Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра инженерной психологии и эргономики

Дисциплина: Компьютерные системы и сети

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему:

СЕТЕВОЕ ФАЙЛОВОЕ ХРАНИЛИЩЕ МУЗЫКИ

БГУИР КП 6-05-0612-01-032 ПЗ

Студент Навросюк М.А.

Руководитель Болтак С.В.

Минск 2025

1 ВВЕДЕНИЕ

Веб-сервисы стали неотъемлемой частью нашей повседневной жизни, с тех пор как компьютеры, телефоны, ноутбуки и другие умные устройства получили широкое распространение. Они не только сделали кино, литературу, музыку и различные формы творчества доступными для каждого, но и вывели эти сферы на совершенно новый уровень взаимодействия и потребления контента. В этом контексте музыка занимает особое место, превратившись из простого набора звуков в постоянного спутника человека.

Теперь музыка сопровождает нас всю жизнь, начиная с песен о дружбе и семье, которые мы слышим в детстве, и заканчивая сложными композициями, отражающими наши взрослые переживания. Музыка становится чем-то большим, чем просто поток звуковых волн или строчки закодированных символов в цифровом файле. Она способна хранить драгоценные воспоминания, вызывать глубокие чувства и эмоции, нести в себе особое настроение и даже становиться якорем для определенных жизненных этапов. Поэтому для людей, по-настоящему ценящих музыку, важно иметь постоянный и надежный доступ к своим любимым композициям. Для них важно, чтобы этот доступ был безопасным, предсказуемым и не зависел от внешних факторов.

Несмотря на популярность современных стриминговых сервисов, предоставляющих доступ к своим огромным музыкальным библиотекам за определенную плату, не все меломаны полностью им доверяют. Существует категория пользователей, которые опасаются быть зависимыми от той или иной платформы. Их беспокоит возможность изменения условий предоставления услуг, удаления треков из каталога по решению правообладателей или самого сервиса, а также потенциальные ограничения на прослушивание определенных исполнителей или жанров. Кроме того, модель подписки, хоть и удобна, не всегда отвечает стремлению к полному владению и контролю над своей музыкальной коллекцией.

В свете вышеизложенного, актуальной становится задача создания такого решения, которое объединило бы надежность и контроль личного хранения с удобством и доступностью облачных технологий.

Целью данной курсовой работы является разработка прототипа сетевого хранилища музыки, которое позволит пользователям загружать, хранить и воспроизводить собственные аудиофайлы в рамках локальной сети. Проект направлен на освоение базовых принципов создания клиент-серверных веб-приложений, работы с базами данных, реализации потоковой передачи данных и построения пользовательского интерфейса, предоставляя пользователю больший контроль над своей музыкальной коллекцией по сравнению с традиционными стриминговыми сервисами.

.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

Сегодня существует огромное количество способов хранить, прослушивать, искать музыку, а также делится ей. Чаще всего для этого используют стриминговые сервисы, такие как Яндекс Музыка, Spotify, VK Музыка и облачные хранилища Яндекс Диск, Google Drive, хранилище Mail.ru.

Рассмотрим самые востребованные приложения.

В первую очередь был рассмотрен самый популярный сервис на территории Беларуси, Яндекс Музыка.

Внешний вид данного приложения представлен на рисунке 1.1.

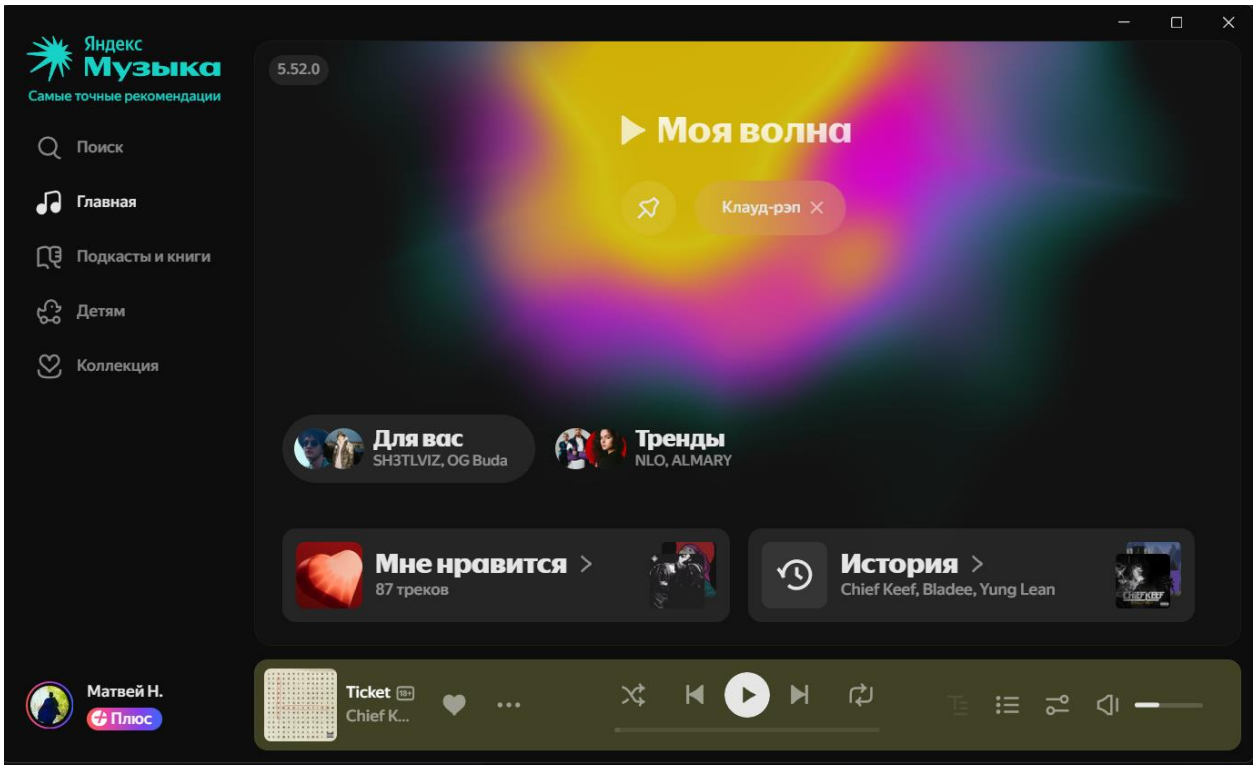

Рисунок 1.1 – музыкальный сервис «Яндекс Музыка»

В визуальной стилистике и по части пользовательского интерфейса «Яндекс Музыка» предлагает современный и интуитивно понятный дизайн, ориентированный на удобный доступ к музыкальному контенту.

Ключевой особенностью сервиса «Яндекс Музыка» является его глубокая интеграция с экосистемой Яндекса. Платформа активно использует собственные алгоритмы для формирования персональных рекомендаций («Моя волна», плейлисты дня), основанных на предпочтениях пользователя и истории его прослушиваний.

Разрабатываемый проект делает акцент на персональном хранилище и контроле пользователя над своей музыкальной библиотекой, в то время как «Яндекс Музыка» ориентирована на предоставление доступа к обширному, но контролируемому каталогу коммерческой музыки.

Далее был рассмотрен популярный сервис облачного хранения «Google Drive».

Внешний вид данного приложения представлен на рисунке 1.2.

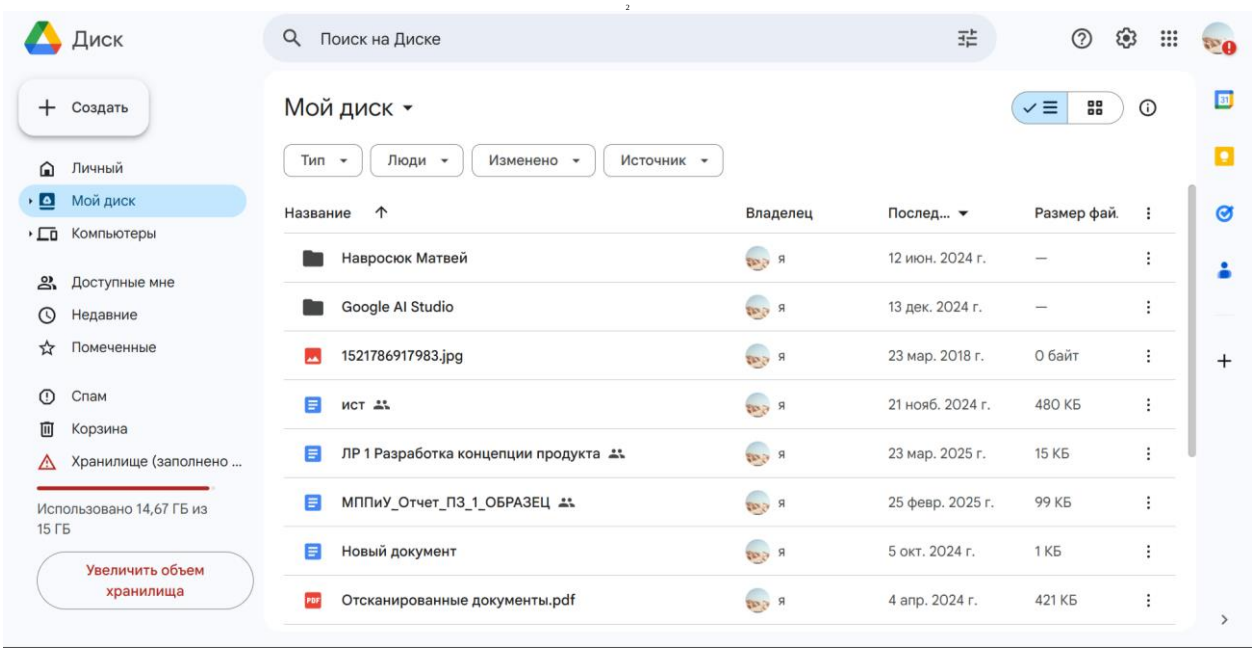

Рисунок 1.2 – облачное хранилище «Google Drive»

Сервис «Google Drive», разработанный всем известной компанией, является универсальным облачным хранилищем, предназначенным для хранения, синхронизации и совместного использования файлов различных типов.

Функционал сервиса включает загрузку и организацию файлов в папки, совместную работу над документами через интеграцию с Google Workspace, а также доступ к данным с любых устройств. Хоть Google Drive позволяет хранить аудиофайлы и имеет базовые средства для их предпрослушивания, он не ориентирован на управление музыкальными коллекциями.

Ключевыми преимуществами этого аналога является его глубокая интеграция с экосистемой сервисов Google, высокий уровень доступности и надежности.

Однако, зависимость от интернет-соединения для полноценного доступа и отсутствие полноценного интерфейса для прослушивания музыки являются заметными недостатками.

1.2 Постановка задачи

В рамках данной курсовой работы планируется разработать облачное хранилище музыки, включающее серверную часть и клиентское приложение для операционной системы Windows.

В процессе разработки должны быть реализованы следующие базовые функции:

– регистрация новых пользователей;
– аутентификация существующих пользователей;
– загрузка аудиофайлов пользователем на сервер;
– удаление аудиофайлов пользователем с сервера;
– хранение аудиофайлов и их метаданных (название, исполнитель, длительность);
– отображение каталога доступной музыки;
– потоковое воспроизведение выбранных аудиофайлов;
– поиск музыкальных треков по названию;
– сохранение данных о любимых треках пользователей.

Нефункциональные требования:

– Локальная работа: Приложение должно функционировать автономно в рамках локальной вычислительной сети, без зависимости от внешних облачных сервисов.
– Простота использования: Интерфейс клиентского приложения должен быть логичным и удобным для освоения пользователем.
– Надежность: Должна быть реализована базовая обработка ошибок на сервере и клиенте.
– Безопасность (базовая): Пароли пользователей должны храниться в хешированном виде. Доступ к защищенным ресурсам API должен осуществляться с использованием токенов.

Используемый стек технологий и средств разработки:

Серверная часть (Backend):
– Язык программирования: C#;
– Фреймворк: ASP.NET Core Web API (с применением Minimal APIs);
– Система управления базами данных: SQLite;
– Механизм аутентификации: Простая токен-аутентификация;
– Хранение файлов: Локальная файловая система сервера.

Клиентская часть (Frontend):
– Тип приложения: Десктопное приложение для ОС Windows.
– Язык программирования: C#
– Технология построения UI: WPF (Windows Presentation Foundation).
– Архитектурный паттерн: MVVM (Model-View-ViewModel).
– Среда разработки: Microsoft Visual Studio 2022.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

Программное средство «Сетевое файловое хранилище музыки» проектируется на основе клиент-серверной архитектуры. Такая архитектура разделяет приложение на два взаимодействующих компонента: сервер (Backend), отвечающий за обработку данных и бизнес-логику, и клиент (Frontend), предоставляющий пользовательский интерфейс.

Серверная часть реализуется как веб-API на платформе ASP.NET Core с использованием языка C#. Она обеспечивает управление пользователями, загрузку и хранение аудиофайлов, взаимодействие с базой данных SQLite и потоковую передачу музыкальных треков.

Клиентская часть представляет собой десктопное приложение для операционной системы Windows, разработанное с использованием технологии WPF и языка C#. Для структурирования кода и разделения логики представления от бизнес-логики применяется архитектурный паттерн MVVM.

Взаимодействие между клиентским приложением и сервером осуществляется по протоколу HTTP. Клиент отправляет запросы к API сервера, передавая данные в формате JSON или как multipart/form-data (для загрузки файлов). Сервер обрабатывает запросы и возвращает ответы в формате JSON или аудиопоток для стриминга.

При разработке приложения было реализовано 8 основных модулей для клиентской части:

- MainWindow – модуль, для отображения главного окна приложения;
- LoginWindow – модуль, обеспечивающий авторизацию и регистрацию;
- AllTracksView – модуль, отображающий список всех музыкальных треков, доступных пользователю в его хранилище;
- ProfileView – модуль, отображающий информацию о текущем аутентифицированном пользователе (никнейм, email);
- FavoritesView – модуль, отображающий список треков, которые пользователь отметил как "понравившиеся";
- UploadTrackView– модуль, предоставляющий пользовательский интерфейс для выбора аудиофайлов с компьютера пользователя, (опционального) ввода метаданных (название, исполнитель) и их последующей загрузки на сервер;
- MainViewModel – центральный модуль ViewModel, управляющий состоянием и логикой представления для MainWindow и координирующий взаимодействие между различными представлениями и сервисом API;

– ApiService – модуль, реализующий всю логику взаимодействия с серверным API.

А также были реализованы 4 основных модуля для серверной части:

– Program.cs – главный модуль сервера, отвечающий за конфигурацию и запуск веб-приложения;
– AuthService – модуль, обеспечивающий авторизацию и регистрацию, проверки пароля, генерации и проверки токенов доступа, получения данных пользователя;
– TrackService – модуль, отвечающий за обработку загрузки файлов, извлечение метаданных (с помощью TagLib#), сохранение и получение информации о треках из базы данных, управление лайками, удаление треков и поиск;
– DatabaseSetup – модуль, отвечающий за инициализацию и структуру базы данных (SQLite) и содержащий логику для создания необходимых таблиц (Users, Tracks, UserLikedTracks) при старте приложения.

2.2 Проектирование интерфейса программного средства

Пользовательский интерфейс клиентского WPF-приложения проектируется с акцентом на простоту, интуитивность и стиль, напоминающий популярные музыкальные сервисы.

2.2.1 Окно аутентификации и регистрации

Данное окно является стартовым для приложения. Оно предоставляет пользователю возможность войти в существующий аккаунт или перейти к форме регистрации. Макет «стартового» окна приложения представлен на рисунке 2.1.

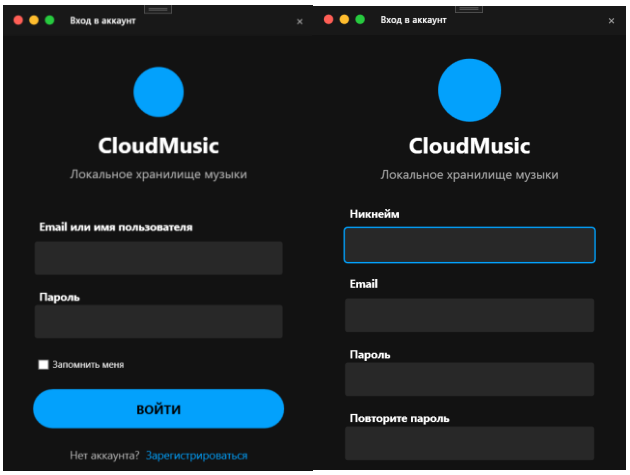

Рисунок 2.1 – Окно аутентификации и регистрации

2.2.2 Главное окно приложения

Отображается после успешной аутентификации пользователя. Содержит основные элементы для взаимодействия с музыкальной библиотекой.

Структура:

– Боковая панель навигации: включает логотип, кнопки «Главная», «Моя медиатека», «Профиль», «Загрузить трек», «Выйти».

– Центральная область контента: отображает список треков (по умолчанию), форму загрузки или другие представления в зависимости от выбора пользователя. Включает строку поиска.

– Нижняя панель плеера: содержит информацию о текущем воспроизводимом треке (название, исполнитель), кнопки управления (Play/Pause, Next, Previous), слайдеры прогресса и громкости, отображение времени.

При наведении на каждый элемент управления появляется всплывающая подсказка, говорящая о предназначении компонента.

Внешний вид главного окна представлен на рисунке 2.2.

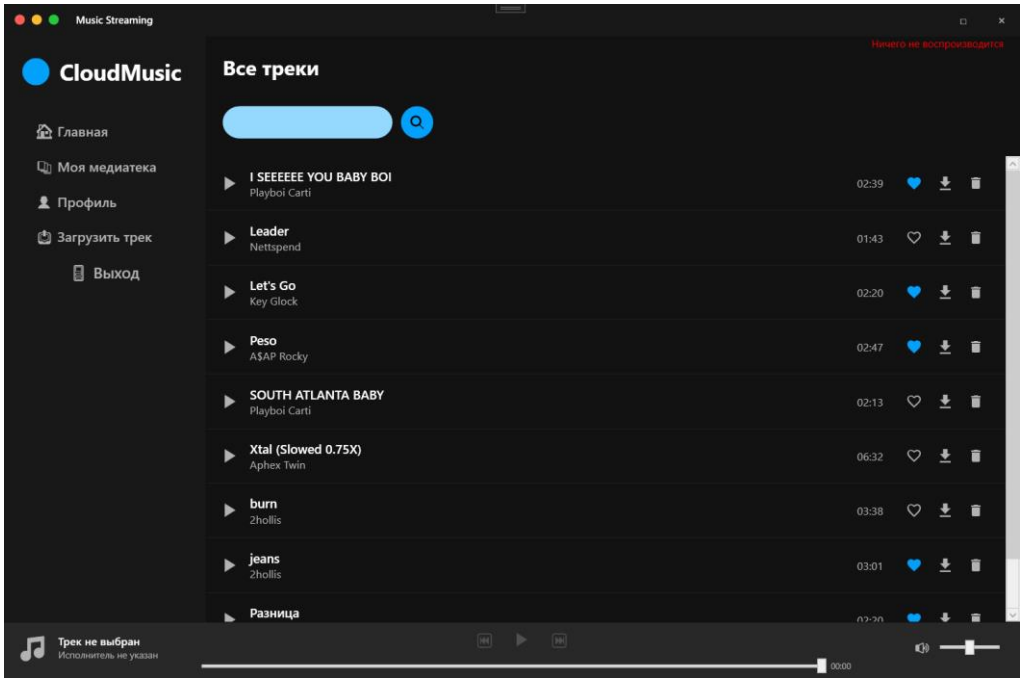

Рисунок 2.2 – Главное окно

2.2.3 Окно «Моя медитатека»

Окно идентично окну «Главное», но в нем отображаютя только понравившиеся пользователю треки. Для этого выполняется GET-запрос на сервер к специальному эндпоинту.

Макет окна медиатеки представлен на рисунке 2.3

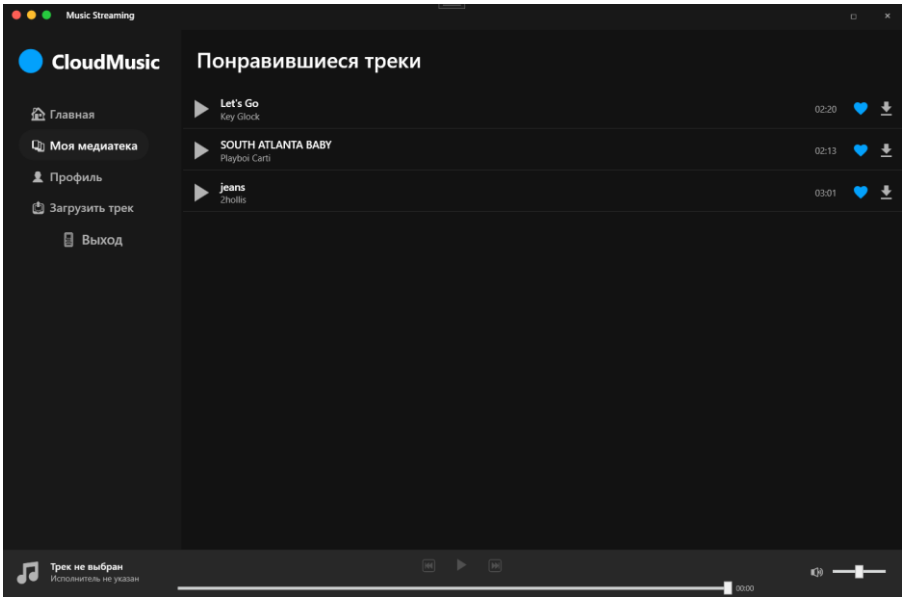

Рисунок 2.3 – Макет окна «Моя медитатека»

2.2.4 Окно «Профиль»

Данное окно предоставляет краткую информацию о текущем авторизованном пользователе. Потенциальнно в этом окне можно отображать количество загруженных песен, минут прослушивания и другую статистику использования.

Внешний вид информационных окон представлен на рисунке 2.4.

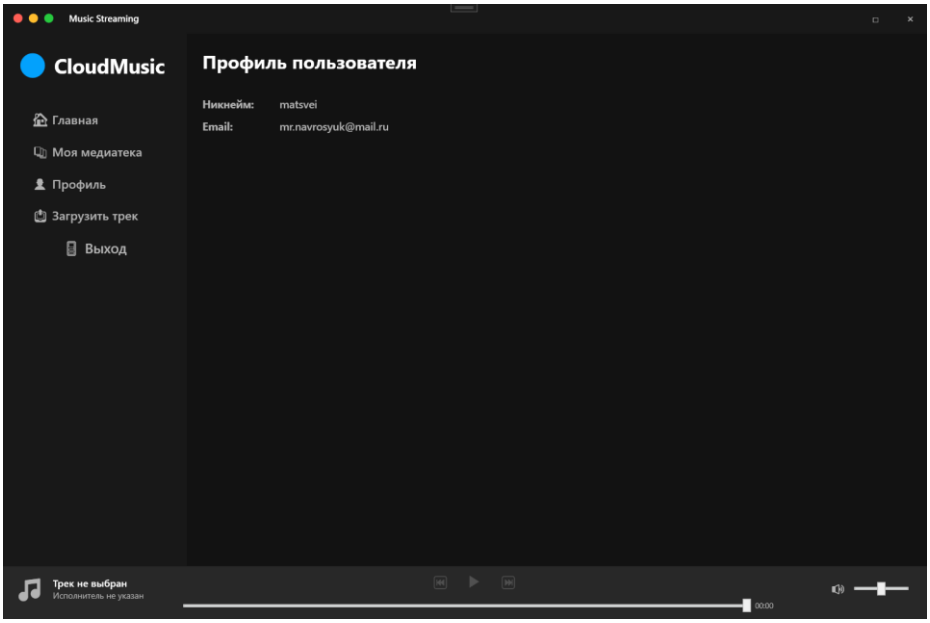

Рисунок 2.4 – Внешний вид окна «Профиль»

2.2.5 Окно «Загрузить трек»

Ключевое окно, позволяющие загрузить трек из хранилища устройства и указать название и исполнителя композиции вручную. При нажатии на кнопку «Выбрать аудиофайл» открывается модальное окно для выбора файла в хранилище компьютера клиента.

Внешний вид информационных окон представлен на рисунке 2.4.

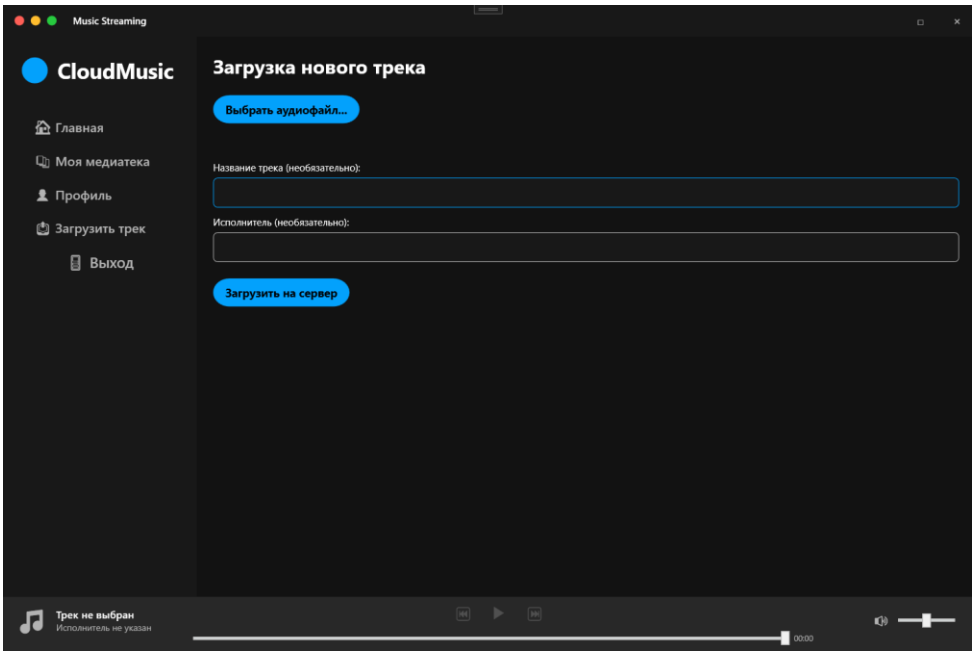

Рисунок 2.5 – Макет окна «Загрузить трек»

2.3 Проектирование функционала программного средства

Грамотно поставленная задача и хорошо составленные алгоритмы – ключевая фаза в проектировании программного средства. Облачное хранилище музыки должно предоставлять пользователю такой минимальный функционал как:

– регистрация и авторизация пользователя;
– загрузка музыкального трека на сервер;
– воспроизведение трека;
– поиск треков в хранилище.

2.3.1 Регистрация пользователей

Регистрация нового пользователя и последующая авторизация существующего являются основополагающими функциями для доступа к персонализированному музыкальному хранилищу.

Регистрация начинается с того, что пользователь вводит уникальный никнейм, email и пароль. Эти данные отправляются на сервер, где происходит их валидация и создание новой учетной записи. Пароль сохраняется в базе данных в хешированном виде.

.Блок-схема кода, осуществляющего регистрацию пользователей представлена на рисунке 2.6.

Рисунок 2.6 – Блок-схема модуля регистрации

2.3.2 Загрузка трека

Загрузка трека на сервер будет происходить из специального раздела приложения по нажатию кнопки «Загрузить на сервер» после выбора аудиофайла пользователем. Пользователь также сможет (опционально) указать название трека и исполнителя.

При нажатии кнопки загрузки клиентское приложение отправляет файл и метаданные на сервер. На сервере будет вызываться метод, например, UploadTrackAsync() класса TrackService. В качестве параметров этот метод

будет принимать файл (объект IFormFile на сервере), ID пользователя, а также опциональные название и исполнителя.

Сервис извлечет из файла дополнительную информацию (длительность трека с помощью библиотеки TagLib#), сохранит сам аудиофайл в специальную папку на сервере (music_uploads) под уникальным именем и запишет все метаданные (путь к файлу, название, исполнителя, длительность, ID пользователя) в базу данных.

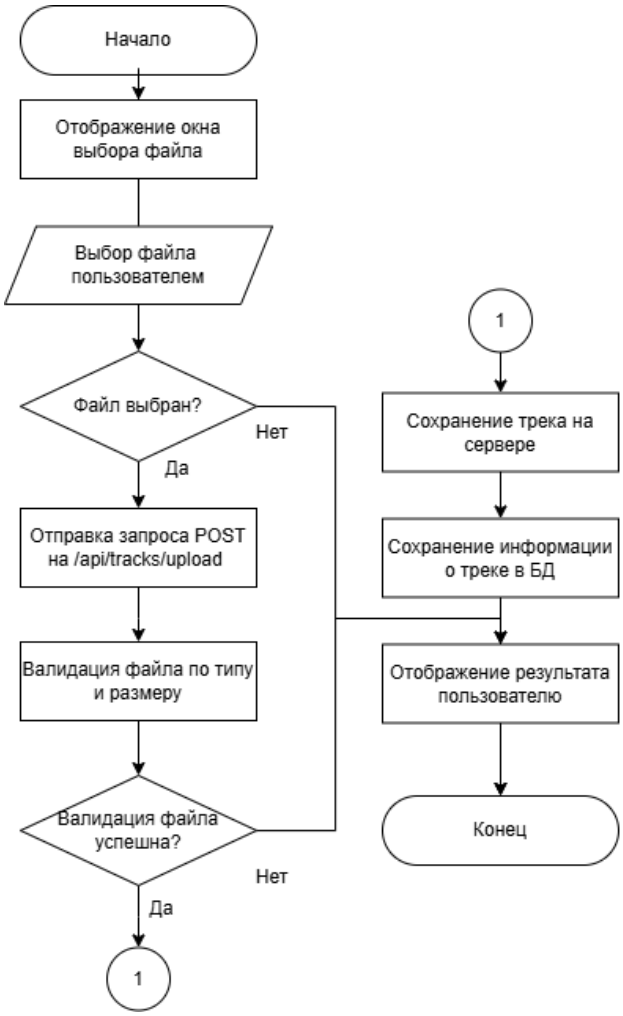

Рисунок 2.7 – Блок-схема метода загрузки трека

2.3.3 Воспроизведение трека

Воспроизведение трека инициируется выбором трека из списка (двойным кликом в AllTracksView или FavoritesView). Клиентское приложение запрашивает у сервера аудиопоток для выбранного трека через эндпоинт /api/tracks/stream/{trackId}. Сервер передает аудиоданные порциями (чанками). Клиент получает эти чанки и обеспечивает их воспроизведение с помощью MediaPlayer. Реализовано управление воспроизведением:

Play/Pause, переключение на следующий/предыдущий трек, перемотка, управление громкостью.

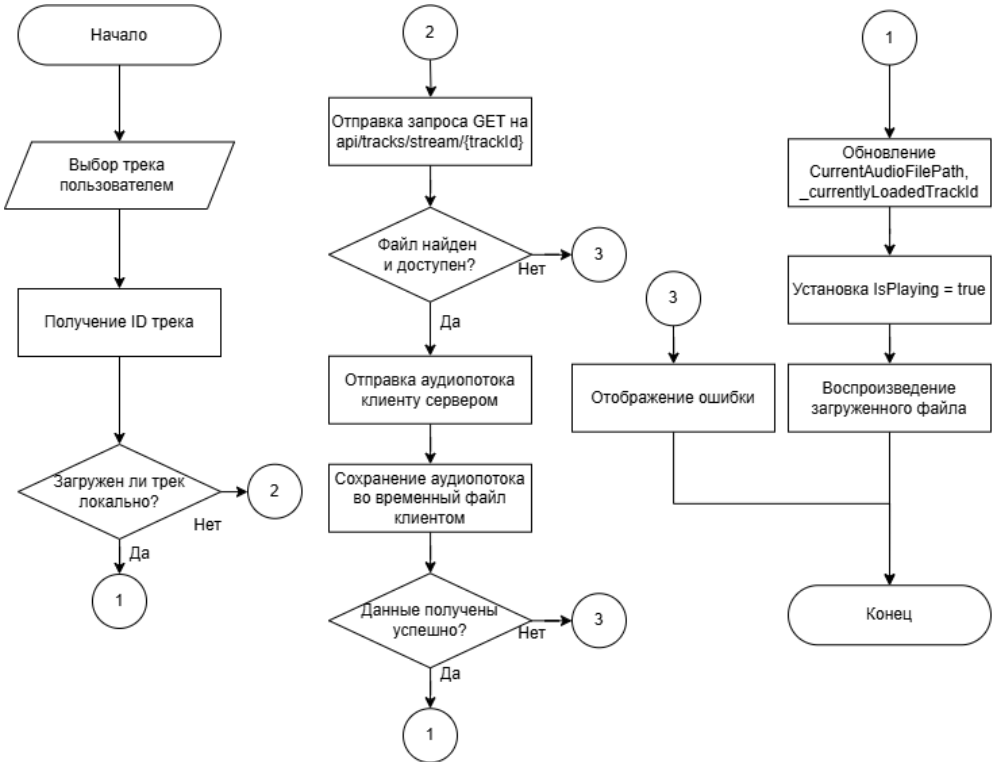

Рисунок 2.8 – Блок-схема воспроизведения трека

3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

3.1 Управление музыкальными треками

Основным функционалом системы является возможность загрузки, хранения, воспроизведения, поиска, лайка/анлайка и удаления музыкальных треков. Все операции, требующие взаимодействия с хранилищем данных, осуществляются через REST API, реализованное на ASP.NET Core, обеспечивающее взаимодействие с WPF-клиентом.

3.1.1 Загрузка треков

Пользователь может загружать аудиофайлы на сервер через клиентское WPF-приложение. Приложение клиента отправляет запрос POST на адрес сервера. На сервере система принимает файл, извлекает метаданные (используя TagLib#), сохраняет файл в специальное хранилище и создает запись о треке в базе данных SQLite. Код функции связанной с навигацией представлен ниже.

```
app.MapPost("/api/tracks/upload",
[RequestFormLimits(MultipartBodyLengthLimit = 100 * 1024 * 1024)]
```

```
async (
[FromForm] IFormFile file,
[FromForm] string? title,
[FromForm] string? artist,
HttpContext httpContext,
TrackService trackService
) => {
var user = httpContext.Items["User"] as User;
if (user == null)
return Results.Unauthorized();
if (file == null || file.Length == 0)
return Results.BadRequest("Файл не загружен или пуст.");
// Проверка расширения файла
var allowedExtensions = new[] { ".mp3", ".wav", ".ogg" };
var ext = Path.GetExtension(file.FileName).ToLowerInvariant();
if (!allowedExtensions.Contains(ext))
return Results.BadRequest("Недопустимый тип файла. Разрешены только mp3, wav, ogg.");
// Проверка MIME-типа
var allowedMimeTypes = new[] { "audio/mpeg", "audio/wav", "audio/ogg" };
if (!allowedMimeTypes.Contains(file.ContentType))

return Results.BadRequest($"Недопустимый MIME-тип файла: {file.ContentType}");
var (success, message, newTrack) = await trackService.UploadTrackAsync(file, user.Id,
title, artist);
if (!success)

return Results.Problem(detail: message, statusCode:
StatusCodes.Status500InternalServerError);

return Results.Ok(new { Message = message, Track = newTrack });
}).DisableAntiforgery();
```

3.1.2 Получение и воспроизведение треков

С помощью GET-запросов клиент может получать списки треков (всех, понравившихся, результаты поиска) и воспроизводить их. Для воспроизведения он запрашивает аудиопоток или путь к файлу. Код соответствующих функций предоставлен ниже.

```
app.MapGet("/api/tracks", async (TrackService trackService, HttpContext httpContext) => {
    var user = httpContext.Items["User"] as User; // Получаем пользователя
    if (user == null) // Проверка пользователя
    {
        return Results.Unauthorized();
    }

    var tracks = await trackService.GetAllTracksAsync(user.Id); // Передаем user.Id
    return Results.Ok(tracks);
       });


app.MapGet("/api/tracks/stream/{trackId}", async (int trackId, TrackService
trackService, HttpContext httpContext) => {
    if (httpContext.Items["User"] == null)
    {
        return Results.Unauthorized();
    }

    if (trackId <= 0)
    {
        return Results.BadRequest("Некорректный ID трека.");
```

```
    }

    var track = await trackService.GetTrackByIdAsync(trackId);
    if (track == null || string.IsNullOrEmpty(track.FilePath) ||
!System.IO.File.Exists(track.FilePath))
    {
        return Results.NotFound("Трек не найден или файл отсутствует на сервере.");
    }

    return Results.File(track.FilePath, contentType: "application/octet-stream",
enableRangeProcessing: true, fileDownloadName: track.FileName);
      });
```

Клиентское WPF-приложение получает список треков и отображает его. При выборе трека для воспроизведения, MainViewModel инициирует запрос к эндпоинту /api/tracks/stream/{trackId}. Полученный поток (или файл) передается в MediaPlayer для воспроизведения.

3.1.3 Поиск треков

Реализована функция поиска треков в хранилище пользователя по названию или исполнителю через HTTP GET-запрос к эндпоинту /api/tracks/search. Код функции предоставлен ниже.

```
app.MapGet("/api/tracks/search", async (string query, TrackService trackService,
HttpContext httpContext) => {
    var user = httpContext.Items["User"] as User;
    if (user == null)
        return Results.Unauthorized();

    if (string.IsNullOrWhiteSpace(query))
        return Results.BadRequest("Поисковый запрос не может быть пустым.");

    var tracks = await trackService.SearchTracksAsync(query, user.Id); // Передаем
userId для потенциального поиска только по трекам пользователя
    return Results.Ok(tracks);
});
```

Получив поисковый запрос от клиента, TrackService выполняет поиск в базе данных SQLite по полям Title и Artist для треков, принадлежащих текущему пользователю. Список найденных треков, включая информацию о том, понравился ли трек пользователю, возвращается клиенту для отображения.

3.1.4 Реализация управления понравившимися треками

Система позволяет пользователям отмечать треки как "понравившиеся" и убирать эту отметку. Эти действия обрабатываются через POST-запросы к соответствующим API-эндпоинтам. Код задействованных функций предоставлен ниже.

Program.cs

```
app.MapPost("/api/userlikedtracks/like/{trackId}", async (int trackId, TrackService
trackService, HttpContext httpContext) => {
```

```
    var user = httpContext.Items["User"] as User; // Предполагается, что класс User
существует и имеет свойство Id
    if (user == null)
        return Results.Unauthorized();

    if (trackId <= 0)
        return Results.BadRequest("Некорректный ID трека.");

    // Предполагается, что TrackService будет иметь метод LikeTrackAsync(int userId,
int trackId)
    var success = await trackService.LikeTrackAsync(user.Id, trackId);

    if (!success)
    {
        // В идеале, TrackService должен возвращать более конкретную информацию об
ошибке
        return Results.Problem(detail: "Не удалось добавить трек в понравившиеся.",
statusCode: StatusCodes.Status500InternalServerError);
    }

    return Results.Ok(new { message = "Трек успешно добавлен в понравившиеся." });
});
TrackService.cs
        public async Task<bool> LikeTrackAsync(int userId, int trackId)
  {
      try
      {
          using var connection = GetConnection();

          var sql = "INSERT OR IGNORE INTO UserLikedTracks (UserId, TrackId) VALUES
(@UserId, @TrackId);";
          await connection.ExecuteAsync(sql, new { UserId = userId, TrackId = trackId
});
          return true;
      }
      catch (Exception ex)
          {

          Console.WriteLine($"Ошибка при добавлении лайка для UserId {userId}, TrackId
{trackId}: {ex.Message}");
          return false;
      }
  }
```

При взаимодействии пользователя с элементом "сердечко" в WPF-клиенте, отправляется запрос на сервер. TrackService обновляет соответствующую запись в таблице UserLikedTracks в базе данных. Эндпоинт GET /api/userlikedtracks позволяет получить список всех понравившихся треков.

3.2 Аутентификация и управление пользователями

Для защиты данных и обеспечения персонализированного доступа реализованы механизмы регистрации и аутентификации.

3.2.1 Реализация регистрации пользователя

Система позволяет создать учетную запись используя клиент через POST запросы на эндпоинт /api/auth/register, предоставив никнейм, email и

пароль. Код данного метода приведен ниже. Фрагмент кода эндпоинта регистрации (Program.cs):

```
app.MapPost("/api/auth/register", async (UserRegisterRequest req, AuthService
authService) =>
{
    if (string.IsNullOrWhiteSpace(req.Nickname) || string.IsNullOrWhiteSpace(req.Email)
|| string.IsNullOrWhiteSpace(req.Password))
    {
        return Results.BadRequest(new { message = "Nickname, email, and password are
required." });
    }
    var (success, message, user) = await authService.RegisterAsync(req.Nickname,
req.Email, req.Password);
    if (success)
    {
        return Results.Ok(new { message, userId = user?.Id, nickname = user?.Nickname,
email = user?.Email });
    }
    return Results.BadRequest(new { message });
});
```

AuthService.cs на сервере производит валидацию данных, проверяет уникальность никнейма/email, хеширует пароль и сохраняет информацию о новом пользователе в базу данных.

3.2.2 Реализация авторизации пользователя

Авторизация зарегистрированных пользователей осуществляется через эндпоинт /api/auth/login по предоставленным никнейму и паролю. Код функции предоставлен ниже.

```
app.MapPost("/api/auth/login", async (UserLoginRequest req, AuthService authService) =>
{
    if (string.IsNullOrWhiteSpace(req.Nickname) ||
string.IsNullOrWhiteSpace(req.Password))
    {
        return Results.BadRequest(new { message = "Nickname and password are required."
});
    }
    var (success, message, token, user) = await authService.LoginAsync(req.Nickname,
req.Password);
    if (success && token != null && user != null)
    {
        return Results.Ok(new { message, token, userId = user.Id, nickname =
user.Nickname, email = user.Email });
    }
    return Results.Json(new  { message }, statusCode:
StatusCodes.Status401Unauthorized);
});
```

После успешной проверки учетных данных, AuthService генерирует Bearer токен доступа. Этот токен возвращается клиенту и используется для аутентификации всех последующих запросов к защищенным API.

3.3 Работа с базой данных

Вся информация о пользователях, музыкальных треках, их метаданных и связях (например, "понравившиеся треки") хранится в реляционной базе данных SQLite. Выбор SQLite обусловлен его простотой интеграции, легковесностью и отсутствием необходимости в отдельном серверном процессе БД, что удобно для данного курсового проекта.

3.3.1 Структура базы данных

База данных спроектирована для эффективного хранения и управления основной информацией приложения и состоит из следующих ключевых таблиц:

– Users: содержит информацию об учетных записях пользователей, включая уникальный идентификатор (ID), никнейм, адрес электронной почты, хешированный пароль и токен аутентификации.

– Tracks: основная таблица для хранения метаданных о музыкальных треках. Включает уникальный идентификатор трека, ID пользователя-владельца, название, исполнителя, название альбома (опционально), имя файла на сервере, полный путь к файлу на сервере, дату загрузки и длительность трека.

– UserLikedTracks: связующая таблица, реализующая отношение "многие-ко-многим" между пользователями и треками. Хранит пары ID пользователя и ID трека, указывая, какие треки понравились конкретным пользователям.

Для представления этих таблиц в коде серверной части используются C#-классы (модели данных). Ниже представлен код этих моделей.

Класс-модель User

```
public class User
{
    public int Id { get; set; }
    public string Nickname { get; set; }
    public string Email { get; set; }
    public string PasswordHash { get; set; }
    public string? AuthToken { get; set; }
}
```

Класс-модель Track

```
public class Track
{
    public int Id { get; set; }
    public int UserId { get; set; }
    public string Title { get; set; }
    public string? Artist { get; set; }
```

Класс-модель Track (продолжение)

```
    public string? Album { get; set; }
    public string FileName { get; set; }
    public string FilePath { get; set; }
    public string UploadedAt { get; set; }
```

```
    public double Duration { get; set; }
    public bool IsLiked { get; set; }
}
```

Структура базы данных логично отражает ключевые сущности музыкального хранилища. Использование внешних ключей (например, между Tracks.UserId и Users.Id, а также в UserLikedTracks) с правилом ON DELETE CASCADE обеспечивает поддержание целостности данных при удалении пользователей или треков.

3.3.2 Реализация взаимодействия с базой данных в коде

Взаимодействие с базой данных SQLite на стороне сервера (test_server) является ключевым аспектом для хранения и извлечения всей информации о пользователях, музыкальных треках и их связях. Для этой цели в проекте используется комбинация провайдера данных System.Data.SQLite.Core и микро-ORM Dapper. Такой подход обеспечивает баланс между производительностью, гибкостью написания SQL-запросов и удобством маппинга результатов на C#-объекты.

Централизованное управление строкой подключения к базе данных реализовано через класс ConnectionStringHolder, который регистрируется в системе внедрения зависимостей (Dependency Injection) как singleton. Это позволяет всем сервисам, которым необходим доступ к базе данных, получать единообразную и актуальную строку подключения. Ниже приведен фрагмент регистрации ConnectionStringHolder в Program.cs:

```
builder.Services.AddSingleton(new ConnectionStringHolder("Data Source=music_library.db"));
```

Основная логика работы с базой данных инкапсулирована в сервисных классах, таких как AuthService (для аутентификации и управления пользователями) и TrackService (для операций с музыкальными треками).

Эти сервисы получают экземпляр ConnectionStringHolder через конструктор благодаря внедрению зависимостей. Внутри методов этих сервисов создается и открывается соединение с базой данных (SQLiteConnection) для выполнения необходимых операций. Ниже приведен пример создания соединения в сервисном классе AuthService:

```
using (var connection = new SQLiteConnection(_connectionString))
{
    await connection.OpenAsync();

    // Проверка, существует ли пользователь с таким Nickname или Email
    var checkUserCommand = connection.CreateCommand();
    checkUserCommand.CommandText = "SELECT COUNT(1) FROM Users WHERE Nickname =
$nickname OR Email = $email";
    checkUserCommand.Parameters.AddWithValue("$nickname", nickname);
    checkUserCommand.Parameters.AddWithValue("$email", email);
    var userExists = Convert.ToInt32(await checkUserCommand.ExecuteScalarAsync()) > 0;

    if (userExists)
```

```
    {
        // Можно добавить более конкретное сообщение, какое поле уже занято, если нужно
        return (false, "User with this nickname or email already exists.", null);
    }

    // Создание нового пользователя
    var command = connection.CreateCommand();
    command.CommandText =
    @"
        INSERT INTO Users (Nickname, Email, PasswordHash)
        VALUES ($nickname, $email, $passwordHash);
        SELECT last_insert_rowid();
    ";
    command.Parameters.AddWithValue("$nickname", nickname);
    command.Parameters.AddWithValue("$email", email);
    command.Parameters.AddWithValue("$passwordHash", passwordHash);

    var userId = Convert.ToInt32(await command.ExecuteScalarAsync());

    var newUser = new User { Id = userId, Nickname = nickname, Email = email,
PasswordHash = passwordHash };
    return (true, "User registered successfully.", newUser);
       }
```

Использование Dapper позволяет писать нативные SQL-запросы, что дает полный контроль над структурой запроса и его оптимизацией, и при этом предоставляет удобные методы расширения. Ниже приведен пример использования Dapper для извлечения данных в TrackService:

```
        public async Task<IEnumerable<Track>> SearchTracksAsync(string query, int
userId)
        {
            using var connection = GetConnection();
            var searchQuery = $"%{query.ToLower()}%";
            var sql = "SELECT Id, Title, Artist, Album, Duration, FilePath, FileName,
UserId, UploadedAt FROM Tracks " +
                      "WHERE UserId = @UserId AND (LOWER(Title) LIKE @Query OR
LOWER(Artist) LIKE @Query) " +
                      "ORDER BY Title;";
            return await connection.QueryAsync<Track>(sql, new { UserId = userId, Query
= searchQuery });
              }
```

В данном примере Dapper выполняет SQL-запрос и автоматически заполняет список объектов Track на основе полученных из базы данных строк. Параметризация запросов (@UserId, @Query) помогает предотвратить SQL-инъекции.

Инициализация схемы базы данных, включая создание таблиц Users, Tracks и UserLikedTracks, если они отсутствуют, а также применение необходимых изменений схемы (например, добавление новых колонок), выполняется классом DatabaseSetup при старте приложения..

4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В процессе тестирования облачного музыкального хранилища были выявлены и устранены различные ошибки и недочёты, затрагивающие как клиентскую (WPF приложение), так и серверную (ASP.NET Core Web API) части программного средства.

Одной из ключевых проблем, обнаруженных на этапе интеграционного тестирования, являлась некорректная работа механизма аутентификации и авторизации при доступе к защищенным API-эндпоинтам. Изначально клиентское приложение испытывало трудности с отправкой Bearer токена или сервер некорректно его обрабатывал, что приводило к ошибкам доступа (например, HTTP 401 Unauthorized) при попытке загрузить трек или получить список понравившихся.

Для диагностики этой проблемы было проведено тестирование серверных эндпоинтов через Postman. В ходе этих тестов было подтверждено, что:

– Эндпоинт /api/auth/login корректно генерирует и возвращает Bearer токен;

– При передаче этого токена в заголовке Authorization: Bearer <token> к защищенным эндпоинтам (например, /api/tracks/upload), сервер правильно идентифицирует пользователя.

Проблема на стороне клиента была связана с инициализацией и передачей токена в ApiService. Была скорректирована логика сохранения токена после входа и его автоматического добавления в заголовки всех последующих запросов к защищенным ресурсам.

После исправления клиентское приложение стало успешно проходить аутентификацию и получать доступ к защищенным функциям сервера.

Другой комплекс проблем был связан с асинхронным взаимодействием и управлением ресурсами, в частности, при загрузке файлов. На клиенте возникала ошибка, указывающая на преждевременное закрытие файлового потока (FileStream) до того, как HttpClient успевал отправить все данные. Это, в свою очередь, приводило к отправке некорректного запроса и ошибке на сервере.

Проблема была решена путем изменения логики управления жизненным циклом потока: в ApiService.cs входящий FileStream теперь копируется в MemoryStream (или используется ByteArrayContent), содержимое которого полностью буферизуется перед отправкой.

В процессе тестирования серверной части через Postman также были проверены и отлажены следующие функции:

– Регистрация новых пользователей (корректность создания учетных записей и обработки уже существующих пользователей);

– Загрузка и получение метаданных треков (успешное сохранение файлов и извлечение информации с помощью TagLib#).

– Управление "понравившимися" треками: Проверка добавления и удаления лайков через эндпоинты /api/userlikedtracks/like/{trackId} и /api/userlikedtracks/unlike/{trackId}, а также получение списка избранного.

– Потоковое воспроизведение и удаление треков: Корректная отдача аудиофайлов и их удаление из системы.

На стороне WPF-клиента также был выявлен и устранен ряд проблем:

– Ошибки привязки данных XAML (XamlParseException): Множественные ошибки, связанные с ненайденными ресурсами (стили, конвертеры), были устранены путем корректного определения и регистрации ресурсов в App.xaml и Resources/Styles.xaml, а также проверкой Build Action для XAML-файлов и файлов конвертеров.

– Некорректное возобновление воспроизведения: Проблема, когда трек начинался заново после постановки на паузу и повторного нажатия Play, была решена путем изменения логики в MainViewModel.cs и MainWindow.xaml.cs для корректного управления MediaPlayer.

Большинство проблем, возникавших на начальных этапах разработки и при интеграции компонентов, были успешно идентифицированы и устранены в процессе тестирования отдельных функций, тестирования взаимодействия клиента и сервера, а также ручного тестирования пользовательских сценариев в WPF-приложении.

Финальная версия программного средства демонстрирует стабильную работу ключевого функционала и корректную обработку основных пользовательских сценариев.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Интерфейс программного средства

5.1.1 Окно регистрации и авторизации

При первом запуске приложения или после выхода из системы пользователю отображается окно входа. В этом окне пользователь может ввести свои учетные данные (никнейм и пароль) для входа. Если у пользователя еще нет учетной записи, он может перейти к форме регистрации, нажав соответствующую кнопку. Форма регистрации содержит следующие поля: никнейм, адрес электронной почты, пароль и повторный пароль для надёжности.

Если поля регистрации будут заполнены правильно, приложение оповестит о том, что вы можете войти в систему со своими данными.

Если же пользователь не заполнит все поля или совершит ошибку при вводе, то по нажатию на «Зарегистрироваться» появится модальное окно, оповещающее о том, что он сделал не так.

Внешний вид окна регистрации и авторизации представлен на рисунке 5.1.

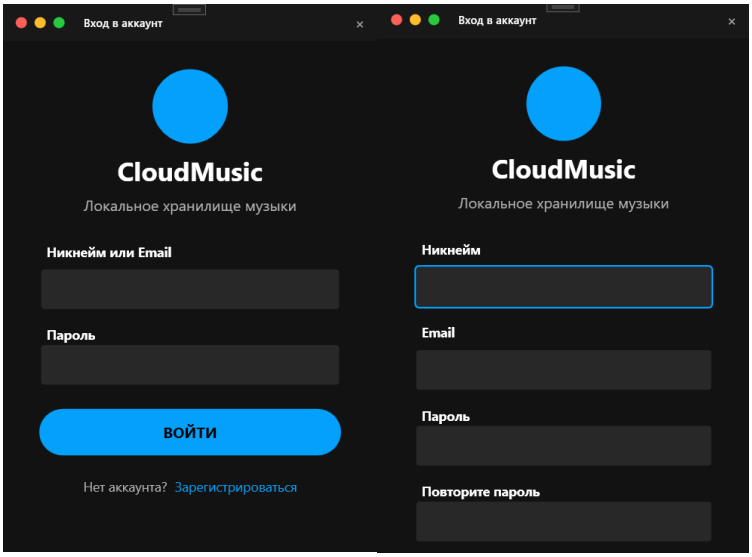

Рисунок 5.1 – Окно регистрации и авторизации

5.1.2 Представление «Все треки» / «Главная»

После успешной авторизации пользователь попадает в главное окно приложения, где по умолчанию отображается представление «Главная», содержащее список всех загруженных им музыкальных треков. В верхней части этого представления располагается поле для ввода поискового запроса и кнопка «Найти», позволяющие фильтровать список по названию трека или имени исполнителя.

Основную часть занимает сам список треков, где каждая композиция представлена с указанием названия, исполнителя и длительности. Рядом с каждым треком находятся интерактивные элементы управления: кнопка для запуска/паузы воспроизведения, иконка "сердечко" для добавления в понравившиеся (или удаления из них), кнопка для скачивания трека и кнопка для его удаления из хранилища. Текущий воспроизводимый трек визуально выделяется в списке.

Внешний вид окна настроек в свернутом и развернутом виде представлен на рисунке 5.2.

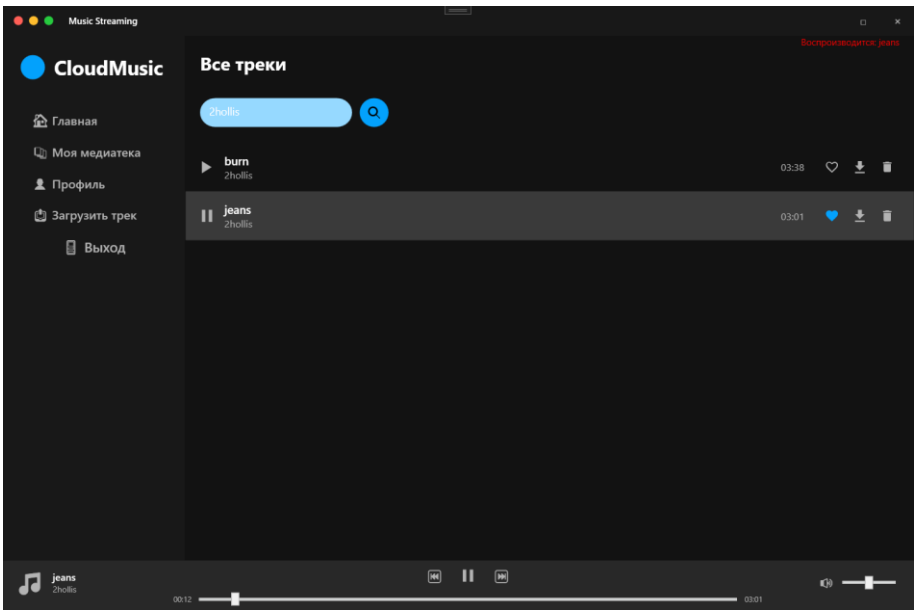

Рисунок 5.2 – Окно представления «Главное»

5.1.3 Представления «Моя медиатека»

Представление «Моя медиатека» по своей структуре и способу отображения списка треков идентично представлению «Главная». Однако ключевым отличием является то, что в данном разделе выводятся только те музыкальные композиции, которые пользователь ранее отметил как "понравившиеся". Если пользователь еще не добавил ни одного трека в понравившиеся, в этом разделе будет отображено соответствующее уведомление. Это позволяет быстро получить доступ к избранной музыке.

Внешний вид окна «Моя медиатека» представлен на рисунке 2.5.

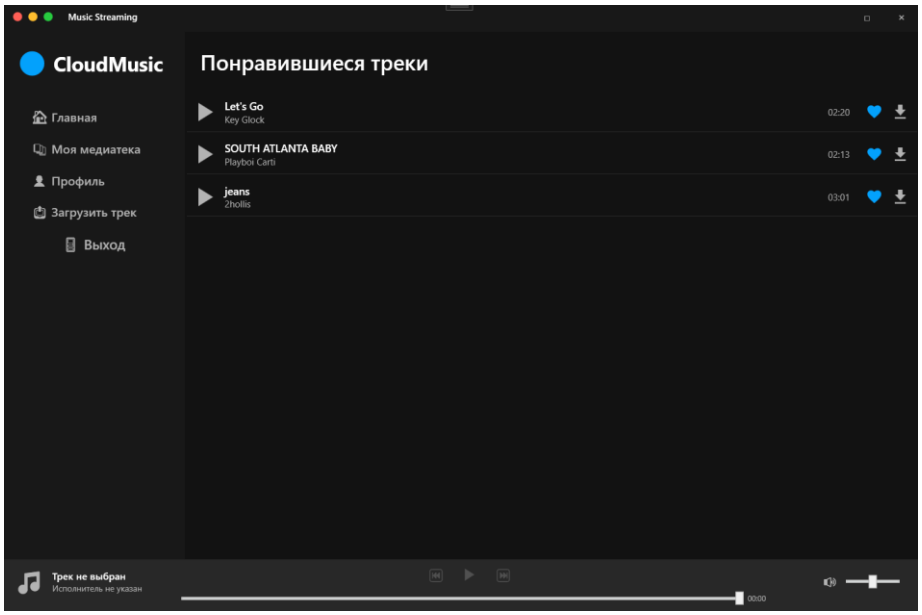

Рисунок 5.3 – Окно «Моя медиатека»

5.1.4 Представление «Загрузить трек»

Для добавления новых аудиофайлов в личное хранилище предназначен раздел «Загрузить трек». В этом представлении пользователю предоставляется кнопка для выбора аудиофайла с локального компьютера (поддерживаются форматы .mp3, .wav, .ogg). После выбора файла отображается его имя, и пользователь имеет возможность опционально указать или скорректировать название трека и имя исполнителя.

Запуск процесса загрузки на сервер осуществляется нажатием кнопки «Загрузить на сервер». Во время загрузки может отображаться индикатор прогресса, а по завершении операции – статусное сообщение об успехе или ошибке.

Внешний вид представления «Загрузить трек» отображен на рисунке 2.4.

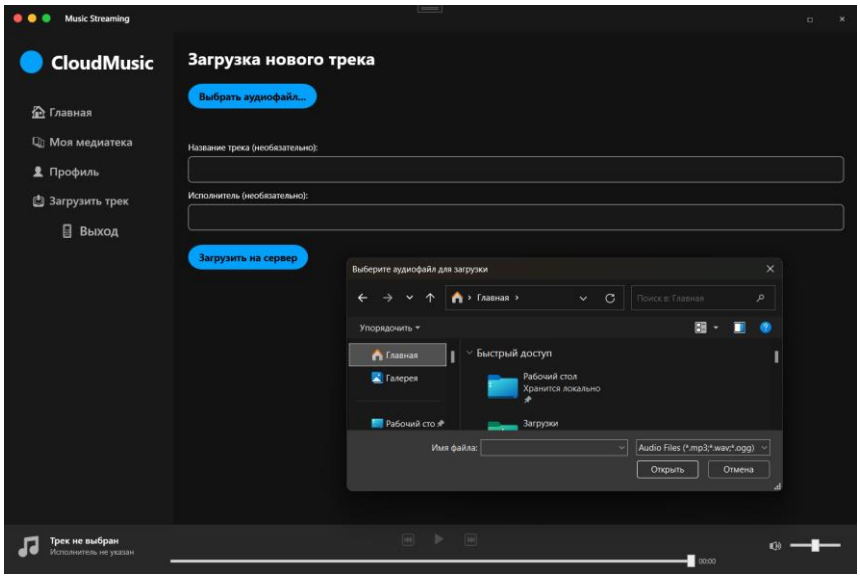

Рисунок 5.4 – Внешний вид окна для загрузки треков

5.2 Управление программным средством

5.2.1 Элементы управления плеером и списками треков

Основные элементы управления сосредоточены в нижней панели плеера и в списках треков. Панель плеера позволяет управлять воспроизведением текущего трека: запускать и приостанавливать воспроизведение, переключаться на предыдущий или следующий трек в текущем отображаемом списке, регулировать громкость и перематывать композицию с помощью слайдера прогресса. В списках треков (в разделах «Главная» и «Моя медиатека») для каждого трека доступны индивидуальные элементы управления:

– Запуск/пауза воспроизведения (также инициируется двойным кликом по треку).
– Добавление трека в «Понравившиеся» или удаление из них (кнопка-«сердечко»).
– Скачивание файла трека на компьютер пользователя.
– Удаление трека из облачного хранилища (с запросом подтверждения).
При наведении на интерактивные элементы могут отображаться всплывающие подсказки, поясняющие их назначение.

ЗАКЛЮЧЕНИЕ

В условиях растущей популярности цифровой музыки, потребность в удобных инструментах для управления личными музыкальными коллекциями остается высокой. Данный курсовой проект был посвящен разработке программного средства «Сетевое файловое хранилище музыки», предназначенного для хранения, организации и совместного прослушивания аудиофайлов в рамках локальной сети.

В ходе работы были успешно решены все ключевые задачи. Разработано клиент-серверное приложение, где серверная часть на ASP.NET Core обеспечивает загрузку треков с автоматическим извлечением метаданных (название, исполнитель, длительность), хранение файлов и их потоковую передачу. Клиентское WPF-приложение предоставляет пользователям интерфейс для доступа к общей музыкальной библиотеке, поиска, воспроизведения и скачивания треков. Реализован плеер с базовыми функциями управления.

Проект продемонстрировал принципы создания современных десктопных приложений и веб-API, работы с базами данных и файловыми потоками.

Существуют перспективы для дальнейшего развития: создание плейлистов, расширение возможностей поиска и интеграция с внешними сервисами метаданных.

Разработанное приложение является практическим примером создания системы для управления музыкальным контентом и позволило углубить знания в области программной инженерии и технологий .NET.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Grinberg M. Flask Web Development: Developing Web Applications with Python. – 2nd ed. – O'Reilly Media, 2018. – 320 p.

[2] Sebastián Ramírez. FastAPI Documentation [Электронный ресурс]. – Режим доступа: https://fastapi.tiangolo.com/

[3] Vasiliev A.A. PostgreSQL для начинающих. – СПб.: БХВ-Петербург, 2019. – 352 с.

[4] Tanenbaum A.S., Van Steen M. Распределённые системы: принципы и парадигмы. – М.: Издательский дом "Вильямс", 2020. – 832 с.

[5] Beazley D. Python. Подробный справочник– СПб.: Питер,
2021. – 880 с.

[6] Соловьев С.В. Проектирование REST API. Практика создания веб-сервисов. – М.: ДМК Пресс, 2021. – 256 с.

[7] Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 4.5 на языке C#. – 4-е изд. – СПб.: Питер, 2013. – 896 с.

[8] Троелсен Э., Джепикс Ф. Язык программирования C# 12 и платформа .NET 8. – 12-е изд. (или наиболее актуальное на момент написания). – СПб.: Диалектика (или М.: Вильямс), 2024 . – 1359 с.

[9] Microsoft. Документация по Windows Presentation Foundation (WPF) [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/dotnet/desktop/wpf/. – Дата доступа: 06.06.2025.

[10] Microsoft. Документация по ASP.NET Core [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/aspnet/core/. – Дата доступа: 06.06.2025.

ПРИЛОЖЕНИЕ А. Исходный код программы

“MainViewModel.cs”

```
using System;
using System.Collections.Generic; // Для List<byte>
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.IO; // Для MemoryStream
using System.Linq;
using System.Net.Http;
using System.Runtime.CompilerServices;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Input;
using MusicClient.Models;
using MusicClient.Services;
using MusicClient.Views;
using Microsoft.Win32;
using System.Windows.Threading;
using System.Diagnostics; // <--- ДОБАВИТЬ

namespace MusicClient.ViewModels
{
   public class MainViewModel : INotifyPropertyChanged, IAsyncDisposable
   {
      private readonly ApiService _apiService;
      private readonly AuthResponseDto _currentUser;
      private TrackDto _selectedTrack;
      private bool _isPlaying;
      private string _nowPlayingText;
      private bool _isLoading;
      private object _currentView;
      private int? _currentlyLoadedTrackId = null; // ID трека, который сейчас загружен в CurrentAudioFilePath

      // Состояния для стриминга
      private TrackDto _streamingTrackInfo; // Информация о треке, который стримится
      private List<byte> _audioBuffer = new List<byte>();
      private bool _isStreamingActive = false;
      private bool _isBuffering = false;
      private string _streamErrorMessage = string.Empty;

      // Свойство для текущего потока аудиоданных, который будет использоваться плеером
      // Это свойство будет установлено, когда стриминг завершен и данные готовы
      private MemoryStream _currentAudioStream;
      public MemoryStream CurrentAudioStream
      {
         get => _currentAudioStream;
         private set
         {
            _currentAudioStream = value;
            OnPropertyChanged();
         }
      }

      public object CurrentView
      {
         get => _currentView;
         private set { _currentView = value; OnPropertyChanged(); }
      }
      public ICommand ShowAllTracksCommand { get; private set; }
      // public ICommand ShowSearchCommand { get; private set; } // <-- ЗАКОММЕНТИРОВАТЬ ЭТУ СТРОКУ
      public ICommand ShowFavoritesCommand { get; private set; }
      public ICommand ShowProfileCommand { get; private set; }
```

```
    public ICommand ShowUploadTrackViewCommand { get; private set; }

    private string _searchQuery;
    public string SearchQuery
    {
      get => _searchQuery;
      set
      {
        _searchQuery = value;
        OnPropertyChanged();
      }
    }

    public ObservableCollection<TrackDto> AllTracks { get; private set; }
    public ObservableCollection<TrackDto> FavoriteTracks { get; private set; }
    public ObservableCollection<PlaylistDto> UserPlaylists { get; private set; }

    public TrackDto SelectedTrack
    {
      get => _selectedTrack;
      set
      {
        bool isNewInstanceOfSameId = _selectedTrack != null && value != null && _selectedTrack.Id == value.Id &&
!ReferenceEquals(_selectedTrack, value);

        if (isNewInstanceOfSameId)
        {
          // Это новый экземпляр DTO для того же трека.
          // Просто обновляем ссылку и IsCurrent, не перезапуская воспроизведение.
          if (_selectedTrack != null)
          {
            _selectedTrack.IsCurrent = false;
          }
          _selectedTrack = value;
          if (_selectedTrack != null)
          {
            _selectedTrack.IsCurrent = true;
          }
          OnPropertyChanged();
        }
        else if (_selectedTrack != value) // Действительно другой трек или изменение на/с null
        {
          if (_selectedTrack != null)
          {
            _selectedTrack.IsCurrent = false; // Снимаем выделение со старого
          }

          var oldTrack = _selectedTrack;
          _selectedTrack = value; // Устанавливаем новый трек

          if (_selectedTrack != null) // Если выбран новый трек (не null)
          {
            _selectedTrack.IsCurrent = true;
            NowPlayingText = $"Выбран: {_selectedTrack.Title} - {_selectedTrack.Artist}";
            OnPropertyChanged(); // Уведомляем, что SelectedTrack изменился

            // Важно: Если это тот же трек, что и играл (например, пользователь кликнул на него снова),
            // то PlayCommand.Execute(null) должен просто переключить Play/Pause, а не перезагружать.
            // TogglePlayPause должен это корректно обработать, если _currentlyLoadedTrackId совпадает.
            // Если это действительно новый трек (ID отличается от _currentlyLoadedTrackId или oldTrack.Id),
            // то нужно сбросить состояние и загрузить.

            bool shouldResetAndPlay = oldTrack == null || oldTrack.Id != _selectedTrack.Id ||
string.IsNullOrEmpty(CurrentAudioFilePath) || _currentlyLoadedTrackId != _selectedTrack.Id;
```

```
            if (shouldResetAndPlay)
            {
                IsPlaying = false; // Останавливаем предыдущий, если он играл и это был другой трек
                CurrentAudioFilePath = null;
                _currentlyLoadedTrackId = null;
                CurrentPosition = 0;
            }

            // Вызываем команду воспроизведения. TogglePlayPause разберется,
            // нужно ли загружать трек или просто возобновить/поставить на паузу.
            if (PlayCommand.CanExecute(null))
            {
                PlayCommand.Execute(null);
            }
        }
        else // Новый выбранный трек - null (например, все треки удалены)
        {
            NowPlayingText = "Трек не выбран";
            IsPlaying = false;
            CurrentAudioFilePath = null;
            _currentlyLoadedTrackId = null;
            CurrentPosition = 0;
            OnPropertyChanged(); // Уведомляем, что SelectedTrack изменился (стал null)
        }
    }
    // Если _selectedTrack == value (тот же самый экземпляр), ничего не делаем.
  }
}

public bool IsPlaying
{
  get => _isPlaying;
  set
  {
    if (_isPlaying != value)
    {
      _isPlaying = value;
      if (_isPlaying && SelectedTrack != null)
      {
        // _progressTimer.Start();
      }
      else
      {
        // _progressTimer.Stop();
      }
      OnPropertyChanged();
      OnPropertyChanged(nameof(PlayButtonText));
    }
  }
}

public string NowPlayingText
{
  get => _nowPlayingText;
  set { _nowPlayingText = value; OnPropertyChanged(); }
}

public bool IsLoading
{
  get => _isLoading;
  set { _isLoading = value; OnPropertyChanged(); }
}

public bool IsStreamingActive
{
```

```
    get => _isStreamingActive;
    private set { _isStreamingActive = value; OnPropertyChanged(); }
}

public bool IsBuffering
{
    get => _isBuffering;
    private set { _isBuffering = value; OnPropertyChanged(); }
}

public string StreamErrorMessage
{
    get => _streamErrorMessage;
    private set { _streamErrorMessage = value; OnPropertyChanged(); }
}

public string PlayButtonText => IsPlaying ? "Пауза" : "Воспроизвести";

public ICommand PlayCommand { get; private set; }
public ICommand ToggleFavoriteCommand { get; private set; }
public ICommand ShuffleCommand { get; private set; }
public ICommand PreviousTrackCommand { get; private set; }
public ICommand NextTrackCommand { get; private set; }
public ICommand RepeatCommand { get; private set; }
public ICommand CreatePlaylistCommand { get; private set; }
public ICommand ShowLikedSongsCommand { get; private set; }
public ICommand ShowRadioCommand { get; private set; }
public ICommand OpenPlaylistCommand { get; private set; }
public ICommand ShowLyricsCommand { get; private set; }
public ICommand ShowQueueCommand { get; private set; }
public ICommand VolumeCommand { get; private set; }
public ICommand DownloadCommand { get; private set; }
public ICommand SelectTrackCommand { get; private set; }
public ICommand SearchCommand { get; private set; }
public ICommand DeleteTrackCommand { get; private set; }

private string _currentAudioFilePath;
public string CurrentAudioFilePath
{
    get => _currentAudioFilePath;
    set { _currentAudioFilePath = value; OnPropertyChanged(); }
}

private double _currentPosition;
public double CurrentPosition
{
    get => _currentPosition;
    set { _currentPosition = value; OnPropertyChanged(); }
}
private double _totalDuration;
public double TotalDuration
{
    get => _totalDuration;
    set { _totalDuration = value; OnPropertyChanged(); }
}
private string _currentTime;
public string CurrentTime
{
    get => _currentTime;
    set { _currentTime = value; OnPropertyChanged(); }
}
private string _totalTime;
public string TotalTime
{
    get => _totalTime;
```

```
        set { _totalTime = value; OnPropertyChanged(); }
    }

    private double _volume = 0.5;
    public double Volume
    {
        get => _volume;
        set { _volume = value; OnPropertyChanged(); }
    }

    public MainViewModel(ApiService apiService, AuthResponseDto currentUser)
    {
        _apiService = apiService ?? throw new ArgumentNullException(nameof(apiService));
        _currentUser = currentUser ?? throw new ArgumentNullException(nameof(currentUser));

        // Подписываемся на события стриминга от ApiService
        _apiService.OnTrackInfoReceived += HandleTrackInfoReceived;
        _apiService.OnAudioChunkReceived += HandleAudioChunkReceived;
        _apiService.OnStreamFinished += HandleStreamFinished;
        _apiService.OnStreamError += HandleStreamError;

        InitializeCollectionsAndCommands();
        LoadInitialData();
    }

    private void InitializeCollectionsAndCommands()
    {
        AllTracks = new ObservableCollection<TrackDto>();
        FavoriteTracks = new ObservableCollection<TrackDto>();
        UserPlaylists = new ObservableCollection<PlaylistDto>();

        NowPlayingText = "Ничего не воспроизводится";

        ShowAllTracksCommand = new RelayCommand(async _ => { await LoadAllTracksAsync(); SetView(typeof(AllTracksView));
});
        // ShowSearchCommand = new RelayCommand(_ => SetView(typeof(SearchView))); // <-- ЗАКОММЕНТИРОВАТЬ ЭТУ
СТРОКУ
        ShowFavoritesCommand = new RelayCommand(async _ => { await LoadFavoriteTracksAsync();
SetView(typeof(FavoritesView)); });
        ShowProfileCommand = new RelayCommand(_ => SetView(typeof(ProfileView)));
        ShowUploadTrackViewCommand = new RelayCommand(_ => SetView(typeof(UploadTrackView)));

        PlayCommand = new RelayCommand(TogglePlayPause, CanPlayPause);
        ToggleFavoriteCommand = new RelayCommand(async param => await ToggleFavoriteTrack(param),
CanToggleFavoriteTrack);
        DownloadCommand = new RelayCommand(async param => await DownloadTrack(param), CanDownloadTrack);
        SelectTrackCommand = new RelayCommand(SelectTrackAction);
        SearchCommand = new RelayCommand(async _ => await ExecuteSearchAsync());
        DeleteTrackCommand = new RelayCommand(async param => await ExecuteDeleteTrackAsync(param), param =>
CanExecuteDeleteTrack(param));
        ShuffleCommand = new RelayCommand(_ => ShowNotImplementedMessage("Перемешивание"));
        PreviousTrackCommand = new RelayCommand(_ => PreviousTrack(), _ => AllTracks.Any() && SelectedTrack != null);
        NextTrackCommand = new RelayCommand(_ => NextTrack(), _ => AllTracks.Any() && SelectedTrack != null);
        RepeatCommand = new RelayCommand(_ => ShowNotImplementedMessage("Повтор трека/плейлиста"));
        CreatePlaylistCommand = new RelayCommand(CreatePlaylistAction);
        ShowLikedSongsCommand = new RelayCommand(async _ => { await LoadFavoriteTracksAsync(); /*
SetView(typeof(FavoritesView)); */ });
        ShowRadioCommand = new RelayCommand(_ => ShowNotImplementedMessage("Радио"));
        OpenPlaylistCommand = new RelayCommand(OpenPlaylistAction, CanOpenPlaylist);
        ShowLyricsCommand = new RelayCommand(_ => ShowNotImplementedMessage("Текст песни"));
        ShowQueueCommand = new RelayCommand(_ => ShowNotImplementedMessage("Очередь воспроизведения"));
        VolumeCommand = new RelayCommand(param => { if (double.TryParse(param?.ToString(), out double vol)) Volume =
vol; });
    }
```

```
        private void LoadInitialData()
        {
            LoadAllTracksAsync(); // Загружаем все треки при старте
            LoadFavoriteTracksAsync(); // Загружаем избранные треки при старте
            LoadUserPlaylistsAsync(); // Загружаем плейлисты пользователя

            // Устанавливаем начальное представление
            SetView(typeof(AllTracksView));
        }
        // Обработчики событий от ApiService для стриминга
        private Task HandleTrackInfoReceived(TrackDto trackInfo)
        {
            _streamingTrackInfo = trackInfo;
            _audioBuffer.Clear();
            IsStreamingActive = true;
            IsBuffering = true;
            StreamErrorMessage = string.Empty;
            NowPlayingText = $"Загрузка: {trackInfo.Title} - {trackInfo.Artist}";
            CurrentAudioStream = null; // Очищаем предыдущий стрим
            OnPropertyChanged(nameof(NowPlayingText)); // Уведомить UI
            return Task.CompletedTask;
        }
        private Task HandleAudioChunkReceived(byte[] chunk)
        {
            if (IsStreamingActive)
            {
                _audioBuffer.AddRange(chunk);
            }
            return Task.CompletedTask;
        }
        private Task HandleStreamFinished()
        {
            if (IsStreamingActive)
            {
                IsBuffering = false;
                CurrentAudioStream = new MemoryStream(_audioBuffer.ToArray());
                NowPlayingText = $"Готово к воспр.: {_streamingTrackInfo?.Title}";
                OnPropertyChanged(nameof(NowPlayingText));
            }
            return Task.CompletedTask;
        }
        private Task HandleStreamError(string errorMessage)
        {
            IsStreamingActive = false;
            IsBuffering = false;
            StreamErrorMessage = errorMessage;
            NowPlayingText = $"Ошибка стриминга: {errorMessage}";
            CurrentAudioStream = null;
            OnPropertyChanged(nameof(NowPlayingText));
            OnPropertyChanged(nameof(StreamErrorMessage));
            Application.Current.Dispatcher.Invoke(() =>
                MessageBox.Show($"Ошибка стриминга: {errorMessage}", "Ошибка", MessageBoxButton.OK,
MessageBoxImage.Error));
            return Task.CompletedTask;
        }
        private bool CanPlayPause(object parameter)
        {
            return (SelectedTrack != null && string.IsNullOrEmpty(StreamErrorMessage)) || IsPlaying;
        }
        private async void TogglePlayPause(object parameter)
        {
            if (IsPlaying) // Если уже играет, то ставим на паузу
            {
                IsPlaying = false;
                NowPlayingText = SelectedTrack != null ? $"Пауза: {SelectedTrack.Title}" : "Пауза";
```

```
        }
        else // Если не играет (стоял на паузе или это первый запуск)
        {
            if (SelectedTrack == null) // Повторная проверка на случай, если он стал null асинхронно
            {
                NowPlayingText = "Трек не выбран (ошибка)";
                IsPlaying = false;
                return;
            }
            if (_currentlyLoadedTrackId == SelectedTrack.Id && !string.IsNullOrEmpty(CurrentAudioFilePath) &&
File.Exists(CurrentAudioFilePath))
            {
                IsPlaying = true;
                NowPlayingText = $"Воспроизводится: {SelectedTrack.Title}";
            }
            else
            {
                if (SelectedTrack == null)
                {
                    NowPlayingText = "Ошибка: Трек стал null перед загрузкой.";
                    IsPlaying = false;
                    IsLoading = false;
                    return;
                }
                IsLoading = true;
                Stream stream = null;
                string oldTempFile = CurrentAudioFilePath;
                try
                {
                    // Перед использованием SelectedTrack, убедимся что он не null
                    if (SelectedTrack == null)
                    {
                        throw new InvalidOperationException("SelectedTrack is null before API call in TogglePlayPause.");
                    }
                    NowPlayingText = $"Загрузка: {SelectedTrack.Title}";
                    stream = await _apiService.GetTrackStreamAsync(SelectedTrack.Id);
                    if (stream == null || stream == Stream.Null || !stream.CanRead)
                    {
                        throw new Exception("Не удалось получить поток для трека от сервера.");
                    }

                    if (SelectedTrack == null)
                    { // Еще одна проверка перед использованием SelectedTrack.Id
                        throw new InvalidOperationException("SelectedTrack is null before Path.Combine in TogglePlayPause.");
                    }
                    string tempFile = Path.Combine(Path.GetTempPath(),
$"music_client_track_{SelectedTrack.Id}_{Guid.NewGuid().ToString().Substring(0, 8)}.mp3");

                    using (var fs = new FileStream(tempFile, FileMode.Create, FileAccess.Write))
                    {
                        await stream.CopyToAsync(fs);
                    }
                    CurrentAudioFilePath = tempFile; // Устанавливаем путь к НОВОМУ файлу
                    _currentlyLoadedTrackId = SelectedTrack.Id; // Запоминаем, что этот трек загружен

                    IsPlaying = true; // Это вызовет Open() и Play() в MainWindow.xaml.cs для нового файла
                    NowPlayingText = $"Воспроизводится: {SelectedTrack.Title}";

                    // Попытка удалить старый временный файл, если он был и отличается от нового
                    if (!string.IsNullOrEmpty(oldTempFile) && oldTempFile != CurrentAudioFilePath && File.Exists(oldTempFile))
                    {
                        try { File.Delete(oldTempFile); } catch (IOException ex) { Console.WriteLine($"Не удалось удалить старый
временный файл {oldTempFile}: {ex.Message}"); }
                    }
                }
```

```
            catch (Exception ex)
            {
                NowPlayingText = $"Ошибка загрузки: {ex.Message}";
                IsPlaying = false;
                CurrentAudioFilePath = null; // Сбрасываем путь, так как загрузка не удалась
                _currentlyLoadedTrackId = null; // Сбрасываем ID
                MessageBox.Show($"Не удалось загрузить трек: {ex.Message}", "Ошибка воспроизведения",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
            finally
            {
                stream?.Dispose();
                IsLoading = false;
            }
        }
    }
}

private async Task LoadAllTracksAsync()
{
    IsLoading = true;
    try
    {
        var tracksFromServer = await _apiService.GetAllTracksAsync();

        Application.Current.Dispatcher.Invoke(() =>
        {
            int? playingTrackIdPriorToClear = null;
            bool wasPlayingPriorToClear = IsPlaying && _selectedTrack != null;
            if (wasPlayingPriorToClear)
            {
                playingTrackIdPriorToClear = _selectedTrack.Id;
            }

            AllTracks.Clear();
            if (tracksFromServer != null)
            {
                foreach (var track in tracksFromServer) AllTracks.Add(track);
            }

            if (wasPlayingPriorToClear && playingTrackIdPriorToClear.HasValue)
            {
                var trackInstanceInNewList = AllTracks.FirstOrDefault(t => t.Id == playingTrackIdPriorToClear.Value);
                if (trackInstanceInNewList != null)
                {
                    if (_selectedTrack == null || !ReferenceEquals(_selectedTrack, trackInstanceInNewList))
                    {
                        if (_selectedTrack != null) _selectedTrack.IsCurrent = false;
                        _selectedTrack = trackInstanceInNewList;
                        _selectedTrack.IsCurrent = true;
                         OnPropertyChanged(nameof(SelectedTrack));
                    }
                    // IsPlaying и NowPlayingText не меняем
                }
                else
                {
                    IsPlaying = false;
                    SelectedTrack = null;
                }
            }
            else if (_selectedTrack != null)
            {
                var previouslySelected = AllTracks.FirstOrDefault(t => t.Id == _selectedTrack.Id);
                if (previouslySelected != null) {
                    if (!ReferenceEquals(_selectedTrack, previouslySelected)){
```

```
                    if (_selectedTrack != null) _selectedTrack.IsCurrent = false;
                    _selectedTrack = previouslySelected;
                     _selectedTrack.IsCurrent = true;
                    OnPropertyChanged(nameof(SelectedTrack));
                    NowPlayingText = $"Выбран: {_selectedTrack.Title} - {_selectedTrack.Artist}";
                }
            } else {
                SelectedTrack = null;
            }
        }
      });
    }
    catch (Exception ex)
    {
      System.Diagnostics.Debug.WriteLine($"[LoadAllTracksAsync] Error: {ex.Message}");
    }
    finally
    {
      IsLoading = false;
    }
  }

  private async Task LoadFavoriteTracksAsync()
  {
    IsLoading = true;
    try
    {
      var favTracksFromServer = await _apiService.GetLikedTracksAsync(); // Получаем актуальный список с сервера

      Application.Current.Dispatcher.Invoke(() =>
      {
        int? currentPlayingTrackId = null;
        bool trackWasPlaying = IsPlaying && _selectedTrack != null;

        if (trackWasPlaying)
        {
          currentPlayingTrackId = _selectedTrack.Id;
          Debug.WriteLine($"[LoadFavoriteTracksAsync] Track was playing: ID {currentPlayingTrackId}");
        }

        FavoriteTracks.Clear();
        if (favTracksFromServer != null)
        {
          foreach (var track in favTracksFromServer) FavoriteTracks.Add(track);
        }
        Debug.WriteLine($"[LoadFavoriteTracksAsync] FavoriteTracks loaded. Count: {FavoriteTracks.Count}");

        if (trackWasPlaying && currentPlayingTrackId.HasValue)
        {
          var playingTrackInNewFavorites = FavoriteTracks.FirstOrDefault(t => t.Id == currentPlayingTrackId.Value);

          if (playingTrackInNewFavorites != null)
          {
            Debug.WriteLine($"[LoadFavoriteTracksAsync] Playing track ID {currentPlayingTrackId} FOUND in new
FavoriteTracks.");
            if (_selectedTrack == null || !ReferenceEquals(_selectedTrack, playingTrackInNewFavorites))
            {
              if (_selectedTrack != null) _selectedTrack.IsCurrent = false;
              _selectedTrack = playingTrackInNewFavorites;
              _selectedTrack.IsCurrent = true;
              OnPropertyChanged(nameof(SelectedTrack));
            }
          }
          else
          {
```

```
                Debug.WriteLine($"[LoadFavoriteTracksAsync] Playing track ID {currentPlayingTrackId} NOT found in new
FavoriteTracks. Checking AllTracks.");
                Debug.WriteLine($"[LoadFavoriteTracksAsync] AllTracks current count: {AllTracks.Count}. First few IDs:
{string.Join(", ", AllTracks.Take(5).Select(t=>t.Id))}");

                var playingTrackInAllTracks = AllTracks.FirstOrDefault(t => t.Id == currentPlayingTrackId.Value);
                if (playingTrackInAllTracks != null)
                {
                    Debug.WriteLine($"[LoadFavoriteTracksAsync] Playing track ID {currentPlayingTrackId} FOUND in AllTracks.
Switching _selectedTrack instance.");
                    if (_selectedTrack == null || !ReferenceEquals(_selectedTrack, playingTrackInAllTracks))
                    {
                        if (_selectedTrack != null) _selectedTrack.IsCurrent = false;
                        _selectedTrack = playingTrackInAllTracks;
                        _selectedTrack.IsCurrent = true;
                        OnPropertyChanged(nameof(SelectedTrack));
                    }
                }
                else
                {
                    Debug.WriteLine($"[LoadFavoriteTracksAsync] Playing track ID {currentPlayingTrackId} NOT found in AllTracks
EITHER. Stopping playback.");
                    IsPlaying = false;
                    SelectedTrack = null;
                }
            }
        }
        else if (_selectedTrack != null && !trackWasPlaying)
        {
            var previouslySelected = FavoriteTracks.FirstOrDefault(t => t.Id == _selectedTrack.Id);
            if (previouslySelected != null)
            {
                if (!ReferenceEquals(_selectedTrack, previouslySelected))
                {
                    if (_selectedTrack != null) _selectedTrack.IsCurrent = false;
                    _selectedTrack = previouslySelected;
                    _selectedTrack.IsCurrent = true;
                    OnPropertyChanged(nameof(SelectedTrack));
                    NowPlayingText = $"Выбран: {_selectedTrack.Title} - {_selectedTrack.Artist}";
                }
            }
            else
            {
                SelectedTrack = null;
            }
        }
    });
}
catch (Exception ex)
{
    Debug.WriteLine($"Ошибка загрузки избранных треков: {ex.Message}");
}
finally
{
    IsLoading = false;
}
}

private async void LoadUserPlaylistsAsync()
{
    IsLoading = true;
UserPlaylists.Clear();
    try
    {
    var playlists = await _apiService.GetUserPlaylistsAsync();
```

```
            if (playlists != null)
                {
                foreach (var playlistDto in playlists)
                {
                    UserPlaylists.Add(playlistDto);
                }
                }
                else
                {
                MessageBox.Show("Плейлисты пользователя не были загружены с сервера.", "Ошибка", MessageBoxButton.OK,
MessageBoxImage.Error);
            }
        }
        catch (HttpRequestException ex)
        {
            MessageBox.Show("Нет соединения с сервером или сервер не отвечает.\n" + ex.Message, "Ошибка",
MessageBoxButton.OK, MessageBoxImage.Error);
        }
        catch (TaskCanceledException)
            {
            MessageBox.Show("Время ожидания ответа от сервера истекло.", "Ошибка", MessageBoxButton.OK,
MessageBoxImage.Error);
            }
            catch (Exception ex)
            {
            MessageBox.Show($"Неизвестная ошибка при загрузке плейлистов: {ex.Message}", "Ошибка",
MessageBoxButton.OK, MessageBoxImage.Error);
            }
            finally
            {
                IsLoading = false;
        }
    }

    private bool CanToggleFavoriteTrack(object parameter)
    {
        return parameter is TrackDto track && track != null;
    }

    private async Task ToggleFavoriteTrack(object parameter)
    {
        if (parameter is TrackDto track)
        {
            bool originalIsLiked = track.IsLiked; // Сохраняем текущее состояние
            // Оптимистично обновляем UI
            // track.IsLiked = !track.IsLiked;
            // Не будем делать оптимистичное обновление, дождемся ответа сервера,
            // так как сервер все равно пришлет актуальное состояние IsLiked при следующем GetAllTracks
            // или можно обновить после успешного ответа.

                bool success;
            if (originalIsLiked) // Если трек был лайкнут, то анлайкаем
                {
                success = await _apiService.UnlikeTrackAsync(track.Id);
                if (success)
                {
                    track.IsLiked = false; // Обновляем состояние после успешного ответа
                    // Удаляем из локальной коллекции FavoriteTracks, если он там был
                    var trackInFavorites = FavoriteTracks.FirstOrDefault(t => t.Id == track.Id);
                    if (trackInFavorites != null)
                    {
                        FavoriteTracks.Remove(trackInFavorites);
                    }
                }
                    }
```

```
            else // Если трек не был лайкнут, то лайкаем
            {
                success = await _apiService.LikeTrackAsync(track.Id);
                if (success)
                    {
                    track.IsLiked = true; // Обновляем состояние после успешного ответа
                    // Добавляем в локальную коллекцию FavoriteTracks, если его там еще нет
                    if (!FavoriteTracks.Any(t => t.Id == track.Id))
                    {
                        FavoriteTracks.Add(track);
                    }
                }
            }
            if (!success)
            {
                // Можно показать сообщение об ошибке, если нужно
                MessageBox.Show("Не удалось обновить статус избранного.", "Ошибка", MessageBoxButton.OK,
MessageBoxImage.Error);
                // track.IsLiked = originalIsLiked; // Вернуть UI к исходному состоянию, если операция не удалась
                // Но мы уже решили положиться на серверное состояние при следующей загрузке треков.
                // Либо можно принудительно перезагрузить все треки, чтобы получить 100% актуальное состояние.
            }
            // После лайка/анлайка, если открыт список избранного, его стоит обновить.
            // Можно, например, снова вызвать LoadFavoriteTracksAsync(), если текущее view - это избранное.
            // Или если используется ShowFavoritesCommand, он должен сам загружать актуальные данные.
        }
    }

    private async void CreatePlaylistAction(object obj)
    {
        ShowNotImplementedMessage("Create Playlist (InputDialogWindow is missing)");
    }

    private bool CanOpenPlaylist(object parameter)
    {
        return parameter is PlaylistDto;
    }

    private async void OpenPlaylistAction(object parameter)
    {
        if (parameter is PlaylistDto playlistToOpen)
        {
            IsLoading = true;
            try
            {
                var playlistWithTracks = await _apiService.GetPlaylistAsync(playlistToOpen.Id);
                if (playlistWithTracks?.Tracks != null)
                {
                    var existingPlaylistInVM = UserPlaylists.FirstOrDefault(p => p.Id == playlistToOpen.Id);
                    if (existingPlaylistInVM != null)
                    {
                        existingPlaylistInVM.Tracks = new ObservableCollection<TrackDto>(playlistWithTracks.Tracks);
                        OnPropertyChanged(nameof(UserPlaylists));
                    }
                    System.Windows.MessageBox.Show($"Открыт плейлист '{playlistWithTracks.Name}'. Количество треков:
{playlistWithTracks.Tracks.Count()}", "Плейлист", System.Windows.MessageBoxButton.OK,
System.Windows.MessageBoxImage.Information);
                }
                else
                {
                    System.Windows.MessageBox.Show($"Не удалось загрузить треки для плейлиста '{playlistToOpen.Name}'.",
"Ошибка", System.Windows.MessageBoxButton.OK, System.Windows.MessageBoxImage.Error);
                }
            }
            catch (Exception ex)
```

```
            {
                System.Windows.MessageBox.Show($"Ошибка при открытии плейлиста: {ex.Message}", "Ошибка",
System.Windows.MessageBoxButton.OK, System.Windows.MessageBoxImage.Error);
            }
            finally
            {
                IsLoading = false;
            }
        }
    }

    private void SetView(Type viewType)
    {
        if (viewType == typeof(AllTracksView))
        {
            CurrentView = new AllTracksView { DataContext = this };
        }
        else if (viewType == typeof(FavoritesView))
        {
            CurrentView = new FavoritesView { DataContext = this };
        }
        else if (viewType == typeof(ProfileView))
        {
             // Предполагаем, что ProfileViewModel создается внутри ProfileView или не требует MainViewModel в качестве
DataContext
            CurrentView = new ProfileView { DataContext = new ProfileViewModel(_currentUser, _apiService) };
        }
        else if (viewType == typeof(UploadTrackView))
        {
            CurrentView = new UploadTrackView { DataContext = new UploadTrackViewModel(_apiService, _currentUser,
this.LoadAllTracksAsync) };
        }
        // else if (viewType == typeof(SearchView))
        // {
        //     CurrentView = new SearchView { DataContext = this }; // Если SearchView использует MainViewModel
        // }
        else
        {
            // Обработка неизвестного типа View или установка по умолчанию
            CurrentView = new AllTracksView { DataContext = this };
        }
    }

    private void ShowNotImplementedMessage(string featureName)
    {
        Application.Current.Dispatcher.Invoke(() =>
            MessageBox.Show($"Функция \"{featureName}\" еще не реализована.", "В разработке", MessageBoxButton.OK,
MessageBoxImage.Information));
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }

    public async ValueTask DisposeAsync()
    {
        if (_apiService != null)
        {
            _apiService.OnTrackInfoReceived -= HandleTrackInfoReceived;
            _apiService.OnAudioChunkReceived -= HandleAudioChunkReceived;
            _apiService.OnStreamFinished -= HandleStreamFinished;
            _apiService.OnStreamError -= HandleStreamError;
```

```
        }
        _currentAudioStream?.Dispose();
        await Task.CompletedTask;
    }

    private bool CanDownloadTrack(object parameter)
    {
        return parameter is TrackDto;
    }
    private async Task DownloadTrack(object parameter)
    {
        if (parameter is TrackDto track)
        {
            var saveDialog = new Microsoft.Win32.SaveFileDialog
            {
                FileName = track.Title + ".mp3",
                Filter = "MP3 files (*.mp3)|*.mp3|All files (*.*)|*.*"
            };
            if (saveDialog.ShowDialog() == true)
            {
                try
                {
                    using (var stream = await _apiService.GetTrackStreamAsync(track.Id))
                    using (var fs = new FileStream(saveDialog.FileName, FileMode.Create, FileAccess.Write))
                    {
                        await stream.CopyToAsync(fs);
                    }
                    MessageBox.Show($"Трек '{track.Title}' успешно скачан!", "Скачивание", MessageBoxButton.OK,
MessageBoxImage.Information);
                }
                catch (Exception ex)
                {
                    MessageBox.Show($"Ошибка при скачивании: {ex.Message}", "Ошибка", MessageBoxButton.OK,
MessageBoxImage.Error);
                }
            }
        }
    }

    private void PreviousTrack()
    {
        if (AllTracks == null || AllTracks.Count == 0) return;
        if (SelectedTrack == null)
        {
            SelectedTrack = AllTracks.First();
            return;
        }
        var idx = AllTracks.IndexOf(SelectedTrack);
        if (idx > 0)
            SelectedTrack = AllTracks[idx - 1];
        else
            SelectedTrack = AllTracks.Last();
    }

    private void NextTrack()
    {
        if (AllTracks == null || AllTracks.Count == 0) return;
        if (SelectedTrack == null)
        {
            SelectedTrack = AllTracks.First();
            return;
        }
        var idx = AllTracks.IndexOf(SelectedTrack);
        if (idx < AllTracks.Count - 1)
            SelectedTrack = AllTracks[idx + 1];
```

```
    else
        SelectedTrack = AllTracks.First();
}

public void Seek(double newPositionInSeconds)
{
    if (TotalDuration > 0)
    {
        CurrentPosition = Math.Max(0, Math.Min(newPositionInSeconds, TotalDuration));
    }
}

private async Task ExecuteSearchAsync()
{
    IsLoading = true;
    AllTracks.Clear();
    try
    {
        if (string.IsNullOrWhiteSpace(SearchQuery))
        {
            await LoadAllTracksAsync();
        }
        else
        {
            var searchResults = await _apiService.SearchTracksAsync(SearchQuery);
            if (searchResults != null)
            {
                foreach (var track in searchResults)
                {
                    AllTracks.Add(track);
                }
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show($"Ошибка поиска: {ex.Message}", "Ошибка", MessageBoxButton.OK, MessageBoxImage.Error);
    }
    finally
    {
        IsLoading = false;
    }
}

private void SelectTrackAction(object parameter)
{
    if (parameter is TrackDto track)
    {
        if (SelectedTrack != null && SelectedTrack.Id == track.Id)
        {
            TogglePlayPause(null);
        }
        else
        {
            SelectedTrack = track;

            if (SelectedTrack != null)
            {
                IsPlaying = false;
                _currentlyLoadedTrackId = null;
                CurrentPosition = 0;
                TogglePlayPause(null);
            }
        }
    }
```

```
    }

    private void ShowProfileView()
    {
        var profileViewModel = new ProfileViewModel(_currentUser, _apiService);
        CurrentView = new ProfileView { DataContext = profileViewModel };
    }

    private async Task ExecuteDeleteTrackAsync(object parameter)
    {
        if (parameter is TrackDto trackToDelete)
        {
            var result = MessageBox.Show($"Вы уверены, что хотите удалить трек '{trackToDelete.Title} - {trackToDelete.Artist}'?
Это действие нельзя будет отменить.",
                                         "Подтверждение удаления",
                                         MessageBoxButton.YesNo,
                                         MessageBoxImage.Warning);

            if (result == MessageBoxResult.Yes)
            {
                IsLoading = true;
                try
                {
                    var (success, message) = await _apiService.DeleteTrackAsync(trackToDelete.Id);
                    if (success)
                    {
                        AllTracks.Remove(trackToDelete);
                        FavoriteTracks.Remove(FavoriteTracks.FirstOrDefault(t => t.Id == trackToDelete.Id));
                        if (SelectedTrack?.Id == trackToDelete.Id)
                        {
                            SelectedTrack = null;
                            IsPlaying = false;
                            CurrentAudioStream?.Dispose();
                            CurrentAudioStream = null;
                            CurrentAudioFilePath = null;
                        }
                        MessageBox.Show(message, "Удаление успешно", MessageBoxButton.OK, MessageBoxImage.Information);
                    }
                    else
                    {
                        MessageBox.Show(message, "Ошибка удаления", MessageBoxButton.OK, MessageBoxImage.Error);
                    }
                }
                catch (Exception ex)
                {
                    MessageBox.Show($"Произошла ошибка при удалении трека: {ex.Message}", "Критическая ошибка",
MessageBoxButton.OK, MessageBoxImage.Error);
                }
                finally
                {
                    IsLoading = false;
                }
            }
        }
    }

    private bool CanExecuteDeleteTrack(object parameter)
    {
        return parameter is TrackDto;
    }
}

public class RelayCommand : ICommand
{
    private readonly Action<object> _execute;
```

```
    private readonly Predicate<object> _canExecute;

    public RelayCommand(Action<object> execute, Predicate<object> canExecute = null)
    {
      _execute = execute ?? throw new ArgumentNullException(nameof(execute));
      _canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
      return _canExecute == null || _canExecute(parameter);
    }

    public void Execute(object parameter)
    {
      _execute(parameter);
    }

    public event EventHandler CanExecuteChanged
    {
      add { CommandManager.RequerySuggested += value; }
      remove { CommandManager.RequerySuggested -= value; }
    }
  }
}
```

“ApiService.cs”

```
using System;
using System.Collections.Generic; // Для IEnumerable<T>
using System.IO; // Для Stream в UploadTrackAsync
using System.Net.Http;
using System.Net.Http.Headers;
using System.Text;
using System.Threading.Tasks;
using Microsoft.AspNetCore.SignalR.Client;
using Newtonsoft.Json;
using Newtonsoft.Json.Linq;
using MusicClient.Models;
using System.Diagnostics;


namespace MusicClient.Services
{
  public class ApiService : IAsyncDisposable
  {
    private readonly HttpClient _httpClientInternal;
    public HttpClient HttpClient => _httpClientInternal;
    private HubConnection _hubConnection;
    private string _jwtToken;
    private bool _isDisposed = false; // Флаг состояния Dispose


    private const string BaseUrl = "http://192.168.1.224:5063";

    // События для стриминга
    public event Func<TrackDto, Task> OnTrackInfoReceived;
    public event Func<byte[], Task> OnAudioChunkReceived;
    public event Func<Task> OnStreamFinished;
    public event Func<string, Task> OnStreamError;

    public ApiService()
    {
      _httpClientInternal = new HttpClient();
```

```
        _httpClientInternal.BaseAddress = new Uri(BaseUrl);
        _httpClientInternal.DefaultRequestHeaders.Accept.Add(new
MediaTypeWithQualityHeaderValue("application/json"));
    }

    public void SetToken(string token)
    {
        _jwtToken = token;
        _httpClientInternal.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", _jwtToken);
    }

    public bool IsUserLoggedIn => !string.IsNullOrEmpty(_jwtToken);

    public async Task<IEnumerable<TrackDto>> GetAllTracksAsync()
    {
        try
        {
            var response = await _httpClientInternal.GetAsync("api/tracks");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStringAsync();
            return JsonConvert.DeserializeObject<IEnumerable<TrackDto>>(content);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка GetAllTracksAsync: {ex.Message}");
            return new List<TrackDto>();
        }
    }

    public async Task<IEnumerable<TrackDto>> GetLikedTracksAsync()
    {
        try
        {
            var response = await _httpClientInternal.GetAsync("api/userlikedtracks");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStringAsync();
            return JsonConvert.DeserializeObject<IEnumerable<TrackDto>>(content);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка GetLikedTracksAsync: {ex.Message}");
            return new List<TrackDto>();
        }
    }

    public async Task<IEnumerable<PlaylistDto>> GetUserPlaylistsAsync()
    {
        try
        {
            var response = await _httpClientInternal.GetAsync("api/playlists");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStringAsync();
            return JsonConvert.DeserializeObject<IEnumerable<PlaylistDto>>(content);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка GetUserPlaylistsAsync: {ex.Message}");
            return new List<PlaylistDto>();
        }
    }

    public async Task<PlaylistDto> GetPlaylistAsync(int playlistId)
    {
        try
        {
```

```
            var response = await _httpClientInternal.GetAsync($"api/playlists/{playlistId}");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStringAsync();
            return JsonConvert.DeserializeObject<PlaylistDto>(content);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка GetPlaylistAsync: {ex.Message}");
            return null;
        }
    }

    public async Task<IEnumerable<TrackDto>> SearchTracksAsync(string query)
    {
        try
        {
            var response = await _httpClientInternal.GetAsync($"api/tracks/search?query={Uri.EscapeDataString(query)}");
            response.EnsureSuccessStatusCode();
            var content = await response.Content.ReadAsStringAsync();
            return JsonConvert.DeserializeObject<IEnumerable<TrackDto>>(content);
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка SearchTracksAsync: {ex.Message}");
            return new List<TrackDto>();
        }
    }

    public async Task<bool> LikeTrackAsync(int trackId)
    {
        try
        {
            var response = await _httpClientInternal.PostAsync($"api/userlikedtracks/like/{trackId}", null);
            if (!response.IsSuccessStatusCode)
            {
                var errorContent = await response.Content.ReadAsStringAsync();
                System.Diagnostics.Debug.WriteLine($"[ApiService] LikeTrackAsync для ID {trackId} провален. Статус:
{response.StatusCode}. Ответ: {errorContent}");
            }
            return response.IsSuccessStatusCode;
        }
        catch (Exception ex)
        {
            System.Diagnostics.Debug.WriteLine($"[ApiService] Исключение в LikeTrackAsync для ID {trackId}:
{ex.Message}");
            return false;
        }
    }

    public async Task<bool> UnlikeTrackAsync(int trackId)
    {
        try
        {
            var response = await _httpClientInternal.PostAsync($"api/userlikedtracks/unlike/{trackId}", null);
            if (!response.IsSuccessStatusCode)
            {
                var errorContent = await response.Content.ReadAsStringAsync();
                System.Diagnostics.Debug.WriteLine($"[ApiService] UnlikeTrackAsync для ID {trackId} провален. Статус:
{response.StatusCode}. Ответ: {errorContent}");
            }
            return response.IsSuccessStatusCode;
        }
        catch (Exception ex)
        {
```

```
            System.Diagnostics.Debug.WriteLine($"[ApiService] Исключение в UnlikeTrackAsync для ID {trackId}:
{ex.Message}");
            return false;
        }
    }

    public async Task<TrackDto> UploadTrackAsync(Stream fileStream, string fileName, string title, string artist, string
album, string genre, string contentType = "application/octet-stream")
    {
        if (_isDisposed)
        {
            Console.WriteLine("[ApiService] UploadTrackAsync вызван для уничтоженного объекта ApiService.");
            throw new ObjectDisposedException(nameof(ApiService));
        }
        try
        {
            using (var memoryStream = new MemoryStream())
            {
                await fileStream.CopyToAsync(memoryStream);
                byte[] fileBytes = memoryStream.ToArray(); // Получаем байты

                using (var content = new MultipartFormDataContent())
                // Удаляем using для streamContent, так как ByteArrayContent будет добавлен напрямую
                {
                    var fileContent = new ByteArrayContent(fileBytes); // Используем ByteArrayContent
                    fileContent.Headers.ContentType = new MediaTypeHeaderValue(contentType);

                    // LoadIntoBufferAsync() не нужен для ByteArrayContent

                    content.Add(fileContent, "file", fileName);
                    content.Add(new StringContent(title ?? string.Empty), "title");
                    content.Add(new StringContent(artist ?? string.Empty), "artist");
                    content.Add(new StringContent(album ?? string.Empty), "album");
                    content.Add(new StringContent(genre ?? string.Empty), "genre");

                    var response = await _httpClientInternal.PostAsync("api/tracks/upload", content);

                    if (response.IsSuccessStatusCode)
                    {
                        var responseContent = await response.Content.ReadAsStringAsync();
                        return JsonConvert.DeserializeObject<TrackDto>(responseContent);
                    }
                    else
                    {
                        var errorContent = await response.Content.ReadAsStringAsync();
                        Console.WriteLine($"[ApiService] Upload failed: {response.StatusCode} - {errorContent}");
                        // Попробуем десериализовать ошибку, если это JSON
                        try
                        {
                            var errorDetails = JsonConvert.DeserializeObject<object>(errorContent); // или конкретный тип
ошибки
                            Console.WriteLine($"[ApiService] Server error details: {JsonConvert.SerializeObject(errorDetails,
Formatting.Indented)}");
                        }
                        catch { /* Не удалось десериализовать, просто вывели как строку */ }
                        return null;
                    }
                }
            }
        }
        catch (ObjectDisposedException odEx)
        {
            Console.WriteLine($"[ApiService] ObjectDisposedException в UploadTrackAsync: {odEx.ToString()} (Object Name:
{odEx.ObjectName})");
            return null;
```

```
        }
        catch (HttpRequestException httpEx)
        {
            Console.WriteLine($"[ApiService] HttpRequestException в UploadTrackAsync: {httpEx.ToString()}");
            if (httpEx.InnerException != null)
            {
                Console.WriteLine($"[ApiService] Inner HttpRequestException: {httpEx.InnerException.ToString()}");
            }
            return null;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Критическая ошибка в UploadTrackAsync: {ex.ToString()}");
            return null;
        }
    }

    public async Task<PlaylistDto> CreatePlaylistAsync(string name)
    {
        try
        {
            var createPlaylistDto = new CreatePlaylistDto { Name = name };
            var json = JsonConvert.SerializeObject(createPlaylistDto);
            var stringContent = new StringContent(json, Encoding.UTF8, "application/json");
            var response = await _httpClientInternal.PostAsync("api/playlists", stringContent);
            if (response.IsSuccessStatusCode)
            {
                var responseContent = await response.Content.ReadAsStringAsync();
                return JsonConvert.DeserializeObject<PlaylistDto>(responseContent);
            }
            Console.WriteLine($"Ошибка создания плейлиста: {response.StatusCode} - {await
response.Content.ReadAsStringAsync()}");
            return null;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Ошибка CreatePlaylistAsync: {ex.Message}");
            return null;
        }
    }

    // TODO: Методы для инициализации SignalR, подключения, вызова серверных методов хаба
    // и обработки сообщений от хаба

    public async Task InitializeHubConnectionAsync()
    {
        if (_hubConnection != null && _hubConnection.State != HubConnectionState.Disconnected)
        {
            Console.WriteLine("Hub connection already initialized or connected.");
            return;
        }

        _hubConnection = new HubConnectionBuilder()
            .WithUrl($"{BaseUrl}/streaminghub", options =>
            {
                if (!string.IsNullOrEmpty(_jwtToken))
                {
                    options.AccessTokenProvider = () => Task.FromResult(_jwtToken);
                }
            })
            .WithAutomaticReconnect()
            .Build();

        // Регистрируем обработчики сообщений от хаба
        _hubConnection.On<TrackDto>("ReceiveTrackInfo", async (trackInfo) =>
```

```
        {
            if (OnTrackInfoReceived != null)
            {
                await OnTrackInfoReceived.Invoke(trackInfo);
            }
        });

        _hubConnection.On<byte[]>("ReceiveAudioChunk", async (chunk) =>
        {
            if (OnAudioChunkReceived != null)
            {
                await OnAudioChunkReceived.Invoke(chunk);
            }
        });

        _hubConnection.On("StreamFinished", async () =>
        {
            if (OnStreamFinished != null)
            {
                await OnStreamFinished.Invoke();
            }
        });

        _hubConnection.On<string>("StreamError", async (errorMessage) =>
        {
            if (OnStreamError != null)
            {
                await OnStreamError.Invoke(errorMessage);
            }
        });

        _hubConnection.Closed += async (error) =>
        {
            // Это событие вызывается, когда соединение закрыто.
            // Можно добавить логику для обработки неожиданного закрытия.
            Console.WriteLine($"SignalR Hub connection closed. Error: {error?.Message}");
            // Возможно, потребуется попытаться переподключиться или уведомить пользователя.
            // WithAutomaticReconnect() должен обрабатывать большинство случаев, но здесь можно добавить доп.
логику.
            await Task.CompletedTask; // Просто для примера, чтобы сделать лямбду async
        };

        try
        {
            await _hubConnection.StartAsync();
            Console.WriteLine("SignalR Hub connected successfully.");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"SignalR Connection Error: {ex.Message}");
        }
    }

    public async Task StartStreamingTrackAsync(int trackId) // Переименовал для ясности, что это команда к хабу
    {
        if (_hubConnection?.State == HubConnectionState.Connected)
        {
            try
            {
                await _hubConnection.InvokeAsync("StreamTrack", trackId);
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error invoking StreamTrack on hub: {ex.Message}");
                if(OnStreamError != null) await OnStreamError.Invoke($"Failed to start streaming: {ex.Message}");
```

```
            }
        }
        else
        {
            Console.WriteLine("SignalR Hub not connected. Cannot stream track.");
            if(OnStreamError != null) await OnStreamError.Invoke("Not connected to streaming server.");
        }
    }

    public async ValueTask DisposeAsync()
    {
        if (_isDisposed) return;

        if (_hubConnection != null)
        {
            // Отписываемся от обработчиков, чтобы избежать утечек памяти, если ApiService будет пересоздаваться
            _hubConnection.Remove("ReceiveTrackInfo");
            _hubConnection.Remove("ReceiveAudioChunk");
            _hubConnection.Remove("StreamFinished");
            _hubConnection.Remove("StreamError");
            // _hubConnection.Closed -= ... ; // Если бы мы присваивали именованный метод

            await _hubConnection.DisposeAsync();
            _hubConnection = null; // Убедимся, что ссылка обнулена
        }
        _httpClientInternal.Dispose();
        _isDisposed = true; // Устанавливаем флаг
        GC.SuppressFinalize(this); // Если есть финализатор, хотя для этого класса он не нужен
    }

    public async Task<(bool Success, string Message, AuthResponseDto AuthResponse)> RegisterAsync(UserRegisterDto
registerDto)
    {
        if (_isDisposed) throw new ObjectDisposedException(nameof(ApiService));
        try
        {
            var json = JsonConvert.SerializeObject(registerDto); // registerDto уже содержит Nickname, Email, Password
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            var response = await _httpClientInternal.PostAsync("api/auth/register", content);
            if (response.IsSuccessStatusCode)
            {
                var responseContent = await response.Content.ReadAsStringAsync();
                // Ответ от сервера может не содержать полный AuthResponseDto при регистрации,
                // а только сообщение и, возможно, Id пользователя.
                // Пока оставим десериализацию в AuthResponseDto, но это можно изменить.
                var authResponse = JsonConvert.DeserializeObject<AuthResponseDto>(responseContent);
                // При успешной регистрации токен обычно не возвращается сразу, пользователь должен войти
отдельно.
                // Так что SetToken(authResponse.Token) здесь может быть лишним.
                // Если сервер возвращает токен при регистрации, то можно оставить.
                // В текущей реализации сервера (Program.cs) токен НЕ возвращается при регистрации.
                return (true, "Registration successful!", authResponse); // authResponse может быть частично пустым
            }
            else
            {
                var errorContent = await response.Content.ReadAsStringAsync();
                string message = $"Registration failed. Status: {response.StatusCode}";
                try
                {
                    var errorJObject = JsonConvert.DeserializeObject<JObject>(errorContent);
                    string extractedMessage = errorJObject?["message"]?.ToString() ??
                                              errorJObject?["title"]?.ToString();
                    if (!string.IsNullOrEmpty(extractedMessage)) message = extractedMessage;
                    else message = errorContent;
                    var errorsToken = errorJObject?["errors"];
```

```
                if (errorsToken != null)
                {
                    message += " Details: " + errorsToken.ToString(Newtonsoft.Json.Formatting.None);
                }
            }
            catch { /* Оставляем message как есть */ }
            return (false, message, null);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"[ApiService] Ошибка RegisterAsync: {ex.Message}");
        return (false, $"Ошибка регистрации: {ex.Message}", null);
    }
}

    public async Task<(bool Success, string Message, AuthResponseDto AuthResponse)> LoginAsync(UserLoginDto
loginDto)
    {
        try
        {
            var json = JsonConvert.SerializeObject(loginDto); // loginDto уже содержит Nickname, Password
            var content = new StringContent(json, Encoding.UTF8, "application/json");
            var response = await _httpClientInternal.PostAsync("api/auth/login", content);
            if (response.IsSuccessStatusCode)
            {
                var responseContent = await response.Content.ReadAsStringAsync();
                var authResponse = JsonConvert.DeserializeObject<AuthResponseDto>(responseContent);
                if (authResponse != null && !string.IsNullOrEmpty(authResponse.Token))
                {
                    SetToken(authResponse.Token);
                    return (true, "Login successful!", authResponse);
                }
                return (false, "Login failed: Invalid response from server.", null);
            }
            else
            {
                var errorContent = await response.Content.ReadAsStringAsync();
                string message = $"Login failed. Status: {response.StatusCode}"; // Сообщение по умолчанию
                try
                {
                    var errorJObject = JsonConvert.DeserializeObject<JObject>(errorContent);
                    string extractedMessage = errorJObject?["message"]?.ToString() ??
                                              errorJObject?["title"]?.ToString();

                    if (!string.IsNullOrEmpty(extractedMessage))
                    {
                        message = extractedMessage;
                    }
                    // Если extractedMessage пуст, а errorContent не является валидным JSON с ошибкой,
                    // или сам errorContent пуст/содержит только пробелы,
                    // то message останется "Login failed. Status: {response.StatusCode}"
                    // или будет заменено на errorContent, если он не пустой и не JSON.
                    // Чтобы избежать пустого окна, если errorContent пуст или не JSON и не содержит текста:
                    else if (string.IsNullOrWhiteSpace(errorContent) || !errorContent.TrimStart().StartsWith("{"))
                    {
                        // Оставляем message = $"Login failed. Status: {response.StatusCode}";
                        // или можно установить более общее сообщение, если errorContent бесполезен
                    }
                    else // errorContent есть, не пустой, и может быть JSON, но без message/title
                    {
                        // Попытка взять весь errorContent, если он не пустой и предположительно JSON
                        // Но если он не содержит полезной текстовой информации, лучше оставить дефолтное
                        if (!string.IsNullOrWhiteSpace(errorContent))
                        {
```

```
                        // Проверим, не является ли errorContent слишком большим или HTML
                        if (errorContent.Length < 500 && !errorContent.TrimStart().StartsWith("<"))
                        {
                            message = errorContent;
                        }
                        // Иначе, оставляем дефолтное сообщение со статусом.
                    }
                }

                var errorsToken = errorJObject?["errors"];
                if (errorsToken != null)
                {
                    // Добавляем детали, если они есть и сообщение не стало слишком длинным
                    string details = errorsToken.ToString(Newtonsoft.Json.Formatting.None);
                    if (message.Length + details.Length < 1000) // Ограничение на длину сообщения
                    {
                        message += " Details: " + details;
                    }
                }
            }
            catch
            {
                // Если парсинг JSON не удался, и errorContent не пустой и не HTML, используем его.
                // Иначе, останется сообщение по умолчанию.
                if (!string.IsNullOrWhiteSpace(errorContent) && errorContent.Length < 500 &&
!errorContent.TrimStart().StartsWith("<"))
                {
                    message = errorContent;
                }
            }
            // Финальная проверка, чтобы сообщение не было пустым
            if (string.IsNullOrWhiteSpace(message))
            {
                message = $"Login failed with status {response.StatusCode}. No further details available.";
            }
            return (false, message, null);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine($"[ApiService] Ошибка LoginAsync: {ex.Message}");
        return (false, $"Ошибка входа: {ex.Message}", null);
    }
}

public void Logout()
{
    _jwtToken = null;
    _httpClientInternal.DefaultRequestHeaders.Authorization = null;
    _hubConnection?.StopAsync();
}

public async Task<Stream> GetTrackStreamAsync(int trackId)
{
    try
    {
        var request = new HttpRequestMessage(HttpMethod.Get, $"api/tracks/stream/{trackId}");
        var response = await _httpClientInternal.SendAsync(request, HttpCompletionOption.ResponseHeadersRead);
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStreamAsync();
    }
    catch (Exception ex)
    {
        Console.WriteLine($"[ApiService] Ошибка GetTrackStreamAsync: {ex.Message}");
        return Stream.Null;
```

```
        }
    }

    public async Task<(bool Success, string Message)> DeleteTrackAsync(int trackId)
    {
        if (_isDisposed)
        {
            Console.WriteLine("[ApiService] DeleteTrackAsync вызван для уничтоженного объекта ApiService.");
            // Consider throwing ObjectDisposedException or returning a specific error tuple
            return (false, "ApiService has been disposed.");
        }
        try
        {
            var response = await _httpClientInternal.DeleteAsync($"api/tracks/{trackId}");
            var responseContent = await response.Content.ReadAsStringAsync();

            if (response.IsSuccessStatusCode)
            {
                // Пытаемся извлечь сообщение об успехе, если оно есть
                // Сервер возвращает объект { message: "..." }
                dynamic? result = JsonConvert.DeserializeObject<dynamic>(responseContent);
                string message = result?.message?.ToString() ?? "Трек успешно удален.";
                return (true, message);
            }
            else
            {
                // Пытаемся извлечь сообщение об ошибке
                string errorMessage = "Неизвестная ошибка при удалении трека.";
                if (!string.IsNullOrWhiteSpace(responseContent))
                {
                    try
                    {
                        dynamic? errorResult = JsonConvert.DeserializeObject<dynamic>(responseContent);
                        errorMessage = errorResult?.message?.ToString() ?? errorResult?.detail?.ToString() ?? $"Ошибка:
{response.StatusCode}";
                    }
                    catch
                    {
                        // Если ответ не JSON, используем статус код
                        errorMessage = $"Ошибка сервера: {response.StatusCode}, Ответ: {responseContent}";
                    }
                }
                else
                {
                    errorMessage = $"Ошибка сервера: {response.StatusCode}";
                }
                Console.WriteLine($"[ApiService] Ошибка DeleteTrackAsync ({trackId}): {response.StatusCode} -
{errorMessage}");
                return (false, errorMessage);
            }
        }
        catch (HttpRequestException httpEx)
        {
            Console.WriteLine($"[ApiService] HTTP Исключение в DeleteTrackAsync ({trackId}): {httpEx.Message}");
            return (false, $"Ошибка сети при удалении: {httpEx.Message}");
        }
        catch (JsonException jsonEx)
        {
            Console.WriteLine($"[ApiService] JSON Исключение в DeleteTrackAsync ({trackId}): {jsonEx.Message}");
            return (false, $"Ошибка обработки ответа сервера: {jsonEx.Message}");
        }
        catch (Exception ex)
        {
            Console.WriteLine($"[ApiService] Общее Исключение в DeleteTrackAsync ({trackId}): {ex.Message}");
    return (false, $"Непредвиденная ошибка: {ex.Message}") }}}}
```