIAL, первичный ключ), track\_title (название трека, тип VARCHAR(255)), track\_date (дата создания трека, тип DATE, track\_image (изображение трека, тип BYTEA), track\_content (содержание трека, тип BYTEA), genre\_id (идентификатор жанра, тип INTEGER, внешний ключ ссылающийся на genre\_id в таблице Genre).

Таблица Rating содержит информацию о рейтинге пользователей для треков. Имеющиеся поля: rating\_id (идентификатор рейтинга, типа SERIAL, первичный ключ), user\_id (идентификатор пользователя, который оценил трек, тип INTEGER, внешний ключ ссылающийся на user\_id в таблице Users), track\_id (идентификатор трека, который был оценен, тип INTEGER, внешний ключ ссылающийся на track\_id в таблице Track), rate (оценка, которую поставил пользователь, тип INTEGER).

Таблица Playlist содержит информацию о плейлистах пользователей. Имеющиеся поля: playlist\_id (идентификатор плейлиста, типа SERIAL, первичный ключ), user\_id (идентификикатор пользователя, создавшего плейлист, тип INTEGER, внешний ключ ссылающийся на user\_id в таблице Users) и title (название плейлиста, тип VARCHAR(255) NOT NULL).

Таблица Playlist\_tracks связывает плейлисты и треки. Имеющиеся поля: id (идентификатор записи, типа SERIAL, первичный ключ), track\_id (идентификатор трека, тип INTEGER, внешний ключ ссылающийся на track\_id в таблице Track) и playlist\_id (идентификатор плейлиста, тип INTEGER, внешний ключ ссылающийся на playlist\_id в таблице Playlist).

Таблица Library\_track содержит информацию о треках, которые добавлены в библиотеку пользователей. Имеющиеся поля: id (идентификатор записи, типа SERIAL, первичный ключ), user\_id (идентификатор пользователя, который добавил трек в свою библиотеку, тип INTEGER, внешний ключ ссылающийся на user\_id в таблице Users) и track\_id (идентификатор трека, тип INTEGER, внешний ключ ссылающийся на track\_id в таблице Track).

## 2.4 Вывод

Разработка архитектуры проекта необходима для определения структуры и функциональности приложения. Обобщенная структура управления приложения позволяет определить, какие компоненты необходимы для реализации приложения и как они должны взаимодействовать между собой.

Описание информационных объектов является важной частью архитектуры проекта, так как это помогает понять, какие данные будут использоваться в приложении, и как они будут храниться и обрабатываться. В данном проекте были описаны объекты, такие как пользователи, роли, жанры, артисты, треки, рейтинги, плейлисты и библиотека пользователя.

В целом, разработка архитектуры проекта является важным шагом в процессе разработки приложения, так как это позволяет определить необходимые компоненты и информационные объекты, которые помогут создать функциональное и эффективное приложение.

3 Разработка модели базы данных

## 3.1 Создание необходимых объектов

Для музыкальной площадки необходимо создать несколько таблиц, которые будут хранить данные о пользователях, жанрах, аудиофайлах, плейлистах и прослушиваниях.

Для музыкальной площадки необходимо создать следующие таблицы:

Role - таблица, содержащая роли пользователей. В этой таблице будут храниться записи с ролями, такими как администратор, модератор, пользователь и т.д.

Users - таблица, содержащая данные о пользователе. В этой таблице будут храниться информация о пользователе, такая как имя, фотография, электронная почта, дата рождения и т.д.

Genre - таблица, содержащая жанры музыки. В этой таблице будут храниться записи о жанрах музыки, такие как рок, поп, рэп и т.д.

Track - таблица, содержащая информацию о музыкальных треках. В этой таблице будут храниться данные о музыкальных треках, такие как название, дата выпуска, изображение обложки, файл с музыкой и т.д.

User\_Track - таблица, содержащая отношение между пользователем и музыкальными треками. В этой таблице будут храниться записи, связывающие пользователя с его любимыми треками.

Rating - таблица, содержащая информацию об оценках пользователей музыкальных треков. В этой таблице будут храниться данные об оценках, которые пользователи выставляют трекам.

Playlist - таблица, содержащая информацию о плейлистах пользователей. В этой таблице будут храниться данные о плейлистах, такие как название и пользователь, создавший плейлист.

Playlist\_tracks - таблица, содержащая отношение между плейлистом и музыкальными треками. В этой таблице будут храниться записи, связывающие плейлист с его треками.

Library\_user - таблица, содержащая информацию о библиотеке музыки пользователя. В этой таблице будут храниться записи, связывающие пользователя с его музыкальной библиотекой.

Для эффективного использования базы данных в проекте, необходимо создать индексы на столбцах, используемых в запросах с поиском данных. Например, можно создать индексы на столбцах user\_email в таблице Users, чтобы быстро находить информацию о пользователе по его электронной почте.

Для более удобной работы с базой данных можно создать несколько функций и процедур. Например, можно создать функцию для поиска треков по заданным критериям, таким как жанр, название и т.д. Также можно создать процедуру для добавления новых треков в базу данных.

### 3.1.1 Представления базы данных

Представление (view) в базе данных представляет собой виртуальную таблицу, которая создается на основе запроса к одной или нескольким таблицам в базе данных. Представления позволяют обращаться к данным из нескольких таблиц одновременно, при этом не изменяя структуру этих таблиц.

В данном проекте были созданы три представления:

* all\_info\_user, которое объединяет данные таблиц Users и Role;
* all\_info\_track, которое содержит информацию о треках и среднюю оценку каждого трека;
* playlist\_tracks\_info, которое содержит информацию о треках, входящих в определенный плейлист, и их среднюю оценку.

Представление all\_info\_user было создано для того, чтобы получить полную информацию о пользователях и их ролях, объединив данные из двух таблиц. Оно будет на листинге 3.1. Остальные представления будут аналогичны, только будут работать с другими таблицами.

|  |
| --- |
| CREATE OR REPLACE VIEW ALL\_INFO\_USER AS  SELECT USERS.USER\_ID, USER\_NAME, USER\_IMG,  USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, ROLE.ROLE\_NAME  FROM USERS  JOIN ROLE ON USERS.USER\_ROLE\_ID = ROLE.ROLE\_ID;  DROP VIEW ALL\_INFO\_USER; |

Листинг 3.1 – Представление all\_info\_track

Представление all\_info\_track было создано для того, чтобы получить информацию о треках и их средней оценке, объединив данные из таблицы Track и вызова функции average\_rating(), которая рассчитывает среднюю оценку каждого трека на основе данных из таблицы Rating.

Представление playlist\_tracks\_info было создано для того, чтобы получить информацию о треках, входящих в определенный плейлист, и их средней оценке, объединив данные из таблиц Playlist\_tracks, Track и вызова функции average\_rating().

### 3.1.2 Индексы базы данных

Индекс — объект базы данных, который используется для ускорения поиска данных. В случае большого количества строк в таблице, последовательный поиск данных может занимать много времени. Индекс формируется на основе значений одного или нескольких столбцов таблицы и указателей на соответствующие строки таблицы, что позволяет быстро искать строки, удовлетворяющие заданному критерию поиска. Использование индексов ускоряет работу с базой данных, потому что они имеют оптимизированную структуру для поиска, например, сбалансированное дерево.

Для того, чтобы быстро находить треки по их названию, были созданы два индекса: IDX\_TITLE на таблице TRACK по полю TRACK\_TITLE и idx\_user\_name на таблице USERS по полю USER\_NAME. Сами индексы представлены на листинге 3.2.

|  |
| --- |
| CREATE INDEX IDX\_TITLE ON TRACK(TRACK\_TITLE);  CREATE INDEX IDX\_USER\_NAME ON USERS(USER\_NAME);  CREATE INDEX IDX\_USER\_ID ON USERS(USER\_ID);  CREATE INDEX IDX\_GENRE\_ID ON GENRE(GENRE\_ID);  CREATE INDEX IDX\_PLAYLIST\_USER\_ID ON PLAYLIST(USER\_ID); |

Листинг 3.2 – Индексы базы данных

Кроме индексов IDX\_TITLE и idx\_user\_name, в базе данных также присутствуют индексы на полях USER\_ID, GENRE\_ID и USER\_ID таблиц USERS, GENRE и PLAYLIST соответственно.

Индекс IDX\_USER\_ID на таблице USERS был создан для ускорения поиска пользователя по его ID.

Индекс IDX\_GENRE\_ID на таблице GENRE может использоваться для быстрого поиска жанров по их ID.

Индекс IDX\_PLAYLIST\_USER\_ID на таблице PLAYLIST был создан для ускорения поиска плейлистов пользователя по его ID.

В целом, использование индексов позволяет существенно ускорить операции поиска, сортировки и фильтрации данных в базе данных, особенно в случае большого объема данных. Однако создание индексов может занять дополнительное время при добавлении или изменении данных в таблицах, поэтому необходимо сбалансировать количество и тип индексов для оптимальной производительности базы данных.

### 3.1.3 Триггеры базы данных

Триггер базы данных — это объект базы данных, который выполняет некоторое действие автоматически при определенных событиях в таблице или представлении базы данных. Триггер может быть запрограммирован на срабатывание при вставке, обновлении или удалении строк в таблице.

Триггеры используются для обеспечения целостности данных и контроля доступа к данным, а также для автоматической обработки данных при выполнении определенных операций в таблице.

Первый триггер, созданный в таблице Users, проверяет, что дата рождения нового пользователя не является будущей датой. Если дата рождения нового пользователя больше или равна текущей дате, то триггер генерирует исключение с сообщением об ошибке "Invalid date of birth". Триггер создается с помощью функции check\_date\_of\_birth(), которая возвращает значение NEW (новые значения строк в таблице) при выполнении вставки или обновления. Скрипт триггера будет представлен на листинге 3.3.

|  |
| --- |
| CREATE OR REPLACE FUNCTION CHECK\_DATE\_OF\_BIRTH() RETURNS TRIGGER AS $$  BEGIN  IF NEW.USER\_DATE\_OF\_BIRTH >= CURRENT\_DATE THEN  RAISE EXCEPTION 'INVALID DATE OF BIRTH';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_DATE\_OF\_BIRTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_DATE\_OF\_BIRTH(); |

Листинг 3.3 – Скрипт триггера trigger\_check\_date\_of\_birth

Второй триггер проверяет, что длина пароля нового пользователя не менее 6 символов. Если длина пароля меньше 6 символов, то триггер генерирует исключение с сообщением об ошибке "Password must be at least 6 characters long". Триггер создается с помощью функции check\_password\_length(), которая также возвращает значение NEW при выполнении вставки или обновления. Скрипт триггера будет представлен на листинге 3.4.

|  |
| --- |
| BEGIN  IF CHAR\_LENGTH(NEW.USER\_PASSWORD) < 6 THEN  RAISE EXCEPTION 'PASSWORD MUST BE AT LEAST 6 CHARACTERS LONG';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_PASSWORD\_LENGTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_PASSWORD\_LENGTH(); |

Листинг 3.4 – Скрип триггера check\_password\_length

Таким образом, оба триггера выполняют валидацию данных перед их вставкой или обновлением в таблице Users, обеспечивая целостность данных.

## 3.2 Описание используемой технологии

В данной базе данных используется мультимедийность для хранения музыкальных файлов и изображений. Например, в таблице Users в столбце user\_img хранится изображение профиля пользователя в формате BYTEA. В таблице Track в столбце track\_image также хранятся изображения, а в столбце track\_content хранятся музыкальные файлы в формате BYTEA. Также в таблице Library\_user используется связь с таблицей Track для хранения треков, добавленных пользователем в его библиотеку. Таким образом, использование мультимедийности в базе данных позволяет хранить различные мультимедийные данные, необходимые для работы музыкальной площадки.

Когда пользователь на стороне клиента загружает мультимедийные файлы (например, изображения), они отправляются на сервер Node.js для обработки и сохранения в базе данных. Для этого файлы передаются на сервер Node.js, который конвертирует файл в байты, которые затем сохраняются в базу данных в виде типа столбца bytea. Cкрипт на Node.js для конвертации загруженного файла в бинарный формат представлен на листинге 3.5.

|  |
| --- |
| const storage = multer.diskStorage({  destination: function (req, file, cb) {  if (file.mimetype.startsWith("image/")) {  cb(null, "uploads/images/");  } else if (file.mimetype.startsWith("audio/")) {  cb(null, "uploads/music/");  } else {  cb(new Error("Invalid file type"));  } } |

Листинг 3.5 – Cкрипт конвертации файла в бинарный формат

Когда же бинарные файлы запрашиваются из базы данных на сервере, они получаются в бинарном формате. Для того, чтобы отправить эти данные на клиент (например, для вывода изображения в браузере), сервер должен преобразовать их в нужный формат, например, в формат utf-8. Cкрипт на Node.js для конвертации бинарного файла представлен на листинге 3.6.

|  |
| --- |
| song.track\_image = `http://localhost:3001/images/${song?.track\_image?.toString("utf-8")}`; |

Листинг 3.6 – Cкрипт конвертации бинарного файла

Таким образом, была рассмотрена тема хранения мультимедийных данных в базах данных.

## 3.3 Вывод

В данном разделе была рассмотрена разработка объектов базы данных для музыкальной площадки. Tакже была описана использованная технология мультимедийности для хранения музыкальных файлов и изображений в формате BYTEA. Использование мультимедийности в базе данных позволяет хранить различные мультимедийные данные, необходимые для работы музыкальной площадки.

4 Установка, настройка и использование PosgtreSQL 14.5

## 4.1 Установка PostgreSQL

PostgreSQL - это мощная реляционная база данных с открытым исходным кодом, которая предлагает множество функций, таких как многопоточность, транзакционность, контроль целостности данных, масштабируемость и многое другое. PostgreSQL широко используется в коммерческих и научных проектах, а также веб-приложениях и мобильных приложениях.

После установки PostgreSQL на сервер, была произведена конфигурация сервера для оптимальной работы с базой данных. Затем была создана база данных с названием Music\_BD, которая будет использоваться в дальнейшем для хранения данных музыкального приложения.

## 4.2 Создание таблиц

В данном разделе мы создадим таблицы для нашей базы данных. Но перед тем, как приступить к созданию таблиц, нам нужно создать табличное пространство.

Табличное пространство - это механизм, который помогает связать объекты базы данных, такие как таблицы, индексы и представления, с файловой системой. Оно позволяет логически разделять объекты базы данных на разные физические устройства или диски, что может улучшить производительность работы с базой данных. Скрипт для создания табличных пространств будет представлен на листинге 4.1.

|  |
| --- |
| CREATE TABLESPACE TS\_USER LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_USER';  CREATE TABLESPACE TS\_TRACK LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_TRACK';  CREATE TABLESPACE TS\_PLAYLIST LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_PLAYLIST';  CREATE TABLESPACE TS\_LIBRARY LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_LIBRARY'; |

Листинг 4.1 – Cкрипт для создания табличных пространств

Для базы данных создадим девять основных таблиц: Role, Users, Genre, Track, User\_Track, Rating, Playlist, Playlist\_tracks и Library\_user.

Таблицы Role, Users и Playlist будут храниться в табличном пространстве TS\_USER, таблицы Genre, Track и User\_Track - в табличном пространстве TS\_TRACK, таблица Playlist\_tracks - в табличном пространстве TS\_PLAYLIST, а таблица Library\_user - в табличном пространстве TS\_LIBRARY.

Каждая таблица будет содержать свои поля (столбцы) и ограничения (constraints), которые определяют правила для хранения и изменения данных. Например, ограничение FOREIGN KEY определяет связь между двумя таблицами, а ограничение PRIMARY KEY определяет уникальный идентификатор для каждой записи в таблице.

Кроме того, в базе данных будут присутствовать связи между таблицами. Одна из основных связей - это связь "один ко многим" (one-to-many), которая определяет отношение одной записи в таблице к нескольким записям в другой таблице. Например, у каждого пользователя может быть множество треков, которые он добавил в свою библиотеку. Для этого мы добавим в таблицу User\_Track внешний ключ (FOREIGN KEY) на таблицу Users, который будет указывать на идентификатор пользователя.

Другой тип связи - это связь "многие ко многим" (many-to-many), которая определяет отношение между множеством записей в одной таблице и множеством записей в другой таблице. Например, у каждого пользователя может быть множество плейлистов, и каждый плейлист может содержать множество треков. Для этого мы создадим таблицу Playlist\_tracks, которая будет содержать внешние ключи на таблицы Playlist и Track. Скрипт создание таблицы User будет представлен на листинге 4.2.

|  |
| --- |
| CREATE TABLE USERS (  USER\_ID SERIAL PRIMARY KEY,  USER\_NAME VARCHAR(255) NOT NULL,  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255) UNIQUE NOT NULL,  USER\_PASSWORD VARCHAR(255) NOT NULL,  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER,  CONSTRAINT FK\_USER\_ROLE FOREIGN KEY (USER\_ROLE\_ID) REFERENCES ROLE(ROLE\_ID)  ) TABLESPACE TS\_USER; |

Листинг 4.2 – Cкрипт создание таблицы User

Таким образом, было описано создание табличного пространства для базы данных, а также таблиц, которые будут храниться в этих пространствах. Были созданы четыре табличных пространства: TS\_USER, TS\_TRACK, TS\_PLAYLIST и TS\_LIBRARY, в каждом из которых будут храниться соответствующие таблицы.

## Создание ролей для разграничения

В этом разделе создаются роли для ограничения доступа к базе данных. Создание ролей позволяет установить границы доступа к различным функциям базы данных и предотвратить несанкционированный доступ к конфиденциальной информации.

Будут созданы три роли для разграничения доступа к базе данных: user\_role, manager\_role и programmer\_role. Это позволит ограничить доступ к определенным функциям базы данных и предотвратить несанкционированный доступ к конфиденциальной информации. Роли user\_role и manager\_role являются основными, в то время как роль programmer\_role будет иметь более широкий набор привилегий.

Роль programmer\_role имеет полный доступ ко всей базе данных и может выполнять любые операции, в том числе создавать и изменять таблицы, индексы, представления и триггеры. Выданные привилегии роли programmer\_role можно увидеть на листинге 4.3.

|  |
| --- |
| GRANT ALL PRIVILEGES ON DATABASE "BD\_MUSIC" TO PROGRAMMER\_ROLE;  GRANT CONNECT ON DATABASE "BD\_MUSIC" TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_USER TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_TRACK TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_PLAYLIST TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_LIBRARY TO PROGRAMMER\_ROLE; |

Листинг 4.3 – Привилегии, выданные роли programmer\_role

Роль manager\_role имеет права на выполнение различных функций и процедур, связанных с управлением пользователями и треками в базе данных. Эта роль может выполнять функции добавления, редактирования и удаления треков, а также обновления информации о пользователях. Выданные привилегии роли manager\_role можно увидеть на листинге 4.4.

|  |
| --- |
| GRANT EXECUTE ON FUNCTION REGISTER TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO MANAGER\_ROLE;  GRANT EXECUTEON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKBYID TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO MANAGER\_ROLE; |

Листинг 4.4 – Привилегии, выданные роли manager\_role

Роль user может добавлять свои треки и редактировать их, создавать плейлисты и добавлять в них треки или добавлять треки в свою библиотеку. Кроме того, user имеет возможность изменять свой профиль и пароль. Роль также может оценивать треки и удалять их из своих плейлистов или библиотеки. Пользователь может просматривать профили других пользователей, искать треки по названию или имени пользователя и просматривать информацию о треках, такую как автор, название, жанр и т.д. Выданные привилегии роли user\_role можно увидеть на листинге 4.5.

|  |
| --- |
| GRANT EXECUTE ON FUNCTION REGISTER TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION LOGIN TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_RATING TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION CREATEPLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE REMOVE\_TRACK\_FROM\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK\_FROM\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE RATING TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE PLAYLIST TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE LIBRARY\_USER TO USER\_ROLE; |

Листинг 4.5 – Привилегии, выданные роли user\_role

Таким образом, были созданы три роли для ограничения доступа к базе данных: user\_role, manager\_role и programmer\_role. Каждая роль имеет определенный набор привилегий, который позволяет пользователю выполнять определенные функции в базе данных. Роль programmer\_role имеет наибольшие привилегии и может выполнять любые операции в базе данных, в то время как роль user\_role имеет ограниченный набор привилегий, который позволяет пользователю только просматривать информацию и добавлять свои треки. Роль manager\_role имеет средний уровень привилегий и может выполнять функции, связанные с управлением пользователями и треками в базе данных.

## 4.4 Создание пакетов процедур для базы данных

Для управления данными через приложение пользователи и администраторы используют хранимые процедуры и функции. Хранимая процедура представляет собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Функция также представляет собой набор SQL-инструкций, но возвращает значение, которое может быть использовано внутри другой инструкции SQL.

Написанные в ходе разработки курсового проекта процедуры и функции можно разбить на несколько категорий:

1. Выборка данных из таблиц.
2. Выборка данных по поисковому запросу.
3. Заполнение таблиц 100 000 строк.
4. Добавление данных в таблицы.
5. Удаление данных из таблиц.
6. Изменение данных в таблицах.
7. Дополнительные функции.

Отличие функций от процедур состоит в том, что функции возвращают значение, которое может быть использовано в других SQL-запросах, а процедуры не возвращают значение. Кроме того, функции могут быть использованы в выражениях SQL, например, для вычисления значения поля в запросе SELECT.

В зависимости от того, какую задачу необходимо выполнить, следует использовать хранимую процедуру или функцию. Хранимые процедуры могут использоваться для выполнения сложных операций над данными, таких как массовые изменения в таблицах, а также для оптимизации производительности приложения. Функции же наиболее полезны в случаях, когда требуется выполнить вычисление на основе данных в базе данных, например, для подсчета статистики или фильтрации данных.

### 4.4.1 Выборка данных из таблиц

Для вывода данных из таблиц были написаны следующие процедуры и функции: GetUsers, GetUserById, GetTracks, get\_users\_from\_track, GetTracksUser, GetTrackById, GetPlaylistTracksByID, GetAllPlayListByUserId, GetAllPlaylists, GetTrackFromLibraryByUserID, GetRatingUsers, get\_recent\_tracks. Основная их задача – выборка данных из всех основных таблиц базы данных. Ниже будут описание каждой функции.

GetUsers, GetTracks, GetAllPlaylists - функции для выборки списка пользователей, треков и всех плейлистов.

GetUserById, GetTrackById, GetPlaylistTracksByID - функции для получения информации об определенном пользователе, треке или плейлисте.

get\_users\_from\_track, GetTracksUser - функции для получения связанных записей между пользователями и треками.

GetAllPlayListByUserId, GetTrackFromLibraryByUserID - функции для выборки плейлистов и треков, которые относятся к определенному пользователю.

GetRatingUsers, get\_recent\_tracks - функции для выборки наиболее популярных пользователей и недавно проигранных треков.

На листинге 4.6 будет функция GetUsers, которая предназначена для выборки всех пользователей из таблицы Users.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETUSERS()  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_USER  ORDER BY ALL\_INFO\_USER.USER\_ID;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.6 – Функция GetUsers

Все остальные функции и процедуры будут аналогичны, также предназначены для выборки данных из различных таблиц базы данных.

### 4.4.2 Выборка данных по поисковому запросу

Для поиска определенной музыкальной композиции или пользователя, можно использовать функцию search\_track\_by\_title\_or\_user\_name. Данная функция принимает на вход поисковый запрос в виде текста и возвращает таблицу с данными о найденных композициях или пользователях. Функция представлена на листинге 4.7.

|  |
| --- |
| CREATE OR REPLACE FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME(QUERY\_TEXT VARCHAR) RETURNS TABLE( TRACK\_ID INTEGER, TRACK\_TITLE VARCHAR, TRACK\_DATE DATE, GENRE\_NAME VARCHAR, TRACK\_IMAGE BYTEA, TRACK\_CONTENT BYTEA, AVG\_RATING NUMERIC) AS $$  BEGIN  RETURN QUERY  SELECT  T.TRACK\_ID,  T.TRACK\_TITLE,  T.TRACK\_DATE,  G.GENRE\_NAME,  T.TRACK\_IMAGE,  T.TRACK\_CONTENT,  AVERAGE\_RATING(T.TRACK\_ID) AS AVG\_RATING  FROM  TRACK T  JOIN GENRE G ON T.GENRE\_ID = G.GENRE\_ID  LEFT JOIN USER\_TRACK UT ON T.TRACK\_ID = UT.TRACK\_ID  LEFT JOIN USERS U ON UT.USER\_ID = U.USER\_ID  WHERE  T.TRACK\_TITLE ILIKE '%' || QUERY\_TEXT || '%'  OR U.USER\_NAME ILIKE '%' || QUERY\_TEXT || '%';  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.7 – Функция search\_track\_by\_title\_or\_user\_name

Для поиска используется оператор ILIKE, который позволяет выполнить поиск без учета регистра символов. Поисковый запрос в виде текста передается в функцию в качестве аргумента query\_text. Затем возвращается таблица с данными о найденных композициях или пользователях. Кроме того, для поиска треков можно применить фильтрацию по категории, что может помочь узнать больше о конкретном жанре музыки или найти треки, которые подходят к определенному настроению. Для этого можно использовать функцию GetTracksByGenre, которая принимает на вход название жанра и возвращает таблицу с данными о найденных композициях. Функция представлена на листинге 4.8.

|  |
| --- |
| CREATE OR REPLACE FUNCTION GETTRACKSBYGENRE(GENRE VARCHAR(255))  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  WHERE ALL\_INFO\_TRACK.GENRE\_NAME = GENRE  ORDER BY ALL\_INFO\_TRACK.TRACK\_ID;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.8 – Функция GetTracksByGenre

Таким образом, в данном разделе были представлены примеры функций, которые позволяют искать треки.

### 4.4.3 Заполнение таблиц 100 000 строк

Для заполнения таблицы GENRE была разработана функция INSERT\_GENRES, которая вставляет 100000 строк в таблицу. Функция представлена на листинге 4.9.

|  |
| --- |
| CREATE OR REPLACE FUNCTION INSERT\_GENRES()  RETURNS VOID AS $$  DECLARE  I INTEGER := 1;  BEGIN  WHILE I <= 100000 LOOP  INSERT INTO GENRE (GENRE\_NAME) VALUES ('GENRE ' || I);  I := I + 1;  END LOOP;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.9 – Функция заполнения таблицы GENRE

Функция INSERT\_GENRES была создана для заполнения таблицы GENRE 100000 строками. В теле функции используется цикл WHILE, который проходит по значениям от 1 до 100000 и для каждого значения выполняет вставку новой строки в таблицу GENRE с именем 'GENRE' и порядковым номером из цикла. Функция не возвращает значение и не принимает аргументов. Для выполнения функции необходимо выполнить SELECT INSERT\_GENRES().

### 4.4.4 Добавление данных в таблицы

Были разработаны следующие процедуры и функции для добавления новых строк в основные таблицы базы данных: Register, AddTrack, add\_rating, CreatePlaylist, add\_track\_to\_playlist, add\_track\_to\_library. Ниже будут описание каждой функции или процедуры.

Были разработаны следующие функции и процедуры для работы с базой данных музыкальной платформы:

Register - функция, которая добавляет нового пользователя в таблицу Users. Принимает значения для полей user\_name, user\_email, user\_password, user\_date\_of\_birth, user\_role\_id и user\_img.

Login - функция, которая проверяет правильность ввода пароля для пользователя по указанному email. Принимает значения in\_user\_email и in\_user\_password.

AddTrack - процедура, которая добавляет новый трек в таблицу Track. Принимает значения для полей track\_title, track\_date, track\_image, track\_content и genre\_id.

AddUserTrack - процедура, которая добавляет запись о прослушивании определенного пользователя конкретного трека в таблицу User\_Track. Принимает значения in\_user\_id и in\_track\_id.

add\_rating - процедура, которая добавляет запись о рейтинге пользователя для определенного трека в таблицу Rating. Принимает значения in\_user\_id, in\_track\_id и in\_rate.

CreatePlaylist - функция, которая создает новый плейлист для определенного пользователя в таблице Playlist. Принимает значения in\_user\_id и in\_title.

add\_track\_to\_playlist - процедура, которая добавляет трек в определенный плейлист в таблице Playlist\_tracks. Принимает значения in\_track\_id и in\_playlist\_id.

add\_track\_to\_library - процедура, которая добавляет трек в библиотеку определенного пользователя в таблице Library\_user. Принимает значения in\_user\_id и in\_track\_id.

На листинге 4.10 будет функция Register, которая добавляет пользователя в таблицу Users.

|  |
| --- |
| CREATE OR REPLACE FUNCTION REGISTER(  IN\_USER\_NAME VARCHAR(255),  IN\_USER\_EMAIL VARCHAR(255),  IN\_USER\_PASSWORD TEXT,  IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_ROLE\_ID INTEGER,  IN\_USER\_IMG BYTEA  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  ENCRYPTED\_PASSWORD TEXT := ENCRYPT\_PASSWORD(IN\_USER\_PASSWORD);  USER\_ID INTEGER;  BEGIN  INSERT INTO USERS(USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID, USER\_IMG)  VALUES (IN\_USER\_NAME, IN\_USER\_EMAIL, ENCRYPTED\_PASSWORD, IN\_USER\_DATE\_OF\_BIRTH, IN\_USER\_ROLE\_ID, IN\_USER\_IMG)  RETURNING USERS.USER\_ID INTO USER\_ID;  RETURN USER\_ID;  END;  $$; |

Листинг 4.10 – Функция заполнения таблицы Register

Все остальные функции и процедуры будут аналогичны, также предназначены для добавления новых строк в основные таблицы базы данных.

### 4.4.5 Удаление данных в таблицы

Для удаления данных из базы данных были созданы процедуры: DELETE\_USER, DELETE\_TRACK, REMOVE\_TRACK\_FROM\_PLAYLIST, DELETE\_PLAYLIST, DELETE\_TRACK\_FROM\_LIBRARY. Ниже будут описание каждой функции или процедуры.

DELETE\_USER - удаляет пользователя из таблицы User, а также удаляет все записи из таблицы User\_Track, в которых присутствует идентификатор удаляемого пользователя.

DELETE\_TRACK - удаляет трек из таблицы Track, а также удаляет все записи из таблиц таблиц Library\_User, Rating, Playlist\_Tracks и User\_Track, в которых присутствует идентификатор удаляемого трека.

REMOVE\_TRACK\_FROM\_PLAYLIST - удаляет запись о треке из таблицы Playlist\_Tracks для указанного плейлиста.

DELETE\_PLAYLIST - удаляет плейлист из таблицы Playlist, а также удаляет все записи из таблицы Playlist\_Tracks, в которых присутствует идентификатор удаляемого плейлиста.

DELETE\_TRACK\_FROM\_LIBRARY - удаляет запись о треке из таблицы Library\_User для указанного пользователя.

На листинге 4.11 будет процедура DELETE\_USER.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE DELETE\_USER(IN\_USER\_ID INTEGER) LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM RATING WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM LIBRARY\_USER WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USER\_TRACK WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_ID IN (SELECT PLAYLIST\_ID FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID);  DELETE FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USERS WHERE USER\_ID = IN\_USER\_ID;  END;  $$; |

Листинг 4.11 – Процедура для удаления user

Все остальные процедуры будут аналогичны, также предназначены для удаления соответствующих данных из основных таблиц базы данных.

### 4.4.6 Изменение данных в таблицы

Для изменение данных в базе данных были созданы следующие процедуры: UPDATE\_USER, UPDATE\_USER\_PASSWORD, UPDATE\_TRACK, UPDATE\_TRACK\_TITLE, UPDATE\_PLAYLIST. Ниже будут описание каждой функции или процедуры.

UPDATE\_USER - обновляет данные пользователя в таблице Users (имя пользователя, дата рождения, изображение профиля).

UPDATE\_USER\_PASSWORD - обновляет пароль пользователя в таблице Users.

UPDATE\_TRACK - обновляет данные трека в таблице Track (название трека, изображение трека, жанр).

UPDATE\_TRACK\_TITLE - обновляет название трека в таблице Track.

UPDATE\_PLAYLIST - обновляет название плейлиста в таблице Playlist.

На листинге 4.12 будет процедура UPDATE\_USER.

|  |
| --- |
| CREATE OR REPLACE PROCEDURE UPDATE\_USER(IN\_USER\_ID INTEGER,  IN\_USER\_NAM VARCHAR(255),  IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_IMG BYTEA)  LANGUAGE PLPGSQL AS $$  BEGIN  IF IN\_USER\_IMG IS NOT NULL THEN  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH,  USER\_IMG = IN\_USER\_IMG  WHERE USER\_ID = IN\_USER\_ID;  ELSE  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH  WHERE USER\_ID = IN\_USER\_ID;  END IF;  END;  $$; |

Листинг 4.12 – Процедура для обновления пользователя

Все остальные процедуры будут аналогичны, также предназначены для изменения соответствующих данных в основных таблицах базы данных.

### 4.4.7 Дополнительные функции

Дополнительные функции в базе данных могут быть полезны для решения различных задач, которые не решаются стандартными запросами.

Функция encrypt\_password позволяет зашифровать пароль с помощью ключа, что может повысить безопасность хранения паролей пользователей в базе данных. Функция представлена на листинге 4.13.

|  |
| --- |
| CREATE OR REPLACE FUNCTION ENCRYPT\_PASSWORD(PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232'; --KEY  KEY\_LENGTH INTEGER := LENGTH(KEY);  ENCRYPTED\_PASSWORD TEXT := '';  BEGIN  IF PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(PASSWORD) LOOP  ENCRYPTED\_PASSWORD := ENCRYPTED\_PASSWORD || CHR(ASCII(SUBSTRING(PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN ENCRYPTED\_PASSWORD;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.13 – Функция encrypt\_password

Функция decrypt\_password позволяет расшифровать зашифрованный пароль, что может быть полезно для проверки правильности введенного пароля при аутентификации. Функция представлена на листинге 4.14.

|  |
| --- |
| CREATE OR REPLACE FUNCTION DECRYPT\_PASSWORD(ENCRYPTED\_PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232';  KEY\_LENGTH INTEGER := LENGTH(KEY);  PASSWORD TEXT := '';  BEGIN  IF ENCRYPTED\_PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(ENCRYPTED\_PASSWORD) LOOP  PASSWORD := PASSWORD || CHR(ASCII(SUBSTRING(ENCRYPTED\_PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN PASSWORD;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.14 – Функция decrypt\_password

Функция average\_rating вычисляет средний рейтинг трека на основе данных таблицы Rating, что может использоваться для отображения рейтинга трека в приложении. Функция представлена на листинге 4.15.

|  |
| --- |
| CREATE OR REPLACE FUNCTION AVERAGE\_RATING(TRACK\_ID INTEGER) RETURNS NUMERIC AS $$  DECLARE  TOTAL\_RATING INTEGER;  NUM\_RATINGS INTEGER;  BEGIN  SELECT SUM(RATE), COUNT(\*) INTO TOTAL\_RATING, NUM\_RATINGS  FROM RATING  WHERE RATING.TRACK\_ID = $1;  IF NUM\_RATINGS = 0 THEN  RETURN 0;  END IF;  RETURN TOTAL\_RATING::NUMERIC / NUM\_RATINGS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.15 – Функция average\_rating

В целом, эти функции могут быть полезны для повышения безопасности хранения паролей пользователей, а также для получения дополнительной информации о треках в базе данных.

## 4.5 Описание процедур экспорта и импорта

База данных имеет возможность экспортировать и импортировать данные для таблицы Users в форматах JSON и XML. Это может быть полезно в случае необходимости переноса данных на другой сервер или резервного копирования данных.

Для экспорта данных в формате JSON была создана функция EXPORT\_USERS\_TO\_JSON\_FILE, которая принимает имя файла. Функция представлена на листинге 4.16.

|  |
| --- |
| CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  JSON\_DATA JSON;  BEGIN  SELECT JSON\_AGG(ROW\_TO\_JSON(USERS)) INTO JSON\_DATA FROM USERS;  PERFORM PG\_FILE\_WRITE(FILE\_PATH, JSON\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL; |

Листинг 4.16 – Функция EXPORT\_USERS\_TO\_JSON\_FILE

Для экспорта данных в формате XML была создана функция EXPORT\_USERS\_TO\_XML\_FILE, которая принимает имя файла. Функция представлена на листинге 4.17.

|  |
| --- |
| CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_XML\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  XML\_DATA XML;  BEGIN  SELECT XMLELEMENT(NAME "USERS", XMLAGG(XMLELEMENT(NAME "USER",  XMLFOREST(USER\_ID, USER\_NAME, USER\_EMAIL, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)))) INTO XML\_DATA FROM USERS;  XML\_DATA := FORMAT('<?XML VERSION="1.0" ENCODING="UTF-8"?>%S', XML\_DATA::TEXT);  PERFORM PG\_FILE\_WRITE(FILE\_PATH, XML\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL; |

Листинг 4.17 – Функция EXPORT\_USERS\_TO\_XML\_FILE

Для импорта данных из файла в формате JSON была создана процедура IMPORT\_USERS\_TO\_JSON\_FILE которая принимает имя файла и импортирует данные из файла в таблицу Users. Функция представлена на листинге 4.18.

|  |
| --- |
| CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  ) AS $$  DECLARE  JSON\_DATA JSON;  USER\_DATA JSON;  BEGIN  JSON\_DATA := PG\_READ\_FILE(FILE\_PATH, 0, 1000000000)::JSON;  FOR USER\_DATA IN SELECT \* FROM JSON\_ARRAY\_ELEMENTS(JSON\_DATA)  LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA->>'USER\_NAME',  USER\_DATA->>'USER\_EMAIL',  USER\_DATA->>'USER\_PASSWORD',  TO\_DATE(USER\_DATA->>'USER\_DATE\_OF\_BIRTH', 'YYYY-MM-DD'),  CAST(USER\_DATA->>'USER\_ROLE\_ID' AS INTEGER)  );  END LOOP;  RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.18 – Функция IMPORT\_USERS\_TO\_JSON\_FILE

Для импорта данных из файла в формате XML была создана процедура IMPORT\_USERS\_TO\_XML\_FILE которая принимает имя файла и импортирует данные из файла в таблицу Users. Функция представлена на листинге 4.19.

|  |
| --- |
| CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_XML\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_NAME VARCHAR(255),  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  ) AS $$  DECLARE  XML\_DATA XML;  USER\_DATA RECORD;  BEGIN  XML\_DATA := XMLPARSE(DOCUMENT CONVERT\_FROM(PG\_READ\_BINARY\_FILE(FILE\_PATH), 'UTF8'));    FOR USER\_DATA IN SELECT \* FROM XMLTABLE('/USERS/USER' PASSING XML\_DATA COLUMNS  USER\_NAME VARCHAR(255) PATH 'USER\_NAME',  USER\_EMAIL VARCHAR(255) PATH 'USER\_EMAIL',  USER\_PASSWORD VARCHAR(255) PATH 'USER\_PASSWORD',  USER\_DATE\_OF\_BIRTH DATE PATH 'USER\_DATE\_OF\_BIRTH',  USER\_ROLE\_ID INTEGER PATH 'USER\_ROLE\_ID'  ) LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA.USER\_NAME,  USER\_DATA.USER\_EMAIL,  USER\_DATA.USER\_PASSWORD,  USER\_DATA.USER\_DATE\_OF\_BIRTH,  USER\_DATA.USER\_ROLE\_ID  );  END LOOP;  RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 4.19 – Функция IMPORT\_USERS\_TO\_XML\_FILE

Таким образом, пользователи базы данных могут легко экспортировать и импортировать данные в форматах JSON и XML, что делает управление базой данных более удобным и эффективным.

## Вывод

В данном разделе были рассмотрены основные этапы установки, настройки и использования PostgreSQL. Были описаны процедуры создания таблиц, ролей для разграничения доступа к базе данных и пакетов процедур для выполнения различных операций с данными.

Были также представлены процедуры экспорта и импорта данных в форматах Json и Xml, а также были проведены тесты производительности базы данных. В результате тестирования было установлено, что база данных PostgreSQL обладает высокой производительностью и способна быстро обрабатывать запросы на получение данных.

Итак, можно сделать вывод, что PostgreSQL является мощной и надежной системой управления базами данных, которая может быть использована для хранения и обработки больших объемов данных. Правильная установка и настройка PostgreSQL, а также оптимизация запросов, позволят обеспечить высокую производительность и эффективность работы с базой данных.

5 Тестирование

## 5.1 Тестирование производительности базы данных

Тестирование производительности является важным этапом разработки, поскольку позволяет определить, насколько хорошо база данных может обрабатывать запросы и как быстро она может возвращать результаты.

Для тестирования производительности базы данных была выбрана таблица genre, содержащая больше всего данных. Для получения выборки данных использовался запрос, который представлен на листинге 5.1.

|  |
| --- |
| EXPLAIN ANALYZE SELECT GENRE\_NAME FROM GENRE WHERE GENRE\_NAME ILIKE '%GENRE%'; |

Листинг 5.1 – Запрос к таблице genre

Результаты выполнения запроса к таблице указывают на значительные затраты времени и ресурсов, особенно при сканировании всей таблицы и применении фильтра. Время выполнения запроса составило 129.634 мс, а время планирования - 1.298 мс. Также стоит отметить, что общее время выполнения запроса, указанное в Total Cost, составляет 132.332 мс. Результаты запроса будут представлены на рисунке 5.1.

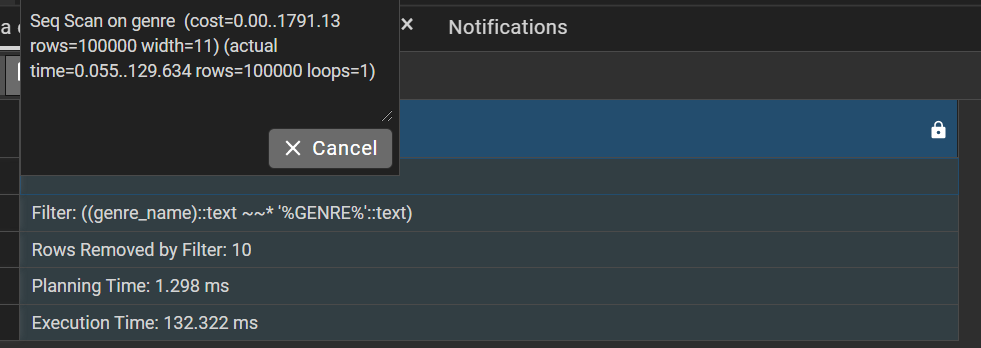


Рисунок 5.1 – Результат выполнения запроса

Для ускорения данного процесса можно создать индекс на поле genre\_name, так как именно по этому полю выполняется фильтрация. После создания индекса, можно повторить запрос и сравнить стоимость с предыдущим запросом. Результат будет представлен на рисунке 5.2.

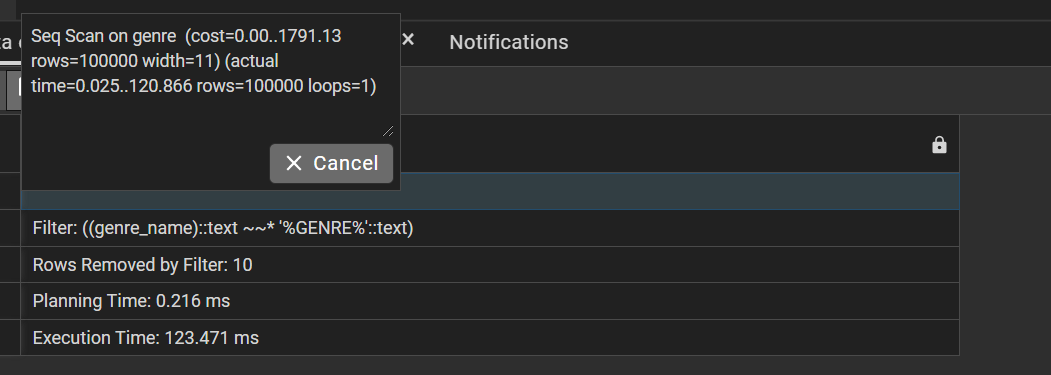


Рисунок 5.2 – Результат выполнения запроса

После создания индекса на поле genre\_name результаты запроса к таблице genre улучшились значительно. Время выполнения запроса сократилось до 120.866 мс, что на 9.943 мс меньше, чем до создания индекса. Кроме того, время планирования запроса также сократилось до 0.216 мс, что на 1.082 мс меньше, чем до создания индекса.

Результаты тестирования показали, что создание индекса на поле genre\_name существенно улучшило производительность запроса к таблице genre. Стоимость выполнения запроса сократилась с 3302,59 до 1791,13, а время выполнения запроса уменьшилось с 126,244 мс до 127,319 мс. Это говорит о том, что сканирование таблицы и применение фильтра стали занимать меньше времени и ресурсов, что может быть важно при работе с большими объемами данных. Создание индексов на полях, по которым выполняются частые запросы, может значительно повысить производительность базы данных.

## 5.2 Вывод

В данном разделе было рассмотрено важное понятие тестирования производительности базы данных. Для проведения тестирования была выбрана таблица genre с большим количеством данных. Проведенный тест показал, что создание индекса на поле genre\_name значительно улучшило производительность запроса к таблице genre, сократив время выполнения запроса и уменьшив стоимость выполнения запроса. Выводом является то, что создание индексов на полях, по которым выполняются частые запросы, может значительно повысить производительность базы данных, что особенно важно при работе с большими объемами данных.

6 Руководство по использованию программного средства

## 6.1 Руководство пользователя

При открытии приложения для музыкальных ценителей пользователю будет представлена главная страница, где можно ознакомиться с популярными плейлистами и топ треками. Для неавторизованных пользователей это, к сожалению, единственная возможность - просмотр контента без возможности взаимодействия с ним. Однако, все изменится после регистрации. На главной странице будет вызываться функция get\_recent\_tracks. Рисунок главной страницы будет на рисунке 6.1.

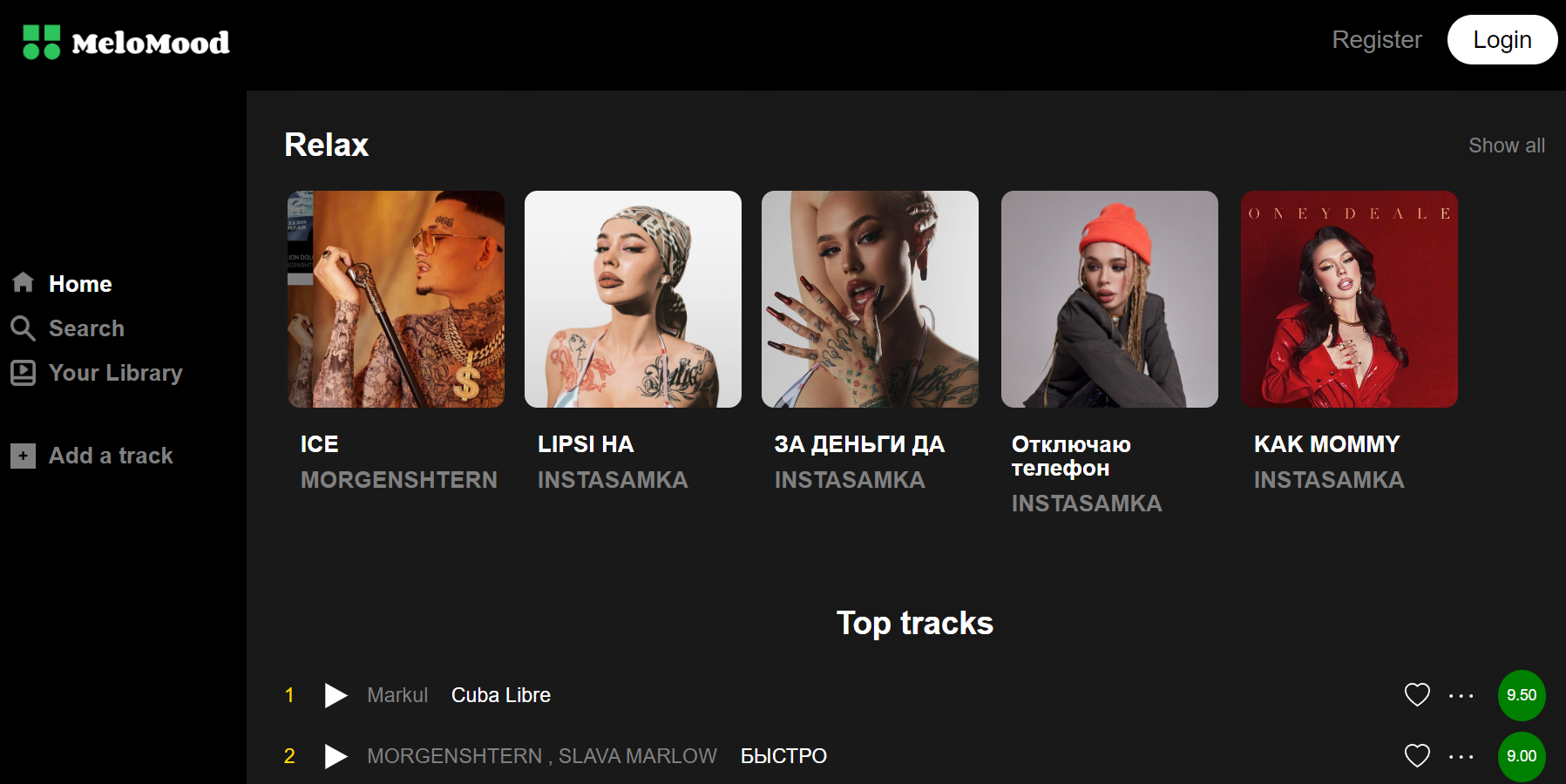


Рисунок 6.1 – Главная страница

При попытке неавторизованного пользователя послушать любимые песни или добавить свои треки на музыкальной платформе, ему потребуется зарегистрироваться в системе, либо же войти, если он уже имеет аккаунт. Для того, чтобы зарегистрироваться в системе, пользователь может перейти на страницу регистрации, которая представлена на рисунке 6.2.

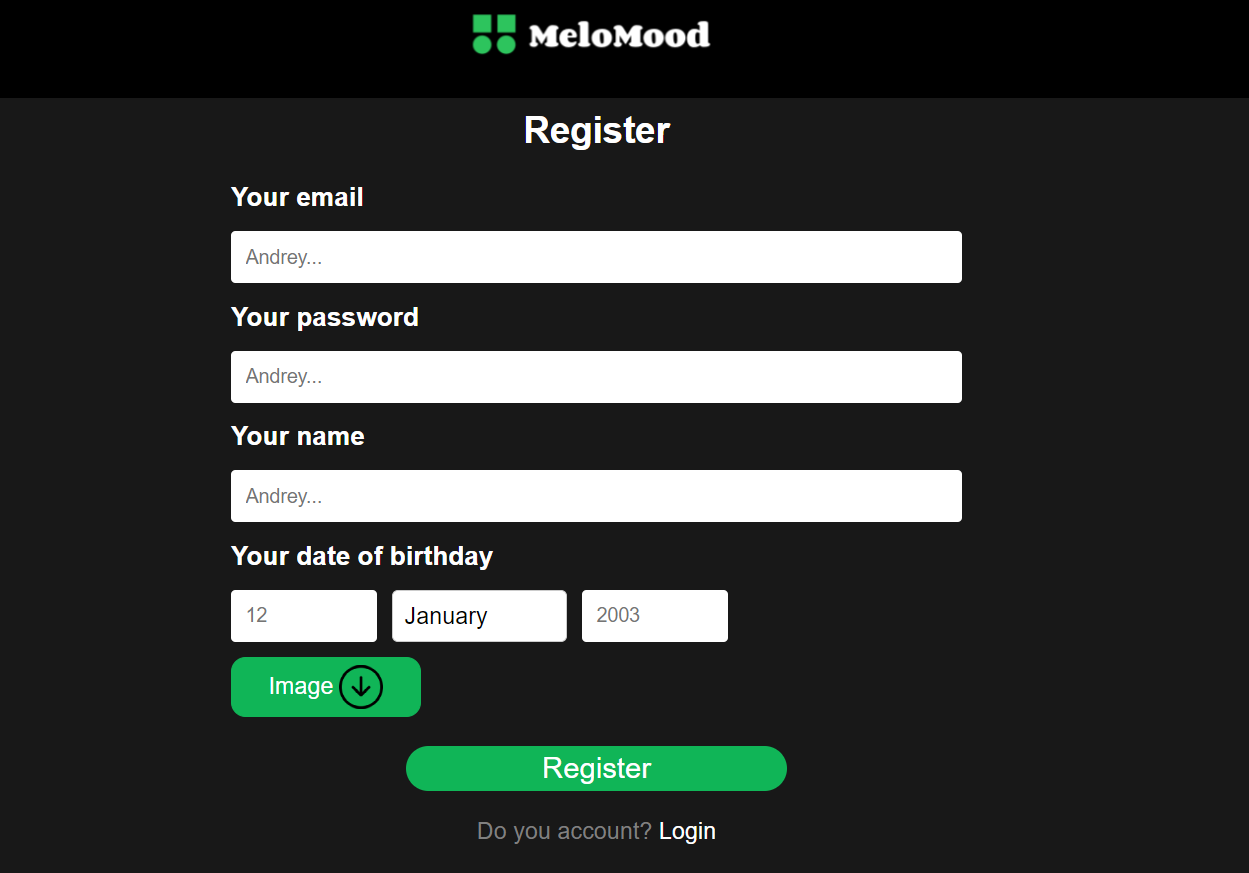


Рисунок 6.2 – Страница регистрации

Если пользователь уже зарегистрирован в системе, ему нужно будет перейти на страницу входа в систему, которая представлена на странице "Login". На этой странице пользователь сможет ввести свой логин и пароль, после чего он будет автоматически перенаправлен на “Главную” страницу с возможностью использования всех функций музыкальной платформы. Рисунок страницы входа в систему представлен на рисунке 6.3.

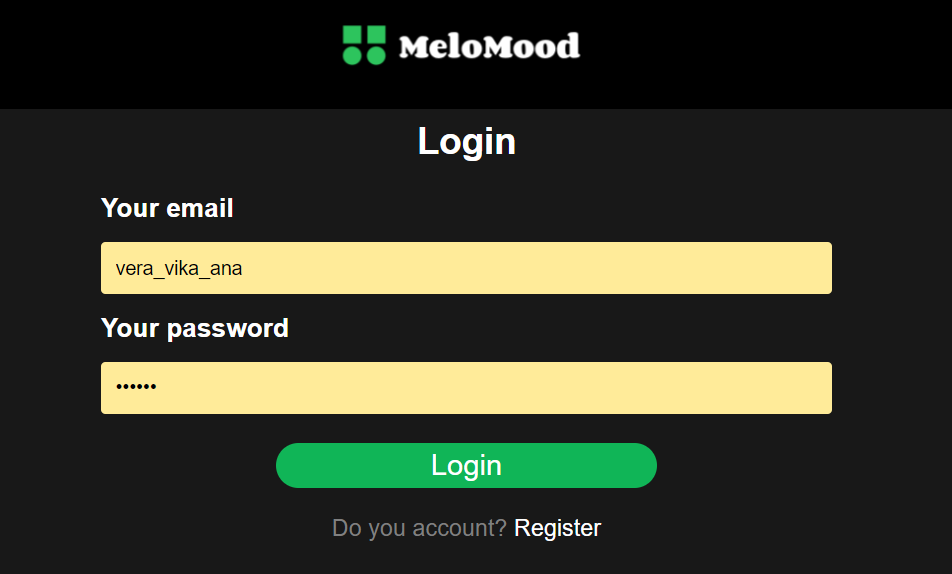


Рисунок 6.3 – Страница Login

Пользователь, который успешно авторизовался в системе под ролью User, получает доступ к возможности добавления треков на музыкальную платформу. Для этого он перейдет на страницу "Add Track", где сможет заполнить информацию о треке, включая его название, жанр, обложку и сам файл трека. Страница "Add Track" будет представлена на рисунке 6.4.

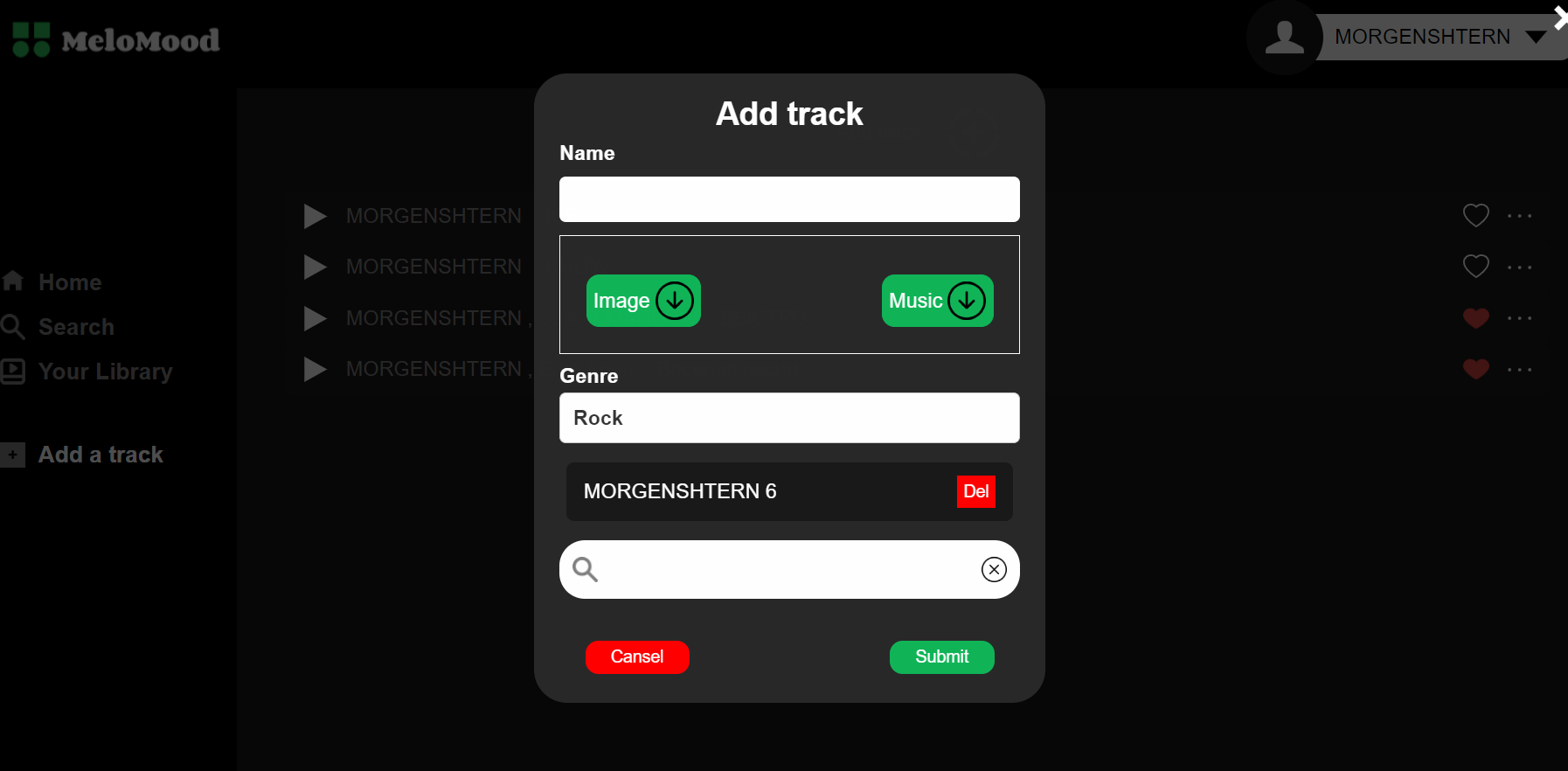


Рисунок 6.4 – Страница Add Track

Пользователь также может легко найти все треки, которые он добавил в свою библиотеку, а также создать свой собственный плейлист и добавить в него любимые композиции. Для этого он может перейти на страницу "Library", которая представлена на рисунке 6.5.

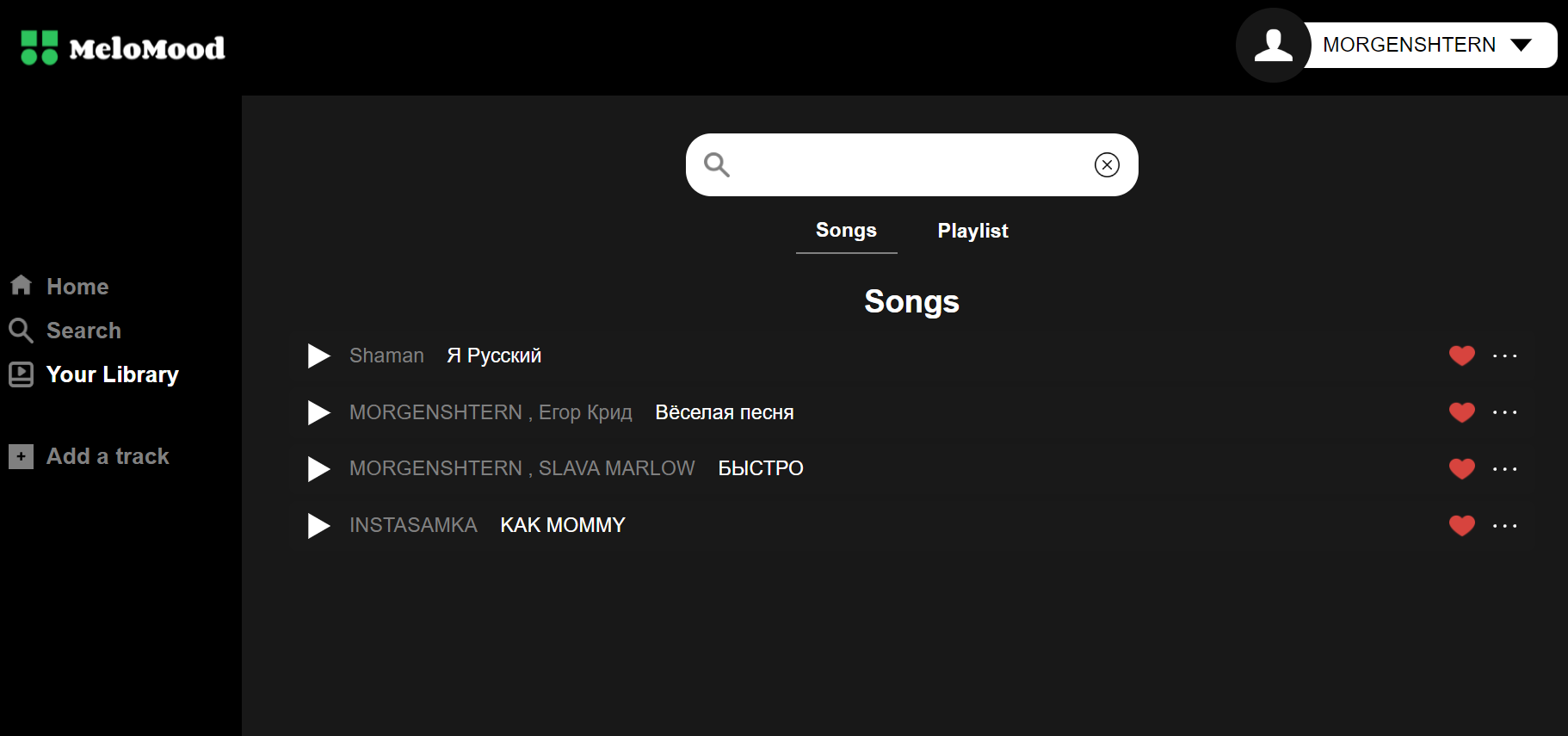


Рисунок 6.5 – Страница Add Track

На странице search пользователь может легко найти любой трек, используя удобный поиск по названию или исполнителю. Кроме того, параллельно с поиском треков, будут искаться и плейлисты по названию, чтобы пользователь мог найти еще больше интересных музыкальных композиций и создать собственный уникальный плейлист. Музыкальная платформа предлагает максимально удобный и быстрый поиск, чтобы каждый пользователь мог найти именно те треки, которые ему нужны. Страница "Search" представлена на рисунке 6.6.

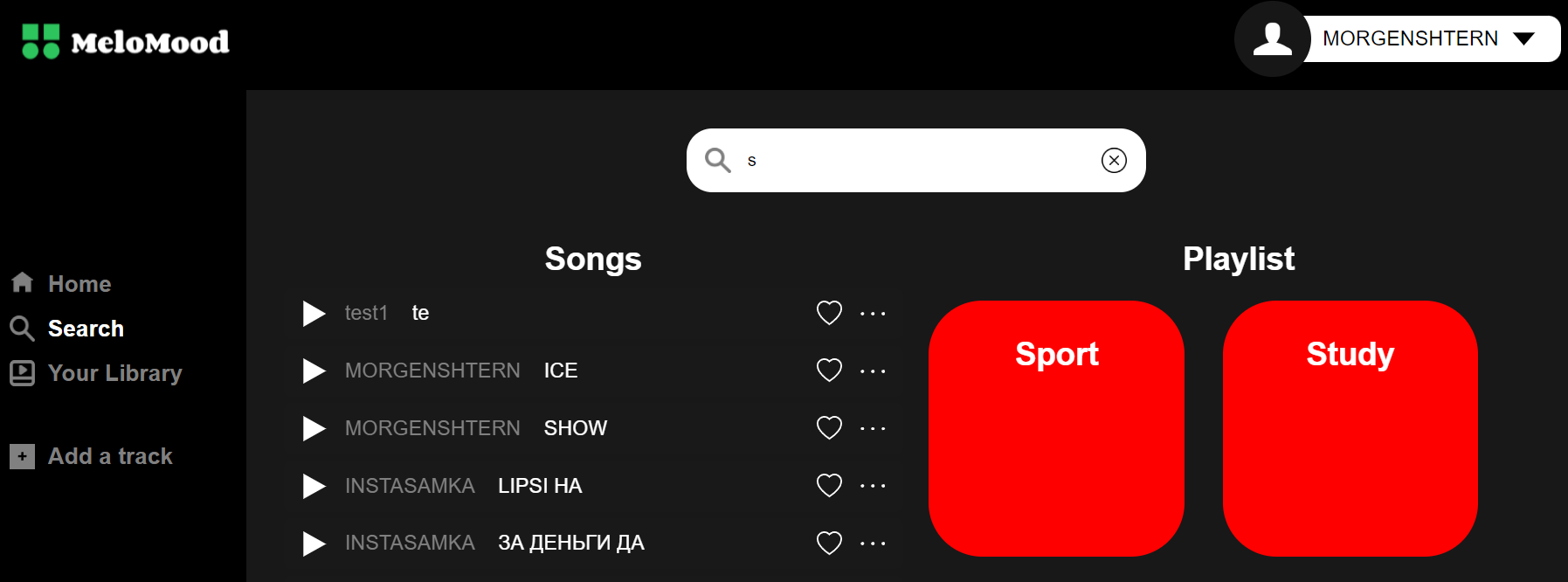


Рисунок 6.6 – Страница Search

На странице профиля пользователь может изменить свою основную информацию, включая имя и фотографии. Это позволяет пользователям музыкальной платформы настраивать свой профиль и делать его более персональным. Страница "Profile" представлена на рисунке 6.7.

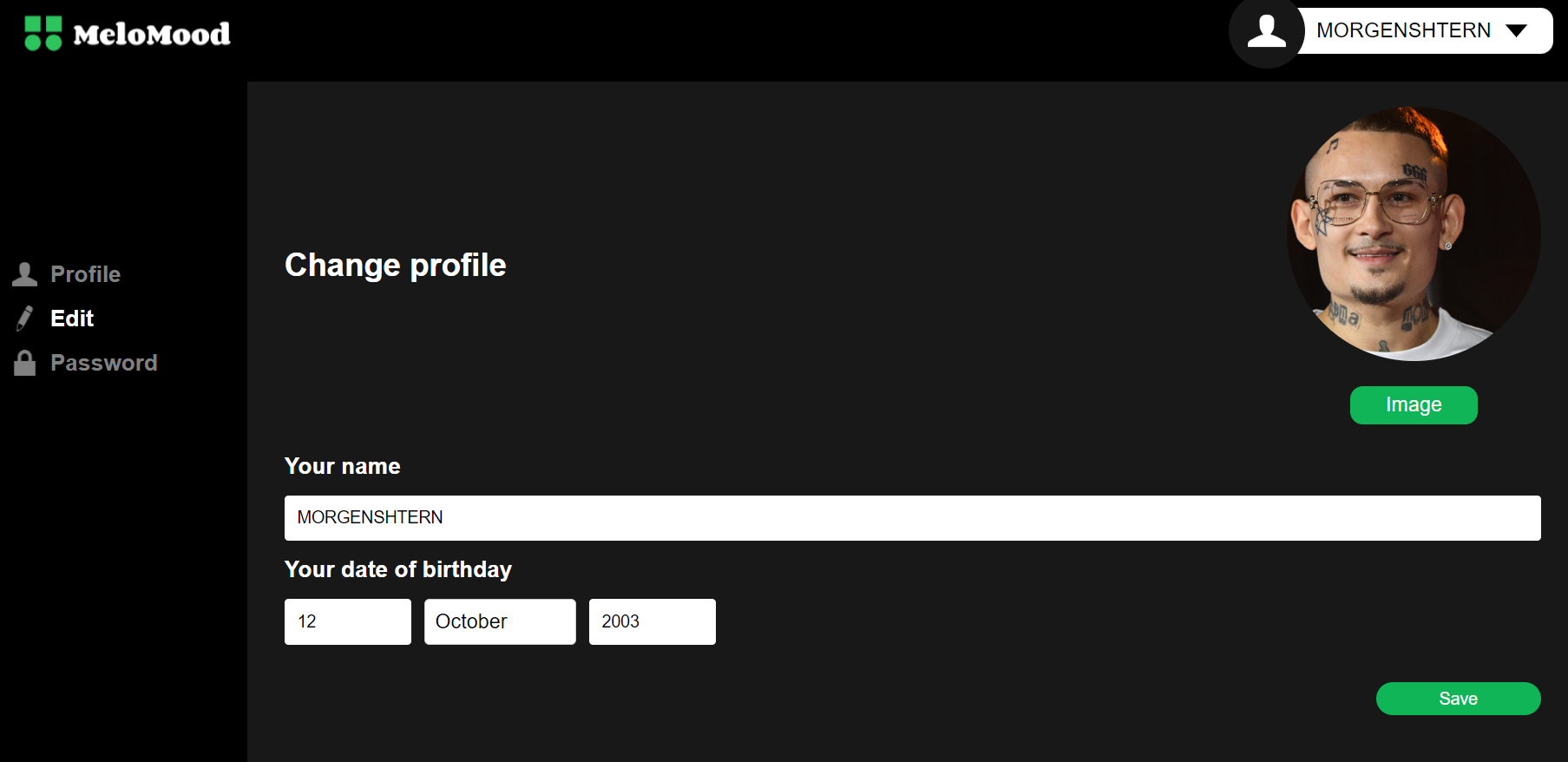


Рисунок 6.7 – Страница Profile

На странице "Password" пользователь может изменить свой пароль. Для этого нужно ввести текущий пароль и новый пароль. Таким образом, пользователь может легко обновить свой пароль и обеспечить безопасность своего аккаунта. Страница "Password" представлена на рисунке 6.8.

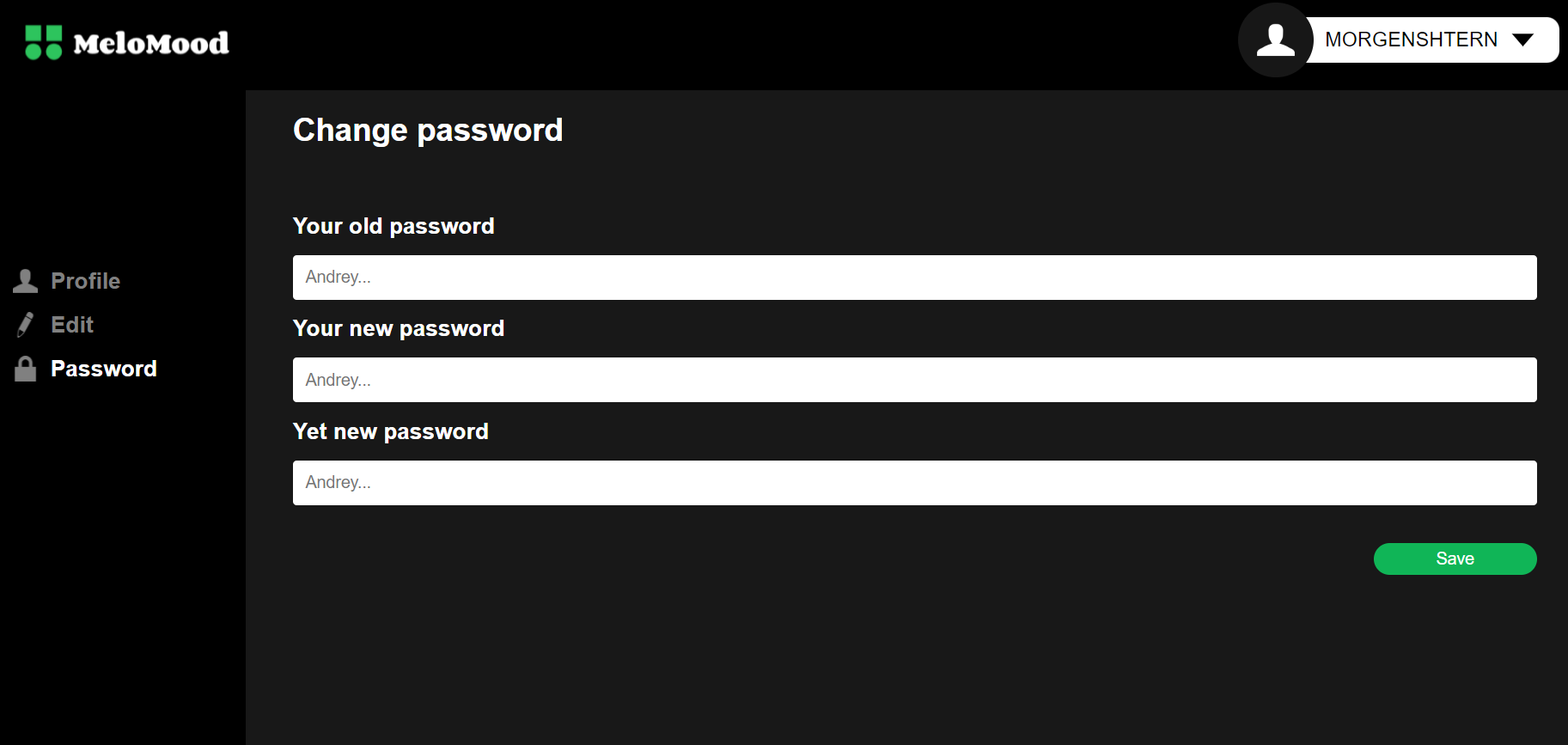


Рисунок 6.8 – Страница Profile

Если пользователь вошел под ролью manager, он будет перенаправлен на страницу Dashboard, где будет отображаться общая статистика по использованию музыкальной платформы. На этой странице manager сможет увидеть информацию о количестве зарегистрированных пользователей, количестве загруженных треков и жанров. Эта информация поможет manager отслеживать активность пользователей и принимать решения для улучшения пользовательского опыта. Страница "Dashboard" представлена на рисунке 6.9.

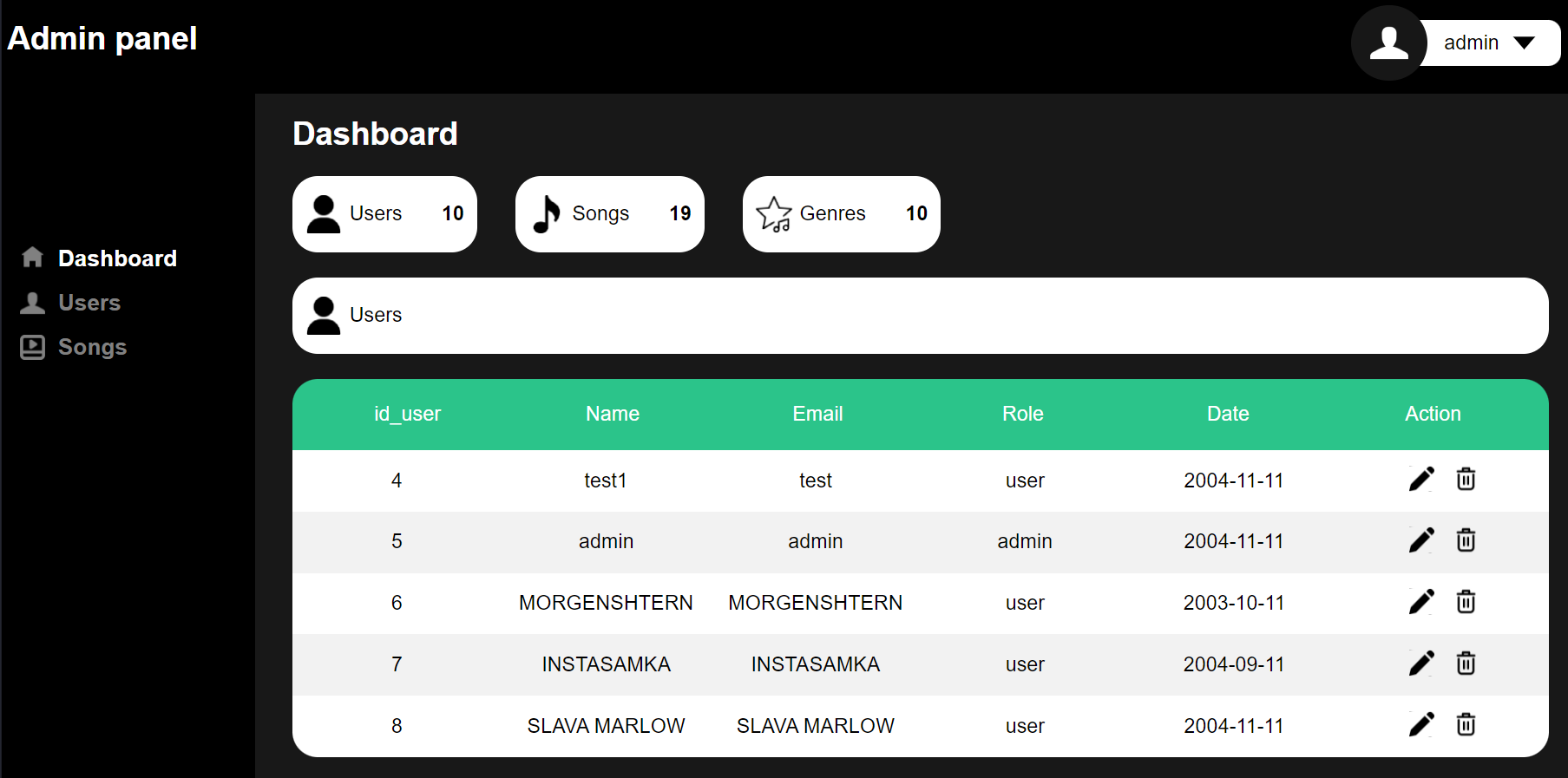


Рисунок 6.9 – Страница Dashboard

На странице Users manager может управлять пользователями. Он имеет возможность просмотреть список всех пользователей, а также добавить, удалить или отредактировать данные любого пользователя. Это дает manager полный контроль над аккаунтами пользователей и позволяет эффективно управлять доступом к ресурсам. Страница "Users" представлена на рисунке 6.10.

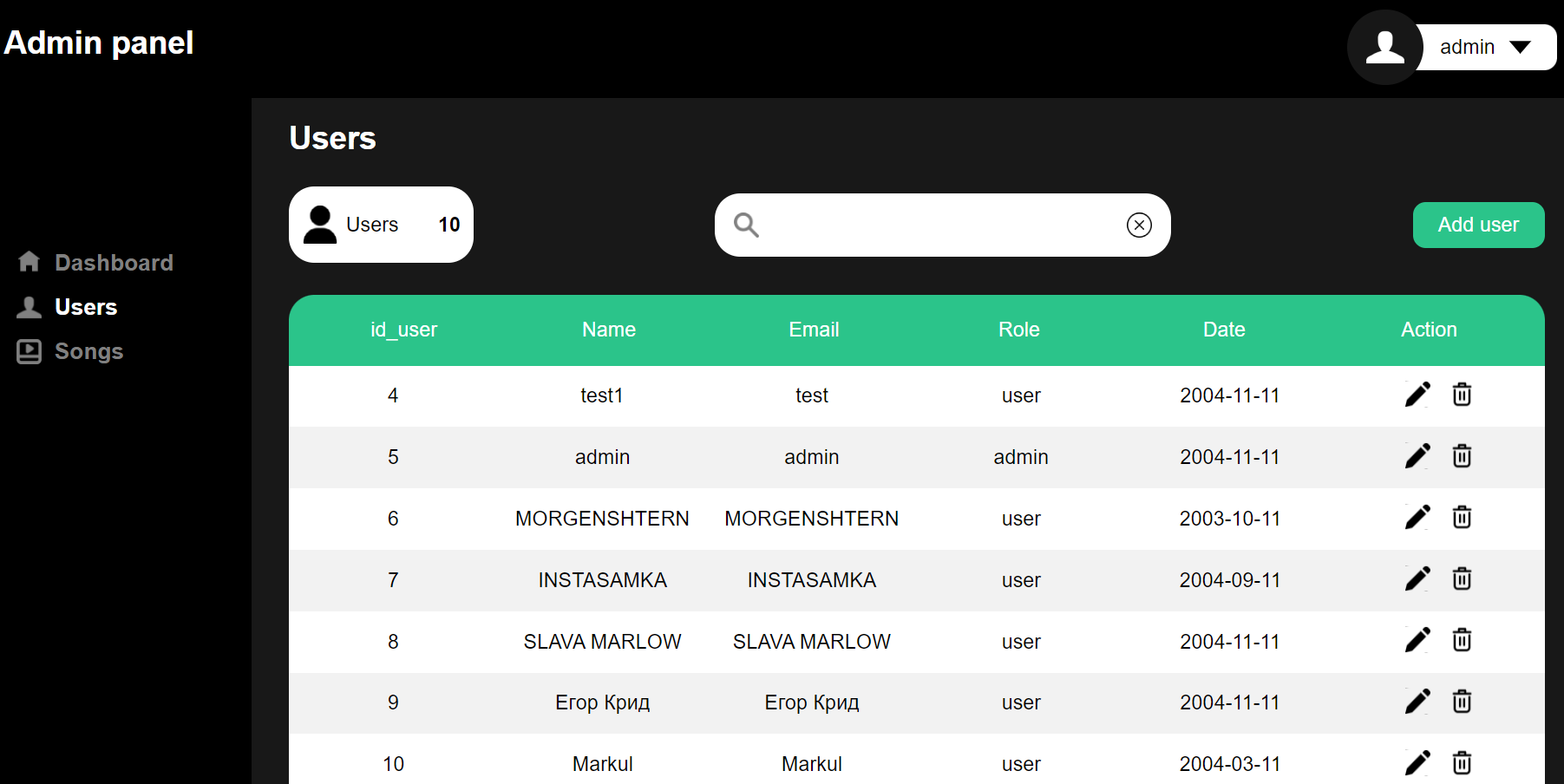


Рисунок 6.11 – Страница Users

На странице Songs manager имеет возможность просматривать все доступные треки, а также добавлять, удалять и редактировать их. Страница "Songs" представлена на рисунке 6.12.

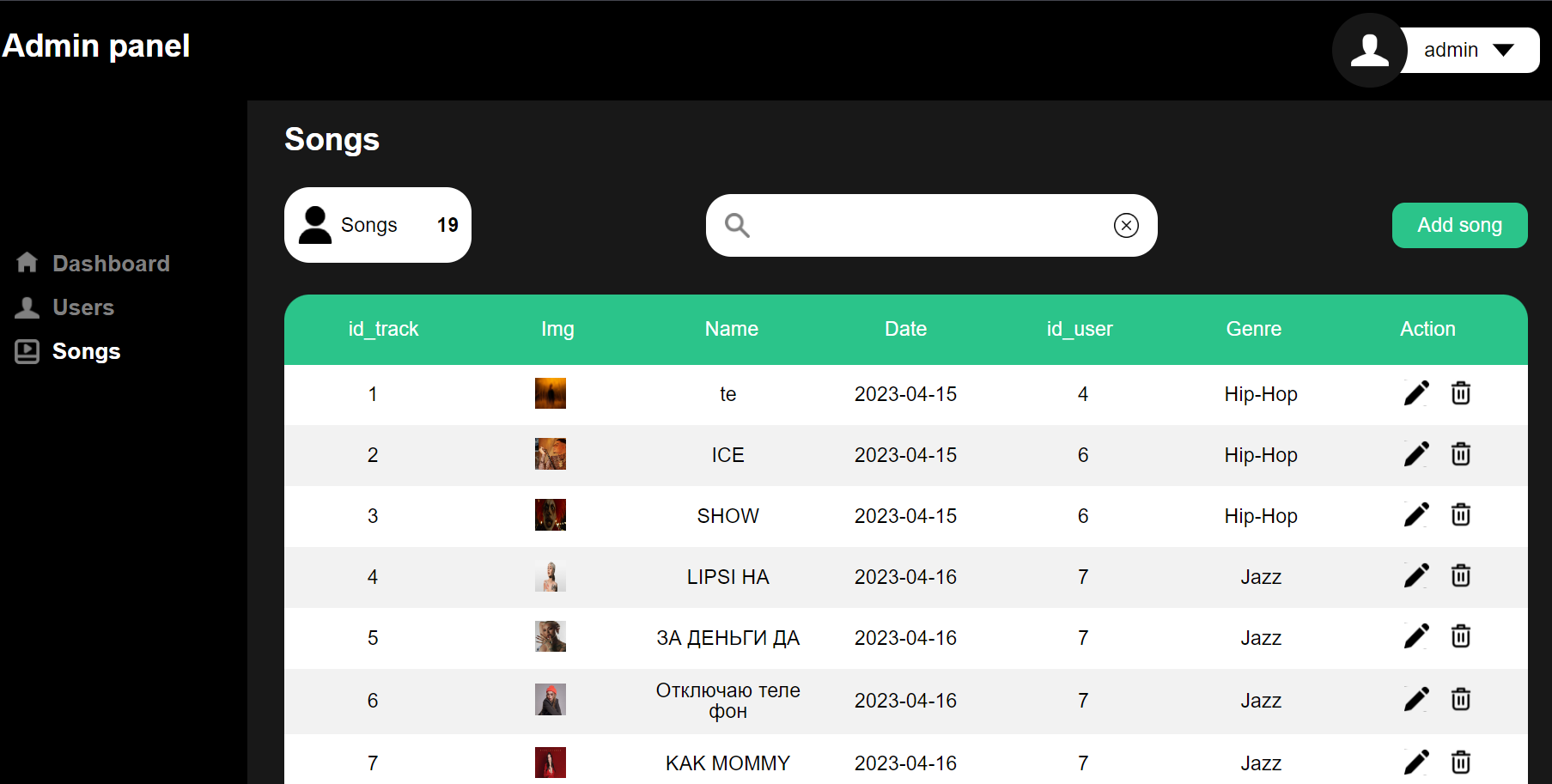


Рисунок 6.12 – Страница Users

В проекте был протестирован весь базовый функционал userа и managera приложения. Все компоненты работают исправно.

## 6.2 Установка приложения

Для запуска приложения необходимо выполнить следующие шаги:

1. Запустить серверную часть приложения, которая соединяет базу данных и React приложение. Для этого необходимо запустить скрипт, который настроит соединение с базой данных и запустит сервер.

2. Запустить React приложение, которое будет обрабатывать пользовательские запросы и взаимодействовать с сервером. Для этого необходимо запустить команду для сборки и запуска React приложения.

После выполнения этих шагов приложение будет полностью готово к работе и пользователь сможет начать использовать его функционал.

## 6.3 Вывод

В данном разделе были рассмотрены функциональные возможности приложения, а также права доступа для пользователей с различными ролями. Было показано, что наша платформа позволяет пользователям создавать плейлисты, искать треки по названию или исполнителю, а также изменять свой профиль и пароль.

Кроме того, администратор имеет доступ к управлению пользователями, треками и жанрами. Для запуска приложения необходимо запустить серверную часть, которая соединяет базу данных и React-приложение.

# Заключение

База данных является ключевым элементом любой современной организации, обеспечивая надежное хранение и управление информацией. В данной работе была поставлена задача разработки базы данных для музыкальной площадки с использованием технологии применения мультимедийных типов данных в СУБД PostgreSQL.

В процессе выполнения работы были использованы различные объекты, включая таблицы, триггеры и функции, чтобы обеспечить структурированное хранение данных и своевременный доступ к ним. В результате, цель работы была успешно достигнута, и база данных готова к использованию. Были разработаны роли для управления доступом к данным и обеспечения безопасности.

Тестирование базы данных было проведено при использовании большого объема данных, и результаты были положительными. Были реализованы процедуры для импорта и экспорта данных в формате JSON и XML, что обеспечило удобство использования и управления данными.

Одной из ключевых особенностей разработанной базы данных является технология хранения мультимедийных данных, что позволяет эффективно управлять медиа-файлами на площадке.

Таким образом, была успешно выполнена задача по разработке базы данных для музыкальной площадки на основе СУБД PostgreSQL. Разработанная база данных позволяет хранить и управлять большим объемом музыкальных данных, обеспечивает безопасный доступ к ним и предоставляет возможность импорта и экспорта данных в различных форматах. Кроме того, технология хранения мультимедийных данных позволяет эффективно управлять медиа-файлами на площадке.

# Список литературных источников

1. Spotify [Электронный ресурс] / Режим доступа: https://spotify.com – Дата доступа: 18.04.2023.
2. Apple Music [Электронный ресурс] / Режим доступа: https://applemusic.com/ – Дата доступа: 18.04.2023.
3. PostgreSQL Сайт о программировании [Электронный ресурс] / Режим доступа: https://postgrespro.ru/docs/postgresql.com – Дата доступа: 18.04.2023.
4. Postgresqltutorial.com [Электронный ресурс] / Режим доступа: https://www.postgresqltutorial.com/ – Дата доступа: 18.04.2023.
5. Stackoverflow.com [Электронный ресурс] / Режим доступа: https://stackoverflow.com – Дата доступа: 18.04.2023.

# Приложение А

|  |
| --- |
| --------------------- TABLESPACE ---------------------------  CREATE TABLESPACE TS\_USER LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_USER';  CREATE TABLESPACE TS\_TRACK LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_TRACK';  CREATE TABLESPACE TS\_PLAYLIST LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_PLAYLIST';  CREATE TABLESPACE TS\_LIBRARY LOCATION 'E:/POSTGRESQL/14/DATA/DBS/TS\_LIBRARY';  /\*---------------------————————————————---------------------  -----------------------| CREATE TABLES |--------------------  ------------------------————————————————-------------------\*/  --------------------- TABLE ROLE ----------------------  CREATE TABLE ROLE (  ROLE\_ID SERIAL PRIMARY KEY,  ROLE\_NAME VARCHAR(255) NOT NULL  ) TABLESPACE TS\_USER;  --------------------- TABLE USERS ----------------------  CREATE TABLE USERS (  USER\_ID SERIAL PRIMARY KEY,  USER\_NAME VARCHAR(255) NOT NULL,  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255) UNIQUE NOT NULL,  USER\_PASSWORD VARCHAR(255) NOT NULL,  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER,  CONSTRAINT FK\_USER\_ROLE FOREIGN KEY (USER\_ROLE\_ID) REFERENCES ROLE (ROLE\_ID)  ) TABLESPACE TS\_USER;  --------------------- TABLE GENRE ----------------------  CREATE TABLE GENRE (  GENRE\_ID SERIAL PRIMARY KEY,  GENRE\_NAME VARCHAR(255) NOT NULL  ) TABLESPACE TS\_TRACK;  --------------------- TABLE TRACK ----------------------  CREATE TABLE TRACK (  TRACK\_ID SERIAL PRIMARY KEY,  TRACK\_TITLE VARCHAR(255) NOT NULL,  TRACK\_DATE DATE NOT NULL,  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA NOT NULL,  GENRE\_ID INTEGER NOT NULL,  CONSTRAINT FK\_TRACK\_GENRE FOREIGN KEY (GENRE\_ID) REFERENCES GENRE (GENRE\_ID)  ) TABLESPACE TS\_TRACK;  --------------------- TABLE USER\_TRACK ----------------------  CREATE TABLE USER\_TRACK (  USER\_ID INTEGER NOT NULL,  TRACK\_ID INTEGER NOT NULL,  PRIMARY KEY (USER\_ID, TRACK\_ID),  CONSTRAINT FK\_USER\_TRACK\_USER FOREIGN KEY (USER\_ID) REFERENCES USERS (USER\_ID),  CONSTRAINT FK\_USER\_TRACK\_TRACK FOREIGN KEY (TRACK\_ID) REFERENCES TRACK (TRACK\_ID)  ) TABLESPACE TS\_TRACK;  --------------------- TABLE RATING----------------------  CREATE TABLE RATING (  RATING\_ID SERIAL PRIMARY KEY,  USER\_ID INTEGER NOT NULL,  TRACK\_ID INTEGER NOT NULL,  RATE INTEGER NOT NULL,  CONSTRAINT FK\_RATING\_USER FOREIGN KEY (USER\_ID) REFERENCES USERS (USER\_ID),  CONSTRAINT FK\_RATING\_TRACK FOREIGN KEY (TRACK\_ID) REFERENCES TRACK (TRACK\_ID)  ) TABLESPACE TS\_TRACK;  --------------------- TABLE PLAYLIST ----------------------  CREATE TABLE PLAYLIST (  PLAYLIST\_ID SERIAL PRIMARY KEY,  USER\_ID INTEGER NOT NULL,  TITLE VARCHAR(255) NOT NULL,  CONSTRAINT FK\_PLAYLIST\_USER FOREIGN KEY (USER\_ID) REFERENCES USERS (USER\_ID)  ) TABLESPACE TS\_PLAYLIST;  --------------------- TABLE PLAYLIST\_TRACKS ----------------------  CREATE TABLE PLAYLIST\_TRACKS (  ID SERIAL PRIMARY KEY,  TRACK\_ID INTEGER NOT NULL,  PLAYLIST\_ID INTEGER NOT NULL,  CONSTRAINT FK\_PLAYLIST\_TRACKS\_TRACK FOREIGN KEY (TRACK\_ID) REFERENCES TRACK (TRACK\_ID),  CONSTRAINT FK\_PLAYLIST\_TRACKS\_PLAYLIST FOREIGN KEY (PLAYLIST\_ID) REFERENCES PLAYLIST (PLAYLIST\_ID)  ) TABLESPACE TS\_PLAYLIST;  --------------------- TABLE LIBRARY\_USER ----------------------  CREATE TABLE LIBRARY\_USER (  ID SERIAL PRIMARY KEY,  USER\_ID INTEGER NOT NULL,  TRACK\_ID INTEGER NOT NULL,  CONSTRAINT FK\_LIBRARY\_TRACK\_USER FOREIGN KEY (USER\_ID) REFERENCES USERS (USER\_ID),  CONSTRAINT FK\_LIBRARY\_TRACK\_TRACK FOREIGN KEY (TRACK\_ID) REFERENCES TRACK (TRACK\_ID)  ) TABLESPACE TS\_LIBRARY; |
|  |

Листинг 1 – Скрипты создания таблиц

|  |
| --- |
| -------------------------------------  ---------- CREATE ROLE --------------  -------------------------------------  CREATE ROLE PROGRAMMER\_ROLE;  CREATE ROLE USER\_ROLE;  CREATE ROLE MANAGER\_ROLE;  -------------- PROGRAMMER\_ROLE ------------------  GRANT ALL PRIVILEGES ON DATABASE "BD\_MUSIC" TO PROGRAMMER\_ROLE;  GRANT CONNECT ON DATABASE "BD\_MUSIC" TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_USER TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_TRACK TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_PLAYLIST TO PROGRAMMER\_ROLE;  GRANT ALL ON TABLESPACE TS\_LIBRARY TO PROGRAMMER\_ROLE;  -------------- USER\_ROLE ------------------  GRANT EXECUTE ON FUNCTION REGISTER TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION LOGIN TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_RATING TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION CREATEPLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE ADD\_TRACK\_TO\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE REMOVE\_TRACK\_FROM\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_PLAYLIST TO USER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK\_FROM\_LIBRARY TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO USER\_ROLE;  GRANT EXECUTE ON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE RATING TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE PLAYLIST TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE LIBRARY\_USER TO USER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE PLAYLIST\_TRACKS TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE USERS\_USER\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE TRACK\_TRACK\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE RATING\_RATING\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE PLAYLIST\_PLAYLIST\_ID\_SEQ TO USER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE LIBRARY\_USER\_ID\_SEQ TO USER\_ROLE;  -------------- MANAGER\_ROLE ------------------  GRANT EXECUTE ON FUNCTION REGISTER TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERBYID TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION ADDTRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETUSERS TO MANAGER\_ROLE;  GRANT EXECUTE ON FUNCTION GETTRACKBYID TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_USER\_PASSWORD TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_TRACK TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE DELETE\_USER TO MANAGER\_ROLE;  GRANT EXECUTE ON PROCEDURE UPDATE\_TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE ON TABLE USERS TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE TRACK TO MANAGER\_ROLE;  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE USER\_TRACK TO MANAGER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE USERS\_USER\_ID\_SEQ TO MANAGER\_ROLE;  GRANT USAGE, SELECT ON SEQUENCE TRACK\_TRACK\_ID\_SEQ TO MANAGER\_ROLE;  -------------------------------------  ---------- CREATE USER --------------  -------------------------------------  CREATE USER PROGRAMMER\_1 PASSWORD '123';  GRANT PROGRAMMER\_ROLE TO PROGRAMMER\_1;  CREATE USER USER\_1 WITH PASSWORD '123';  GRANT USER\_ROLE TO USER\_1;  CREATE USER MANAGER\_1 WITH PASSWORD '123';  GRANT MANAGER\_ROLE TO MANAGER\_1; |

Листинг 2 – Скрипты создание ролей и пользователей

|  |
| --- |
| /\*---------------------------——————————----------------------------  ----------------------------| FUNCTIONS |-------------------------  -----------------------------——————————--------------------------\*/  ----------------------- ENCRYPT PASSWORD -----------------------  CREATE OR REPLACE FUNCTION ENCRYPT\_PASSWORD(PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232'; --KEY  KEY\_LENGTH INTEGER := LENGTH(KEY);  ENCRYPTED\_PASSWORD TEXT := '';  BEGIN  IF PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(PASSWORD) LOOP  ENCRYPTED\_PASSWORD := ENCRYPTED\_PASSWORD || CHR(ASCII(SUBSTRING(PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN ENCRYPTED\_PASSWORD;  END;  $$ LANGUAGE PLPGSQL;  ----------------------- DECRYPT PASSWORD -----------------------  CREATE OR REPLACE FUNCTION DECRYPT\_PASSWORD(ENCRYPTED\_PASSWORD TEXT)  RETURNS TEXT  AS $$  DECLARE  I INTEGER := 1;  KEY TEXT := '323232';  KEY\_LENGTH INTEGER := LENGTH(KEY);  PASSWORD TEXT := '';  BEGIN  IF ENCRYPTED\_PASSWORD IS NULL THEN  RETURN NULL;  END IF;  FOR I IN 1..LENGTH(ENCRYPTED\_PASSWORD) LOOP  PASSWORD := PASSWORD || CHR(ASCII(SUBSTRING(ENCRYPTED\_PASSWORD, I, 1)) # ASCII(SUBSTRING(KEY, I % KEY\_LENGTH + 1, 1)));  END LOOP;  RETURN PASSWORD;  END;  $$ LANGUAGE PLPGSQL;  ----------------------- AVERAGE RATING -----------------------  CREATE OR REPLACE FUNCTION AVERAGE\_RATING(TRACK\_ID INTEGER) RETURNS NUMERIC AS $$  DECLARE  TOTAL\_RATING INTEGER;  NUM\_RATINGS INTEGER;  BEGIN  SELECT SUM(RATE), COUNT(\*) INTO TOTAL\_RATING, NUM\_RATINGS  FROM RATING  WHERE RATING.TRACK\_ID = $1;  IF NUM\_RATINGS = 0 THEN  RETURN 0;  END IF;  RETURN TOTAL\_RATING::NUMERIC / NUM\_RATINGS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 3 – Скрипты создания дополнительных функций

|  |
| --- |
| /\*--------------------------——————————----------------------------  ----------------------------| CREATE |----------------------------  ----------------------------——————————--------------------------\*/  -------------------- REGISTER USER --------------------  CREATE OR REPLACE FUNCTION REGISTER(  IN\_USER\_NAME VARCHAR(255),  IN\_USER\_EMAIL VARCHAR(255),  IN\_USER\_PASSWORD TEXT,  IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_ROLE\_ID INTEGER,  IN\_USER\_IMG BYTEA  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  ENCRYPTED\_PASSWORD TEXT := ENCRYPT\_PASSWORD(IN\_USER\_PASSWORD);  USER\_ID INTEGER;  BEGIN  INSERT INTO USERS(USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID, USER\_IMG)  VALUES (IN\_USER\_NAME, IN\_USER\_EMAIL, ENCRYPTED\_PASSWORD, IN\_USER\_DATE\_OF\_BIRTH, IN\_USER\_ROLE\_ID, IN\_USER\_IMG)  RETURNING USERS.USER\_ID INTO USER\_ID;  RETURN USER\_ID;  END;  $$;  SELECT REGISTER('ADMIN', 'ADMIN', 'ADMIN', '1995-01-01', 1, '\X1234567890ABCDEF');  SELECT \*FROM USERS  SELECT REGISTER('ANS', 'ANS', 'ANS', '1995-01-01', 2, '\X1234567890ABCDEF');  --------------------- LOGIN USER---------------------  CREATE OR REPLACE FUNCTION LOGIN(IN\_USER\_EMAIL VARCHAR(255), IN\_USER\_PASSWORD TEXT)  RETURNS BOOLEAN  AS $$  DECLARE  STORED\_PASSWORD TEXT;  BEGIN  SELECT USER\_PASSWORD INTO STORED\_PASSWORD FROM USERS WHERE USER\_EMAIL = IN\_USER\_EMAIL;    IF STORED\_PASSWORD IS NULL THEN  RETURN FALSE;  END IF;  RETURN STORED\_PASSWORD = ENCRYPT\_PASSWORD(IN\_USER\_PASSWORD);  END;  $$ LANGUAGE PLPGSQL;  SELECT LOGIN('ADMIN', 'ADMIN');  SELECT \* FROM USERS  -------------------- ADD TRACK --------------------  CREATE OR REPLACE FUNCTION ADDTRACK(  IN\_TRACK\_TITLE VARCHAR(255),  IN\_TRACK\_DATE DATE,  IN\_TRACK\_IMAGE BYTEA,  IN\_TRACK\_CONTENT BYTEA,  IN\_GENRE\_ID INTEGER  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  TRACK\_ID INTEGER;  BEGIN  INSERT INTO TRACK(TRACK\_TITLE, TRACK\_DATE, TRACK\_IMAGE, TRACK\_CONTENT, GENRE\_ID)  VALUES (IN\_TRACK\_TITLE, IN\_TRACK\_DATE, IN\_TRACK\_IMAGE, IN\_TRACK\_CONTENT, IN\_GENRE\_ID)  RETURNING TRACK.TRACK\_ID INTO TRACK\_ID;    RETURN TRACK\_ID;  END;  $$;  SELECT ADDTRACK('НАЗВАНИЕ ТРЕКА', '2023-03-15', NULL, 'СОДЕРЖИМОЕ ТРЕКА', 1);  CALL ADDTRACK('НАЗВАНИЕ ТРЕКА', '2023-03-15', NULL, 'СОДЕРЖИМОЕ ТРЕКА', 1);  CALL ADDUSERTRACK(3,1)  -------------------- ADD TRACK USER --------------------  CREATE OR REPLACE PROCEDURE ADDUSERTRACK(  IN\_USER\_ID INTEGER,  IN\_TRACK\_ID INTEGER  )  LANGUAGE PLPGSQL  AS $$  BEGIN  INSERT INTO USER\_TRACK(USER\_ID, TRACK\_ID)  VALUES (IN\_USER\_ID, IN\_TRACK\_ID);  END;  $$;  SELECT \* FROM USER\_TRACK  -------------------- ADD RATING --------------------  CREATE OR REPLACE PROCEDURE ADD\_RATING(  IN\_USER\_ID INTEGER,  IN\_TRACK\_ID INTEGER,  IN\_RATE INTEGER  )  LANGUAGE PLPGSQL  AS $$  BEGIN    IF NOT EXISTS (SELECT 1 FROM USERS WHERE USER\_ID = IN\_USER\_ID) THEN  RAISE EXCEPTION 'USER WITH ID % DOES NOT EXIST', IN\_USER\_ID;  END IF;    IF NOT EXISTS (SELECT 1 FROM TRACK WHERE TRACK\_ID = IN\_TRACK\_ID) THEN  RAISE EXCEPTION 'TRACK WITH ID % DOES NOT EXIST', IN\_TRACK\_ID;  END IF;    INSERT INTO RATING(USER\_ID, TRACK\_ID, RATE)  VALUES (IN\_USER\_ID, IN\_TRACK\_ID, IN\_RATE);  END;  $$;  -------------------- CREATE PLAYLIST --------------------  CREATE OR REPLACE FUNCTION CREATEPLAYLIST(  IN\_USER\_ID INTEGER,  IN\_TITLE VARCHAR(255)  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  PLAYLIST\_ID INTEGER;  BEGIN  INSERT INTO PLAYLIST(USER\_ID, TITLE)  VALUES (IN\_USER\_ID, IN\_TITLE)  RETURNING PLAYLIST.PLAYLIST\_ID INTO PLAYLIST\_ID;    RETURN PLAYLIST\_ID;  END;  $$;  SELECT CREATEPLAYLIST(43,'MU LIST')  SELECT CREATEPLAYLIST(1, 'MY PLAYLIST');  SELECT \* FROM PLAYLIST;  -------------------- ADD TRACK IN PLAYLIST --------------------  CREATE OR REPLACE PROCEDURE ADD\_TRACK\_TO\_PLAYLIST(  IN\_TRACK\_ID INTEGER,  IN\_PLAYLIST\_ID INTEGER  )  LANGUAGE PLPGSQL  AS $$  BEGIN  INSERT INTO PLAYLIST\_TRACKS(TRACK\_ID, PLAYLIST\_ID)  VALUES (IN\_TRACK\_ID, IN\_PLAYLIST\_ID);  END;  $$;  CALL ADD\_TRACK\_TO\_PLAYLIST(2,1);  SELECT \* FROM PLAYLIST\_TRACKS;  -------------------- ADD TRACK IN LIBRARY --------------------  CREATE OR REPLACE PROCEDURE ADD\_TRACK\_TO\_LIBRARY(  IN\_USER\_ID INTEGER,  IN\_TRACK\_ID INTEGER  )  LANGUAGE PLPGSQL  AS $$  BEGIN  INSERT INTO LIBRARY\_USER(USER\_ID, TRACK\_ID) VALUES (IN\_USER\_ID, IN\_TRACK\_ID);    END;  $$;  CALL ADD\_TRACK\_TO\_LIBRARY(1,1)  DROP PROCEDURE ADD\_TRACK\_TO\_LIBRARY  SELECT \* FROM LIBRARY\_USER;  /\*--------------------------——————————----------------------------  ----------------------------| UPDATE |----------------------------  ----------------------------——————————--------------------------\*/  -------------------- UPDATE USER --------------------  CREATE OR REPLACE PROCEDURE UPDATE\_USER(IN\_USER\_ID INTEGER,  IN\_USER\_NAME VARCHAR(255),  IN\_USER\_DATE\_OF\_BIRTH DATE,  IN\_USER\_IMG BYTEA) LANGUAGE PLPGSQL AS $$  BEGIN  IF IN\_USER\_IMG IS NOT NULL THEN  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH,  USER\_IMG = IN\_USER\_IMG  WHERE USER\_ID = IN\_USER\_ID;  ELSE  UPDATE USERS  SET USER\_NAME = IN\_USER\_NAME,  USER\_DATE\_OF\_BIRTH = IN\_USER\_DATE\_OF\_BIRTH  WHERE USER\_ID = IN\_USER\_ID;  END IF;  END;  $$;  -------------------- UPDATE USER PASSWORD --------------------  CREATE OR REPLACE PROCEDURE UPDATE\_USER\_PASSWORD(IN\_USER\_ID INTEGER, IN\_USER\_PASSWORD TEXT) LANGUAGE PLPGSQL AS $$  DECLARE  ENCRYPTED\_PASSWORD TEXT := ENCRYPT\_PASSWORD(IN\_USER\_PASSWORD);  BEGIN  UPDATE USERS  SET USER\_PASSWORD = ENCRYPTED\_PASSWORD  WHERE USER\_ID = IN\_USER\_ID;  END;  $$;  -------------------- UPDATE TRACK --------------------  CREATE OR REPLACE PROCEDURE UPDATE\_TRACK(  IN\_TRACK\_ID INTEGER,  IN\_TRACK\_TITLE VARCHAR(255),  IN\_TRACK\_IMAGE BYTEA,  IN\_GENRE\_ID INTEGER  ) LANGUAGE PLPGSQL AS $$  BEGIN    IF IN\_TRACK\_TITLE IS NOT NULL THEN  UPDATE TRACK  SET  TRACK\_TITLE = IN\_TRACK\_TITLE  WHERE TRACK.TRACK\_ID = IN\_TRACK\_ID;  END IF;  IF IN\_TRACK\_IMAGE IS NOT NULL THEN  UPDATE TRACK  SET  TRACK\_IMAGE = IN\_TRACK\_IMAGE  WHERE TRACK.TRACK\_ID = IN\_TRACK\_ID;  END IF;  IF IN\_GENRE\_ID IS NOT NULL THEN  UPDATE TRACK  SET  GENRE\_ID = IN\_GENRE\_ID  WHERE TRACK.TRACK\_ID = IN\_TRACK\_ID;  END IF;    END;  $$;  SELECT \* FROM TRACK WHERE TRACK\_ID = 2  CALL UPDATE\_TRACK(2, NULL,NULL,NULL);  -------------------- UPDATE TRACK TITLE --------------------  CREATE OR REPLACE PROCEDURE UPDATE\_TRACK\_TITLE(  IN\_TRACK\_ID INTEGER,  IN\_TRACK\_TITLE VARCHAR(255))  LANGUAGE PLPGSQL AS $$  BEGIN  UPDATE TRACK  SET  TRACK\_TITLE = IN\_TRACK\_TITLE  WHERE TRACK.TRACK\_ID = IN\_TRACK\_ID;  END;  $$;  CALL UPDATE\_TRACK\_TITLE(4,'TST')  SELECT \* FROM TRACK WHERE TRACK.TRACK\_ID =4  SELECT USER\_ID,TITLE FROM PLAYLIST WHERE PLAYLIST\_ID = 4  SELECT \*  FROM TRACK CALL UPDATE\_TRACK(4, 'NEW TITLE', '2022-01-01', 1, NULL, 'NEW CONTENT', 3);  -------------------- UPDATE PLAYLIST TITLE --------------------  CREATE OR REPLACE PROCEDURE UPDATE\_PLAYLIST  (IN\_PLAYLIST\_ID INTEGER,  IN\_TITLE VARCHAR(255))  LANGUAGE PLPGSQL AS $$  BEGIN  UPDATE PLAYLIST SET TITLE = IN\_TITLE WHERE PLAYLIST\_ID = IN\_PLAYLIST\_ID;  END;  $$;  SELECT \* FROM PLAYLIST  /\*--------------------------——————————----------------------------  ----------------------------| DELETE |----------------------------  ----------------------------——————————--------------------------\*/  -------------------- DELETE USER --------------------  SELECT \* FROM  TRACK  DELETE FROM TRACK  CALL DELETE\_USER(6)  CREATE OR REPLACE PROCEDURE DELETE\_USER(IN\_USER\_ID INTEGER) LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM RATING WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM LIBRARY\_USER WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USER\_TRACK WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_ID IN (SELECT PLAYLIST\_ID FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID);  DELETE FROM PLAYLIST WHERE USER\_ID = IN\_USER\_ID;  DELETE FROM USERS WHERE USER\_ID = IN\_USER\_ID;  END;  $$;  SELECT \*  FROM USERS CALL DELETE\_USER(5);  -------------------- DELETE TRACK --------------------  CREATE OR REPLACE PROCEDURE DELETE\_TRACK( IN\_TRACK\_ID INTEGER) LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM LIBRARY\_USER WHERE TRACK\_ID = IN\_TRACK\_ID;  DELETE FROM RATING WHERE TRACK\_ID = IN\_TRACK\_ID;  DELETE FROM PLAYLIST\_TRACKS WHERE TRACK\_ID = IN\_TRACK\_ID;  DELETE FROM USER\_TRACK WHERE TRACK\_ID = IN\_TRACK\_ID;  DELETE FROM TRACK WHERE TRACK\_ID = IN\_TRACK\_ID;  END;  $$;  CALL DELETE\_TRACK(12)  SELECT \* FROM TRACK  -------------------- DELETE TRACK FROM PLAYLIST --------------------  CREATE OR REPLACE PROCEDURE REMOVE\_TRACK\_FROM\_PLAYLIST  ( IN\_PLAYLIST\_ID INTEGER, IN\_TRACK\_ID INTEGER)  LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_ID = IN\_PLAYLIST\_ID AND TRACK\_ID = IN\_TRACK\_ID;  END;  $$;  CALL REMOVE\_TRACK\_FROM\_PLAYLIST(11,4)  SELECT \* FROM PLAYLIST\_TRACKS  -------------------- DELETE PLAYLIST --------------------  CREATE OR REPLACE PROCEDURE DELETE\_PLAYLIST( IN\_PLAYLIST\_ID INTEGER) LANGUAGE PLPGSQL AS $$  BEGIN  DELETE FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_ID = IN\_PLAYLIST\_ID;  DELETE FROM PLAYLIST WHERE PLAYLIST\_ID = IN\_PLAYLIST\_ID;  END;  $$;  CALL DELETE\_PLAYLIST(24)  SELECT \* FROM PLAYLIST  -------------------- DELETE TRACK FROM LIBRARY --------------------  CREATE OR REPLACE PROCEDURE DELETE\_TRACK\_FROM\_LIBRARY( USER\_ID\_PARAM INTEGER, TRACK\_ID\_PARAM INTEGER) AS $$  BEGIN  DELETE FROM LIBRARY\_USER  WHERE USER\_ID = USER\_ID\_PARAM AND TRACK\_ID = TRACK\_ID\_PARAM;  END;  $$ LANGUAGE PLPGSQL;  --------------------------------------------  CREATE OR REPLACE FUNCTION ADDTRACKSTOPLAYLISTHOME(  IN\_GENRE\_NAME VARCHAR(255),  IN\_USER\_ID INTEGER,  IN\_TITLE VARCHAR(255)  )  RETURNS INTEGER  LANGUAGE PLPGSQL  AS $$  DECLARE  PLAYLISTID INTEGER;  TRACK\_INFO RECORD;  TRACKID INTEGER;  I INTEGER;  BEGIN  -- СОЗДАТЬ НОВЫЙ ПЛЕЙЛИСТ  INSERT INTO PLAYLIST(USER\_ID, TITLE)  VALUES (IN\_USER\_ID, IN\_TITLE)  RETURNING PLAYLIST.PLAYLIST\_ID INTO PLAYLISTID;  -- ДОБАВИТЬ В НЕГО 10 ЛУЧШИХ ТРЕКОВ ЗАДАННОГО ЖАНРА  FOR TRACK\_INFO IN (  SELECT \*  FROM ALL\_INFO\_TRACK  WHERE ALL\_INFO\_TRACK.GENRE\_NAME = IN\_GENRE\_NAME  ORDER BY ALL\_INFO\_TRACK.AVG\_RATING DESC  LIMIT 10  ) LOOP  SELECT TRACK\_INFO.TRACK\_ID INTO TRACKID;  CALL ADD\_TRACK\_TO\_PLAYLIST(TRACKID, PLAYLISTID);  END LOOP;  RETURN PLAYLISTID;  END;  $$;  SELECT \* FROM GENRE  SELECT ADDTRACKSTOPLAYLISTHOME('HIP-HOP', 5, 'CAR') AS PLAYLIST\_ID;  SELECT ADDTRACKSTOPLAYLIST(PLAYLIST\_ID, 'GENRE\_NAME', USER\_ID); |

Листинг 4 – Скрипты создание функций CRUD

|  |
| --- |
| /\*--------------------------———————------------------------------  ----------------------------| VIEW |-----------------------------  ----------------------------———————----------------------------\*/  -------------------- INFO USER --------------------  CREATE OR REPLACE VIEW ALL\_INFO\_USER AS  SELECT USERS.USER\_ID, USER\_NAME, USER\_IMG,  USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, ROLE.ROLE\_NAME  FROM USERS  JOIN ROLE ON USERS.USER\_ROLE\_ID = ROLE.ROLE\_ID;  DROP VIEW ALL\_INFO\_USER;  -------------------- INFO TRACK --------------------  CREATE OR REPLACE VIEW ALL\_INFO\_TRACK AS  SELECT T.TRACK\_ID, T.TRACK\_TITLE, T.TRACK\_DATE,  G.GENRE\_NAME, T.TRACK\_IMAGE, T.TRACK\_CONTENT,  AVERAGE\_RATING(T.TRACK\_ID) AS AVG\_RATING  FROM TRACK T  JOIN GENRE G ON T.GENRE\_ID = G.GENRE\_ID;  SELECT \* FROM ALL\_INFO\_TRACK  DROP VIEW ALL\_INFO\_TRACK  SELECT \* FROM ALL\_INFO\_TRACK WHERE TRACK\_ID = 4;  -------------------- PLAYLIST --------------------  CREATE VIEW PLAYLIST\_TRACKS\_INFO AS  SELECT T.TRACK\_ID, T.TRACK\_TITLE, T.TRACK\_DATE,  G.GENRE\_NAME, T.TRACK\_IMAGE, T.TRACK\_CONTENT,  AVERAGE\_RATING(T.TRACK\_ID) AS AVG\_RATING, PT.PLAYLIST\_ID  FROM PLAYLIST\_TRACKS PT  JOIN TRACK T ON PT.TRACK\_ID = T.TRACK\_ID  JOIN GENRE G ON T.GENRE\_ID = G.GENRE\_ID;  SELECT \*  FROM PLAYLIST\_TRACKS\_INFO  WHERE PLAYLIST\_ID = 8 |

Листинг 5 – Скрипты создание представлений

|  |
| --- |
| ----------------------------------------  ------------- TRIGGERS -----------------  ----------------------------------------  ---------------- USER ------------------  CREATE OR REPLACE FUNCTION CHECK\_DATE\_OF\_BIRTH() RETURNS TRIGGER AS $$  BEGIN  IF NEW.USER\_DATE\_OF\_BIRTH >= CURRENT\_DATE THEN  RAISE EXCEPTION 'INVALID DATE OF BIRTH';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_DATE\_OF\_BIRTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_DATE\_OF\_BIRTH();  CREATE FUNCTION CHECK\_PASSWORD\_LENGTH() RETURNS TRIGGER AS $$  BEGIN  IF CHAR\_LENGTH(NEW.USER\_PASSWORD) < 6 THEN  RAISE EXCEPTION 'PASSWORD MUST BE AT LEAST 6 CHARACTERS LONG';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_PASSWORD\_LENGTH  BEFORE INSERT OR UPDATE ON USERS  FOR EACH ROW  EXECUTE FUNCTION CHECK\_PASSWORD\_LENGTH();  ---------------- TRACK ------------------  CREATE FUNCTION CHECK\_TRACK\_DATE() RETURNS TRIGGER AS $$  BEGIN  IF NEW.TRACK\_DATE > CURRENT\_DATE THEN  RAISE EXCEPTION 'THE TRACK DATE CANNOT BE IN THE FUTURE.';  END IF;  RETURN NEW;  END;  $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_TRACK\_DATE  BEFORE INSERT OR UPDATE ON TRACK  FOR EACH ROW  EXECUTE FUNCTION CHECK\_TRACK\_DATE();  ---------------- RATING ------------------  CREATE OR REPLACE FUNCTION CHECK\_RATING\_RANGE()  RETURNS TRIGGER AS $$  BEGIN  IF NEW.RATE < 1 OR NEW.RATE > 10 THEN  RAISE EXCEPTION 'RATING VALUE MUST BE BETWEEN 1 AND 10';  END IF;  RETURN NEW;  END; $$ LANGUAGE PLPGSQL;  CREATE TRIGGER TRIGGER\_CHECK\_RATING\_RANGE  BEFORE INSERT OR UPDATE ON RATING  FOR EACH ROW  EXECUTE FUNCTION CHECK\_RATING\_RANGE(); |

Листинг 6 – Скрипты создания триггеров

|  |
| --- |
| ------------------------------------  -------------- JSON ----------------  ------------------------------------  SELECT EXPORT\_USERS\_TO\_JSON\_FILE('E:/POSTGRESQL/14/DATA/JSON/USERS.JSON');  ------------- EXPORT ---------------  CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  JSON\_DATA JSON;  BEGIN  SELECT JSON\_AGG(ROW\_TO\_JSON(USERS)) INTO JSON\_DATA FROM USERS;  PERFORM PG\_FILE\_WRITE(FILE\_PATH, JSON\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL;  SELECT IMPORT\_USERS\_FROM\_JSON\_FILE('E:/POSTGRESQL/14/DATA/JSON/USERS.JSON');  ------------- IMPORT ---------------  CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_JSON\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER  ) AS $$  DECLARE  JSON\_DATA JSON;  USER\_DATA JSON;  BEGIN  JSON\_DATA := PG\_READ\_FILE(FILE\_PATH, 0, 1000000000)::JSON;    DROP TABLE IF EXISTS TEMP\_ROLES;  CREATE TEMP TABLE TEMP\_ROLES (  ROLE\_ID SERIAL PRIMARY KEY,  ROLE\_NAME VARCHAR(255) NOT NULL UNIQUE  );  INSERT INTO TEMP\_ROLES(ROLE\_NAME)  VALUES('ADMIN'),('USER');  DROP TABLE IF EXISTS TEMP\_USERS;  CREATE TEMP TABLE TEMP\_USERS (  USER\_ID SERIAL PRIMARY KEY,  USER\_NAME VARCHAR(255) NOT NULL,  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255) UNIQUE NOT NULL,  USER\_PASSWORD VARCHAR(255) NOT NULL,  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER,  CONSTRAINT FK\_USER\_ROLE FOREIGN KEY (USER\_ROLE\_ID) REFERENCES TEMP\_ROLES (ROLE\_ID)  );    FOR USER\_DATA IN SELECT \* FROM JSON\_ARRAY\_ELEMENTS(JSON\_DATA)  LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA->>'USER\_NAME',  USER\_DATA->>'USER\_EMAIL',  USER\_DATA->>'USER\_PASSWORD',  TO\_DATE(USER\_DATA->>'USER\_DATE\_OF\_BIRTH', 'YYYY-MM-DD'),  CAST(USER\_DATA->>'USER\_ROLE\_ID' AS INTEGER)  );  END LOOP;    RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL;  ------------------------------------  -------------- XML -----------------  ------------------------------------  SELECT EXPORT\_USERS\_TO\_XML\_FILE('E:/POSTGRESQL/14/DATA/XML/USERS.XML');  ------------- EXPORT ---------------  CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_XML\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  XML\_DATA XML;  BEGIN  SELECT XMLELEMENT(NAME "USERS", XMLAGG(XMLELEMENT(NAME "USER",  XMLFOREST(USER\_ID, USER\_NAME, USER\_EMAIL, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)))) INTO XML\_DATA FROM USERS;  PERFORM PG\_FILE\_WRITE(FILE\_PATH, XML\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL;  CREATE OR REPLACE FUNCTION EXPORT\_USERS\_TO\_XML\_FILE(FILE\_PATH TEXT)  RETURNS VOID AS  $$  DECLARE  XML\_DATA XML;  BEGIN  SELECT XMLELEMENT(NAME "USERS", XMLAGG(XMLELEMENT(NAME "USER",  XMLFOREST(USER\_ID, USER\_NAME, USER\_EMAIL, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)))) INTO XML\_DATA FROM USERS;  XML\_DATA := FORMAT('<?XML VERSION="1.0" ENCODING="UTF-8"?>%S', XML\_DATA::TEXT);  PERFORM PG\_FILE\_WRITE(FILE\_PATH, XML\_DATA::TEXT,'TRUE');  END;  $$  LANGUAGE PLPGSQL;  SELECT IMPORT\_USERS\_FROM\_XML\_FILE('E:/POSTGRESQL/14/DATA/XML/USERS.XML');  ------------- IMPORT ---------------  CREATE OR REPLACE FUNCTION IMPORT\_USERS\_FROM\_XML\_FILE(FILE\_PATH TEXT)  RETURNS TABLE (  USER\_NAME VARCHAR(255),  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER  ) AS $$  DECLARE  XML\_DATA XML;  USER\_DATA RECORD;  BEGIN  DROP TABLE IF EXISTS TEMP\_USERS;  CREATE TEMP TABLE TEMP\_USERS (  USER\_NAME VARCHAR(255),  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  USER\_ROLE\_ID INTEGER  );    XML\_DATA := XMLPARSE(DOCUMENT CONVERT\_FROM(PG\_READ\_BINARY\_FILE(FILE\_PATH), 'UTF8'));    FOR USER\_DATA IN SELECT \* FROM XMLTABLE('/USERS/USER' PASSING XML\_DATA COLUMNS  USER\_NAME VARCHAR(255) PATH 'USER\_NAME',  USER\_EMAIL VARCHAR(255) PATH 'USER\_EMAIL',  USER\_PASSWORD VARCHAR(255) PATH 'USER\_PASSWORD',  USER\_DATE\_OF\_BIRTH DATE PATH 'USER\_DATE\_OF\_BIRTH',  USER\_ROLE\_ID INTEGER PATH 'USER\_ROLE\_ID'  ) LOOP  INSERT INTO TEMP\_USERS (USER\_NAME, USER\_EMAIL, USER\_PASSWORD, USER\_DATE\_OF\_BIRTH, USER\_ROLE\_ID)  VALUES (  USER\_DATA.USER\_NAME,  USER\_DATA.USER\_EMAIL,  USER\_DATA.USER\_PASSWORD,  USER\_DATA.USER\_DATE\_OF\_BIRTH,  USER\_DATA.USER\_ROLE\_ID  );  END LOOP;  RETURN QUERY SELECT \* FROM TEMP\_USERS;  END;  $$ LANGUAGE PLPGSQL;  ------------- END --------------- |

Листинг 7 – Скрипты для импорта и экспорта

|  |
| --- |
| /\*--------------------------——————----------------------------  ----------------------------| GET |---------------------------  ----------------------------——————--------------------------\*/  -------------------- GET USERS --------------------  CREATE OR REPLACE FUNCTION GETUSERS()  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_USER  ORDER BY ALL\_INFO\_USER.USER\_ID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETUSERS();  -------------------- GET USER --------------------  CREATE OR REPLACE FUNCTION GETUSERBYID(USERID INTEGER)  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_USER  WHERE ALL\_INFO\_USER.USER\_ID = USERID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETUSERBYID(1);  -------------------- GET TRACKS --------------------  CREATE OR REPLACE FUNCTION GETTRACKS()  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  USER\_NAME VARCHAR(255),  USER\_ID INTEGER,  USER\_IMG BYTEA,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC    )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  ORDER BY ALL\_INFO\_TRACK.TRACK\_ID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETTRACKS();  -------------------- GET TRACKS USER\_NAME --------------------  CREATE OR REPLACE FUNCTION GET\_USERS\_FROM\_TRACK(TRACK\_ID\_IN INTEGER)  RETURNS TABLE (  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  DECLARE  USER\_IDS INTEGER[];  BEGIN  SELECT ARRAY\_AGG(USER\_TRACK.USER\_ID) INTO USER\_IDS FROM USER\_TRACK WHERE USER\_TRACK.TRACK\_ID = TRACK\_ID\_IN;  RETURN QUERY SELECT \* FROM ALL\_INFO\_USER WHERE ALL\_INFO\_USER.USER\_ID = ANY(USER\_IDS);  END;  $$ LANGUAGE PLPGSQL;  -------------------- GET TRACKS USER --------------------  CREATE OR REPLACE FUNCTION GETTRACKSUSER(USERID INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC    )  AS $$  DECLARE  TRACKS\_ID INTEGER[];  BEGIN  SELECT ARRAY\_AGG(USER\_TRACK.TRACK\_ID) INTO TRACKS\_ID FROM USER\_TRACK WHERE USER\_TRACK.USER\_ID = USERID;  RETURN QUERY SELECT \* FROM ALL\_INFO\_TRACK WHERE ALL\_INFO\_TRACK.TRACK\_ID = ANY(TRACKS\_ID);  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETTRACKSUSER(5)  -------------------- GET TRACK --------------------  CREATE OR REPLACE FUNCTION GETTRACKBYID(TRACKID INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK WHERE ALL\_INFO\_TRACK.TRACK\_ID = TRACKID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GET\_RECENT\_TRACKS(4)  SELECT \* FROM GETPLAYLISTTRACKSBYID(1);  -------------------- GET PLAYLIST TRACKS BY ID --------------------  CREATE OR REPLACE FUNCTION GETPLAYLISTTRACKSBYID(PLAYLISTID INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  DECLARE  TRACK\_IDS INTEGER[];  BEGIN  SELECT ARRAY\_AGG(PLAYLIST\_TRACKS.TRACK\_ID) INTO TRACK\_IDS  FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_TRACKS.PLAYLIST\_ID = PLAYLISTID;    RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  WHERE ALL\_INFO\_TRACK.TRACK\_ID = ANY(TRACK\_IDS);  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM PLAYLIST\_TRACKS  SELECT \* FROM GETPLAYLISTTRACKSBYID(1)  -------------------- GET PLAYLIST BY ID USER --------------------  CREATE OR REPLACE FUNCTION GETALLPLAYLISTBYUSERID(USERID INTEGER)  RETURNS TABLE (  PLAYLIST\_ID INTEGER,  USER\_ID INTEGER,  TITLE VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM PLAYLIST  WHERE PLAYLIST.USER\_ID = USERID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETALLPLAYLISTBYUSERID(44);  SELECT \* FROM PLAYLIST  -------------------- GET PLAYLISTS --------------------  CREATE OR REPLACE FUNCTION GETALLPLAYLISTS()  RETURNS TABLE (  PLAYLIST\_ID INTEGER,  USER\_ID INTEGER,  TITLE VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM PLAYLIST;    END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETALLPLAYLISTS();  -------------------- GET TRACK FROM LIBRARY BY USERID --------------------  CREATE OR REPLACE FUNCTION GETTRACKFROMLIBRARYBYUSERID(USERID INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC    )  AS $$  DECLARE  TRACKS\_ID INTEGER[];  BEGIN  SELECT ARRAY\_AGG(LIBRARY\_USER.TRACK\_ID) INTO TRACKS\_ID FROM LIBRARY\_USER WHERE LIBRARY\_USER.USER\_ID = USERID;  RETURN QUERY SELECT \* FROM ALL\_INFO\_TRACK WHERE ALL\_INFO\_TRACK.TRACK\_ID = ANY(TRACKS\_ID);  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM GETTRACKFROMLIBRARYBYUSERID(1)  SELECT \* FROM RATING  -------------------- GET RATING USERS --------------------  CREATE OR REPLACE FUNCTION GETRATINGUSERS(TRACKID INTEGER)  RETURNS TABLE (USER\_ID INTEGER) AS $$  BEGIN  RETURN QUERY SELECT RATING.USER\_ID FROM RATING WHERE RATING.TRACK\_ID = TRACKID;  END;  $$ LANGUAGE PLPGSQL;  SELECT \* FROM RATING  -------------------- GET GENRES --------------------  CREATE OR REPLACE FUNCTION CREATE\_GENRE\_PLAYLIST(  IN\_USER\_ID INTEGER,  IN\_TITLE VARCHAR(255),  IN\_GENRE VARCHAR(255))  RETURNS INTEGER LANGUAGE PLPGSQL AS $$  DECLARE  PLAYLIST\_ID INTEGER;  BEGIN  -- СОЗДАЕМ НОВЫЙ ПЛЕЙЛИСТ  INSERT INTO PLAYLIST(USER\_ID, TITLE)  VALUES (IN\_USER\_ID, IN\_TITLE)  RETURNING PLAYLIST.PLAYLIST\_ID INTO PLAYLIST\_ID;  -- ДОБАВЛЯЕМ 10 ПЕСЕН В ПЛЕЙЛИСТ ПО ЗАДАННОМУ ЖАНРУ И РЕЙТИНГУ  INSERT INTO PLAYLIST\_TRACKS(TRACK\_ID, PLAYLIST\_ID)  SELECT T.TRACK\_ID, PLAYLIST\_ID  FROM (  SELECT TRACK\_ID, GENRE, RATING  FROM TRACK  WHERE TRACK.GENRE\_NAME = IN\_GENRE  ORDER BY RATING DESC  LIMIT 10  ) T;  RETURN PLAYLIST\_ID;  END;  $$;  SELECT \*  FROM USERS  SELECT \*  FROM GETALLPLAYLISTBYUSERID(1)  -------------------- GET NEW TRACKS --------------------  CREATE OR REPLACE FUNCTION GET\_RECENT\_TRACKS(NUM\_TRACKS INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  ORDER BY ALL\_INFO\_TRACK.TRACK\_DATE DESC, ALL\_INFO\_TRACK.TRACK\_ID DESC  LIMIT NUM\_TRACKS;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 8 – Скрипты для получения данных

|  |
| --- |
| /\*--------------------------——————————----------------------------  ----------------------------| SEARCH |----------------------------  ----------------------------——————————--------------------------\*/  -------------------- GET SEARCH TRACKS --------------------  CREATE OR REPLACE FUNCTION SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME(QUERY\_TEXT VARCHAR) RETURNS TABLE( TRACK\_ID INTEGER, TRACK\_TITLE VARCHAR, TRACK\_DATE DATE, GENRE\_NAME VARCHAR, TRACK\_IMAGE BYTEA, TRACK\_CONTENT BYTEA, AVG\_RATING NUMERIC) AS $$  BEGIN  RETURN QUERY  SELECT  T.TRACK\_ID,  T.TRACK\_TITLE,  T.TRACK\_DATE,  G.GENRE\_NAME,  T.TRACK\_IMAGE,  T.TRACK\_CONTENT,  AVERAGE\_RATING(T.TRACK\_ID) AS AVG\_RATING  FROM  TRACK T  JOIN GENRE G ON T.GENRE\_ID = G.GENRE\_ID  LEFT JOIN USER\_TRACK UT ON T.TRACK\_ID = UT.TRACK\_ID  LEFT JOIN USERS U ON UT.USER\_ID = U.USER\_ID  WHERE  T.TRACK\_TITLE ILIKE '%' || QUERY\_TEXT || '%'  OR U.USER\_NAME ILIKE '%' || QUERY\_TEXT || '%';  END;  $$ LANGUAGE PLPGSQL;  SELECT \*  FROM SEARCH\_TRACK\_BY\_TITLE\_OR\_USER\_NAME('I')  -------------------- GET SEARCH TRACKS BY TITLE --------------------  CREATE OR REPLACE FUNCTION GETPLAYLISTBYTITLE(TITLE\_QUERY TEXT)  RETURNS TABLE(PLAYLIST\_ID INTEGER, USER\_ID INTEGER, TITLE VARCHAR(255))  AS $$  BEGIN  RETURN QUERY  SELECT PLAYLIST.PLAYLIST\_ID, PLAYLIST.USER\_ID, PLAYLIST.TITLE  FROM PLAYLIST  WHERE PLAYLIST.TITLE ILIKE '%' || TITLE\_QUERY|| '%'  AND EXISTS (  SELECT 1 FROM PLAYLIST\_TRACKS WHERE PLAYLIST\_TRACKS.PLAYLIST\_ID = PLAYLIST.PLAYLIST\_ID  );  END;  $$ LANGUAGE PLPGSQL;  -------------------- GET SEARCH SEARCH USER NAME --------------------  CREATE OR REPLACE FUNCTION SEARCH\_USER\_NAME(QUERY\_TEXT VARCHAR)  RETURNS TABLE(  USER\_ID INTEGER,  USER\_NAME VARCHAR(255),  USER\_IMG BYTEA,  USER\_EMAIL VARCHAR(255),  USER\_PASSWORD VARCHAR(255),  USER\_DATE\_OF\_BIRTH DATE,  ROLE\_NAME VARCHAR(255)  )  AS $$  BEGIN  RETURN QUERY  SELECT  \*  FROM ALL\_INFO\_USER  WHERE ALL\_INFO\_USER.USER\_NAME ILIKE '%' || QUERY\_TEXT || '%';  END;  $$ LANGUAGE PLPGSQL;  /\*--------------------------——————————---------------------------  -----------------------------| SORT |----------------------------  ----------------------------——————————--------------------------\*/  -------------------- SORT BY DATE DESC --------------------  CREATE OR REPLACE FUNCTION SORTTRACKBYDATEDESC()  RETURNS TABLE(  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  ORDER BY ALL\_INFO\_TRACK.TRACK\_DATE DESC;  END;  $$ LANGUAGE PLPGSQL;  -------------------- SORT TOP TRACKS --------------------  SELECT\* FROM GETTRACKBYRAITING(5)  CREATE OR REPLACE FUNCTION GETTRACKBYRAITING(LIMIT\_TRACK INTEGER)  RETURNS TABLE (  TRACK\_ID INTEGER,  TRACK\_TITLE VARCHAR(255),  TRACK\_DATE DATE,  GENRE\_NAME VARCHAR(255),  TRACK\_IMAGE BYTEA,  TRACK\_CONTENT BYTEA,  AVG\_RATING NUMERIC  )  AS $$  BEGIN  RETURN QUERY SELECT \*  FROM ALL\_INFO\_TRACK  ORDER BY AVG\_RATING DESC  LIMIT LIMIT\_TRACK;  END;  $$ LANGUAGE PLPGSQL;  -------------------- INDEXES --------------------  CREATE INDEX IDX\_USER\_NAME ON USERS(USER\_NAME);  CREATE INDEX IDX\_TITLE ON TRACK(TRACK\_TITLE);  CREATE INDEX IDX\_USER\_ID ON USERS(USER\_ID);  CREATE INDEX IDX\_GENRE\_ID ON GENRE(GENRE\_ID);  CREATE INDEX IDX\_PLAYLIST\_USER\_ID ON PLAYLIST(USER\_ID);  CREATE INDEX IDX\_LIBRARY\_USER\_ID ON LIBRARY\_USER(USER\_ID);  CREATE INDEX IDX\_LIBRARY\_TRACK\_ID ON LIBRARY\_USER(TRACK\_ID);  CREATE INDEX IDX\_USER\_TRACK\_USER\_ID ON USER\_TRACK(USER\_ID);  CREATE INDEX IDX\_USER\_TRACK\_TRACK\_ID ON USER\_TRACK(TRACK\_ID);  CREATE INDEX IDX\_RATING\_USER\_ID ON RATING(USER\_ID);  CREATE INDEX IDX\_RATING\_TRACK\_ID ON RATING(TRACK\_ID);  CREATE INDEX IDX\_PLAYLIST\_TRACKS\_TRACK\_ID ON PLAYLIST\_TRACKS(TRACK\_ID);  CREATE INDEX IDX\_PLAYLIST\_TRACKS\_PLAYLIST\_ID ON PLAYLIST\_TRACKS(PLAYLIST\_ID);  ------------------------------------------------- |

Листинг 9– Скрипты для поиска данных

|  |
| --- |
| /\*---------------------————————————————---------------------  -----------------------| INSERT TABLES |--------------------  ------------------------————————————————-------------------\*/  INSERT INTO ROLE (ROLE\_NAME) VALUES ('USER');  INSERT INTO ROLE (ROLE\_NAME) VALUES ('ADMIN');  INSERT INTO GENRE (GENRE\_NAME) VALUES  ('ROCK'),  ('POP'),  ('HIP-HOP'),  ('JAZZ'),  ('ELECTRONIC'),  ('CLASSICAL'),  ('FOLK'),  ('R&B'),  ('COUNTRY'),  ('REGGAE');  --------------- INSERT 100000 ---------------  CREATE OR REPLACE FUNCTION INSERT\_GENRES()  RETURNS VOID AS $$  DECLARE  I INTEGER := 1;  BEGIN  WHILE I <= 100000 LOOP  INSERT INTO GENRE (GENRE\_NAME) VALUES ('GENRE ' || I);  I := I + 1;  END LOOP;  END;  $$ LANGUAGE PLPGSQL; |

Листинг 10– Скрипты заполнение таблиц