|  |  |
| --- | --- |
|  | **Министерство науки и высшего образования Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**  **«Московский государственный технический университет имени Н.Э. Баумана**  **(национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Теоретическая информатика и технологии

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К КУРСОВОЙ РАБОТЕ**

**ПО КУРСУ БАЗЫ ДАННЫХ**

**НА ТЕМУ:**

«Проектирование базы данных музыкальной платформы»

Студент

*подпись, дата фамилия, и.о.*

Научный руководитель

*подпись, дата фамилия, и.о.*

*2022 г*

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc107175902)

[1. Обзор предметной области 4](#_Toc107175903)

[1.1. Формирование требований к приложению 4](#_Toc107175904)

[1.2. ER-модель 4](#_Toc107175905)

[1.2.1. Описание сущностей 6](#_Toc107175906)

[1.2.2. Обоснование выбора кардинальных связей 8](#_Toc107175907)

[2. Проектирование клиент-серверного приложения 10](#_Toc107175908)

[2.1. Реляционная модель 10](#_Toc107175909)

[2.1.1. Свойства отношений 12](#_Toc107175910)

[2.1.2. Ограничения минимальной кардинальности 16](#_Toc107175911)

[2.2. Клиент-серверное взаимодействие 18](#_Toc107175912)

[2.3. Протокол клиент-серверного приложения 20](#_Toc107175913)

[2.4. Сжатие аудио-файлов 22](#_Toc107175914)

[3. Реализация 24](#_Toc107175915)

[3.1. Реализация базы данных 24](#_Toc107175916)

[3.2. Реализация клиента 27](#_Toc107175917)

[3.2.1. Главная страница 28](#_Toc107175918)

[3.2.2. Индивидуальные страницы 29](#_Toc107175919)

[3.2.3. Проигрывание песен 31](#_Toc107175920)

[3.3. Реализация сервера 32](#_Toc107175921)

[3.4. Реализация приложения администратора 32](#_Toc107175922)

[4. Тестирование 34](#_Toc107175923)

[ЗАКЛЮЧЕНИЕ 37](#_Toc107175924)

[CПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 38](#_Toc107175925)

[ПРИЛОЖЕНИЕ А 39](#_Toc107175926)

[ПРИЛОЖЕНИЕ Б 41](#_Toc107175927)

# ВВЕДЕНИЕ

С развитием Интернета и компьютерных технологий становилось возможным упрощать жизнь их пользователей. Например, раньше, чтобы узнать о какой-либо интересной человеку музыкальной группе, приходилось искать и покупать журналы, в которых возможно и не было нужной информации. Чтобы послушать музыку, приходилось покупать кассеты и винил, храня их у себя дома. При этом они могли занимать много места у особых меломанов.

В начале 2000-х все чаще музыка стала распространяться в цифровом формате. Были разные варианты: скачивание после покупки в онлайн-магазине или через торрент. Теперь музыкальные композиции можно было хранить на компьютере, что было довольно удобно, так как экономилось место в доме и можно было прослушивать музыку без Интернета. Однако возникала проблема с синхронизацией песен, если предполагалось прослушивание с разных устройств. Развитие процессоров и технологий Интернета сделали возможным так называемую потоковую передачу (стриминг) аудио.

На сегодняшний день стриминговые сервисы обладают наибольшей популярностью среди разных вариантов прослушивания песен. Заходя в приложение с разных устройств, пользователь получает доступ к одной и той же библиотеке песен с возможными рекомендациями. Зачастую в таких сервисах также присутствует информация о группах, музыкантах, концертах и т.д. По причине хранения всей нужной информации в одном месте они и обрели популярность.

Целью данной курсовой работы является разработка базы данных для музыкальной платформы и соответствующего ей клиент-серверного приложения, реализующего потоковую передачу аудио и информации о группах, музыкантах, концертах и т.д.

# Обзор предметной области

## Формирование требований к приложению

В качестве предметной области была выбрана музыкальная индустрия. База данных должна хранить информацию о музыкантах, историю участия музыкантов в группе и учитывать то, что музыкант мог играть в ней несколько раз. Модель данных должна содержать информацию о существующих и уже распавшихся группах. Помимо этого, для каждой группы должно быть возможным посмотреть прошедшие и запланированные концерты. Альбомы должны содержать информацию о песнях в них. При просмотре альбома песни должны быть отсортированы в исходном, заданном создателями порядке.

Предполагается, что пользователь из приложения сможет просматривать различную информацию о группах, альбомах, песнях и музыкантах. Также параллельно пользователь сможет прослушивать музыку. Помимо этого представляется возможным поиск музыкантов, групп, концертов, песен по имени/названию.

Добавлять в базу данных различные сущности должен администратор, для этого нужно реализовать отдельное приложение. В основном, оно будет предназначено для добавления песен и альбомов, так как просто SQL-запросами делать это неудобно (нужно добавлять все песни альбома в базу данных, проще это автоматизировать). Помимо этого, для удобства данная утилита не должна зависеть от определенного формата добавляемых аудио-файлов.

## ER-модель

На основе сформированных требований была создана модель «сущность-связь», представленная на рисунке 1.1, в нотации «Crow’s Foot» Дж. Мартина.

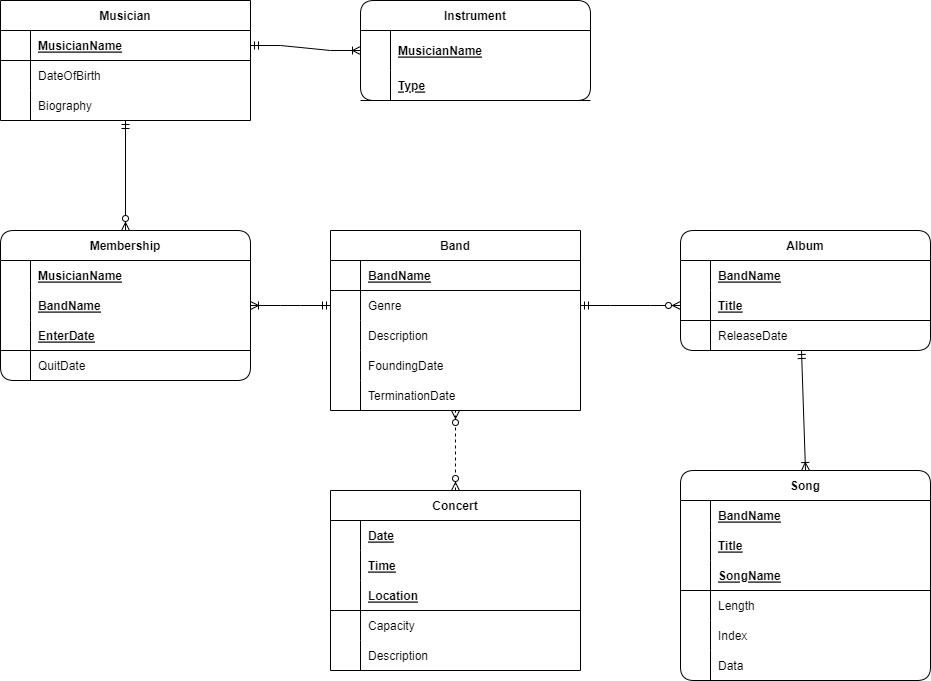


Рисунок 1.1 - ER-модель

В результате получилось 7 сущностей, 3 сильные и 4 слабые. К сильным сущностям относятся:

1. Musician (музыкант)
2. Band (группа)
3. Concert (концерт)

К слабым относятся:

1. Instrument (инструмент)
2. Membership (членство в группе)
3. Album (альбом)
4. Song (песня в альбоме)

### Описание сущностей

Далее описывается каждая из сущностей ER-модели.

1. Musician – сущность, описывающая отдельного музыканта.

Идентификатор:

MusicianName: имя музыканта

Атрибуты:

DateOfBirth – день рождения

Biography – биография

1. Instrument – многозначный атрибут у Musician, обозначающий инструменты, на которых играет музыкант.

Составной идентификатор:

MusicianName – имя соответствующего музыканта

Type – тип инструмента (гитара, вокал и т.д.)

1. Membership – сущность, хранящая членство музыканта в группе в определенный период времени. Используется шаблон «сопряжение».

Составной идентификатор:

MusicianName – имя музыканта

BandName – имя группы

EnterDate – дата вступления в группу

Атрибуты:

QuitDate – дата, когда музыкант покинул группу

Стоит отметить, что наличие в составном идентификаторе атрибута EnterDate обеспечивает случай, когда музыкант присоединялся к группе несколько раз.

1. Band – сущность, обозначающая группу.

Идентификатор:

BandName – имя группы

Атрибуты:

Genre – преобладающий жанр музыки у группы

Description – описание группы

FoundingDate – дата основания группы

TerminationDate – дата распада группы

1. Concert – сущность, обозначающая концерт.

Составной идентификатор:

Date – дата проведения

Time – время проведения

Location – место проведения

Атрибуты:

Capacity – вместимость

Description – описание концерта

В данном случае место проведения концерта является и его названием, так как зачастую они совпадают.

1. Album – сущность, обозначающая альбом, является многозначным атрибутом у группы.

Составной идентификатор:

BandName – имя группы, выпустившей альбом

Title – название альбома

Атрибуты:

ReleaseDate – дата выпуска альбома

1. Song – сущность, обозначающая песню, является многозначным атрибутом у альбома.

Составной идентификатор:

BandName – название группы

Title – название альбома

SongName – название песни

Атрибуты:

Length – длительность песни

Index – номер песни в альбоме, начиная с 1

Data – блок бинарных данных песни

### Обоснование выбора кардинальных связей

Далее рассматриваются кардинальные связи между сущностями ER-модели.

1. Связь 1:N, M-M, идентифицирующая Musician – Instrument

Musician: музыкант может владеть несколькими инструментами, но при этому хотя бы одним, иначе он не музыкант.

Instrument: у инструмента музыкант ровно один по шаблону многозначный атрибут.

1. Связь 1:N, M-O, идентифицирующая Musician – Membership

Musician: музыкант мог участвовать во многих группах или не участвовать ни в одной.

Membership: по шаблону «сопряжение» сущность связана ровно с одной родительской сущностью.

1. Связь 1:N, M-M, идентифицирующая Band – Membership

Band: в группе должен быть хотя бы один музыкант, при этом их может быть несколько.

Membership: аналогично предыдущему случаю, по шаблону «сопряжение» ровно одна родительская сущность.

1. Связь N:M, O-O, неидентифицирующая Band-Concert

Band: группа могла выступать (будет выступать) как нескольких концертах, так и ни на одном.

Concert: концерт может быть запланирован, но не иметь расписания, поэтому на нем может не играть ни одной группы в текущий момент. Также понятно, что на концерте может играть больше одной группы.

1. Связь 1:N, M-O, идентифицирующая Band-Album

Band: группа может иметь как ноль альбомов, так и несколько.

Album: у альбома по шаблону «многозначный атрибут» ровно одна группа.

1. Связь 1:N, M-M, идентифицирующая Album-Song

Album: в альбоме может быть много песен, но как минимум одна.

Song: песня принадлежит какому-либо альбому, поэтому он должен быть ровно один.

# Проектирование клиент-серверного приложения

Основной задачей данного этапа является разработка архитектуры клиент-серверного приложения, реализующего просмотр клиентом информации о разных сущностях базы данных и потоковую передачу аудио-файлов. Все приложение можно разделить на 4 части:

1. Клиент, реализующий обмен данными с сервером
2. Сервер, взаимодействующий с клиентом и выполняющий запросы в базу данных
3. Приложение для администратора, позволяющее добавлять альбомы в базу данных, а также некоторые другие сущности
4. База данных, представляющая собой совокупность таблиц, представлений и триггеров

Прежде всего, нужно определить, как сущности будут храниться в базе данных. Для этого необходимо создать реляционную модель базы данных путем преобразования ER-модели.

## Реляционная модель

Реляционная модель, получившаяся из ER-модели, изображена на рисунке 2.1.

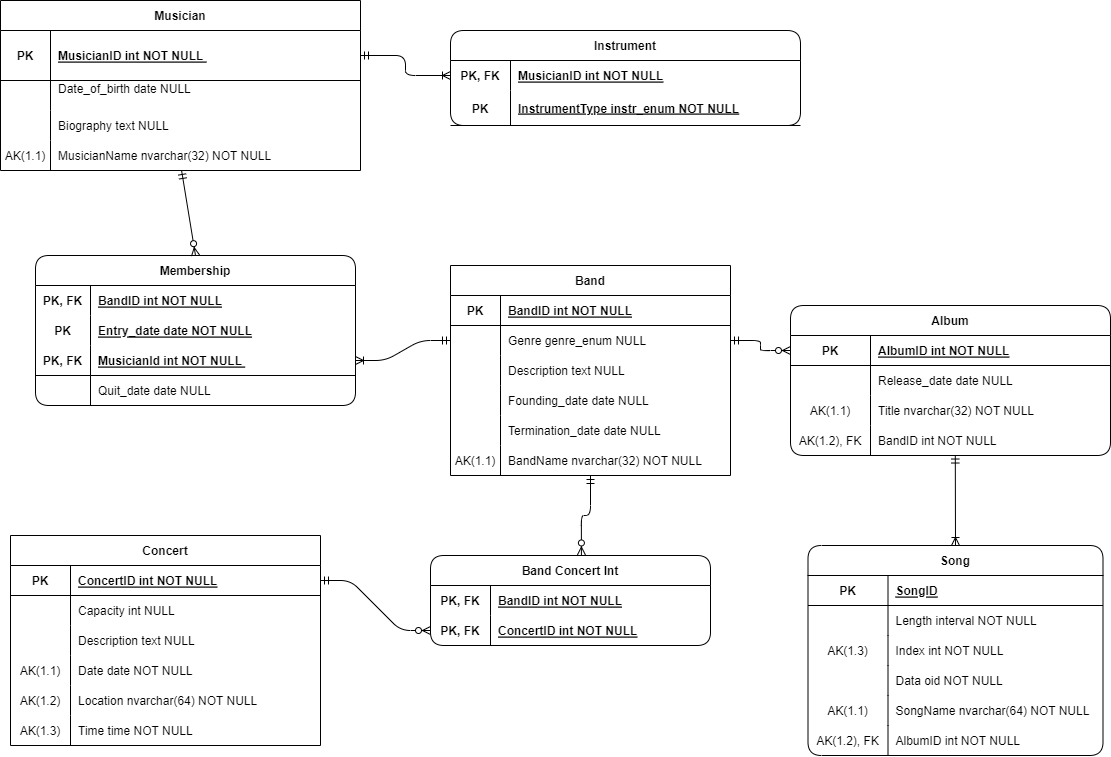
После преобразования, помимо таблиц, соответствующих сущностям ER-модели, добавилась еще одна промежуточная таблица для связи «многие-ко-многим»: Band Concert Int, связывающая группу и концерт. Также в угоду производительности, где возможно, у отношений в качестве первичного ключа или его части (при составном ключе) выступает целочисленное значение. Аналогично и с внешними ключами. Выигрыш будет при осуществлении поиска записей. Существенный прирост производительности будет, например, в случае, если нужно узнать на каких концертах играют группы. Для этого необходимо сделать двойное присоединение таблиц Band, Band Concert Int и Concert. При этом идентификатор сущности преобразовался в альтернативный ключ в соответствующих таблицах.

Рисунок 2.1 - Реляционная модель

Для полей, значение которых может принадлежать только определенному множеству значений, задан тип Enum (перечисление) для поддержания целостности базы данных. К ним относятся поля жанра музыки у группы и тип инструмента музыканта.

Поскольку подразумевается потоковая передача аудио-файлов, для их хранения нужно выбрать тип данных, позволяющий считывать блок данных из определенного места. То есть должны поддерживаться аналоги функций read, seek, tell стандартной библиотеки языка Си для чтения файлов. В разных СУБД такие типы данных называются по-разному, поэтому на данном этапе под типом данных поля Data отношения Song понимается некий абстрактный тип BLOB бинарных данных.

При получении запроса аудио-файла от клиента сервер начинает с определенной периодичностью (между другими запросами) посылать в ответ блоки данных одинакового размера. Перед отправкой сервер каждый раз запрашивает новый блок из базы данных с определенной позиции, а не хранит целый аудио-файл и не отправляет его частями. Таким образом, это решение является оптимальным по памяти, а значит использование описанного типа данных BLOB обосновано.

Также возможен вариант хранения файлов на диске, а не в базе данных непосредственно. При этом в базе данных хранятся названия файлов, соответствующих записям. Плюсом данного подхода является меньший размер базы данных и, соответственно, более быстрое резервное копирование. Однако из этого вытекают минусы – отсутствие атомарности операций над базой данных и целостности. Также более трудно переносить базу данных с одной машины на другую. Помимо этого файлы уязвимы к действиям администратора. Исходя из этих минусов, был выбран первый вариант хранения аудио-файлов непосредственно в БД.

### Свойства отношений

В таблицах 2.1 – 2.8 представлена информация об отношениях в базе данных: типы полей, типы ключей, допустимость неопределенных значений и дополнительная информация.

Таблица 2.1 – Отношение Musician

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| MusicianID | int | PK | Нет | Суррогатный ключ |
| DateOfBirth | date | Нет | Да | Дата без времени |
| Biography | text | Нет | Да |  |
| MusicianName | nvarchar(32) | AK | Нет | Уникальный |

Таблица 2.2 – Отношение Instrument

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| MusicianID | int | PK, FK | Нет | Суррогатный ключ |
| InstrumentType | instr\_enum | PK | Нет |  |

Таблица 2.3 – Отношение Membership

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| MusicianID | int | PK, FK | Нет |  |
| BandID | int | PK, FK | Нет |  |
| EntryDate | date | PK | Нет | Дата без времени |
| QuitDate | date | Нет | Да | Дата без времени |

Таблица 2.4 – Отношение Band

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| BandID | int | PK | Нет | Суррогатный ключ |
| Genre | genre\_enum | Нет | Да | Тип «жанр» |
| Description | text | Нет | Да |  |
| FoundingDate | date | Нет | Да | Дата без времени |
| TerminationDate | date | Нет | Да | Дата без времени |
| BandName | nvarchar(32) | AK | Нет | Уникальный |

Таблица 2.5 – Отношение Band Concert Int

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| BandID | int | PK, FK | Нет |  |
| ConcertID | int | PK, FK | Нет |  |

Таблица 2.6 – Отношение Concert

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| ConcertID | int | PK | Нет | Суррогатный ключ |
| Capacity | int | Нет | Да |  |
| Description | text | Нет | Да |  |
| Date | date | AK(1.1) | Нет | Уникальный (составной ключ), дата без времени |
| Location | nvarchar(64) | AK(1.2) | Нет | Уникальный, (составной ключ) |
| Time | time | AK(1.3) | Нет | Уникальный, (составной ключ), время |

Таблица 2.7 – Отношение Album

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| AlbumID | int | PK | Нет | Суррогатный ключ |
| BandID | int | FK, AK(1.1) | Нет | Уникальный (составной ключ) |
| ReleaseDate | date | Нет | Да | Дата без времени |
| Title | nvarchar(32) | AK(1.2) | Нет | Уникальный (составной ключ) |

Таблица 2.8 – Отношение Song

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Название столбца | Тип | Ключ | Неопр. значение | Дополнительно |
| SongID | int | PK | Нет | Суррогатный ключ |
| Length | interval | Нет | Нет | Интервал времени в секундах |
| Data | blob | Нет | Нет | Бинарные данные песни |
| SongName | nvarchar(64) | AK(1.1) | Нет | Уникальный (составной ключ) |
| AlbumID | int | FK, AK(1.2) | Нет | Уникальный (составной ключ) |
| Index | int | AK(1.3) | Нет | Уникальный (составной ключ) |

### Ограничения минимальной кардинальности

В таблицах 2.9 – 2.15 представлены действия, нужные для выполнения ограничений минимальной кардинальности связей между отношениями.

Таблица 2.9 - Связь Musician-Instrument

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Musician | Действие над Instrument |
| Вставка | Одновременная вставка хотя бы одной записи в дочернюю таблицу | Ограничение ссылочной целостности + MusicianID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | Если единственный инструмент, то запрет |

Таблица 2.10 - Связь Musician-Membership

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Musician | Действие над Membership |
| Вставка | - | Ограничение ссылочной целостности + MusicianID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 2.11 - Связь Band-Membership

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Band | Действие над Membership |
| Вставка | Одновременная вставка хотя бы одной записи в дочернюю таблицу | Ограничение ссылочной целостности + BandID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | Если единственный Membership, то запрет |

Таблица 2.12 - Связь Band-Album

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Band | Действие над Album |
| Вставка | - | Ограничение ссылочной целостности + BandID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 2.13 – Связь Album-Song

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Album | Действие над Song |
| Вставка | Одновременная вставка хотя бы одной записи в дочернюю таблицу | Ограничение ссылочной целостности + AlbumID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | Запрет |

Здесь стоит отметить, что удаление записи в таблице Song возможно только при удалении всего альбома, так как множество песен на альбоме неразрывно связано с ним.

Таблица 2.14 – Связь Concert - BandConcertInt

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Concert | Действие над BandConcertInt |
| Вставка | - | Ограничение ссылочной целостности + ConcertID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

Таблица 2.15 – Связь Band - BandConcertInt

|  |  |  |
| --- | --- | --- |
| Операция | Действие над Band | Действие над BandConcertInt |
| Вставка | - | Ограничение ссылочной целостности + BandID NOT NULL (внешний ключ) |
| Изменение ключа (первичного/внешнего) | Запрет | Запрет |
| Удаление | Каскадное удаление | - |

## Клиент-серверное взаимодействие

Клиент и сервер взаимодействуют между собой посредством сети. Как было указано в требованиях к приложению в главе 1, клиент должен иметь возможность одновременно прослушивать музыку и осуществлять навигацию по приложению. Это требование накладывает ограничение на запросы к серверу – они должны быть асинхронными. В ином случае пришлось бы ждать выполнение текущего запроса для выполнения следующего. Если это запрос на передачу поточных данных, то ожидание увеличивается в разы по сравнению, например, с простым запросом таблицы музыкантов. Даже вынесение сетевого взаимодействия в отдельный поток не помогло бы решить проблему. Таким образом, приложение нужно проектировать исходя из того, что оно должно реализовывать асинхронные запросы к серверу.

Реализовать его можно путем создания идентификатора запроса, который передается на сервер, и включается в его ответ. В таком случае на стороне клиента нужно хранить отображение идентификатора в некоторые данные, которые необходимы для обработки ответа.

Также предполагается, что у клиента будет возможность просмотреть список записей некоторых сущностей базы данных в виде таблицы и сделать по ним поиск. Например, запросить список песен. Понятно, что загружать все песни из базы данных за один запрос вряд ли нужно. Таким образом, в протоколе взаимодействия нужно предусмотреть запросы ограниченного числа записей. В предположении, что записи отсортированы, это можно реализовать путем добавления в запрос полей offset (смещение), size (максимальное число записей в ответе), filter (строка, по которой осуществляется поиск). Такому набору параметров будет соответствовать запрос: «select \* from table where col ilike <filter> order by col limit <size> offset <offset>».

Для снижения нагрузки с сервера нужно предусмотреть отмену поточной передачи данных. Это можно сделать, послав на сервер запрос отмены с нужным идентификатором. Такой случай возможен, когда клиент, не дослушав текущую песню, захочет запросить другую.

Так как клиент ожидает быстрого отклика и отсутствия задержки при проигрывании песни, передача аудио-файла должна быть как можно более эффективной. Это можно реализовать комбинацией двух приемов: использование в качестве нижележащего протокола TCP для реализации собственного протокола уровня приложения стека TCP/IP [1] и сжатием исходных файлов. Таким образом, в базе данных должны храниться песни в сжатом формате. На стороне клиента песня должна декодироваться в «сырой формат» PCM [2], состоящий из сэмплов звука определенной частоты дискретизации и который воспринимается аудиокартой. Также нужно сделать декодирование по запросу, например, когда звука для передачи в аудиокарту осталось менее чем на 100 мс. Это позволит избежать накопления большого буфера данных.

Также использование сжатых данных уменьшит размер базы данных.

## Протокол клиент-серверного приложения

Как было сказано ранее, для реализации взаимодействия между клиентом и сервером стоит использовать протокол TCP. Так как протокол будет бинарным для улучшения производительности, необходимо придумать способ сериализации запросов для записи данных в TCP соединение. Одним из способов является использование протокола Google Protobuf (Protocol Buffers).

Protobuf представляет собой описание структур, по которому генерируется код для их сериализации и десериализации [3]. Этот протокол является независимым от платформы и языка программирования. Это позволяет реализовать сервер и клиент на разных ЯП. Также Protobuf отличается быстрым разбором пакета и оптимизацией его размера (поле не занимает места, если для него нет данных).

Недостатком данного протокола является необходимость знать точный размер сообщения для его декодирования. Решить эту проблему можно путем добавления 4 байт (размер unsigned int) перед началом пакета – неотрицательное число в little-endian, которое будет обозначать размер бинарных данных в кодировке Protobuf. Таким образом, парсер сообщений сети у клиента и сервера должен работать в двух состояниях: первое – состояние считывания размера пакета, второе – считывание самого пакета и его десериализация.

Для создания сообщения необходимо определить имя следующего вида: «message <имя типа>», и внутри тела описать поля. Также можно сделать тип перечисление с помощью следующей команды: «enum <имя типа>».

Протокол приложения будет иметь 3 типа сообщений: запрос таблицы, запрос потоковой передачи аудио-файла и простой запрос какой-либо сущности. Первый запрос, как исходит из названия, выполняется, когда пользователь захочет увидеть список записей одной таблицы, например, список всех песен в базе данных. Второй – при запросе песни. Третий – при запросе какой-либо страницы, например, страницы группы. Аналогично, определено три типа ответа – на каждый тип запроса.

У всех запросов и ответов есть общие поля – номер запроса и факт отмены запроса (логическое значение). Protobuf позволяет добавлять вложенные сообщения, поэтому можно определить структуры Request и Answer с этими двумя полями и хранить в них уже конкретный запрос. Кроме этого, ключевое слово «oneof» сообщает компилятору Protobuf, что в поле хранится только одно из перечисленных значений. Под этот случай как раз подходит описанные ранее 3 типа сообщений.

Пример сообщения Request изображен на листинге 1.

message Request {

uint64 reqId = 1;

bool cancel = 3; //send again with cancel = true to cancel request

oneof msg {

TableReq tableReq = 4;

SimpleReq simpleReq = 5;

StreamReq streamReq = 6;

}

}

Листинг 1 – сообщение Request в Protobuf

Полный список сообщений протокола находится в приложении А.

## Сжатие аудио-файлов

Как было сказано ранее, аудио-файлы перед загрузкой в базу данных необходимо сжать каким-либо алгоритмом. Одним из способов является использование кодека Opus. Opus является кодеком для сжатия с потерями и предназначен специально для звуковых файлов, передающихся по Интернету, в особенности, для потоковой передачи музыки [4]. Отличительными особенностями являются:

1. Поддержка переменного битрейта – позволяет использовать меньше данных в частях файла, где они не нужны. Как следствие, большее сжатие.
2. Поддержка исходного и декодированного аудио-файла разной частоты дискретизации
3. Поддержка моно и стерео звука
4. Быстрое декодирование пакета – важное свойство в приложениях реального времени.

Части файла кодируются в так называемые фреймы, хранящие звуковые данные за определенный промежуток. Обычно это 20 мс. Недостатком кодека Opus является отсутствие разграничений между фреймами. Эту проблему решает формат контейнера мультимедиа Ogg [5].

Формат Ogg служит для хранения разнообразных файлов мультимедиа. В начале каждого пакета существует заголовок, который разные кодеки могут использовать для своих нужд. После заголовка располагаются разграниченные инкапсулированные пакеты.

Кодек Opus также использует данный контейнер для хранения закодированных пакетов из исходного файла.

Таким образом, в базе данных можно хранить пакеты Ogg Opus и передавать их на клиент для декодирования. Небольшой размер данных относительно исходного аудио-файла позволяет быстро передавать и даже хранить весь пакет на клиенте в случае необходимости.

# Реализация

## Реализация базы данных

В качестве СУБД была взята PostgreSQL. Это постоянно развивающаяся реляционная СУБД с открытым кодом, которая используется во многих компаниях.

Для хранения аудио-файлов в базе данных используется тип Large Object. Он предоставляет потоковый доступ к данным, которые хранятся в системной таблице pg\_largeobject [6]. Для доступа к нему используется ключ типа oid (object id), ответственность за хранение которого лежит на программисте. Таким образом, при удалении записи в триггере необходимо явно удалить объекты, которые связаны с этой записью, с помощью вызова функции lo\_unlink(oid).

Непростая ситуация возникает при соблюдении ограничений минимальной кардинальности для связей M-M. Например, связь Album-Song. Нельзя произвести вставку альбома без существования песен, связанных с ним, но и наоборот тоже нельзя, так как у дочернего элемента при вставке обязательно должен быть корректный внешний ключ, ссылающийся на родителя. В PostgreSQL эту проблему можно решить, отложив вызов триггера на вставку альбома (родителя), который проверяет существование дочерних записей, на конец транзакции. Для этого при создании триггера необходимо прописать ключевое слово initially deffered. Таким образом, внутри транзакции сначала происходит вставка альбома, после чего возможна вставка песен с нужным внешним ключом. После вставки последней песни транзакция завершается и вызывается триггер.

Пример триггера на вставку записи в таблицу Album изображен на листинге 2.

create or replace function album\_insert\_trig() returns trigger as $album\_insert\_trig$

begin

if not exists(

select 1 from Song

where albumID = new.albumID

) then

raise exception 'cannot insert album: album % must have at least 1 song', new.title;

end if;

return null;

end;

$album\_insert\_trig$ language plpgsql;

drop trigger if exists album\_insert\_trig on Album;

create constraint trigger album\_insert\_trig

after insert on album

initially deferred

for each row

execute procedure album\_insert\_trig();

Листинг 2 – Триггер на вставку в Album

Данный триггер обеспечивает проверку существования записей в дочерней таблице Song, связанных с данным альбомом. При создании триггера указаны ключевые слова initially deferred, с помощью которых вызов функции триггера откладывается на конец транзакции. Команда raise exception создает сообщение об ошибке и прерывает транзакцию, делая откат.

Аналогичные триггеры, реализованы для всех связей M-M.

Также стоит рассмотреть триггер на удаление записи в таблице Song. Как было отмечено ранее, удаление песни возможно только в случае удаления альбома (каскадное удаление). Благодаря тому, что Postgres удаляет сначала запись родительского отношения и только потом связанные с ней дочерние, в триггере на удаление песни можно выполнить проверку на существование альбома. В случае если его нет, песню можно удалить. Иначе – вызов ошибки. Данный триггер представлен на листинге 3.

create or replace function song\_delete\_trig() returns trigger as $song\_delete\_trig$

begin

if (exists(

select 1 from album a where a.albumID = old.albumID)

) then

raise exception 'cannot delete song: album with id % must have at least 1 song', old.albumID;

end if;

perform lo\_unlink(old.data);

return null;

end;

$song\_delete\_trig$ language plpgsql;

drop trigger if exists song\_delete\_trig on Song;

create constraint trigger song\_delete\_trig

after delete on Song

initially deferred

for each row

execute procedure song\_delete\_trig();

Листинг 3 – Триггер на удаление песни

Если проверка прошла успешно, удаляется связанный с песней Large Object с помощью вызова lo\_unlink.

Также для удобства вставки некоторых записей были реализованы процедуры. Одна из таких процедур – insert\_band. Она представлена на листинге 4.

create or replace procedure insert\_band(bandName varchar(64), genre genre\_enum, description text,

foundingDate date, terminationDate date,

musicianNameP varchar(32), enterDate date,

quitDate date)

language plpgsql

as $$

declare

id int;

begin

insert into band(bandName, genre, description, foundingDate, terminationDate) values

(bandName, genre, description, foundingDate, terminationDate) returning bandID into id;

insert into Membership(musID, bandID, enterDate, quitDate) values

((select musicianID from musician m where m.musicianName = musicianNameP),

id, enterDate, quitDate);

end

$$;

Листинг 4 – Процедура для создания группы

Она принимает параметры для создания группы. Так как у группы должен быть хотя бы один участник, она также принимает данные для создания записи в Membership. Код процедуры выполняется в рамках одной транзакции, поэтому триггер на проверку обеспечения минимальной кардинальности будет выполнен в конце процедуры. PostgreSQL позволяет вернуть некоторые колонки после вставки записи с помощью ключевых слов returning into. В случае данной процедуры при создании группы можно сразу получить авто-инкрементный ключ bandID и обойтись без дополнительного запроса.

Для удобства запросов таблиц с сервера были реализованы представления. На листинге 5 представлено одно из них.

create view AlbumTable as

select title, (select bandname from band b where b.bandid = a.bandid),

(select count(\*) from song s where s.albumid = a.albumid), releaseDate

from album a order by title;

Листинг 5 – Представление для запроса таблицы альбомов

Данное представление используется при отображении таблицы альбомов на клиенте. Оно включает в себя название альбома, группу, дату выпуска и количество песен. Также данное представление сортирует записи по колонке title (название альбома), что необходимо для корректных запросов с ограничением количества записей.

## Реализация клиента

Для создания клиента был выбран фреймворк Qt для C++. Qt является кросс-платформенным инструментом создания приложений рабочего стола с богатой внутренней библиотекой [7]. Его отличительной особенностью является асинхронное взаимодействие между объектами. Оно работает через механизм сигналов и слотов. Когда объект желает оповестить другие объекты о каком-либо событии, он отправляет сигнал, и Qt обеспечивает вызов слотов, которые являются методами у подписавшихся на этот сигнал объектов. Таким образом, отсутствует необходимость, например, в синхронизации потоков. Так как клиент не захочет ждать конца выполнения запроса, желательно переместить взаимодействие с сетью в отдельный поток. С помощью механизма сигналов и слотов сделать это нетрудно.

Глобально клиент состоит из нескольких частей:

1. Графическая составляющая – множество классов для отображения и обработки нажатий на кнопки.
2. Аудио-плеер – используется для записи данных в звуковую карту через мультимедиа библиотеку Qt.
3. Декодер пакетов Ogg Opus – передает декодированные данные по запросу плеера.
4. Парсер сообщений сети – кодирует запросы и декодирует ответы сервера в формате Protobuf.

Для повышения производительности каждая из частей работает в отдельном потоке.

### Главная страница

На главной странице по умолчанию отображается первая из таблиц – таблица групп. Можно нажать на одну из четырех кнопок слева, чтобы отобразить другие таблицы вместо текущей. Также доступен поиск по текущей таблице в виде строки сверху нее. На рисунке 3.1. изображена главная страница приложения.

Запросы таблиц реализованы с помощью слота на прокрутку. Когда пользователь прокручивает таблицу, ячейки проверяются на пустоту. Если ячейка пустая, то она добавляется в запрос. Когда все ячейки проверены, вычисляются пределы запроса, после чего он отправляется на сервер. Пока ответ на него не пришел, интервал, который он покрывает, запрашиваться не будет (при новой прокрутке). На рисунке 3.2. изображена таблица песен. Всего в ней 19 записей, но в базе данных 21 песня. Если прокрутить дальше, загрузятся оставшиеся две.

Рисунок 3.1 – Главная страница

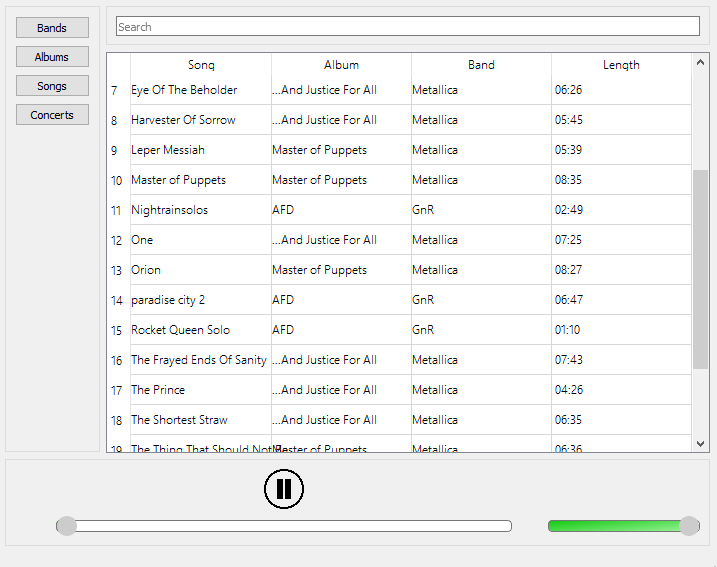


Рисунок 3.2 – Список песен

### Индивидуальные страницы

Нажав на некоторые колонки таблиц можно перейти на разные страницы, отображающие данные о некоторой сущности: группа, альбом или музыкант. При этом делается запрос на сервер с названием сущности и ее типом. При получении ответа страница генерируется и становится активной. На рисунке 3.3. представлена страница группы.

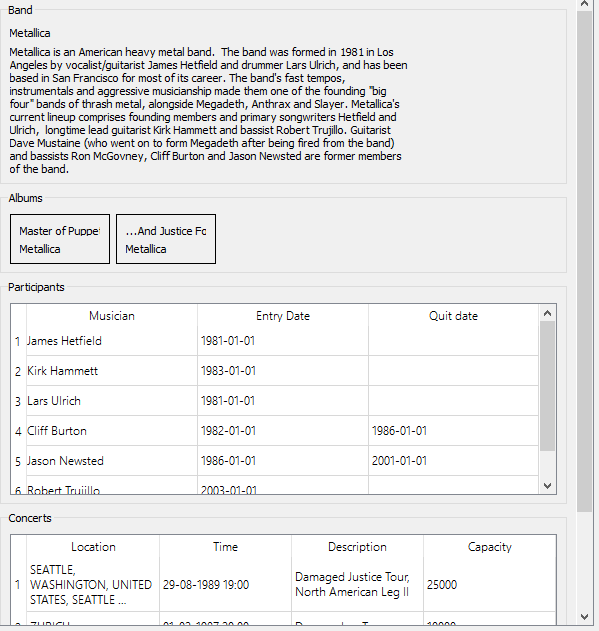


Рисунок 3.3 – Страница группы

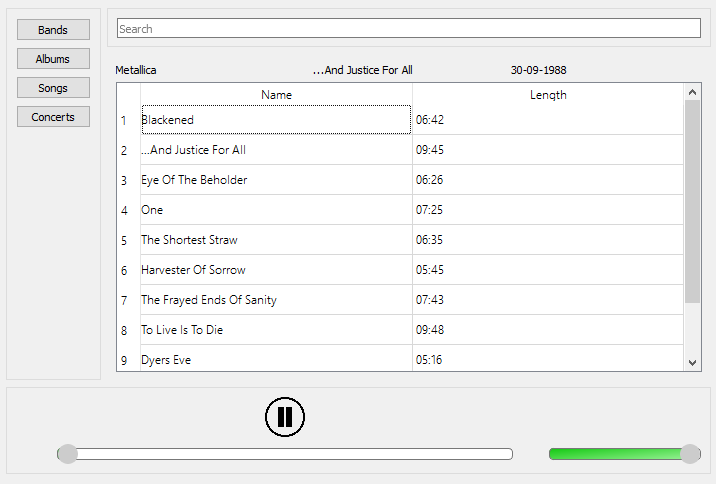
Она состоит из названия, описания, альбомов, участников и концертов группы. С этой страницы можно перейти, например, на страницу альбома, изображенную на рисунке 3.4.

Рисунок 3.4 – Страница альбома

На ней присутствует название группы, альбома, дата выпуска и список песен. Нажав на песню в данной или в общей таблице песен, начнется ее воспроизведение.

### Проигрывание песен

После нажатия на песню на сервер отправляется запрос о потоковой передаче песни. На нижней панели отображается информация о текущей песне: название группы, песни, длительность и текущее время проигрывания. Также присутствует возможность приостановки/воспроизведения песни и изменения громкости звука. На рисунке 3.5. изображен пример проигрывания песни.



Рисунок 3.5 – Воспроизведение песни

При нажатии на другую песню, на сервер придет информация об отмене старого запроса и новой передаче.

Декодирование осуществлялось с помощью библиотеки opusfile[8]. Она позволяет декодировать пакет Ogg Opus. Главным понятием данной библиотеки является поток. Под ним подразумевается сжатый пакет. Для открытия потока можно использовать либо готовые функции (чтение файла или буфера в памяти и др.), либо реализовать свои функции, удовлетворяющие абстрактному интерфейсу чтения потока (функции read, seek, tell, close). Так как данные передаются в реальном времени, ни одна из готовых функций не подходит. Например, для функции чтения буфера должен быть доступен весь пакет на момент открытия потока, поэтому пришлось бы ждать момента окончания передачи. По этой причине пришлось реализовать свои функции абстрактного интерфейса.

## Реализация сервера

В качества языка реализации сервера был выбран Go. Go идеально подходит для реализации сетевых приложений, так как располагает богатой стандартной библиотекой и удобной для работы конкурентностью. Для работы с СУБД PostgreSQL используется драйвер pgx [9].

Для удобства работы с сетью разработана небольшая библиотека tcpbuf, которая абстрагирует пользователя от вызовов сетевых функций стандартной библиотеки Go. Вместо этого реализуется вызов пользовательских функций при определенных событиях, например, при подключении нового клиента. Для записи и чтения выделяется отдельный буфер, в который копируются пришедшие из сети данные. Этот буфер можно считывать частично или полностью в функции-коллбэке. Таким образом удобно обрабатывать пакеты, когда пришла только часть данных. В данном случае считывается ровно столько, сколько занимает места один пакет, и буферизация со стороны пользователя не нужна. При появлении новых данных или при возможности записать данные вызываются соответствующие функции.

Для сервера разработан файл конфигурации в формате JSON, который при желании можно изменить. В нем содержатся параметры подключения к базе данных, уровень логирования, IP-адрес сервера и конфигурация библиотеки tcpbuf. Пример конфигурационного файла расположен в приложении Б.

## Реализация приложения администратора

Так как добавление песен является нетривиальной задачей, было решено разработать приложение для администратора для добавления альбома в базу данных. Так как приложение не предполагает сложного взаимодействия, его можно сделать консольным. Для реализации графического интерфейса использовалась библиотека promptui [10]. На рисунке изображен процесс добавления альбома в базу данных.

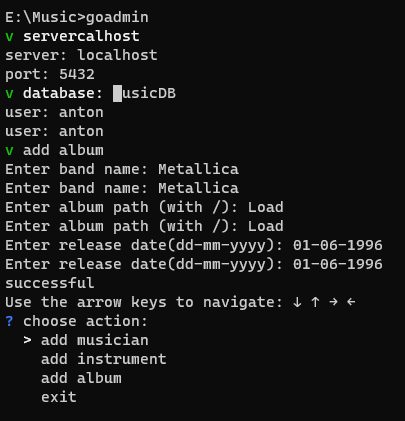


Рисунок 3.6 – Добавление альбома

Сначала считываются параметры подключения к базе данных. Дальше выбирается действие «Добавить альбом» и вводится информация о нем: группа, дата выпуска в определенном формате (средствами библиотеки реализуется проверка введенного результата) и путь до папки с композициями. Название альбома будет как у папки, названия песен должны быть с добавленным к ним префиксом «индекс\_», обозначающими индекс в альбоме. Дальше с помощью утилиты ffmpeg аудио-файл сжимается с помощью кодека Opus. В приложении поддерживаются файлы mp3 и wav, однако легко ослабить это ограничение и реализовать добавление других форматов. Главное, чтобы они поддерживались ffmpeg и кодеком Opus.

Помимо этого, реализовано добавление музыкантов и инструментов у них.

# Тестирование

Все части клиент-серверного приложения запускались в ОС Windows 10. В качестве системы контроля версий использовался Git. Для проверки работоспособности после нововведений использовалось ручное тестирование. Важной частью тестирования сетевых приложений, особенно реального времени, является нагрузочное тестирование. Оно может помочь выявить неверные решения при проектировании и места программы, в которых требуется оптимизация. Однако его провести не удалось по причине наличия лишь одной локальной машины.

Результаты итогового тестирования триггеров и процедур базы данных представлены в таблице 4.1.

Таблица 4.1 – Итоговое тестирование в БД

|  |  |  |  |
| --- | --- | --- | --- |
| Действие | Ожидаемый результат | Получившийся результат | Совпадение |
| Вставка записи в таблицу Musician с помощью процедуры insert\_musician | Успешная вставка музыканта | - | + |
| Вставка музыканта простым запросом INSERT без указания инструмента | Ошибка | Ошибка: «ERROR: musician Eddie Vedder must have at least 1 instrument» | + |
| Удаление одного из нескольких инструментов музыканта | Успешное удаление | - | + |
| Удаление инструмента у музыканта с единственным инструментом | Ошибка | Ошибка: «ERROR: cannot delete instrument because musician must have at least 1 instrument» | + |
| Удаление музыканта | Удаление музыканта, инструмента и его участия в группах | - | + |
| Вставка группы с помощью процедуры insert\_band | Успешная вставка группы | - | + |
| Вставка группы без указания хотя бы одного участника | Ошибка | Ошибка: « ERROR: cannot insert band: band GnR must have at least 1 participant» | + |
| Удаление единственного участника группы | Ошибка | Ошибка: « ERROR: cannot delete membership: band 1 must have at least 1 participant» | + |
| Вставка участия музыканта в группе с помощью процедуры insert\_membership | Успешная вставка | - | + |
| Вставка альбома с помощью вспомогательного приложения | Успешная вставка | - | + |
| Вставка альбома без песен с помощью простого insert | Ошибка | Ошибка: «ERROR: cannot insert album: album Load must have at least 1 song» | + |
| Удаление альбома | Успешное удаление альбома и песен | - | + |
| Удаление песни | Ошибка | Ошибка: «ERROR: cannot delete song Battery: only cascade delete is permitted» | + |

# ЗАКЛЮЧЕНИЕ

В рамках данной курсовой работы были спроектированы и реализованы база данных музыкальной платформы и соответствующее ей клиент-серверное приложение, реализующее просмотр клиентом сущностей базы данных и осуществляющее потоковую передачу звука для прослушивания.

Также были приобретены и улучшены навыки разработки многопоточных приложений, использования СУБД PostgreSQL и протокола Protobuf. Были изучены различные способы потоковой передачи данных, кодирования аудио-файлов.

Дальнейшая разработка может быть направлена на:

1. добавление возможности перемотки песен
2. добавление новых сущностей в базу данных, например, синглов
3. улучшение графического интерфейса приложения
4. добавление новых страниц, например, страницы с описанием концерта
5. добавление кэширования запросов у клиента

CПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

* + 1. Официальная документация IBM про стек протоколов TCP/IP [Электронный ресурс] - URL: https://www.ibm.com/docs/ru/aix/7.2?topic= protocol-tcpip-protocols (дата обращения: 15.Февраль.2022)
    2. Статья Sony про цифровое аудио [Электронный ресурс] - URL: https://www.sony.ru/electronics/audio-vysokogo-razresheniya-ot-a-do-ja-lyubiteli-audiotekhnologij (дата обращения: 15.Февраль.2022)
    3. Официальная документация Google про протокол Protobuf [Электронный ресурс] - URL: https://developers.google.com/protocol-buffers/docs/overview (дата обращения: 27.Февраль.2022)
    4. Официальная документация кодека Opus [Электронный ресурс] - URL: https://opus-codec.org/ (дата обращения: 7.Март.2022)
    5. Официальная документация контейнера Ogg [Электронный ресурс] - URL: https://datatracker.ietf.org/doc/html/rfc3533 (дата обращения: 7.Март.2022)
    6. Официальная документация PostgreSQL про Large Objects Ogg [Электронный ресурс] - URL: https://www.postgresql.org/docs/current/lo-intro.html (дата обращения: 20.Март.2022)
    7. Официальная документация Qt [Электронный ресурс] - URL: https://www.qt.io/product/qt-for-desktop (дата обращения: 10.Апрель.2022)
    8. Официальная документация библиотеки opusfile [Электронный ресурс] - URL: https://opus-codec.org/docs/opusfile\_api-0.12/index.html (дата обращения: 12.Апрель.2022)
    9. Документация библиотеки pgx [Электронный ресурс] - URL: https://pkg.go.dev/github.com/jackc/pgx (дата обращения: 25.Апрель.2022)
    10. Документация библиотеки promptui [Электронный ресурс] - URL: https://pkg.go.dev/github.com/manifoldco/promptui (дата обращения: 15.май.2022)

# ПРИЛОЖЕНИЕ А

Протокол клиент-серверного приложения в кодировке Protobuf.

syntax = "proto3";

option go\_package = "./proto";

enum EntityType {

BAND = 0;

ALBUM = 1;

SONG = 2;

CONCERT = 3;

MUSICIAN = 4;

MEMBERSHIP = 5;

}

message Request {

uint64 reqId = 1;

bool cancel = 3; //send again with cancel = true to cancel request

oneof msg {

TableReq tableReq = 4;

SimpleReq simpleReq = 5;

StreamReq streamReq = 6;

}

}

message Answer {

uint64 reqId = 1;

bool cancel = 2; //failed req on server side

oneof msg {

TableAns tableAns = 3;

SimpleAns simpleAns = 4;

StreamAns streamAns = 5;

}

}

message TableReq {

uint32 first = 1;

uint32 last = 2;

string filter = 3;

EntityType type = 4;

}

message SimpleReq {

string reqString = 1;

EntityType type = 4;

}

message StreamReq {

string reqString = 1;

int32 suggestedSize = 2;

EntityType type = 3;

}

message TableAns {

repeated Band bands = 1;

repeated Album albums = 2;

repeated Concert concerts = 3;

repeated Song songs = 4;

EntityType type = 5;

}

message SimpleAns {

oneof msg {

Band band = 1;

Album album = 2;

Song song = 3;

Concert concert = 4;

Musician musician = 5;

Membership membership = 6;

}

}

message StreamAns {

bytes data = 1;

bool isFinal = 2;

}

enum Genre {

ROCK = 0;

ALTERNATIVE = 1;

INDIE = 2;

BLUES = 3;

METAL = 4;

}

message Membership {

string musName = 1;

string bandName = 2;

int64 unixEntryDate = 3;б

int64 unixQuitDate = 4;

}

message Concert {

int32 capacity = 1;

string description = 2;

int64 unixDateTime = 3;

string location = 4;

repeated string bandNames = 5;

}

message Band {

string bandName = 1;

Genre genre = 2;

int64 unixFoundDate = 3;

int64 unixTermDate = 4;

string description = 5;

repeated string albumNames = 6;

repeated Membership participants = 7;

repeated Concert concerts = 8;

}

message Song {

string songName = 1;

int32 lengthSec = 2;

string albumName = 3;

string bandName = 4;

}

message Album {

string title = 1;

int64 unixReleaseDate = 2;

repeated Song songs = 3;

string bandName = 4;

}

message Musician {

string musName = 1;

int64 unixDateOfBirth = 2;

string bio = 3;

repeated Membership memberships = 4;

}

# ПРИЛОЖЕНИЕ Б

Пример конфигурационного файла сервера.

{

"host": "192.168.1.30",

"port": 3018,

"ms\_loop": 100, //loop interval in ms

"in\_size": 65536, //input buffer size

"out\_size": 65536, //output buffer size

"db\_server": "localhost",

"db\_port": 5432,

"db\_name": "MusicDB",

"db\_user": "postgres",

"db\_pass": "<pass>",

"log\_lvl": "debug", //debug, info, warn, error, crit

}