МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ ДНР

ДОНЕЦКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Институт КНТ

Факультет ФИСП

Кафедра ПИ им. Л.П. Фельдмана

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к проекту по курсу

«Практика»

Выполнили:

ст. гр. ПИ-18А

Данильчук К.М.

ст. гр. ПИ–18Б

Моргунов А.Г.

Руководитель:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ДОНЕЦК – 2021

РЕФЕРАТ

Пояснительная записка содержит: 104 страницы, 16 рисунков, 4 источников, 7 приложений.

Цель проектирования: выполнить обоснованный проект телеграмм-бота, который удовлетворяет ряду специальных требований и реализует функции, описанные в техническом задании (приложение А-Б).

Для реализации цели курсового проекта необходимо: выявить требования к пользовательскому интерфейсу бота и сравнить его с уже существующими аналогами, выбрать средства реализации, которые удовлетворяют требованиям будущей системы, спроектировать систему, позволяющую реализовать все функции, описанные в техническом задании, спроектировать и реализовать базу данных, которая хранит информацию о подписках (какую группу прослушивать, куда отсылать, по каким ключевым словам фильтровать и т.д.).

Результатом работы является программа, которая позволяет подписываться на обновления групп в телеграмме и пересылать новые сообщения в указанные в подписке группы.

ТЕЛЕГРАММ, ТЕЛЕГРАММ-БОТ, БАЗА ДАННЫХ, СЕРВЕР, НОВОСТЬ, ОБНОВЛЕНИЕ, ПЕРЕСЫЛКА, СООБЩЕНИЕ, ВЛОЖЕНИЕ

СОДЕРЖАНИЕ

[Ведение 5](#_Toc83052527)

[1 Постановка задачи 6](#_Toc83052528)

[1.1 Задача заказчика 6](#_Toc83052529)

[1.2 Идея команды разработчиков 6](#_Toc83052530)

[1.3 Экономическое обоснование 7](#_Toc83052531)

[2 Архитектура 8](#_Toc83052532)

[2.1 Сущности 8](#_Toc83052533)

[2.1.1 Телеграмм 8](#_Toc83052534)

[2.1.2 Телеграмм-бот 8](#_Toc83052535)

[2.1.3 База данных 9](#_Toc83052536)

[2.1.4 Медиатор 9](#_Toc83052537)

[2.2 Диаграмма классов 10](#_Toc83052538)

[2.3 Диаграмма прецедентов 11](#_Toc83052539)

[2.4 Диаграмма размещения 11](#_Toc83052540)

[2.5 Диаграмма объектов 12](#_Toc83052541)

[2.6 Диаграмма состояний 12](#_Toc83052542)

[2.7 Диаграмма последовательности 14](#_Toc83052543)

[2.8 Диаграмма компонентов 14](#_Toc83052544)

[3 Выбор средств 16](#_Toc83052545)

[3.1 Платформа и окружение 16](#_Toc83052546)

[3.2 Язык программирования 16](#_Toc83052547)

[3.3 Third-party 17](#_Toc83052548)

[3.3.1 TdLib 17](#_Toc83052549)

[3.3.2 reo7sp/tgbot-cpp 17](#_Toc83052550)

[3.4 СУБД 18](#_Toc83052551)

[3.4.1 Выбор СУБД 18](#_Toc83052552)

[3.4.2 Выбор средств для работы с СУБД 20](#_Toc83052553)

[4 Реализация 21](#_Toc83052554)

[4.1 Взаимодействие с telegram API 21](#_Toc83052555)

[4.2 Взаимодействие с Bot API 22](#_Toc83052556)

[4.3 Взаимодействие с PQXX 23](#_Toc83052557)

[4.4 Логирование 24](#_Toc83052558)

[5 Перспективы развития 25](#_Toc83052559)

[5.1 Идея 25](#_Toc83052560)

[5.2 Архитектура 25](#_Toc83052561)

[5.2.1 Сервер 26](#_Toc83052562)

[5.2.2 Сайт 26](#_Toc83052563)

[Выводы 27](#_Toc83052564)

[Литература 28](#_Toc83052565)

[Приложение А. Техническое задание (сервер) 29](#_Toc83052566)

[Приложение Б. Техническое задание (клиент) 40](#_Toc83052567)

[Приложение В. Экранные формы 50](#_Toc83052568)

[Приложение Г. Статистика приложения 52](#_Toc83052569)

[Приложение Д. Руководство пользователя 53](#_Toc83052570)

[Приложение Е. Руководство администратора 54](#_Toc83052571)

[Приложение Ж. Код приложения 55](#_Toc83052572)

# Ведение

Telegram — кроссплатформенная система мгновенного обмена сообщениями (мессенджер) с функциями VoIP, позволяющая обмениваться текстовыми, голосовыми и видеосообщениями, стикерами и фотографиями, файлами многих форматов. Клиентские приложения Telegram доступны для Android, IOS, Windows Phone, Windows, MacOS и GNU/Linux. Количество ежемесячных активных пользователей сервиса по состоянию на январь 2021 года составляет около 500 млн человек. [1]

Телеграмм предоставляет API для создания ботов – специальных аккаунтов телеграмм, которые могут быть запрограммированы и выполнять определенные действия на заранее сформулированные команды или действия.

При выполнении проекта были проанализированы интерфейсы различных телеграмм-ботов и выбран наиболее понятный для пользователя, спроектирована структура базы данных, необходимая для хранения информации о пользователях и созданных ими подписках, распределены функции системы между ботом и сервером, а также был внедрен механизм подписки на использование услуг бота, который автоматически отключает пользователя от системы при отсутствии продления аренды.

# 1 Постановка задачи

## 1.1 Задача заказчика

Задача заказчика заключается в частичной автоматизации процесса введения каналов в телеграмме (аналог группы в ВК).

Для реализации автоматизации было предложено сделать поисковой бот, который является интерфейсом для серверной части. Серверная часть занимается тем, что производит поиск сообщений по ключевой фразе в определенном списке телеграмм каналов.

Таким образом для того, чтобы опубликовать новость в определенный канал, необходимо создать список каналов, где мы будем производить поиск, далее произвести поиск и опубликовать понравившиеся записи в другой канал.

После чего было сразу понятно, что необходимо сделать данную функциональную возможность доступную для большего круга людей, тогда появилась идея создать арендную систему и разграничивать права доступа к сервису.

После озвучивания данного требования, согласования сроков и стоимости проекта было составлено техническое задание.

После первой реализации, заказчик захотел сделать сортировку по просмотрам для каждой найденной записи.

Следует учесть, что для удобства использования результатов поиска потребовалось создать уникальный интерфейс с постраничной навигацией при помощи средств телеграмма.

## 1.2 Идея команды разработчиков

Идея команды разработчиков состояло в том, чтобы полностью автоматизировать процесс введения каналов. Для этого требовалось получать все новые сообщения из каналов, фильтровать их и рассылать в соответствующие каналы (получатели). Задача сильно усложняется, но при реализации получится полностью отказаться от ручного поиска и выбора новостей с дальнейшей публикацией.

Если говорить более формально, то необходимо получить новости из всех интересующих телеграмм каналов, посмотреть какие телеграмм каналы подписаны на данные новости, проверить новости на содержание ключевых фраз, и далее если новость проходит полную проверку, то опубликовать в канал (получатель), таким образом полностью заменить человеческий труд. Человеку необходимо только настроить связи между каналами-отправителями и каналами-получателями, а также установить ключевые фразы.

Данная идея очень понравилась заказчику, и после реализации части заказчика, команда разработчиков сразу преступила к реализации новой функциональности, которая в будущем полностью затмила старую потребность в поиске новостей.

## 1.3 Экономическое обоснование

Зарплата одного администратора составляет 20.000 руб. – 100.000 руб. в месяц, за 5-10 каналов.

Данные траты постоянные, т.к. если не платить зарплату администраторам, то каналы простаивают.

В случае использования решения с поиском заработная плата минимизируется, т.к. поиск осуществляется автоматически.

Если использовать наше решение, то затраты сводятся к 0, т.к. для работы канала требуется одна настройка, которую может осуществить владелец канала, за несколько часов (в зависимости от сложности канала), и далее только немного изменять конфигурации по предпочтениям.

В связи с этим пользователь может ввести несколько каналов за фиксированную сумму в месяц, которая явно меньше, чем платить человеку (приложение ориентированно на идеи с подписками – временный сервис за определенную сумму).

# 2 Архитектура

## 2.1 Сущности

В процессе анализа требований к проекту были выявлены следующие сущности:

* Телеграмм;
* Телеграмм-бот;
* База данных;
* Медиатор.

## 2.1.1 Телеграмм

Эта сущность отвечает за работу с информацией в телеграмме, а именно за просмотр обновлений каналов, на которые была осуществлена подписка, получение служебной информации из опубликованного сообщения и выделение из нее значимой части, пересылку сообщений из одного канала в другой, осуществляет поиск, по ключевым словам, в сообщениях сообщества.

## 2.1.2 Телеграмм-бот

Телеграмм-бот выступает пользовательским интерфейсом, то есть позволяет пользователю взаимодействовать с системой. Эта сущность интерпретирует команды, которые вводит пользователь и выполняет связанные с этой командой действия. Расширяет функциональность базового API бота путем предоставления возможности ведения цепочки сообщений для диалога с пользователем.

## 2.1.3 База данных

База данных – сущность, которая предназначена для хранения информации, которая может понадобится системе или администраторам во время работы. База данных хранит в себе: информацию о пользователях, оформленных ими подписках, сроке окончания аренды для каждого пользователя, а также информацию о наличии прав администратора каждого пользователя. Также система сохраняет историю запросов в удобном для просмотра виде.

## 2.1.4 Медиатор

Медиатор является связующим звеном системы. Он принимает сообщения, которые приходят из сущности телеграмма, берет из базы данных связи, которые относятся к сообществу, из которого пришло сообщение, фильтрует его, по ключевым словам, определенным для каждой связи, и рассылает ко всем адресатам, которые подходят по связям и ключевым словам. Телеграмм-бот является интерфейсом к данной сущности и позволяет менять конфигурацию связей в базе данных, регистрировать новых пользователей и администраторов, продлевать аренду. Преобразует список найденных новостей в сообщение с постраничным просмотром всех выданных результатов.

## 2.2 Диаграмма классов

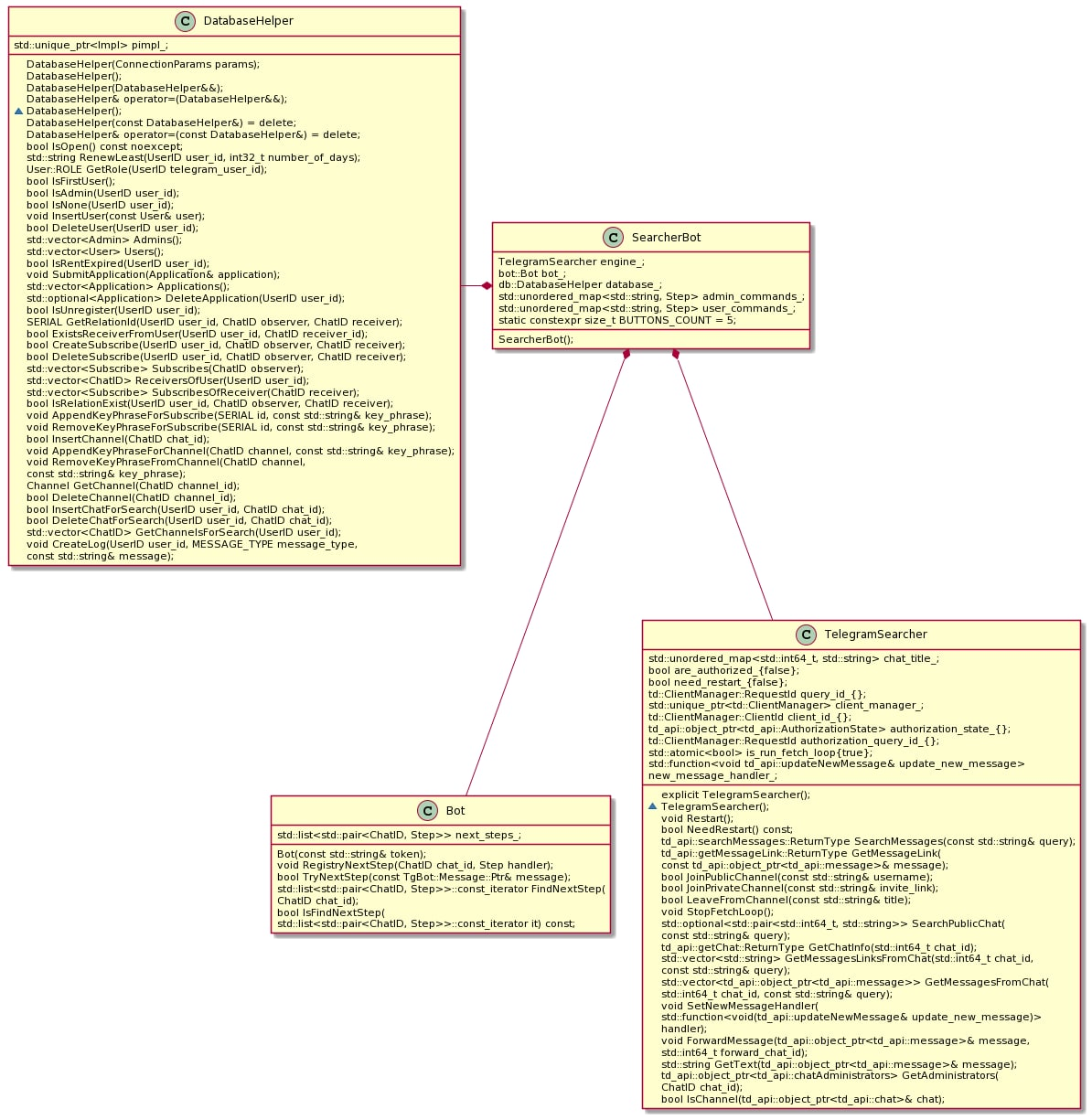


Рисунок 2.1 – Диаграмма классов

## 2.3 Диаграмма прецедентов

Варианты использования системы пользователем и администратором приведены на рисунках 2.2 и 2.3 соответственно. Администратор обладает всем функционалом пользователя, а также может администрировать систему.

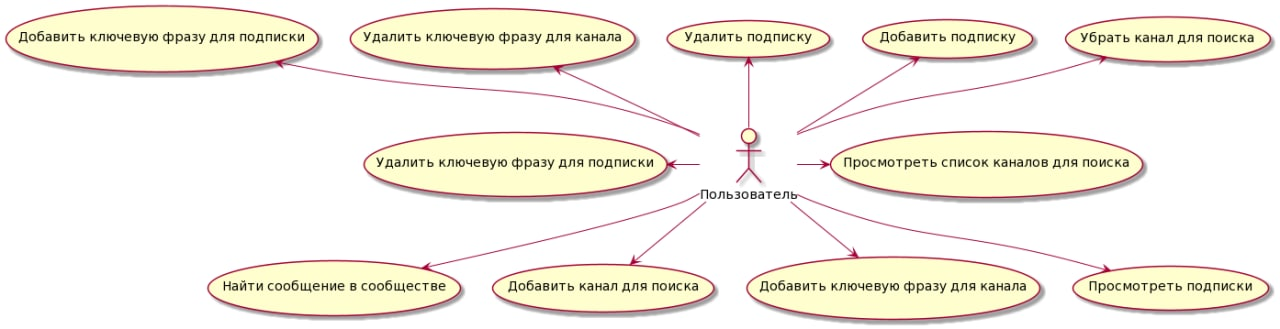


Рисунок 2.2 – Диаграмма прецедентов (Пользователь)

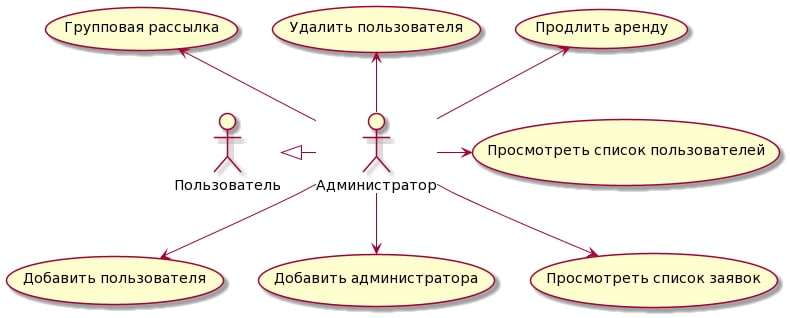


Рисунок 2.3 – Диаграмма прецедентов (Администратор)

## 2.4 Диаграмма размещения

Все модули системы располагаются на одной машине, которая имеет связь с сервером Telegram.

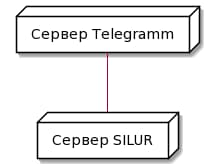


Рисунок 2.4 – Диаграмма размещения

## 2.5 Диаграмма объектов

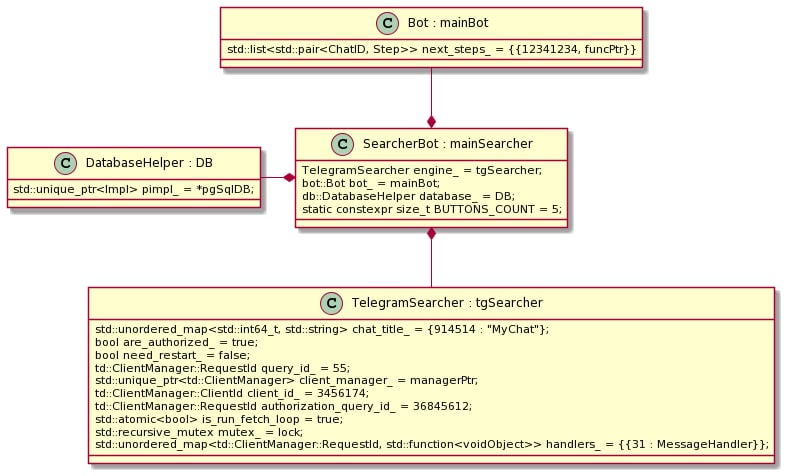


Рисунок 2.5 – Диаграмма объектов

## 2.6 Диаграмма состояний

На рисунке 2.6 приведен пример диаграммы состояний для функции «Добавить ключевую фразу к связи».

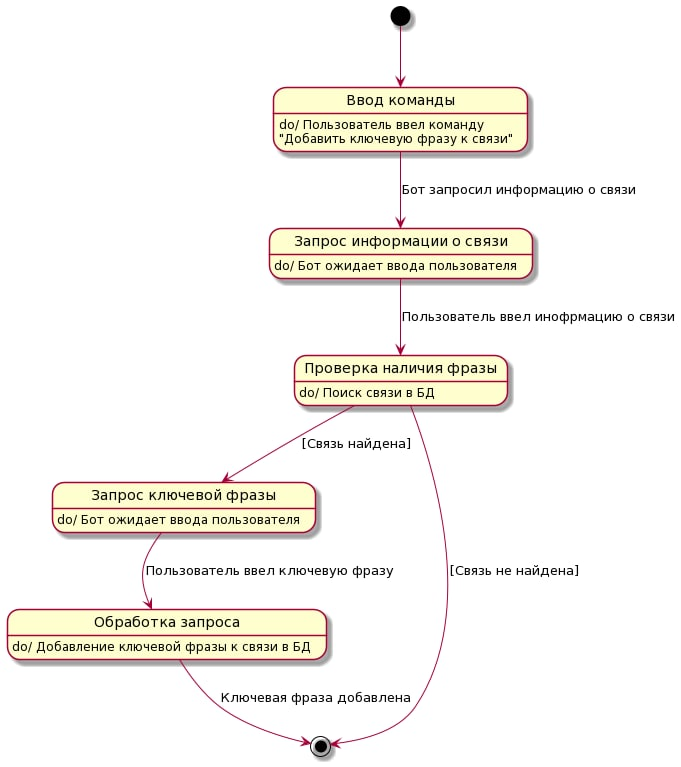


Рисунок 2.6 – Диаграмма состояний

## 2.7 Диаграмма последовательности

На рисунке 2.7 приведен пример диаграммы последовательности для функции «Добавить ключевую фразу к связи».

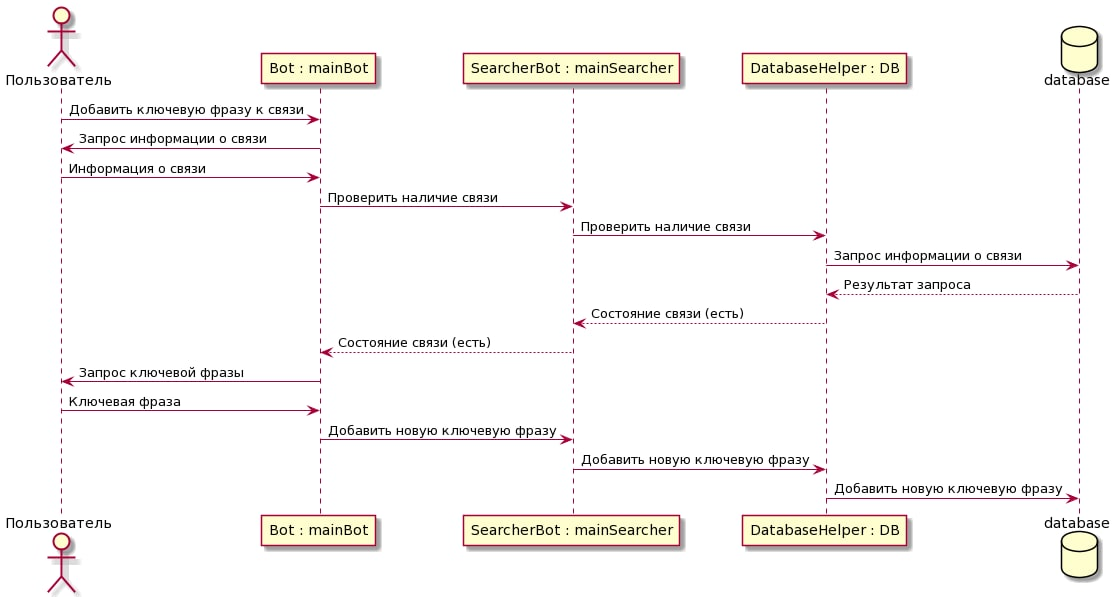


Рисунок 2.7 – Диаграмма последовательности

## 2.8 Диаграмма компонентов

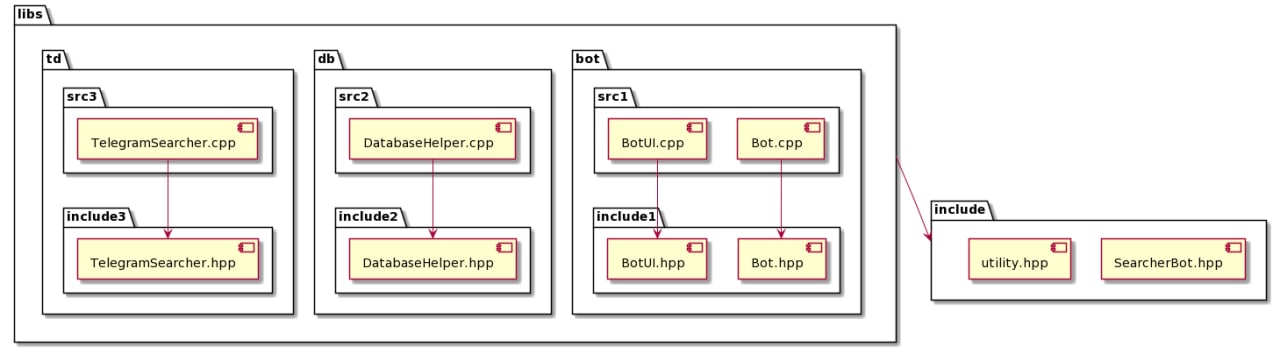


Рисунок 2.8 – Диаграмма компонентов

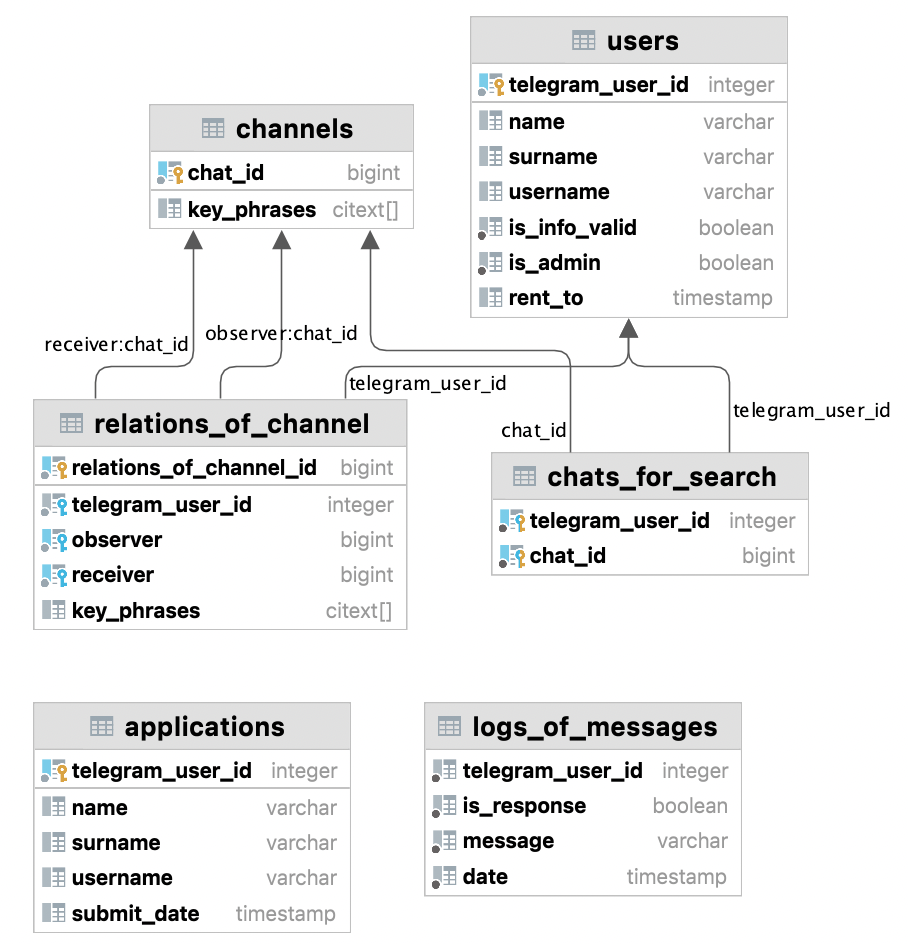


Рисунок 2.9 – Схема базы данных

# 3 Выбор средств

Выбор средств является ключевой особенностью реализации проекта, т.к. при фиксации данного параметра от него сложно в будущем отказаться.

В связи важности данного решения была выбрана UNIX платформа, т.к. данная платформа предоставляет большее количество возможностей, чем Windows или MacOS и является более общедоступной и дешевой по сравнению с конкурентами.

## 3.1 Платформа и окружение

В качестве операционной системы была выбрана Ubuntu 20.04 LTS, т.к. именно её предоставляют большинство облачных серверов.

Архитектура x86\_64, из-за своей популярности и пока-что монопольного владения рынком компьютеров. В будущем лучше подойдут ARM архитектура из-за своей производительности.

Т.к. команда разработчиков состоит из нескольких человек и работают на разных устройствах, то данное решение будет работать внутри контейнера Docker, с архитектурой аналогичной архитектуре сервера, что позволит быстро разрабатывать программный продукт и не терять в производительности из-за контейнеризации.

Выбор облачного сервера обуславливается тем, что непонятна начальная загрузка системы, поэтому за заказчиком остается возможность арендовать сервер с минимальными характеристиками и далее расширять технические возможности арендуя большую мощность.

## 3.2 Язык программирования

Для работы с telegram API и telegram BOT API можно воспользоваться многими языками программирования, но т.к. система тесно связанная с многопоточностью и производительностью, то множество языков существенно сокращается: C, RUST, C++.

Язык программирования C – не предоставляет современных абстракций (динамического полиморфизма), что существенно усложняет масштабируемость и разработку проекта, ну и также данный язык имеет меньший набор инструментов, чем остальные, хоть и предоставляет отличную производительность.

Язык программирования RUST – очень хороший кандидат на разработку, но у RUST есть ряд особенностей, во-первых, он (пока что) не такой производительный как C/C++, не такой популярный, а также он не был изучен командой разработки, до начала проекта.

Язык программирования C++ – идеальный вариант, который сочетает и производительность, и предоставление всех современных абстракций для написания быстрого и простого кода с дальнейшей поддержкой. Также ключевым аспектом являлось то, что team-leader отлично знал данный язык и мог помочь остальным участникам проекта в его быстром освоении.

## 3.3 Third-party

## 3.3.1 TdLib

Для работы с telegram API была выбрана библиотека TdLib предоставляемая разработчиками телеграмма, которая реализует протоколы передачи данных и авторизации, а также реализует криптографическую защиту данных. Данная библиотека предоставляет функциональность для написание пользовательских клиентов. [2]

## 3.3.2 reo7sp/tgbot-cpp

Данная библиотека является обёрткой на Bot API от telegram, что позволяет работать с ботом не через HTTP протокол и JSON, а при помощи методов на C++.

Выбор на данную библиотеку был также обоснован тем, что team-leader программного продукта является contributor`ом библиотеки.

Интерфейс библиотеки предоставляется в лучших традициях современного C++, что позволяет разрабатывать с большой производительностью. [3]

## 3.4 СУБД

## 3.4.1 Выбор СУБД

На данный момент в мире существует большое количество СУБД: MySQL, SQLite, Oracle, PostgreSQL и т.д. Также существует No SQL СУБД, ярким представителем является Mongo, который хорошо себя показывает в стартапах и небольших хранилищах данных.

MySQL не обеспечивает параллельной обработки данных, а также не полностью соответствует SQL, но самый главный недостаток данной базы данных – отсутствие ролей, более новые версии начали поддержку данной функциональности, но там отсутствует row level sequrity (RLS), который необходим для реализации проекта.

SQLite не годится по многим причинам: производительность, отсутствие многопоточной поддержки, отсутствие ролей, плохая масштабируемость. Обычно SQLite используют как локальную базу пользователя, что не годится под данную задачу.

Oracle достаточно современная СУБД и во много превосходит PostgreSQL, но есть один очень существенный недостаток: это не opensource проект, то есть мы полагаемся на безопасность данных полностью на другую компанию, что недопустимо в случае со страховыми компаниями.

В качестве СУБД был выбран PostgreSQL. PostgreSQL идеально подходит для решения задачи, т.к. имеет функциональность:

* для реализации разграничения доступа на уровне ролей пользователей;
* может содержать, неограниченные СУБД, объемы данных;
* имеет возможность разрабатывать логику по управлению с данными на сервере (при помощи хранимых процедур и функций), что позволяет не изменять, по возможности, клиентскую часть программного комплекса;
* существует поддержка большого количество встроенных типов, а также возможность создавать свои типы с поддержкой встраивания ограничения в тип (домены);
* существует возможность оптимизировать запросы при помощи индексов в следствии анализа предметной области и выявления данных, к которым имеется большое количество запросов и по каким данным происходит сортировка и фильтрация, при помощи индексов;
* в будущем существует возможность горизонтального масштабирования, при помощи партицирования распределять данные на разные кластера.
* кроссплатформенность, конечно, редко происходит «переезд» базы данных с одной операционной системы на другую, но данная функциональность не является минусом – значит плюс.
* Открытый исходный код – то, из-за чего выбор пал на данную СУБД, а не на Oracle, ведь перед выпуском в продакшн специалисты могут проверить код СУБД на уязвимости и в случае обнаружения устранить.

## 3.4.2 Выбор средств для работы с СУБД

Для взаимодействия с СУБД была выбрана библиотека PQXX, которая является обёрткой над библиотекой PQ. Данная библиотека позволяет производить множество подключений к серверу PostgreSQL и производить параллельные запросы, а также поддерживает современные стандарты С++, что является огромным преимуществом данной библиотеки.

API очень хорошо документировано, сообщество развивается уже несколько лет, а архитектура библиотеки интуитивно понятна из её назначения.

Хочется также отметить, что в библиотеки продуманны разные виды транзакций: для чтения, записи, и запросы без транзакции (для более производительных запросов), особенность данной детали является то, что библиотека не навязывает все технические тонкости взаимодействия с такой СУБД как PostgreSQL, а предоставляет минимальное API, для взаимодействия, что позволяет быстро написать прототипы взаимодействия не думая о производительности и безопасности на ранних стадиях разработки, но позволяет в будущем произвести очень тонкую конфигурацию работы с СУБД.

# 4 Реализация

При реализации компонентов системы необходимо взаимодействовать с другими API, которые предоставляют некие гарантии только при соблюдении неких правил, что вносит некие функциональные ограничения, то есть проблемы, из-за которых приходится либо подстраиваться под их правила, либо подстраивать их API под правила нашей системы.

## 4.1 Взаимодействие с telegram API

Telegram API предоставляет возможности работы с телеграммов на самом низком уровне, что позволяет делать некоторые действия, которые недоступны даже в официальных клиентах, или автоматизировать некие процессы пользователя, выполняя шаблонные действия.

Telegram создал TdLib (Telegram Database library), которая предоставляет большой функционал для работы, шифрование, запросы к серверу Telegram, асинхронные операции.

Первая трудность – TdLib не предоставляет синхронные операции, которые позволяют сообщать о результатах выполнения действия пользователю, или например, дожидаться ответа от сервера информации о каком-то канале.

Вторая трудность – через одна подключение нельзя отправлять несколько запросов параллельно, что накладывает ограничения на приложение, если оно работает в нескольких потоках.

Решение первой трудности, реализуется за счет создания своего синхронного API. В данном проекте реализация не составила труда, т.к. C++ предоставляет возможность работы с шаблонами, таким образом удалось создать шаблон синхронного запроса к серверу, для разных типов запросов.

Вторая проблема более сложная и требует понимания примитивов синхронизации, если кратко, то проблема решилась за счет рекурсивных mutex’ов, почему нельзя было воспользоваться стандартным средством синхронизации, потому что в ходе обработки одного запроса, может возникнуть необходимость отослать еще один запрос на сервер, но отправить запрос на сервер означает что необходимо еще раз захватить mutex, чтобы избежать deadlock необходимо воспользоваться mutex или пересмотреть архитектуру приложения, т.к. рекурсивный mutex не является хорошим тоном, но в данном случае это решения самое оптимальное, если поменять архитектуру приложения, то возникнут ограничения на то, что нельзя вызывать запросы к серверу внутри кода, который обрабатывает запросы к серверу, т.е. страдает клиентский код, что не допустимо.

Данная проблема сразу стала очевидной, когда приложение зависло при получения нового сообщения, потому что в обработчике сообщения проверяется, что сообщение пришло из чата, группы или канала, для этого необходим дополнительный запрос к серверу. Можно было ограничить пользователя, например складывать сообщения в очередь, а обработки сообщения еще одним потоком разбирать очередь и обрабатывать другие запросы к серверу, но в этом случае очень сильно страдает клиентский код.

## 4.2 Взаимодействие с Bot API

Взаимодействие с ботом от телеграмма проходило через API телеграмма, которое реализовано в библиотеке tgbot-cpp, проблема заключалась только в том, что у бота есть event-loop, и для работы сервера необходим event-loop (в данном случае, он наш). Существует несколько вариантов как решить данную проблему, можно сделать event-loop не бесконечными, что позволит нам время от времени выходить из одного event-loop и заходить в другой и так по очереди до бесконечности. Решение хорошее, но очень плохое, проблема в том, что в любом из event-loop можно очень долго находится, если, например нет событий, хотя в другом event-loop уже накопились необработанные события.

Для решения проблемы необходимо разбить приложение на несколько потоков, один поток разбирает event-loop сервера, а другой event-loop бота. Если учесть то, что синхронизация критических секций в приложении уже реализована в классах, где это важно, то синхронизация между двумя event-loop не требуется. Обработчики для разных event-loop работают с классами, где уже все критические секции защищены.

Еще одна очень важна проблема, которую необходимо решить, получение от одного пользователя нескольких сообщений подряд.

Существует несколько решений данной проблемы, самое лучшее – coroutines, но они появятся только в C++20, когда компиляторы начнут их поддерживать, а пока это нецелесообразно.

Но понимания устройства coroutines позволяет понять, как решить данную задачу.

Решение: создадим очередь, где будем хранить id чата (пользователя) и обработчик на сообщение. Далее необходимо написать функцию, которая умеет регистрировать обработку следующего сообщение пользователя, и дальше, когда приходит сообщение от пользователя, для начала нужно посмотреть существует для него следующих шаг, если да, то обработать сообщение данным шагом, если нет, то обработать пользователя по общему правилу или можно считать, что пользователь находится в начальном состоянии и имеет стартовый обработчик.

На этом проблемы с телеграмм ботом не заканчиваются, например в процессе разработки, team-leader проекта правил баги в библиотеке, при этом став contributor’от библиотеки.

## 4.3 Взаимодействие с PQXX

PQXX – библиотека для взаимодействия с сервером базы данных PostgreSQL. Данная библиотека предоставляет очень хорошее API. Но в ней существует 2 особенности.

Первая особенность, на MacOS и на Ubuntu ставятся разные версии, которые несовместимы, поэтому необходимо писать код под меньшую версию.

Вторая особенность, через одно соединение нельзя делать параллельные запросы, первое решение – делать много соединение (плохое решение), второе решение, код с запросами вынести в отдельный метод, и сам метод сделать критической секцией. [4]

## 4.4 Логирование

Для системы необходимо логирование, при чем так как данный продукт является первородным, то не существует аналогичного примера, который смог бы стать основой проекта. В связи с этим необходимо логирования всех действий пользователя и реакции сервера на них.

Сначала было принято делать логирование в файлы, для этого существуют хорошие библиотеки, что позволяет с минимальными затратами настроить логирование проекта.

Проблема данного подхода в том, что поиск по логам пришлось бы делать самостоятельно, да и хранение в файлах не самое лучшее.

В связи с этим было предусмотрено решение логировать данные в базе данных, удобно сделать это в той же базе, где хранятся данные пользователя.

При таком подходе можно обращаться к логам через обычные запросы к БД – сортировать по времени, отбирать логи от определенного пользователя и т.д. с минимальными усилиями.

# 5 Перспективы развития

## 5.1 Идея

Во время автоматизации пересылки сообщений из одного канала в другой было выделено три этапа обработки сообщения:

1. Получение нового сообщения;
2. Обработка сообщения;
3. Отправка сообщения.

Каждый из этих этапов проходит последовательно, а главное оперирует с содержимым сообщения, и использует Telegramm как инструмент для получения и отправки сообщений. Соответственно можно отделить получение, обработку и отправку сообщения, что позволит считывать сообщения из одного источника, обрабатывать сообщение и отправлять сообщение в абсолютно другой информационный источник.

## 5.2 Архитектура

Идейно архитектура сервера будет заключатся в разделении системы на модули:

* Модуль приема сообщений;
* Модуль обработки сообщений;
* Модуль отправки сообщений.

Связи модулей изображены на рисунке 5.1.

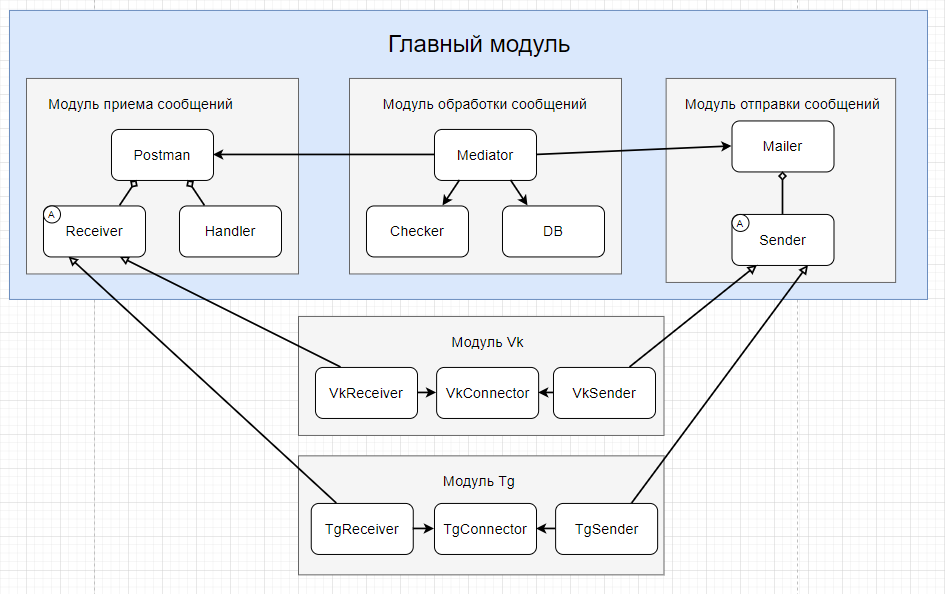


Рисунок 5.1 – Архитектура системы

## 5.2.1 Сервер

На сервере реализуется вся логика проекта, хранится информация в базе данных, происходит обмен информацией с клиентом (сайтом), принимаются и обрабатываются запросы с клиента.

## 5.2.2 Сайт

На сайте реализуется пользовательский интерфейс для администраторов системы. Для управления пользователь должен пройти авторизацию, после чего у него будет возможность добавлять, удалять, просматривать связи между источниками информации. Для администраторов будет возможность добавлять, удалять, просматривать пользователей системы.

# Выводы

Результат работы – система, реализующая агрегатор для пересылки сообщений из одних каналов в другие в мессенджере Telegram, с возможностью поиска сообщений по ключевым фразам, реализованная при помощи языка программирования С++.

К преимуществам программы относятся: многоуровневый пользовательский интерфейс, обеспечивающий интуитивно понятную навигацию по меню команд, ведение диалога с пользователем при выполнении команд, стабильная работа системы, возможность добавлять, удалять, просматривать связи между каналами и их ключевыми фразами, возможность для администраторов добавлять, удалять, просматривать пользователей системы, наличие групповой рассылки для администраторов, защита системы от несанкционированного использования, возможность искать сообщения в каналах по указанным ключевым словам или фразам.

При дальнейшей работе над системой можно масштабировать ее до более крупных размеров, а именно добавить возможность работать не только с Telegram, а и с другими информационными источниками.

# Литература

1. Telegram – Википедия [Электронный ресурс] // Telegram. – Режим доступа: <https://ru.wikipedia.org/wiki/Telegram>.
2. TDLib – GitHub [Электронный ресурс] // Telegram API. – Режим доступа: <https://github.com/tdlib/td>.
3. TgBot – GitHub [Электронный ресурс] // Бот. – Режим доступа: <https://github.com/reo7sp/tgbot-cpp>.
4. PQXX – GitHub [Электронный ресурс] // База данных. – Режим доступа: <https://github.com/jtv/libpqxx>.

# Приложение А. Техническое задание (сервер)

1

2

3

4

5

6

7

8

9

10

# Приложение Б. Техническое задание (клиент)

1

2

3

4

5

6

7

8

9

# Приложение В. Экранные формы

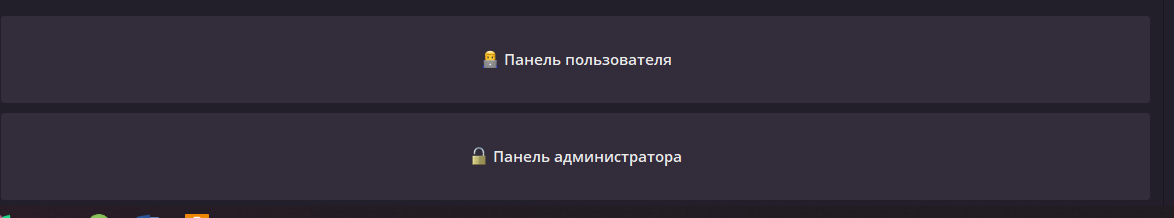


Рисунок Б.1 – Начальное меню

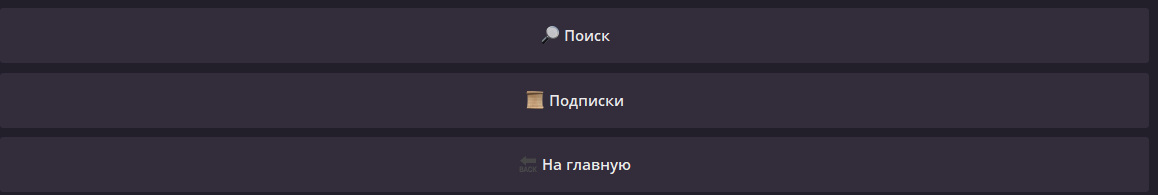


Рисунок Б.2 – Панель пользователя

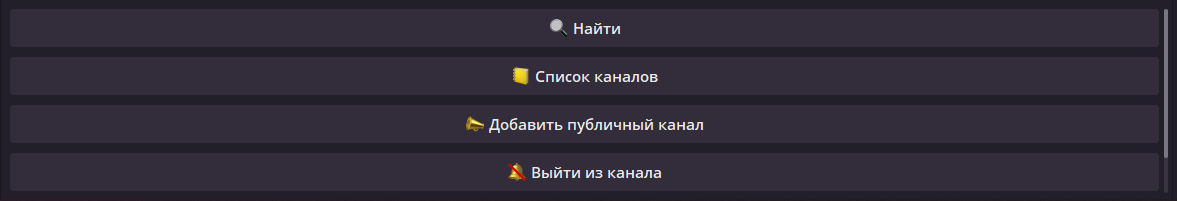


Рисунок Б.3 – Меню «Поиск»

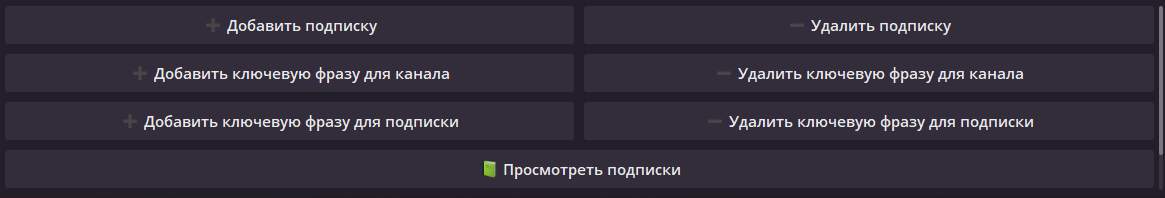


Рисунок Б.4 – Меню «Подписки»

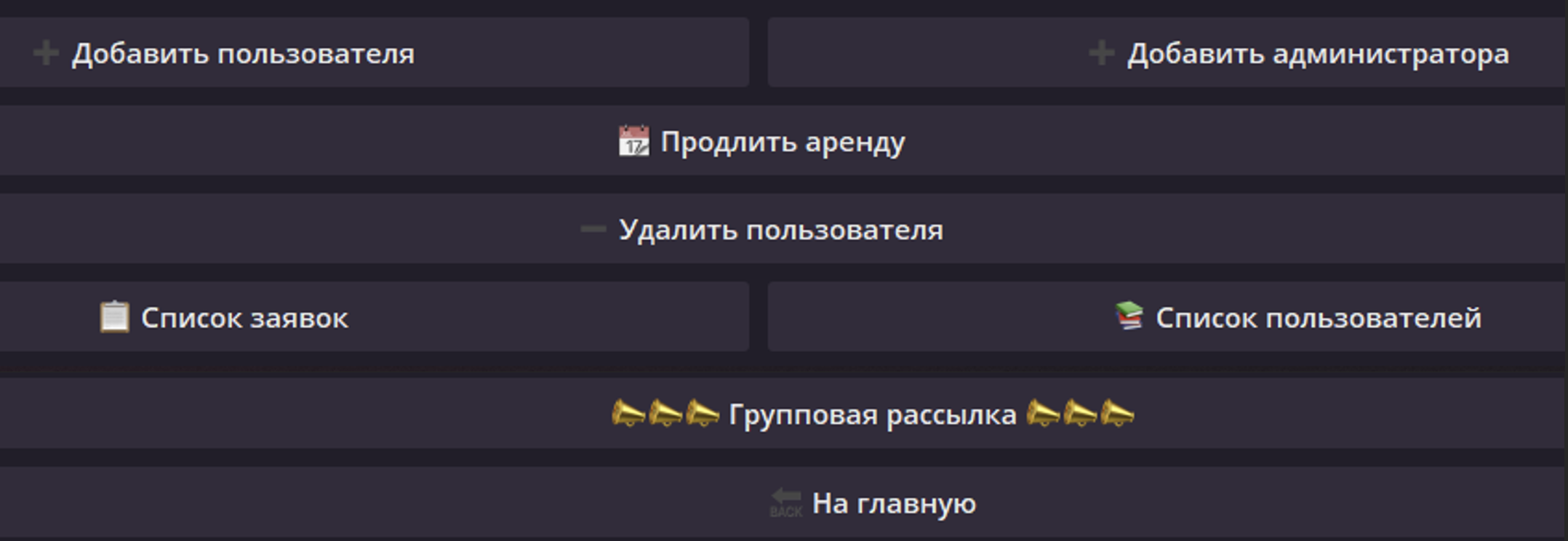


Рисунок Б.5 – Панель администратора

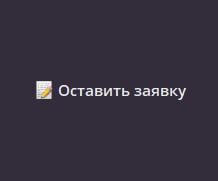


Рисунок Б.6 – Оставить заявку для нового пользователя

# Приложение Г. Статистика приложения

Приложение функционирует более 3х месяцев, за все время было обнаружено несколько критических ошибок, то есть изначально система работала нестабильно, но после проведения нескольких рефакторингов, удалось урегулировать стабильность системы.

Сейчас приложение обслуживает 10 каналов в телеграмме, при этом получая информацию с более 100 каналов, при этом в день обрабатывается несколько тысяч сообщений, и поддерживается полная история взаимодействия пользователей с системой. При этом сервер обладает минимальной конфигурацией, что позволило сэкономить затраты на старте проекта.

Также в приложении присутствует 7 пользователей (4 из которых – администраторы).

Без перебойно приложение работает более 2х месяцев.

Приблизительно, за все время было опубликовано 3 \* 30 \* 1000 = 90000 сообщений.

Переработано сообщений еще больше, что показывает полезность данного проекта, ведь стоимость данной работы в человеко-часах будет оцениваться в несколько лет, а может и десятки лет, а приложение всего за 3 месяца на 10 каналах смогло показать такие результаты.

Приложение показало свою эффективность с точки зрения статистики и далее будет только улучшать данные показатели.

# Приложение Д. Руководство пользователя

Для начала использования необходимо подать заявку на использования системы при помощи Telegram-бота @test\_silur\_bot, а также продлить аренду использования бота. После того, как заявка принята открывается доступ к системе.

Для того чтобы найти сообщение в определенном сообществе необходимо зайти в подменю «Поиск». Перед поиском необходимо добавить канал в список каналов для поиска при помощи пункта «Добавить публичный канал». После того, как требуемый канал добавлен в список каналов нужно нажать на кнопку «Найти». После этого нужно ввести ключевую фразу, по которой будет осуществляться поиск. Результаты поиска выводятся в сообщении с постраничной ориентацией. Для перехода между результатами поиска необходимо нажать на кнопки с цифрами под сообщением. Для удаления канала из списка каналов нужно нажать кнопку «Выйти из канала», после чего ввести название канала. Для просмотра списка каналов нужно нажать на «Список каналов». Для выхода в главное меню нужно нажать «Выбор функции».

Для того чтобы создать подписку нужно перейти в подменю «Подписки». Для добавления подписки нужно нажать на кнопку «Добавить подписку», ввести какой канал слушать, ввести в какой канал публиковать.

Удаление подписки происходит аналогичным образом, как и добавление, для этого существует команда «Удалить подписку».

Для работы с ключевыми фразами существуют следующие команды: «Добавить ключевую фразу для канала», «Удалить ключевую фразу для канала», «Добавить ключевую фразу для подписки», «Удалить ключевую фразу для подписки». Все команды с ключевыми фразами требуют информации о канале (каналах), а также ключевую фразу для добавления или удаления.

# Приложение Е. Руководство администратора

Панель пользователя была рассмотрена в руководстве пользователя (Приложение Д).

Для использования функций администрирования нужно нажать «Панель администратора».

Для добавления пользователя нужно нажать «Добавить пользователя», после чего ввести ID пользователя. Для добавления администратора нужно нажать «Добавить администратора», после чего ввести ID пользователя. Для продления аренды пользователю нужно нажать «Продлить аренду», ввести ID пользователя и срок, на который будет продлена аренда. Для удаления пользователя нужно нажать на «Удалить пользователя» и ввести ID пользователя. Для просмотра списка заявок нужно нажать «Список заявок». Для просмотра списка пользователей нужно нажать на «Список пользователей». Для осуществления групповой рассылки нужно нажать «Групповая рассылка», после чего ввести сообщение, которое будет отослано всем пользователям.

# Приложение Ж. Код приложения

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 28/06/2021.

//

#pragma once

#include <tgbot/tgbot.h>

#include <Bot.hpp>

#include <DatabaseHelper.hpp>

#include <TelegramSearcher.hpp>

#include <functional>

#include <list>

class SearcherBot {

using MessagePtr = TgBot::Message::Ptr;

using ChatID = std::int64\_t;

using UserID = std::int32\_t;

using Step = std::function<void(const MessagePtr&)>;

public:

SearcherBot();

private:

TelegramSearcher engine\_;

bot::Bot bot\_;

static constexpr auto token =

"1444180840:AAGOGiwsL\_h9cRFkmbvSTwtvRg3MOAuIfl4";

static constexpr auto test\_token =

"1875664122:AAHGHJ0RgYBY7zLWlY0YmLD9TFUx-ZH4Imk";

db::DatabaseHelper database\_;

std::unordered\_map<std::string, Step> admin\_commands\_;

std::unordered\_map<std::string, Step> user\_commands\_;

static constexpr size\_t BUTTONS\_COUNT = 5;

private:

// Helpers

void InitDatabase() noexcept;

void InitBot();

void StartLongPoll();

void SetCommands();

void InsertUser(UserID telegram\_user\_id, db::User::ROLE role,

const std::string& first\_name, const std::string& second\_name,

const std::string& username);

bool CreatingFirstAdmin(const MessagePtr& message);

void SubmitYourApplication(const MessagePtr& message);

void NotifyAdmins(const db::Application& application);

static bool InvokeCommand(

const MessagePtr& message,

const std::unordered\_map<std::string, Step>& commands);

static TgBot::ReplyKeyboardMarkup::Ptr GetStartMarkup(db::User::ROLE role);

bool DeleteUser(UserID telegram\_user\_id);

bool RenewLease(UserID telegram\_user\_id, int32\_t number\_of\_days);

bool AcceptApplication(UserID telegram\_user\_id, db::User::ROLE role);

void InitAdminCommands();

void InitUserCommands();

bool InsertChannelForSearch(UserID telegram\_user\_id, ChatID chat\_id);

std::optional<std::string> ErasePublicChatForSearch(

UserID telegram\_user\_id, const std::string& channel\_username);

std::vector<std::pair<std::string, std::int64\_t>> GetMessagesLinks(

UserID telegram\_user\_id, const std::string& query);

void EditMessage(int64\_t chat\_id, int32\_t message\_id,

const std::string& message,

TgBot::GenericReply::Ptr keyboard);

void SetCurrentPage(TgBot::Message::Ptr message, size\_t page);

std::pair<bool, std::string> InsertSubscribe(UserID telegram\_user\_id,

ChatID observer,

ChatID receiver);

std::string ToString(const db::User& user);

void InitEngine();

void NewMessageHandler(td\_api::updateNewMessage& new\_message);

bool HasAccessToKeyPhrases(UserID telegram\_user\_id, ChatID receiver);

void ShowReceivers(UserID telegram\_user\_id);

TgBot::InputFile::Ptr DocForRole(db::User::ROLE role);

bool IsRentExpired(UserID telegram\_user\_id);

bool IsAdmin(UserID telegram\_user\_id, ChatID chat\_id);

void SendMessage(ChatID chat\_id, const std::string& message,

TgBot::GenericReply::Ptr reply\_markup = nullptr);

//////////////////////////////////////////////////////////////////////////////

// Slots (is similar to slots from Qt)

void on\_non\_command\_message(const MessagePtr& message);

void on\_applications\_list(const MessagePtr& message);

void on\_insert\_admin(const MessagePtr& message);

void on\_insert\_user(const MessagePtr& message);

void on\_users\_list(const MessagePtr& message);

void on\_delete\_user(const MessagePtr& message);

void on\_renew\_lease(const MessagePtr& message);

void on\_back\_to\_start\_markup(const MessagePtr& message);

void on\_channels\_for\_search\_show(const MessagePtr& message);

void on\_leave\_from\_channel(const MessagePtr& message);

void on\_insert\_channel\_for\_search(const MessagePtr& message);

void on\_search\_messages(const MessagePtr& message);

void on\_right\_rotate(const TgBot::CallbackQuery::Ptr& callback\_query);

void on\_left\_rotate(const TgBot::CallbackQuery::Ptr& callback\_query);

void on\_change\_page(const TgBot::CallbackQuery::Ptr& callback\_query);

void on\_insert\_subscribe(const MessagePtr& message);

void on\_delete\_subscribe(const MessagePtr& message);

void on\_show\_subscribes(const MessagePtr& message);

void on\_append\_key\_phrase\_for\_channel(const MessagePtr& message);

void on\_remove\_key\_phrase\_for\_channel(const MessagePtr& message);

void on\_feedback\_command(const MessagePtr& message);

void on\_broadcast(const MessagePtr& message);

void on\_start\_command(const MessagePtr& message);

void on\_help\_command(const MessagePtr& message);

void on\_append\_key\_phrase\_for\_subscribe(const MessagePtr& message);

void on\_remove\_key\_phrase\_for\_subscribe(const MessagePtr& message);

};

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 20/06/2021.

//

#pragma once

#include <string>

#include <utility>

#include <vector>

std::vector<std::string> Split(const std::string& string,

const std::string& separator);

bool Check(const std::string& text,

const std::vector<std::string>& key\_phrases);

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 10/08/2021.

//

#pragma once

#include <tgbot/Bot.h>

#include <functional>

#include <list>

#include <memory>

namespace bot {

using ChatID = std::int64\_t;

using UserID = std::int32\_t;

using Step = std::function<void(const TgBot::Message::Ptr&)>;

class Bot : public TgBot::Bot {

public:

Bot(const std::string& token);

void RegistryNextStep(ChatID chat\_id, Step handler);

/\*\*

\* @brief find next\_step handler and invoke it

\* @param message

\* @return true if next\_step is exists else return false

\*/

bool TryNextStep(const TgBot::Message::Ptr& message);

private:

std::list<std::pair<ChatID, Step>> next\_steps\_;

private:

[[nodiscard]] std::list<std::pair<ChatID, Step>>::const\_iterator FindNextStep(

ChatID chat\_id) noexcept;

[[nodiscard]] bool IsFindNextStep(

std::list<std::pair<ChatID, Step>>::const\_iterator it) const noexcept;

};

} // namespace bot

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 11/08/2021.

//

#pragma once

#include <tgbot/Bot.h>

namespace ui {

TgBot::ReplyKeyboardMarkup::Ptr FirstAdminMarkup();

TgBot::ReplyKeyboardMarkup::Ptr AdminStartMarkup();

TgBot::ReplyKeyboardMarkup::Ptr AdminMarkup();

TgBot::ReplyKeyboardMarkup::Ptr RegistrationMarkup();

TgBot::ReplyKeyboardMarkup::Ptr UserStartMarkup();

TgBot::ReplyKeyboardMarkup::Ptr UserSearchingMarkup();

TgBot::ReplyKeyboardMarkup::Ptr UserStartMarkupForAdmin();

TgBot::ReplyKeyboardMarkup::Ptr UserSubscribeMarkup();

TgBot::InlineKeyboardMarkup::Ptr CreateInlineKeyboardMarkup(size\_t link\_count);

} // namespace ui

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 10/08/2021.

//

#include "Bot.hpp"

#include <algorithm>

namespace bot {

Bot::Bot(const std::string& token) : TgBot::Bot{token} {

// pass

}

void Bot::RegistryNextStep(

ChatID chat\_id, std::function<void(const TgBot::Message::Ptr&)> handler) {

next\_steps\_.emplace\_back(chat\_id, handler);

}

auto Bot::FindNextStep(ChatID chat\_id) noexcept

-> std::list<std::pair<ChatID, Step>>::const\_iterator {

return std::find\_if(next\_steps\_.cbegin(), next\_steps\_.cend(),

[chat\_id](const std::pair<ChatID, Step>& value) {

return value.first == chat\_id;

});

}

bool Bot::IsFindNextStep(

std::list<std::pair<ChatID, Step>>::const\_iterator it) const noexcept {

return it != next\_steps\_.cend();

}

bool Bot::TryNextStep(const TgBot::Message::Ptr& message) {

auto it = FindNextStep(message->chat->id);

bool is\_find = IsFindNextStep(it);

if (is\_find) {

it->second(message);

next\_steps\_.erase(it);

}

return is\_find;

}

} // namespace bot

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 11/08/2021.

//

#include <BotUI.hpp>

namespace {

std::vector<TgBot::KeyboardButton::Ptr> CreateButtonRow(

const std::vector<std::string>& button\_row) {

std::vector<TgBot::KeyboardButton::Ptr> row;

for (auto& button\_text : button\_row) {

auto button = std::make\_shared<TgBot::KeyboardButton>();

button->text = button\_text;

row.push\_back(button);

}

return row;

}

TgBot::ReplyKeyboardMarkup::Ptr CreateKeyboard(

const std::vector<std::vector<std::string>>& buttons) {

auto board = std::make\_shared<TgBot::ReplyKeyboardMarkup>();

for (auto& button\_row : buttons) {

CreateButtonRow(button\_row);

board->keyboard.push\_back(CreateButtonRow(button\_row));

}

return board;

}

TgBot::ReplyKeyboardMarkup::Ptr PushBackLine(

const TgBot::ReplyKeyboardMarkup::Ptr& board,

const std::vector<std::string>& row) {

auto new\_board = std::make\_shared<TgBot::ReplyKeyboardMarkup>(\*board);

new\_board->keyboard.push\_back(CreateButtonRow(row));

return new\_board;

}

} // namespace

namespace ui {

TgBot::ReplyKeyboardMarkup::Ptr RegistrationMarkup() {

return CreateKeyboard({{"📝 Оставить заявку"}});

}

TgBot::ReplyKeyboardMarkup::Ptr FirstAdminMarkup() {

return CreateKeyboard({{"🥇 Добавить меня"}});

}

TgBot::ReplyKeyboardMarkup::Ptr AdminStartMarkup() {

return CreateKeyboard(

{{"👨‍💻 ‍Панель пользователя"},

{"🔓 Панель администратора"}});

}

TgBot::ReplyKeyboardMarkup::Ptr AdminMarkup() {

return CreateKeyboard(

{{"➕ Добавить пользователя", "➕ Добавить администратора"},

{"📆 Продлить аренду"},

{"➖ Удалить пользователя"},

{"📋 Список заявок", "📚 Список пользователей"},

{"📣📣📣 Групповая рассылка 📣📣📣"},

{"🔙 На главную"}});

}

TgBot::ReplyKeyboardMarkup::Ptr UserStartMarkup() {

return CreateKeyboard({{"🔎 Поиск"}, {"📜 Подписки"}});

}

TgBot::ReplyKeyboardMarkup::Ptr UserSearchingMarkup() {

return CreateKeyboard({{"🔍 Найти"},

{"📒 Список каналов"},

{"📣 Добавить публичный канал"},

{"🔕 Выйти из канала"},

{"🔙 Выбор функции"}});

}

TgBot::ReplyKeyboardMarkup::Ptr UserStartMarkupForAdmin() {

auto user\_start\_markup = UserStartMarkup();

return PushBackLine(user\_start\_markup, {"🔙 На главную"});

}

TgBot::ReplyKeyboardMarkup::Ptr UserSubscribeMarkup() {

return CreateKeyboard({{"➕ Добавить подписку", "➖ Удалить подписку"},

{"➕ Добавить ключевую фразу для канала",

"➖ Удалить ключевую фразу для канала"},

{"➕ Добавить ключевую фразу для подписки",

"➖ Удалить ключевую фразу для подписки"},

{"📗 Просмотреть подписки"},

{"🔙 Выбор функции"}});

}

TgBot::InlineKeyboardMarkup::Ptr CreateInlineKeyboardMarkup(size\_t link\_count) {

// Create beloved markup

// [1] [2] [3] [4] [5]

// [<-] [->]

auto keyboard = std::make\_shared<TgBot::InlineKeyboardMarkup>();

constexpr size\_t BUTTONS\_COUNT{5};

// Create [1] [2] [3] [4] [5] or [1] [2] [3] [4] or ... [1] or empty

std::vector<TgBot::InlineKeyboardButton::Ptr> row0;

for (size\_t i = 0; i < std::min(link\_count, BUTTONS\_COUNT); ++i) {

auto button = std::make\_shared<TgBot::InlineKeyboardButton>();

button->text = std::to\_string(i + 1);

button->callbackData = "page\_" + std::to\_string(i);

row0.push\_back(std::move(button));

}

keyboard->inlineKeyboard.push\_back(std::move(row0));

if (link\_count <= BUTTONS\_COUNT) {

return keyboard;

}

// Create [<-] [->]

std::vector<TgBot::InlineKeyboardButton::Ptr> row1;

auto left = std::make\_shared<TgBot::InlineKeyboardButton>();

left->text = "👈";

left->callbackData = "left";

row1.push\_back(std::move(left));

auto right = std::make\_shared<TgBot::InlineKeyboardButton>();

right->text = "👉";

right->callbackData = "right";

row1.push\_back(std::move(right));

keyboard->inlineKeyboard.push\_back(std::move(row1));

return keyboard;

}

} // namespace ui

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 30/07/2021.

//

#include <gtest/gtest.h>

#include <Bot.hpp>

TEST(A, B) {

EXPECT\_EQ(42, 42);

}

cmake\_minimum\_required(VERSION 3.14)

project(bot\_test)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES

${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.cpp

${CMAKE\_CURRENT\_SOURCE\_DIR}/../src/\*.cpp)

add\_executable(${PROJECT\_NAME} ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PRIVATE ${CMAKE\_CURRENT\_SOURCE\_DIR}/../include)

# dependencies

# GTest

find\_package(GTest REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE GTest::GTest GTest::Main)

# TgBot

if (CURL\_FOUND)

include\_directories(${CURL\_INCLUDE\_DIRS})

add\_definitions(-DHAVE\_CURL)

endif()

set(Boost\_USE\_MULTITHREADED ON)

find\_package(Td REQUIRED)

find\_package(Threads REQUIRED)

find\_package(OpenSSL REQUIRED)

find\_package(Boost COMPONENTS system REQUIRED)

find\_package(CURL)

target\_include\_directories(${PROJECT\_NAME} PRIVATE /usr/local/include

${OpenSSL\_INCLUDE\_DIR} ${Boost\_INCLUDE\_DIR}

)

if (CURL\_FOUND)

include\_directories(${CURL\_INCLUDE\_DIRS})

add\_definitions(-DHAVE\_CURL)

endif()

find\_library(TGBOT\_LIBRARY

NAMES TgBot)

target\_link\_libraries(${PROJECT\_NAME} PUBLIC

${TGBOT\_LIBRARY}

${CMAKE\_THREAD\_LIBS\_INIT}

OpenSSL::SSL

${Boost\_LIBRARIES}

${CURL\_LIBRARIES})

cmake\_minimum\_required(VERSION 3.14)

project(bot)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES ${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp)

add\_library(${PROJECT\_NAME} STATIC ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR}/include)

# dependencies

# TgBot

if (CURL\_FOUND)

include\_directories(${CURL\_INCLUDE\_DIRS})

add\_definitions(-DHAVE\_CURL)

endif()

set(Boost\_USE\_MULTITHREADED ON)

find\_package(Threads REQUIRED)

find\_package(OpenSSL REQUIRED)

find\_package(Boost COMPONENTS system REQUIRED)

find\_package(CURL)

target\_include\_directories(${PROJECT\_NAME} PUBLIC

${OpenSSL\_INCLUDE\_DIR} ${Boost\_INCLUDE\_DIR}

)

if (CURL\_FOUND)

include\_directories(${CURL\_INCLUDE\_DIRS})

add\_definitions(-DHAVE\_CURL)

endif()

find\_library(TGBOT\_LIBRARY

NAMES TgBot)

target\_link\_libraries(${PROJECT\_NAME} PUBLIC

${TGBOT\_LIBRARY}

OpenSSL::SSL

${CMAKE\_THREAD\_LIBS\_INIT}

${Boost\_LIBRARIES}

${CURL\_LIBRARIES})

add\_subdirectory(test)

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 30/07/2021.

//

#pragma once

#include <memory>

#include <string>

#include <optional>

#include <vector>

namespace db {

struct ConnectionParams {

std::string host;

std::string user;

std::string password;

std::string port;

};

using SERIAL = std::int64\_t;

using UserID = std::int32\_t;

using ChatID = std::int64\_t;

struct User {

UserID user\_id;

std::string name;

std::string surname;

std::string username;

bool is\_info\_valid;

bool is\_admin;

std::string rent\_to; // std::chrono don't support by default

enum class ROLE { USER, ADMIN, NONE, UNREGISTER };

};

struct Application {

UserID user\_id;

std::string name;

std::string surname;

std::string username;

std::string submit\_date;

};

struct Subscribe {

SERIAL id;

UserID user\_id;

ChatID observer;

ChatID receiver;

std::vector<std::string> key\_phrases;

};

struct Channel {

ChatID channel\_id;

std::vector<std::string> key\_phrases;

};

// Admin equal user by struct within database

using Admin = User;

/\*\*

\* @brief for CreateLog

\*/

enum class MESSAGE\_TYPE { USER\_MESSAGE, BOT\_MESSAGE };

/\*\*

\* @brief helper which provide struct date via database api such as pqxx.

\* @property noncopyable

\* @property movable

\* @property thread-unsafe but you may create several equals connection for uses

\* from difference thread. Use thread\_local instance.

\*/

class DatabaseHelper {

public:

DatabaseHelper(ConnectionParams params);

DatabaseHelper();

DatabaseHelper(DatabaseHelper&&);

DatabaseHelper& operator=(DatabaseHelper&&);

~DatabaseHelper();

DatabaseHelper(const DatabaseHelper&) = delete;

DatabaseHelper& operator=(const DatabaseHelper&) = delete;

/\*\*

\* @brief open connection of database

\* @return true if the connection have already is opened or reopening

\* connection is success else return false

\*/

bool Open(ConnectionParams connection\_params);

/\*\*

\* @brief After construct instance connection maybe either open or close

\* @return true if connection is open else return false

\*/

[[nodiscard]] bool IsOpen() const noexcept;

//////////////////////////////////////////////////////////////////////////////

// Users' table handlers

/\*\*

\* @brief change rent\_to parameter into database

\* @param user\_id

\* @param number\_of\_days maybe both positive and negative number

\* @return new date

\*/

std::string RenewLeast(UserID user\_id, int32\_t number\_of\_days);

/\*\*

\* @brief get role of user from database

\* @param telegram\_user\_id

\* @return role of user

\*/

[[nodiscard]] User::ROLE GetRole(UserID telegram\_user\_id);

/\*\*

\* @brief

\* @return true if table of users is empty else false

\*/

bool IsFirstUser();

/\*\*

\* @brief Check user is admin or not

\* @param user\_id

\* @return true is user is admin else return false

\*/

[[nodiscard]] bool IsAdmin(UserID user\_id);

/\*\*

\* @brief check user is none or not

\* @param user\_id

\* @return false if this is user else return false

\*/

[[nodiscard]] bool IsNone(UserID user\_id);

/\*\*

\* @brief insert user to database

\* @param user struct which describes user's state

\*/

void InsertUser(const User& user);

/\*\*

\* @brief delete user from database by user\_id == telegram\_user\_id

\* @param user\_id is telegram\_user\_id

\* @return true if user have deleted else return false

\*/

bool DeleteUser(UserID user\_id);

/\*\*

\* @brief get admins from database

\* @return vector of admins

\*/

[[nodiscard]] std::vector<Admin> Admins();

/\*\*

\* @brief get users and admins (all users) from database

\* @return vector of both users and admins

\*/

std::vector<User> Users();

/\*\*

\* @brief check if the user has access

\* @param user\_id

\* @return true if rent is expired else return false

\*/

bool IsRentExpired(UserID user\_id);

//////////////////////////////////////////////////////////////////////////////

// Applications' table handlers

/\*\*

\* @brief

\* @param application are both in and out parameter. All without submit\_date

\* are in parameters but submit\_date is out parameter

\* @example

\* @code

\* Application app;

\* app.id = 554123412;

\* app.name = "Kirill";

\* app.surname = "Danilchuk";

\* app.username = "KirillDanilchuk";

\* SubmitApplication(app);

\* Log("Submit user at " + app.submit\_date);

\* NotifyAdmins(app);

\* @endcode

\*/

void SubmitApplication(Application& application);

/\*\*

\* @brief get application from database

\* @return vector of applications

\*/

[[nodiscard]] std::vector<Application> Applications();

/\*\*

\* @brief delete application from database

\* @param user\_id

\* @return true if application is exists else return false

\*/

std::optional<Application> DeleteApplication(UserID user\_id);

/\*\*

\* @brief check user introduce application or not

\* @param user\_id

\* @return true if user application is exists

\*/

[[nodiscard]] bool IsUnregister(UserID user\_id);

//////////////////////////////////////////////////////////////////////////////

// relations\_of\_channel's table handler

/\*\*

\* @brief get relation\_of\_channel\_id from database

\* @param user\_id

\* @param observer

\* @param receiver

\* @return id of relation if it exists else return -1;

\*/

SERIAL GetRelationId(UserID user\_id, ChatID observer, ChatID receiver);

/\*\*

\* @brief Check user create a relations between any observer and a receiver

\* @param user\_id

\* @param receiver\_id

\* @return true if user create a relations else return false

\*/

[[nodiscard]] bool ExistsReceiverFromUser(UserID user\_id, ChatID receiver\_id);

/\*\*

\* @brief insert into relation\_of\_channel table

\* @param user\_id is telegram\_user\_id

\* @param observer is chat\_id of observer

\* @param receiver is chat\_id of receiver

\*/

bool CreateSubscribe(UserID user\_id, ChatID observer, ChatID receiver);

/\*\*

\* @brief delete from relation\_of\_channel table

\* @param user\_id is telegram\_user\_id

\* @param observer is chat\_id of observer

\* @param receiver is chat\_id of receiver

\*/

bool DeleteSubscribe(UserID user\_id, ChatID observer, ChatID receiver);

/\*\*

\* @brief get array of receivers from database

\* @param observer is chat\_id of observer

\* @return vector of subscribes

\*/

std::vector<Subscribe> Subscribes(ChatID observer);

/\*\*

\* @brief get array of receivers of user from database

\* @param user\_id is user\_id of user

\* @return vector of receivers

\*/

std::vector<ChatID> ReceiversOfUser(UserID user\_id);

/\*\*

\* @brief get array of subscribes of receiver from database

\* @param receiver

\* @return vector of subscribes

\*/

std::vector<Subscribe> SubscribesOfReceiver(ChatID receiver);

/\*\*

\* @brief Check is relation exist

\* @param user\_id

\* @param observer

\* @param receiver

\* @return true if exists else return false

\*/

bool IsRelationExist(UserID user\_id, ChatID observer, ChatID receiver);

/\*\*

\* @brief append key phrases for one subscribe

\* @param id is relations\_of\_channel\_id from database

\* @param key\_phrase any text

\*/

void AppendKeyPhraseForSubscribe(SERIAL id, const std::string& key\_phrase);

/\*\*

\* @brief remove key phrases for one subscribe

\* @param id is relations\_of\_channel\_id from database

\* @param key\_phrase any text

\*/

void RemoveKeyPhraseForSubscribe(SERIAL id, const std::string& key\_phrase);

//////////////////////////////////////////////////////////////////////////////

// channel's table handler

/\*\*

\* @brief insert channel into database without error

\* @param chat\_id

\* @return true if channels doesn't exist else return false

\*/

bool InsertChannel(ChatID chat\_id);

/\*\*

\* @brief append key phrase for channel

\* @param channel

\* @param key\_phrase

\*/

void AppendKeyPhraseForChannel(ChatID channel, const std::string& key\_phrase);

/\*\*

\* @brief remove key phrase from channel

\* @param channel

\* @param key\_phrase

\*/

void RemoveKeyPhraseFromChannel(ChatID channel,

const std::string& key\_phrase);

/\*\*

\* @brief select channel by id from database

\* @param channel\_id

\* @return into of channel

\*/

Channel GetChannel(ChatID channel\_id);

/\*\*

\* @brief delete channel from database

\* @param channel\_id

\* @return true if channel is existed else return false

\*/

bool DeleteChannel(ChatID channel\_id);

//////////////////////////////////////////////////////////////////////////////

// chats\_for\_search's table handler

/\*\*

\* @brief insert the chat for search for the user

\* @param user\_id

\* @param chat\_id

\* @return true if success else return false

\*/

bool InsertChatForSearch(UserID user\_id, ChatID chat\_id);

/\*\*

\* @brief delete the chat for search for the user

\* @param user\_id

\* @param chat\_id

\* @return true if the chat for search for the user exists else return false

\*/

bool DeleteChatForSearch(UserID user\_id, ChatID chat\_id);

/\*\*

\* @brief

\* @param user\_id

\* @return

\*/

std::vector<ChatID> GetChannelsForSearch(UserID user\_id);

//////////////////////////////////////////////////////////////////////////////

// logs\_of\_messages's table

/\*\*

\* @brief create log

\* @param user\_id

\* @param message\_type

\* @param message

\*/

void CreateLog(UserID user\_id, MESSAGE\_TYPE message\_type,

const std::string& message);

private:

// https://en.cppreference.com/w/cpp/language/pimpl

class Impl;

std::unique\_ptr<Impl> pimpl\_;

// Helpers

DatabaseHelper(std::string\_view host, std::string\_view user,

std::string\_view password, std::string\_view port);

};

} // namespace db

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 30/07/2021.

//

#include "DatabaseHelper.hpp"

#include <cassert>

#include <mutex>

#include <pqxx/pqxx>

namespace {

std::string MakeConnectionString(std::string\_view host, std::string\_view user,

std::string\_view password,

std::string\_view port) {

std::string result;

result.append("host=").append(host).append(" ");

result.append("user=").append(user).append(" ");

result.append("password=").append(password).append(" ");

result.append("port=").append(port);

return result;

}

template <typename T>

T GetScalar(const pqxx::result& result) {

assert(result.size() == 1 && "Number of rows aren't one");

assert(result.front().size() == 1 &&

"Number of members of a tuple aren't one");

return result.front().front().as<T>();

}

std::vector<db::User> VectorOfUsers(pqxx::result result) {

std::vector<db::User> users;

users.reserve(result.size());

db::User user;

for (pqxx::row db\_user : result) {

user.user\_id = db\_user["telegram\_user\_id"].as<db::UserID>();

user.name = db\_user["name"].c\_str();

user.surname = db\_user["surname"].c\_str();

user.username = db\_user["username"].c\_str();

user.is\_info\_valid = db\_user["is\_info\_valid"].as<bool>();

user.is\_admin = db\_user["is\_admin"].as<bool>();

user.rent\_to = db\_user["rent\_to"].c\_str();

users.push\_back(std::move(user));

}

return users;

}

std::vector<std::string> GetArray(pqxx::field postgresql\_field) {

std::vector<std::string> result;

pqxx::array\_parser array\_parser = postgresql\_field.as\_array();

std::pair<pqxx::array\_parser::juncture, std::string> element;

do {

element = array\_parser.get\_next();

if (element.first == pqxx::array\_parser::juncture::string\_value) {

result.push\_back(std::move(element.second));

}

} while (element.first != pqxx::array\_parser::juncture::done);

return result;

}

std::vector<db::Subscribe> VectorOfSubscribes(pqxx::result db\_subscribes) {

std::vector<db::Subscribe> subscribes;

db::Subscribe subscribe; // temporary value for subscribes

for (pqxx::row db\_subscribe : db\_subscribes) {

subscribe.id = db\_subscribe["relations\_of\_channel\_id"].as<db::SERIAL>();

subscribe.user\_id = db\_subscribe["telegram\_user\_id"].as<db::UserID>();

subscribe.observer = db\_subscribe["observer"].as<db::ChatID>();

subscribe.receiver = db\_subscribe["receiver"].as<db::ChatID>();

subscribe.key\_phrases = GetArray(db\_subscribe["key\_phrases"]);

subscribes.push\_back(std::move(subscribe));

}

return subscribes;

}

} // namespace

namespace db {

class DatabaseHelper::Impl {

public:

void PrepareQueries();

pqxx::result Exec(const std::string& query);

template <typename... Args>

pqxx::result ExecPrepared(const std::string& statement, Args&&... args) {

std::lock\_guard guard{mutex\_};

pqxx::nontransaction work{\*connection\_};

return work.exec\_prepared(statement, std::forward<Args>(args)...);

}

std::unique\_ptr<pqxx::connection> connection\_;

bool is\_open\_{false};

Impl() = default;

private:

std::mutex mutex\_;

};

DatabaseHelper& DatabaseHelper::operator=(DatabaseHelper&&) = default;

DatabaseHelper::DatabaseHelper(DatabaseHelper&&) = default;

DatabaseHelper::DatabaseHelper() = default;

DatabaseHelper::~DatabaseHelper() = default;

DatabaseHelper::DatabaseHelper(std::string\_view host, std::string\_view user,

std::string\_view password,

std::string\_view port) {

try {

pimpl\_ = std::make\_unique<Impl>();

pimpl\_->connection\_ = std::make\_unique<pqxx::connection>(

MakeConnectionString(host, user, password, port));

pimpl\_->PrepareQueries();

pimpl\_->is\_open\_ = true;

} catch (...) {

pimpl\_->is\_open\_ = false;

}

}

DatabaseHelper::DatabaseHelper(ConnectionParams params)

: DatabaseHelper(params.host, params.user, params.password, params.port) {

// pass

}

void DatabaseHelper::Impl::PrepareQueries() {

//////////////////////////////////////////////////////////////////////////////

// Users' table

connection\_->prepare(

"is\_none",

"SELECT NOT EXISTS(SELECT \* FROM users where telegram\_user\_id = $1)");

connection\_->prepare("is\_first\_user",

"SELECT NOT EXISTS (SELECT \* FROM users)");

connection\_->prepare("is\_admin",

"SELECT EXISTS(SELECT \* FROM users WHERE "

"telegram\_user\_id = $1 AND is\_admin = true)");

connection\_->prepare(

"insert\_user", "INSERT INTO users VALUES($1, $2, $3, $4, $5, $6, now())");

connection\_->prepare("select\_admins",

"SELECT \* FROM users where is\_admin = TRUE");

connection\_->prepare(

"delete\_user",

"delete from users where telegram\_user\_id = $1 RETURNING \*");

connection\_->prepare(

"select\_users\_with\_ordering",

"SELECT \* FROM users ORDER BY is\_admin DESC, surname, name");

connection\_->prepare("is\_rent\_expired",

"SELECT now() < rent\_to OR is\_admin = true FROM users "

"where telegram\_user\_id = $1");

//////////////////////////////////////////////////////////////////////////////

// Application's table

connection\_->prepare(

"submit\_application",

"INSERT INTO applications VALUES($1, $2, $3, $4, now()) RETURNING \*");

connection\_->prepare("select\_applications",

"select telegram\_user\_id, name, surname, username, "

"submit\_date "

"from applications");

connection\_->prepare("is\_unregister",

"SELECT EXISTS(SELECT \* FROM applications WHERE "

"telegram\_user\_id = $1)");

connection\_->prepare(

"delete\_application",

"delete FROM applications WHERE telegram\_user\_id = $1 RETURNING \*");

//////////////////////////////////////////////////////////////////////////////

// relations\_of\_channel's table

connection\_->prepare(

"get\_relation\_id",

"SELECT relations\_of\_channel\_id FROM relations\_of\_channel "

"WHERE telegram\_user\_id = $1 AND observer = $2 AND receiver = $3");

connection\_->prepare("exists\_receiver\_from\_user",

"select exists(select \* from relations\_of\_channel where "

"telegram\_user\_id = $1 and receiver = $2)");

connection\_->prepare(

"insert\_relations\_of\_channel",

"INSERT INTO relations\_of\_channel VALUES(default, $1, $2, $3)");

connection\_->prepare(

"delete\_subscribe",

"DELETE FROM relations\_of\_channel WHERE telegram\_user\_id = $1 AND "

"observer = $2 AND receiver = $3");

connection\_->prepare("select\_receivers",

"SELECT \* FROM relations\_of\_channel WHERE observer = "

"$1");

connection\_->prepare("select\_receiver\_distinct",

"SELECT DISTINCT receiver FROM relations\_of\_channel "

"WHERE telegram\_user\_id = $1");

connection\_->prepare("select\_observers\_for\_receiver",

"SELECT \* FROM relations\_of\_channel WHERE receiver = "

"$1");

connection\_->prepare("check\_relations",

"SELECT EXISTS(SELECT \* FROM relations\_of\_channel WHERE "

"telegram\_user\_id = $1 AND "

"observer = $2 AND receiver = $3)");

connection\_->prepare(

"append\_key\_phrase\_for\_subscribe",

"UPDATE relations\_of\_channel SET key\_phrases = "

"array\_append(key\_phrases, $2) WHERE relations\_of\_channel\_id = $1");

connection\_->prepare(

"remove\_key\_phrase\_for\_subscribe",

"UPDATE relations\_of\_channel SET key\_phrases = "

"array\_remove(key\_phrases, $2) WHERE relations\_of\_channel\_id = $1");

//////////////////////////////////////////////////////////////////////////////

// channel's table

connection\_->prepare(

"insert\_channel\_without\_error",

"INSERT INTO channels VALUES($1) ON CONFLICT DO NOTHING");

connection\_->prepare("append\_key\_phrases\_for\_channel",

"update channels set key\_phrases = "

"array\_append(key\_phrases, $2) where chat\_id = $1");

connection\_->prepare("remove\_key\_phrases\_from\_channel",

"update channels set key\_phrases = "

"array\_remove(key\_phrases, $2) where chat\_id = $1");

connection\_->prepare("select\_channel",

"SELECT \* FROM channels where chat\_id = $1");

connection\_->prepare("delete\_channel",

"DELETE FROM channels WHERE chat\_id = $1");

//////////////////////////////////////////////////////////////////////////////

// chats\_for\_search's table

connection\_->prepare("insert\_chats\_for\_search",

"INSERT INTO chats\_for\_search VALUES($1, $2)");

connection\_->prepare("delete\_from\_channels\_for\_search",

"DELETE FROM chats\_for\_search WHERE telegram\_user\_id = "

"$1 AND chat\_id = $2 RETURNING \*");

connection\_->prepare(

"select\_channels\_for\_search",

"SELECT chat\_id FROM chats\_for\_search WHERE telegram\_user\_id = $1");

//////////////////////////////////////////////////////////////////////////////

// logs\_of\_messages's table

connection\_->prepare(

"create\_log", "INSERT INTO logs\_of\_messages VALUES($1, $2, $3, now())");

}

pqxx::result DatabaseHelper::Impl::Exec(const std::string& query) {

std::lock\_guard guard{mutex\_};

pqxx::nontransaction work{\*connection\_};

return work.exec(query);

}

bool DatabaseHelper::IsOpen() const noexcept {

return pimpl\_ != nullptr && pimpl\_->is\_open\_;

}

////////////////////////////////////////////////////////////////////////////////

// Users' table handler

bool DatabaseHelper::IsNone(UserID user\_id) {

return GetScalar<bool>(pimpl\_->ExecPrepared("is\_none", user\_id));

}

std::string DatabaseHelper::RenewLeast(UserID user\_id, int32\_t number\_of\_days) {

pqxx::result result = pimpl\_->Exec(

"update users set rent\_to = greatest(rent\_to, now()::timestamp) + \'" +

std::to\_string(number\_of\_days) +

" "

"days\' where telegram\_user\_id = " +

std::to\_string(user\_id) +

" RETURNING to\_char(rent\_to, 'DD.MM.YYYY HH24:MM:SS')");

return GetScalar<std::string>(result);

}

User::ROLE DatabaseHelper::GetRole(UserID telegram\_user\_id) {

if (IsUnregister(telegram\_user\_id)) {

return User::ROLE::UNREGISTER;

}

if (IsNone(telegram\_user\_id)) {

return User::ROLE::NONE;

}

if (IsAdmin(telegram\_user\_id)) {

return User::ROLE::ADMIN;

} else {

return User::ROLE::USER;

}

}

bool DatabaseHelper::IsFirstUser() {

pqxx::result db\_is\_first\_user = pimpl\_->ExecPrepared("is\_first\_user");

return GetScalar<bool>(db\_is\_first\_user);

}

bool DatabaseHelper::IsAdmin(UserID user\_id) {

pqxx::result result = pimpl\_->ExecPrepared("is\_admin", user\_id);

return GetScalar<bool>(result);

}

void DatabaseHelper::InsertUser(const User& user) {

pimpl\_->ExecPrepared("insert\_user", user.user\_id, user.name, user.surname,

user.username, user.is\_info\_valid, user.is\_admin);

}

bool DatabaseHelper::DeleteUser(UserID user\_id) {

pqxx::result result = pimpl\_->ExecPrepared("delete\_user", user\_id);

return !result.empty();

}

std::vector<Admin> DatabaseHelper::Admins() {

pqxx::result db\_admins = pimpl\_->ExecPrepared("select\_admins");

std::vector<Admin> admins = VectorOfUsers(db\_admins);

return admins;

}

std::vector<User> DatabaseHelper::Users() {

pqxx::result db\_users = pimpl\_->ExecPrepared("select\_users\_with\_ordering");

return VectorOfUsers(db\_users);

}

bool DatabaseHelper::IsRentExpired(UserID user\_id) {

pqxx::result db\_result = pimpl\_->ExecPrepared("is\_rent\_expired", user\_id);

return GetScalar<bool>(db\_result);

}

////////////////////////////////////////////////////////////////////////////////

// Applications' handlers

void DatabaseHelper::SubmitApplication(Application& application) {

pqxx::result db\_applications = pimpl\_->ExecPrepared(

"submit\_application", application.user\_id, application.name,

application.surname, application.username);

pqxx::row db\_application = db\_applications.front();

application.submit\_date = db\_application["submit\_date"].c\_str();

}

std::vector<Application> DatabaseHelper::Applications() {

pqxx::result db\_applications = pimpl\_->ExecPrepared("select\_applications");

std::vector<Application> applications;

Application app;

for (pqxx::row db\_application : db\_applications) {

app.user\_id = db\_application["telegram\_user\_id"].as<UserID>();

app.name = db\_application["name"].c\_str();

app.surname = db\_application["surname"].c\_str();

app.username = db\_application["username"].c\_str();

app.submit\_date = db\_application["submit\_date"].c\_str();

applications.push\_back(std::move(app));

}

return applications;

}

std::optional<Application> DatabaseHelper::DeleteApplication(UserID user\_id) {

pqxx::result db\_deleted\_applications =

pimpl\_->ExecPrepared("delete\_application", user\_id);

if (db\_deleted\_applications.empty()) {

return std::nullopt;

}

pqxx::row db\_deleted\_application = db\_deleted\_applications.front();

Application application;

application.user\_id = db\_deleted\_application["telegram\_user\_id"].as<UserID>();

application.name = db\_deleted\_application["name"].c\_str();

application.surname = db\_deleted\_application["surname"].c\_str();

application.username = db\_deleted\_application["username"].c\_str();

application.submit\_date = db\_deleted\_application["submit\_date"].c\_str();

return application;

}

bool DatabaseHelper::IsUnregister(UserID user\_id) {

pqxx::result result = pimpl\_->ExecPrepared("is\_unregister", user\_id);

return GetScalar<bool>(result);

}

////////////////////////////////////////////////////////////////////////////////

// relations\_of\_channel's handlers

SERIAL DatabaseHelper::GetRelationId(UserID user\_id, ChatID observer,

ChatID receiver) {

pqxx::result db\_id =

pimpl\_->ExecPrepared("get\_relation\_id", user\_id, observer, receiver);

return GetScalar<SERIAL>(db\_id);

}

bool DatabaseHelper::ExistsReceiverFromUser(UserID user\_id,

ChatID receiver\_id) {

pqxx::result result =

pimpl\_->ExecPrepared("exists\_receiver\_from\_user", user\_id, receiver\_id);

return GetScalar<bool>(result);

}

bool DatabaseHelper::CreateSubscribe(UserID user\_id, ChatID observer,

ChatID receiver) {

return pimpl\_

->ExecPrepared("insert\_relations\_of\_channel", user\_id, observer, receiver)

.affected\_rows();

}

bool DatabaseHelper::DeleteSubscribe(UserID user\_id, ChatID observer,

ChatID receiver) {

return pimpl\_->ExecPrepared("delete\_subscribe", user\_id, observer, receiver)

.affected\_rows();

}

std::vector<Subscribe> DatabaseHelper::Subscribes(ChatID observer) {

pqxx::result db\_subscribes =

pimpl\_->ExecPrepared("select\_receivers", observer);

return VectorOfSubscribes(db\_subscribes);

}

std::vector<ChatID> DatabaseHelper::ReceiversOfUser(UserID user\_id) {

pqxx::result db\_receivers =

pimpl\_->ExecPrepared("select\_receiver\_distinct", user\_id);

std::vector<ChatID> receivers;

for (pqxx::row db\_receiver : db\_receivers) {

receivers.push\_back(db\_receiver["receiver"].as<ChatID>());

}

return receivers;

}

std::vector<Subscribe> DatabaseHelper::SubscribesOfReceiver(ChatID receiver) {

pqxx::result db\_subscribes =

pimpl\_->ExecPrepared("select\_observers\_for\_receiver", receiver);

return VectorOfSubscribes(db\_subscribes);

}

bool DatabaseHelper::IsRelationExist(UserID user\_id, ChatID observer,

ChatID receiver) {

pqxx::result db\_is\_relation\_exist =

pimpl\_->ExecPrepared("check\_relations", user\_id, observer, receiver);

return GetScalar<bool>(db\_is\_relation\_exist);

}

void DatabaseHelper::AppendKeyPhraseForSubscribe(

SERIAL id, const std::string& key\_phrase) {

pimpl\_->ExecPrepared("append\_key\_phrase\_for\_subscribe", id, key\_phrase);

}

void DatabaseHelper::RemoveKeyPhraseForSubscribe(

SERIAL id, const std::string& key\_phrase) {

pimpl\_->ExecPrepared("remove\_key\_phrase\_for\_subscribe", id, key\_phrase);

}

////////////////////////////////////////////////////////////////////////////////

// channel's handlers

bool DatabaseHelper::InsertChannel(ChatID chat\_id) {

return pimpl\_->ExecPrepared("insert\_channel\_without\_error", chat\_id)

.affected\_rows();

}

void DatabaseHelper::AppendKeyPhraseForChannel(ChatID channel,

const std::string& key\_phrase) {

pimpl\_->ExecPrepared("append\_key\_phrases\_for\_channel", channel, key\_phrase);

}

void DatabaseHelper::RemoveKeyPhraseFromChannel(ChatID channel,

const std::string& key\_phrase) {

pimpl\_->ExecPrepared("remove\_key\_phrases\_from\_channel", channel, key\_phrase);

}

Channel DatabaseHelper::GetChannel(ChatID channel\_id) {

pqxx::result db\_channels = pimpl\_->ExecPrepared("select\_channel", channel\_id);

if (db\_channels.empty()) {

return {};

}

pqxx::row db\_channel = db\_channels.front();

Channel channel;

channel.channel\_id = db\_channel["chat\_id"].as<ChatID>();

channel.key\_phrases = GetArray(db\_channel["key\_phrases"]);

return channel;

}

bool DatabaseHelper::DeleteChannel(ChatID channel\_id) {

pqxx::result db\_delete\_channels =

pimpl\_->ExecPrepared("delete\_channel", channel\_id);

return db\_delete\_channels.affected\_rows() == 1;

}

bool DatabaseHelper::Open(ConnectionParams connection\_params) {

// if connection have already is opened

if (IsOpen()) {

return true;

}

// try to reopen connection

try {

pimpl\_ = std::make\_unique<Impl>();

pimpl\_->connection\_ =

std::make\_unique<pqxx::connection>(MakeConnectionString(

connection\_params.host, connection\_params.user,

connection\_params.password, connection\_params.port));

pimpl\_->PrepareQueries();

pimpl\_->is\_open\_ = true;

} catch (...) {

pimpl\_->is\_open\_ = false;

}

return IsOpen();

}

//////////////////////////////////////////////////////////////////////////////

// chats\_for\_search's table handler

bool DatabaseHelper::InsertChatForSearch(UserID user\_id, ChatID chat\_id) {

return pimpl\_->ExecPrepared("insert\_chats\_for\_search", user\_id, chat\_id)

.affected\_rows();

}

bool DatabaseHelper::DeleteChatForSearch(UserID user\_id, ChatID chat\_id) {

pqxx::result db\_delete =

pimpl\_->ExecPrepared("delete\_from\_channels\_for\_search", user\_id, chat\_id);

return !db\_delete.empty();

}

std::vector<ChatID> DatabaseHelper::GetChannelsForSearch(UserID user\_id) {

pqxx::result db\_chat\_ids =

pimpl\_->ExecPrepared("select\_channels\_for\_search", user\_id);

std::vector<ChatID> chat\_ids;

chat\_ids.reserve(db\_chat\_ids.size());

for (pqxx::row db\_chat\_id : db\_chat\_ids) {

chat\_ids.push\_back(db\_chat\_id["chat\_id"].as<ChatID>());

}

return chat\_ids;

}

void DatabaseHelper::CreateLog(UserID user\_id, MESSAGE\_TYPE message\_type,

const std::string& message) {

bool is\_response;

switch (message\_type) {

case MESSAGE\_TYPE::USER\_MESSAGE:

is\_response = false;

break;

case MESSAGE\_TYPE::BOT\_MESSAGE:

is\_response = true;

break;

}

pimpl\_->ExecPrepared("create\_log", user\_id, is\_response, message);

}

} // namespace db

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 30/07/2021.

//

#include <gtest/gtest.h>

#include <DatabaseHelper.hpp>

#include <thread>

db::ConnectionParams connection\_params{/\*host=\*/"localhost",

/\*user=\*/"postgres",

/\*password=\*/"1234",

/\*port=\*/"5432"};

bool operator==(const db::User& lhs, const db::User& rhs) noexcept {

return lhs.user\_id == rhs.user\_id && lhs.name == rhs.name &&

lhs.surname == rhs.surname && lhs.username == rhs.username &&

lhs.is\_info\_valid == rhs.is\_info\_valid && lhs.is\_admin == rhs.is\_admin;

}

bool operator==(const db::Application& lhs, const db::Application& rhs) {

return lhs.user\_id == rhs.user\_id && lhs.name == rhs.name &&

lhs.surname == rhs.surname && lhs.submit\_date == rhs.submit\_date;

}

TEST(OpenDatabase, database\_is\_open) {

db::DatabaseHelper db{connection\_params};

ASSERT\_TRUE(db.IsOpen());

}

////////////////////////////////////////////////////////////////////////////////

// Users' table tests

TEST(UsersTable, GetRole) {

db::DatabaseHelper db{connection\_params};

EXPECT\_EQ(db.GetRole(579488735), db::User::ROLE::ADMIN);

EXPECT\_EQ(db.GetRole(545288135), db::User::ROLE::USER);

EXPECT\_EQ(db.GetRole(9), db::User::ROLE::NONE);

}

TEST(UsersTable, IsAdmin) {

db::DatabaseHelper db{connection\_params};

EXPECT\_TRUE(db.IsAdmin(579488735));

EXPECT\_TRUE(db.IsAdmin(3953360));

EXPECT\_FALSE(db.IsAdmin(545288135));

}

TEST(UsersTable, InsertAndDeleteUser) {

db::DatabaseHelper db{connection\_params};

db::User user;

user.user\_id = 1;

user.name = "name test";

user.surname = "surname test";

user.username = "username test";

user.is\_info\_valid = 1;

user.is\_admin = false;

EXPECT\_FALSE(db.DeleteUser(user.user\_id));

EXPECT\_EQ(db.GetRole(user.user\_id), db::User::ROLE::NONE);

db.InsertUser(user);

EXPECT\_EQ(db.GetRole(user.user\_id), db::User::ROLE::USER);

EXPECT\_TRUE(db.DeleteUser(user.user\_id));

EXPECT\_FALSE(db.DeleteUser(user.user\_id));

}

TEST(Users, GetAllUsers) {

db::DatabaseHelper db{connection\_params};

auto users = db.Users();

EXPECT\_EQ(users.size(), 3);

}

TEST(Users, GetAdmins) {

db::DatabaseHelper db{connection\_params};

auto admins = db.Admins();

EXPECT\_EQ(admins.size(), 2);

for (const db::Admin& admin : admins) {

EXPECT\_TRUE(admin.is\_admin);

}

}

TEST(Users, IsRentExpired) {

db::DatabaseHelper db{connection\_params};

EXPECT\_TRUE(db.IsRentExpired(545288135));

EXPECT\_TRUE(db.IsRentExpired(3953360));

EXPECT\_TRUE(db.IsRentExpired(579488735));

db::User user;

user.user\_id = -1;

user.is\_admin = false;

user.name = "test";

user.surname = "test";

user.is\_info\_valid = true;

db.InsertUser(user);

EXPECT\_FALSE(db.IsRentExpired(-1));

EXPECT\_TRUE(db.DeleteUser(-1));

}

////////////////////////////////////////////////////////////////////////////////

// Applications' table tests

TEST(Applications, SubmitAndDeleteApplication) {

db::DatabaseHelper db{connection\_params};

db::Application app;

app.user\_id = 3;

app.name = "Kirill";

app.surname = "Danilchuk";

app.username = "KirillDanilchuk";

EXPECT\_FALSE(db.IsUnregister(app.user\_id));

EXPECT\_EQ(db.GetRole(app.user\_id), db::User::ROLE::NONE);

EXPECT\_FALSE(db.DeleteApplication(app.user\_id));

EXPECT\_FALSE(db.IsUnregister(app.user\_id));

db.SubmitApplication(app);

EXPECT\_EQ(db.GetRole(app.user\_id), db::User::ROLE::UNREGISTER);

EXPECT\_TRUE(db.IsUnregister(app.user\_id));

std::vector<db::Application> applications = db.Applications();

EXPECT\_TRUE(db.IsUnregister(app.user\_id));

EXPECT\_TRUE(applications.front() == app);

EXPECT\_TRUE(db.DeleteApplication(app.user\_id));

EXPECT\_FALSE(db.IsUnregister(app.user\_id));

EXPECT\_EQ(db.GetRole(app.user\_id), db::User::ROLE::NONE);

EXPECT\_FALSE(db.DeleteApplication(app.user\_id));

EXPECT\_FALSE(db.IsUnregister(app.user\_id));

}

////////////////////////////////////////////////////////////////////////////////

// relations\_of\_channel's table tests

TEST(relations\_of\_channel, ExistsReceiverFromUser) {

db::DatabaseHelper db{connection\_params};

EXPECT\_TRUE(db.ExistsReceiverFromUser(579488735, -1001554950442));

EXPECT\_TRUE(db.ExistsReceiverFromUser(3953360, -1001465764418));

EXPECT\_FALSE(db.ExistsReceiverFromUser(579488735, -1001465764418));

}

TEST(relations\_of\_channel, CreateAndDeleteSubscribe) {

db::DatabaseHelper db{connection\_params};

EXPECT\_FALSE(db.ExistsReceiverFromUser(1, 101));

db.CreateSubscribe(1, 100, 101);

EXPECT\_TRUE(db.ExistsReceiverFromUser(1, 101));

db.DeleteSubscribe(1, 100, 101);

EXPECT\_FALSE(db.ExistsReceiverFromUser(1, 101));

}

TEST(relations\_of\_channel, Subscribes) {

db::DatabaseHelper db{connection\_params};

std::vector<db::Subscribe> subscribes{db.Subscribes(-1001101170442)};

ASSERT\_EQ(subscribes.size(), 2);

std::sort(subscribes.begin(), subscribes.end(),

[](const db::Subscribe& lhs, const db::Subscribe& rhs) {

return lhs.receiver > rhs.receiver;

});

ASSERT\_EQ(subscribes.front().receiver, -1001383495939);

ASSERT\_EQ(subscribes.back().receiver, -1001554950442);

EXPECT\_EQ(subscribes.front().user\_id, 579488735);

EXPECT\_EQ(subscribes.front().user\_id, subscribes.back().user\_id);

EXPECT\_TRUE(subscribes.front().key\_phrases.empty());

ASSERT\_EQ(subscribes.back().key\_phrases.size(), 1);

EXPECT\_EQ(subscribes.back().key\_phrases.front(), "Олимпиада");

}

TEST(relations\_of\_channel, ReceiversOfUser) {

db::DatabaseHelper db{connection\_params};

std::vector<db::ChatID> receivers{db.ReceiversOfUser(579488735)};

EXPECT\_EQ(receivers.size(), 5);

}

TEST(relations\_of\_channel, SubscribesOfReceiver) {

db::DatabaseHelper db{connection\_params};

std::vector<db::Subscribe> subscribes =

db.SubscribesOfReceiver(-1001593194144);

EXPECT\_EQ(subscribes.size(), 6);

for (const db::Subscribe& subscribe : subscribes) {

EXPECT\_EQ(subscribe.receiver, -1001593194144);

}

EXPECT\_EQ(db.SubscribesOfReceiver(0).size(), 0);

}

TEST(relations\_of\_channel, IsRelationExist) {

db::DatabaseHelper db{connection\_params};

EXPECT\_TRUE(db.IsRelationExist(579488735, -1001006869491, -1001593194144));

EXPECT\_FALSE(db.IsRelationExist(123, -1001006869492, -1001593194144));

}

TEST(relations\_of\_channel, AppendAndRemoveKeyPhraseForSubscribe) {

db::DatabaseHelper db{connection\_params};

const std::string test\_phrases = "тестовой ключевая фраза";

// Append

{

std::vector<db::Subscribe> subscribes = db.Subscribes(-1001197755674);

ASSERT\_EQ(subscribes.size(), 1);

db::Subscribe alone\_subscription = std::move(subscribes.front());

db.AppendKeyPhraseForSubscribe(alone\_subscription.id, test\_phrases);

}

{

std::vector<db::Subscribe> subscribes = db.Subscribes(-1001197755674);

ASSERT\_EQ(subscribes.size(), 1);

db::Subscribe alone\_subscription = std::move(subscribes.front());

ASSERT\_EQ(alone\_subscription.key\_phrases.size(), 1);

EXPECT\_EQ(alone\_subscription.key\_phrases.front(), test\_phrases);

}

// Remove

{

std::vector<db::Subscribe> subscribes = db.Subscribes(-1001197755674);

ASSERT\_EQ(subscribes.size(), 1);

db::Subscribe alone\_subscription = std::move(subscribes.front());

db.RemoveKeyPhraseForSubscribe(alone\_subscription.id, test\_phrases);

}

{

std::vector<db::Subscribe> subscribes = db.Subscribes(-1001197755674);

ASSERT\_EQ(subscribes.size(), 1);

db::Subscribe alone\_subscription = std::move(subscribes.front());

db.RemoveKeyPhraseForSubscribe(alone\_subscription.id, test\_phrases);

ASSERT\_EQ(alone\_subscription.key\_phrases.size(), 0);

}

}

////////////////////////////////////////////////////////////////////////////////

// channel's table tests

TEST(channels, common\_case) {

db::DatabaseHelper db{connection\_params};

constexpr db::ChatID channel\_id{111};

const std::string test\_phrase = "тест или не тест";

db.InsertChannel(channel\_id);

db.AppendKeyPhraseForChannel(channel\_id, test\_phrase);

db::Channel channel = db.GetChannel(channel\_id);

EXPECT\_EQ(channel.channel\_id, channel\_id);

ASSERT\_EQ(channel.key\_phrases.size(), 1);

EXPECT\_EQ(channel.key\_phrases.front(), test\_phrase);

db.RemoveKeyPhraseFromChannel(channel\_id, test\_phrase);

channel = db.GetChannel(channel\_id);

EXPECT\_EQ(channel.channel\_id, channel\_id);

EXPECT\_TRUE(channel.key\_phrases.empty());

EXPECT\_TRUE(db.DeleteChannel(channel\_id));

}

////////////////////////////////////////////////////////////////////////////////

// Multi consumers test

TEST(MultiConsumers, common\_case) {

db::DatabaseHelper db;

db.Open(connection\_params);

constexpr db::ChatID channel\_id = 111;

db.InsertChannel(channel\_id);

std::vector<std::thread> threads;

constexpr size\_t count\_threads = 5;

constexpr size\_t number\_of\_operations = 10;

threads.reserve(count\_threads);

for (size\_t i = 0; i < count\_threads; ++i) {

threads.push\_back(std::thread([]() {

db::DatabaseHelper db;

ASSERT\_TRUE(db.Open(connection\_params));

for (size\_t i = 0; i < number\_of\_operations; ++i) {

EXPECT\_FALSE(db.IsUnregister(-124));

}

}));

}

for (auto&& thread : threads) {

thread.join();

}

db.DeleteChannel(channel\_id);

}

//////////////////////////////////////////////////////////////////////////////

// chats\_for\_search's test

TEST(chats\_for\_search, common\_case) {

db::DatabaseHelper db{connection\_params};

db::UserID user\_id = -1;

db::ChatID chat\_id = 1;

db.InsertChatForSearch(user\_id, chat\_id);

std::vector<db::ChatID> chat\_ids = db.GetChannelsForSearch(user\_id);

ASSERT\_EQ(chat\_ids.size(), 1);

EXPECT\_EQ(chat\_ids.front(), chat\_id);

EXPECT\_TRUE(db.DeleteChatForSearch(user\_id, chat\_id));

EXPECT\_TRUE(db.GetChannelsForSearch(user\_id).empty());

}

//////////////////////////////////////////////////////////////////////////////

// logs\_of\_messages's test

TEST(logs\_of\_messages, simpl\_test) {

db::DatabaseHelper db{connection\_params};

db.CreateLog(0, db::MESSAGE\_TYPE::USER\_MESSAGE, "a user send message");

db.CreateLog(0, db::MESSAGE\_TYPE::BOT\_MESSAGE, "the bot response to a user");

}

cmake\_minimum\_required(VERSION 3.14)

project(database\_test)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES

${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.cpp

${CMAKE\_CURRENT\_SOURCE\_DIR}/../src/\*.cpp)

add\_executable(${PROJECT\_NAME} ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PRIVATE ${CMAKE\_CURRENT\_SOURCE\_DIR}/../include)

# dependencies

# GTest

find\_package(GTest REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE GTest::GTest GTest::Main)

# pqxx

find\_library(PQXX\_LIB pqxx)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE ${PQXX\_LIB})

cmake\_minimum\_required(VERSION 3.14)

project(db)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES ${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp)

add\_library(${PROJECT\_NAME} STATIC ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR}/include)

# dependencies

# pqxx

find\_library(PQXX\_LIB pqxx)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE ${PQXX\_LIB})

target\_include\_directories(${PROJECT\_NAME} PRIVATE /usr/local/include)

add\_subdirectory(test)

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 22/06/2021.

//

#pragma once

#include <td/telegram/Client.h>

#include <td/telegram/td\_api.h>

#include <atomic>

#include <functional>

#include <mutex>

#include <string>

#include <td/telegram/td\_api.hpp>

#include <unordered\_map>

#include <vector>

namespace td\_api = td::td\_api;

class TelegramSearcher {

using ChatID = std::int64\_t;

using UserID = std::int32\_t;

public:

explicit TelegramSearcher();

~TelegramSearcher();

/\*\*

\* @brief Restart connection

\*/

void Restart();

/\*\*

\* @brief Need restart connect?

\* @return need\_restart\_;

\*/

[[nodiscard]] bool NeedRestart() const;

/\*\*

\* @brief Search messages in telegram channels

\* @param query is simple text for search

\* @return td\_api::object\_ptr<td\_api::messages>

\*/

td\_api::searchMessages::ReturnType SearchMessages(const std::string& query);

/\*\*

\* @brief Get link for message as string

\* @param message

\* @return object\_ptr<messageLink>

\*/

td\_api::getMessageLink::ReturnType GetMessageLink(

const td\_api::object\_ptr<td\_api::message>& message);

/\*\*

\* @brief Join to public channel (for private channel use JoinPrivateChannel)

\* @param username Username to be resolved

\* @return returns true if join to channel else returns false

\*/

bool JoinPublicChannel(const std::string& username);

/\*\*

\* @brief Join to private channel (for public channel use JoinPublicChannel)

\* @param invite\_link Invite link to import; should begin with

\* "https://t.me/joinchat/", "https://telegram.me/joinchat/", or

\* "https://telegram.dog/joinchat/"

\* @return returns true if join to channel else returns false

\*/

bool JoinPrivateChannel(const std::string& invite\_link);

/\*\*

\* @brief delete user from member list of channel.

\* @param title is title of channel

\* @return returns true if leave from channel else returns false

\*/

bool LeaveFromChannel(const std::string& title);

/\*\*

\* @brief Run loop which fetch response and handle it

\*/

void RunFetchLoop();

/\*\*

\* @brief stop loop

\*/

void StopFetchLoop();

/\*\*

\* @param query is param for example "@KirillDanilchuk\_channel" or

\* "KirillDanilchuk\_channel"

\* @return chat\_id is search public chat is success else std::nullopt

\*/

std::optional<std::pair<std::int64\_t, std::string>> SearchPublicChat(

const std::string& query);

/\*\*

\* @brief

\* @param chat\_id

\* @return info for chat

\*/

td\_api::getChat::ReturnType GetChatInfo(std::int64\_t chat\_id);

/\*\*

\* @brief search messages which have query via substring into one chat with

\* chat\_id

\* @param chat\_id

\* @param query is simple text for search

\* @return array of the links (as strings)

\*/

std::vector<std::string> GetMessagesLinksFromChat(std::int64\_t chat\_id,

const std::string& query);

/\*\*

\* @brief search messages which have query via substring into one chat with

\* chat\_id

\* @param chat\_id

\* @param query is simple text for search

\* @return array of the messages

\*/

std::vector<td\_api::object\_ptr<td\_api::message>> GetMessagesFromChat(

std::int64\_t chat\_id, const std::string& query);

/\*\*

\* @brief Setup handler for new message and invoke handler for each new

\* message

\* @param handler is function for handle new message

\*/

void SetNewMessageHandler(

std::function<void(td\_api::updateNewMessage& update\_new\_message)>

handler);

/\*\*

\* @brief Forward message from message->chat\_ to forward\_chat\_id

\* @param message message which will be forwarded

\* @param forward\_chat\_id chat where message forward

\*/

void ForwardMessage(td\_api::object\_ptr<td\_api::message>& message,

std::int64\_t forward\_chat\_id);

/\*\*

\* @brief get text content from message (if message is plain text that return

\* it else return caption of message)

\* @param message message with formattedText content

\* @return text or caption

\*/

std::string GetText(td\_api::object\_ptr<td\_api::message>& message);

/\*\*

\* @brief get administrators within chat

\* @param chat\_id where find administrators

\* @return vector of chat administrators

\*/

td\_api::object\_ptr<td\_api::chatAdministrators> GetAdministrators(

ChatID chat\_id);

/\*\*

\* @brief check is the chat channel or not

\* @param chat

\* @return true if the chat is channel else return false

\*/

bool IsChannel(td\_api::object\_ptr<td\_api::chat>& chat);

private:

std::unordered\_map<std::int64\_t, std::string> chat\_title\_;

bool are\_authorized\_{false};

bool need\_restart\_{false};

td::ClientManager::RequestId query\_id\_{};

std::unique\_ptr<td::ClientManager> client\_manager\_;

td::ClientManager::ClientId client\_id\_{};

td\_api::object\_ptr<td\_api::AuthorizationState> authorization\_state\_{};

td::ClientManager::RequestId authorization\_query\_id\_{};

std::atomic<bool> is\_run\_fetch\_loop{true};

std::function<void(td\_api::updateNewMessage& update\_new\_message)>

new\_message\_handler\_;

using Object = td\_api::object\_ptr<td\_api::Object>;

std::unordered\_map<td::ClientManager::RequestId, std::function<void(Object)>>

handlers\_;

std::recursive\_mutex mutex\_;

void Init();

td::ClientManager::RequestId NextQueryId() noexcept;

td::ClientManager::RequestId SendQuery(

td\_api::object\_ptr<td\_api::Function> function,

std::function<void(Object)> handler);

td::ClientManager::Response FetchResponse(double timeout = 0);

void ProcessResponse(td::ClientManager::Response response);

void StartAuthorizationLoop();

void ProcessUpdate(Object update);

void OnAuthorizationStateUpdate();

std::function<void(Object)> CreateAuthorizationQueryHandler();

void CheckAuthorizationError(Object object);

void SendQueryAndFetchResponse(td\_api::object\_ptr<td\_api::Function> function,

std::function<void(Object)> handler);

static bool CheckError(const Object& object);

std::vector<td\_api::object\_ptr<td\_api::chat>> GetChats();

template <typename T, typename... Args>

auto Receive(Args&&... args) -> typename T::ReturnType {

typename T::ReturnType result{nullptr};

SendQueryAndFetchResponse(

td\_api::make\_object<T>(std::forward<Args>(args)...),

[&result](Object object) {

if (CheckError(object)) {

return;

}

using ElementType = typename T::ReturnType::element\_type;

result = td\_api::move\_object\_as<ElementType>(object);

});

return result;

}

/\*\*

\* @brief Simple case is get the all links of messages as strings

\* @param query is simple text for search

\* @return array of the links (as strings)

\* @example

\* @code{.cpp}

\* TelegramSearcher engine;

\* auto links = engine.GetMessagesLinks("Погода");

\* for (const auto& link : links) {

\* DoSomething(link);

\* }

\* @endcode

\*/

std::vector<std::string> GetMessagesLinks(const std::string& query);

};

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 22/06/2021.

//

#include "TelegramSearcher.hpp"

#include <iostream>

#include <utility>

namespace {

template <typename... Functions>

class Overloaded : public Functions... {

public:

explicit Overloaded(Functions&&... functions) : Functions{functions}... {

// pass

}

using Functions::operator()...;

};

template <typename... Functions>

auto ViaOverloaded(Functions&&... functions) {

return Overloaded<Functions...>(std::forward<Functions>(functions)...);

}

}

TelegramSearcher::TelegramSearcher() : are\_authorized\_{false} {

Init();

}

void TelegramSearcher::Init() {

td::ClientManager::execute(

td\_api::make\_object<td\_api::setLogVerbosityLevel>(1));

client\_manager\_ = std::make\_unique<td::ClientManager>();

client\_id\_ = client\_manager\_->create\_client\_id();

SendQuery(td\_api::make\_object<td\_api::getOption>("version"), {});

StartAuthorizationLoop();

}

td::ClientManager::RequestId TelegramSearcher::SendQuery(

td\_api::object\_ptr<td\_api::Function> function,

std::function<void(Object)> handler) {

auto current\_query\_id = NextQueryId();

if (handler) {

handlers\_.emplace(current\_query\_id, std::move(handler));

}

client\_manager\_->send(client\_id\_, current\_query\_id, std::move(function));

return current\_query\_id;

}

td::ClientManager::RequestId TelegramSearcher::NextQueryId() noexcept {

return ++query\_id\_;

}

td::ClientManager::Response TelegramSearcher::FetchResponse(double timeout) {

auto response = client\_manager\_->receive(timeout);

// std::cout <<"Request id: " << response.request\_id << std::endl;

return response;

}

void TelegramSearcher::ProcessResponse(td::ClientManager::Response response) {

// std::cout << to\_string(response.object) << std::endl;

if (!response.object) {

return;

}

if (response.request\_id == 0) {

return ProcessUpdate(std::move(response.object));

}

auto it = handlers\_.find(response.request\_id);

if (it != handlers\_.end()) {

it->second(std::move(response.object));

handlers\_.erase(it);

}

}

void TelegramSearcher::StartAuthorizationLoop() {

while (!are\_authorized\_) {

if (need\_restart\_) {

Restart();

}

auto response = FetchResponse(10);

ProcessResponse(std::move(response));

}

}

void TelegramSearcher::ProcessUpdate(Object update) {

td\_api::downcast\_call(

\*update,

ViaOverloaded(

[this](td\_api::updateAuthorizationState& update\_authorization\_state) {

authorization\_state\_ =

std::move(update\_authorization\_state.authorization\_state\_);

OnAuthorizationStateUpdate();

},

[this](td\_api::updateNewMessage& update\_new\_message) {

new\_message\_handler\_(update\_new\_message);

},

[](auto&&) {

// pass

}));

}

void TelegramSearcher::OnAuthorizationStateUpdate() {

++authorization\_query\_id\_;

td\_api::downcast\_call(

\*authorization\_state\_,

ViaOverloaded(

[this](td\_api::authorizationStateReady&) {

are\_authorized\_ = true;

std::cout << "Got authorization" << std::endl;

},

[this](td\_api::authorizationStateLoggingOut&) {

are\_authorized\_ = false;

std::cout << "Logging out" << std::endl;

},

[](td\_api::authorizationStateClosing&) {

std::cout << "Closing" << std::endl;

},

[this](td\_api::authorizationStateClosed&) {

are\_authorized\_ = false;

need\_restart\_ = true;

std::cout << "Terminated" << std::endl;

},

[this](td\_api::authorizationStateWaitCode&) {

std::cout << "Enter authentication code: " << std::flush;

std::string code;

std::cin >> code;

SendQuery(

td\_api::make\_object<td\_api::checkAuthenticationCode>(code),

CreateAuthorizationQueryHandler());

},

[this](td\_api::authorizationStateWaitRegistration&) {

std::string first\_name;

std::string last\_name;

std::cout << "Enter your first name: " << std::flush;

std::cin >> first\_name;

std::cout << "Enter your last name: " << std::flush;

std::cin >> last\_name;

SendQuery(td\_api::make\_object<td\_api::registerUser>(first\_name,

last\_name),

CreateAuthorizationQueryHandler());

},

[this](td\_api::authorizationStateWaitPassword&) {

std::cout << "Enter authentication password: " << std::flush;

std::string password;

std::getline(std::cin, password);

std::cin.ignore(std::numeric\_limits<short>::max(), '\n');

SendQuery(td\_api::make\_object<td\_api::checkAuthenticationPassword>(

password),

CreateAuthorizationQueryHandler());

},

[](td\_api::authorizationStateWaitOtherDeviceConfirmation& state) {

std::cout << "Confirm this login link on another device: "

<< state.link\_ << std::endl;

},

[this](td\_api::authorizationStateWaitPhoneNumber&) {

std::cout << "Enter phone number: " << std::flush;

std::string phone\_number;

std::cin >> phone\_number;

SendQuery(td\_api::make\_object<td\_api::setAuthenticationPhoneNumber>(

phone\_number, nullptr),

CreateAuthorizationQueryHandler());

},

[this](td\_api::authorizationStateWaitEncryptionKey&) {

std::cout << "Enter encryption key or DESTROY: " << std::flush;

std::string key;

std::getline(std::cin, key);

if (key == "DESTROY") {

SendQuery(td\_api::make\_object<td\_api::destroy>(),

CreateAuthorizationQueryHandler());

} else {

SendQuery(td\_api::make\_object<td\_api::checkDatabaseEncryptionKey>(

std::move(key)),

CreateAuthorizationQueryHandler());

}

},

[this](td\_api::authorizationStateWaitTdlibParameters&) {

auto parameters = td\_api::make\_object<td\_api::tdlibParameters>();

parameters->database\_directory\_ = "tdlib";

parameters->use\_message\_database\_ = true;

parameters->use\_secret\_chats\_ = true;

parameters->api\_id\_ = 6047365;

parameters->api\_hash\_ = "e5028a34e32b21c4baa1a9e8fcab6c70";

parameters->system\_language\_code\_ = "en";

parameters->device\_model\_ = "Desktop";

parameters->application\_version\_ = "1.0";

parameters->enable\_storage\_optimizer\_ = true;

SendQuery(td\_api::make\_object<td\_api::setTdlibParameters>(

std::move(parameters)),

CreateAuthorizationQueryHandler());

}));

}

auto TelegramSearcher::CreateAuthorizationQueryHandler()

-> std::function<void(Object)> {

return [this, id = authorization\_query\_id\_](Object object) {

if (id == authorization\_query\_id\_) {

CheckAuthorizationError(std::move(object));

}

};

}

void TelegramSearcher::Restart() {

Init();

}

void TelegramSearcher::CheckAuthorizationError(Object object) {

if (object->get\_id() == td\_api::error::ID) {

auto error = td::move\_tl\_object\_as<td\_api::error>(object);

std::cout << to\_string(error) << std::flush;

OnAuthorizationStateUpdate();

}

}

td\_api::searchMessages::ReturnType TelegramSearcher::SearchMessages(

const std::string& query) {

constexpr int MAX\_MESSAGE\_COUNT = 100;

return Receive<td\_api::searchMessages>(nullptr, query, 0, 0, 0,

MAX\_MESSAGE\_COUNT, nullptr, 0, 0);

}

void TelegramSearcher::SendQueryAndFetchResponse(

td\_api::object\_ptr<td\_api::Function> function,

std::function<void(Object)> handler) {

std::lock\_guard lg{mutex\_};

auto query\_id = SendQuery(std::move(function), std::move(handler));

while (true) {

auto response = FetchResponse();

if (response.object == nullptr) {

continue;

}

ProcessResponse(std::move(response));

auto it = handlers\_.find(query\_id);

if (it == handlers\_.end()) {

return;

}

}

}

td\_api::getMessageLink::ReturnType TelegramSearcher::GetMessageLink(

const td\_api::object\_ptr<td\_api::message>& message) {

return Receive<td\_api::getMessageLink>(message->chat\_id\_, message->id\_, 0, false,

false);

}

std::vector<std::string> TelegramSearcher::GetMessagesLinks(

const std::string& query) {

std::vector<std::string> links;

auto messages = SearchMessages(query);

if (messages == nullptr) {

return {};

}

for (auto& message : messages->messages\_) {

auto link = GetMessageLink(message);

if (link == nullptr) {

continue;

}

links.push\_back(link->link\_);

}

return links;

}

bool TelegramSearcher::NeedRestart() const {

return need\_restart\_;

}

bool TelegramSearcher::JoinPublicChannel(const std::string& username) {

std::optional<std::pair<std::int64\_t, std::string>> chat =

SearchPublicChat(username);

if (!chat.has\_value()) {

return false;

}

bool is\_ok = Receive<td\_api::joinChat>(chat.value().first) != nullptr;

return is\_ok;

}

std::optional<std::pair<std::int64\_t, std::string>>

TelegramSearcher::SearchPublicChat(const std::string& query) {

Object object = Receive<td\_api::searchPublicChat>(query);

if (CheckError(object)) {

return std::nullopt;

}

auto chat = td\_api::move\_object\_as<td\_api::chat>(object);

return std::pair<std::int64\_t, std::string>{chat->id\_,

std::move(chat->title\_)};

}

bool TelegramSearcher::CheckError(const Object& object) {

if (object == nullptr) {

return true;

}

if (object->get\_id() == td\_api::error::ID) {

// auto\* error = static\_cast<const td\_api::error\*>(object.get());

return true;

}

return false;

}

bool TelegramSearcher::JoinPrivateChannel(const std::string& invite\_link) {

auto chat = Receive<td\_api::joinChatByInviteLink>(invite\_link);

if (!chat) {

return false;

}

// LogInfo("Пользователь вступил в приватный канал: " + chat->title\_);

return true;

}

bool TelegramSearcher::LeaveFromChannel(const std::string& title) {

auto chats = GetChats();

std::int64\_t chat\_id;

bool is\_find = false;

for (auto& chat : chats) {

if (chat->title\_ == title) {

chat\_id = chat->id\_;

is\_find = true;

}

}

if (!is\_find) {

return false;

}

bool is\_ok = Receive<td\_api::leaveChat>(chat\_id) != nullptr;

if (is\_ok) {

// LogInfo("Пользователь покинул канал: " + title);

}

return is\_ok;

}

std::vector<td\_api::object\_ptr<td\_api::chat>> TelegramSearcher::GetChats() {

std::vector<td\_api::object\_ptr<td\_api::chat>> result;

auto chat\_ids = Receive<td\_api::getChats>(

nullptr, std::numeric\_limits<std::int64\_t>::max(), 0, 20);

if (chat\_ids == nullptr) {

return {};

}

for (auto& id : chat\_ids->chat\_ids\_) {

auto chat = Receive<td\_api::getChat>(id);

if (chat) {

result.push\_back(std::move(chat));

}

}

return result;

}

void TelegramSearcher::RunFetchLoop() {

is\_run\_fetch\_loop.store(true);

td::ClientManager::Response response;

while (is\_run\_fetch\_loop.load()) {

std::lock\_guard lg{mutex\_};

response = FetchResponse();

ProcessResponse(std::move(response));

}

}

void TelegramSearcher::StopFetchLoop() {

is\_run\_fetch\_loop.store(false);

}

TelegramSearcher::~TelegramSearcher() {

StopFetchLoop();

}

td\_api::getChat::ReturnType TelegramSearcher::GetChatInfo(

std::int64\_t chat\_id) {

return Receive<td\_api::getChat>(chat\_id);

}

std::vector<std::string> TelegramSearcher::GetMessagesLinksFromChat(

std::int64\_t chat\_id, const std::string& query) {

auto messages = GetMessagesFromChat(chat\_id, query);

std::vector<std::string> links;

links.reserve(messages.size());

for (auto& message : messages) {

auto link = GetMessageLink(message);

if (link) {

links.push\_back(link->link\_);

}

}

return links;

}

std::vector<td\_api::object\_ptr<td\_api::message>>

TelegramSearcher::GetMessagesFromChat(std::int64\_t chat\_id,

const std::string& query) {

constexpr int MAX\_MESSAGE\_COUNT = 100;

auto messages = Receive<td\_api::searchChatMessages>(

chat\_id, query, nullptr, 0, 0, MAX\_MESSAGE\_COUNT, nullptr, 0);

if (!messages) {

return {};

}

return std::move(messages->messages\_);

}

void TelegramSearcher::SetNewMessageHandler(

std::function<void(td\_api::updateNewMessage& update\_new\_message)> handler) {

new\_message\_handler\_ = std::move(handler);

}

void TelegramSearcher::ForwardMessage(

td\_api::object\_ptr<td\_api::message>& message,

std::int64\_t forward\_chat\_id) {

Receive<td\_api::forwardMessages>(

forward\_chat\_id, message->chat\_id\_,

std::vector<std::int64\_t>{message->id\_}, /\*option = \*/nullptr,

/\*send\_copy\_ = \*/true, /\*remove\_caption\_ = \*/false);

}

std::string TelegramSearcher::GetText(

td\_api::object\_ptr<td\_api::message>& message) {

std::string result{};

td\_api::downcast\_call(\*message->content\_,

ViaOverloaded(

[&result](td\_api::messageText& message\_text) {

result = message\_text.text\_->text\_;

},

[&result](td\_api::messageAnimation& message) {

result = message.caption\_->text\_;

},

[&result](td\_api::messageAudio& message) {

result = message.caption\_->text\_;

},

[&result](td\_api::messageDocument& message) {

result = message.caption\_->text\_;

},

[&result](td\_api::messagePhoto& message) {

result = message.caption\_->text\_;

},

[&result](td\_api::messageVideo& message) {

result = message.caption\_->text\_;

},

[&result](td\_api::messageVoiceNote& message) {

result = message.caption\_->text\_;

},

[](auto&&) {

;

}));

return result;

}

td\_api::object\_ptr<td\_api::chatAdministrators>

TelegramSearcher::GetAdministrators(ChatID chat\_id) {

return Receive<td\_api::getChatAdministrators>(chat\_id);

}

bool TelegramSearcher::IsChannel(td\_api::object\_ptr<td\_api::chat>& chat) {

if (!(chat && chat->type\_)) {

return false;

}

bool is\_channel{false};

td\_api::downcast\_call(

\*chat->type\_, ViaOverloaded(

[&is\_channel](td\_api::chatTypeSupergroup& supergroup) {

is\_channel = supergroup.is\_channel\_;

},

[&is\_channel](auto&&) {

is\_channel = false;

}));

return is\_channel;

}

cmake\_minimum\_required(VERSION 3.14)

project(td\_test)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES

${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.cpp)

add\_executable(${PROJECT\_NAME} ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PRIVATE ${CMAKE\_CURRENT\_SOURCE\_DIR}/../include)

# dependencies

# GTest

find\_package(GTest REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE GTest::GTest GTest::Main)

# TdLib

find\_package(Td REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PUBLIC Td::TdStatic)

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 30/07/2021.

//

#include <gtest/gtest.h>

#include "../src/TelegramSearcher.cpp"

TEST(Overloaded, any\_case) {

auto overloaded = ViaOverloaded(

[](int) {

return 1;

},

[](float) {

return 2.0f;

},

[](double) {

return 3.0;

});

EXPECT\_EQ(overloaded(1), 1);

EXPECT\_EQ(overloaded(1.0f), 2.0f);

EXPECT\_EQ(overloaded(1.0), 3.0);

}

cmake\_minimum\_required(VERSION 3.14)

project(td)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES ${CMAKE\_CURRENT\_SOURCE\_DIR}/src/\*.cpp)

add\_library(${PROJECT\_NAME} STATIC ${SOURCE\_FILES})

target\_include\_directories(${PROJECT\_NAME} PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR}/include)

add\_subdirectory(test)

# dependencies

# TdLib

find\_package(Td REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PUBLIC Td::TdStatic)

cmake\_minimum\_required(VERSION 3.14)

project(libs)

add\_subdirectory(bot)

add\_subdirectory(db)

add\_subdirectory(td)

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 28/06/2021.

//

#include "SearcherBot.hpp"

#include <BotUI.hpp>

#include <utility.hpp>

#include <utility>

SearcherBot::SearcherBot() : bot\_{test\_token} {

InitUserCommands();

InitAdminCommands();

InitDatabase();

InitBot();

InitEngine();

StartLongPoll();

}

void SearcherBot::InitDatabase() noexcept {

db::ConnectionParams connection\_params;

connection\_params.host = "194.87.92.205";

connection\_params.port = "51001";

connection\_params.user = "postgres";

connection\_params.password = "1234";

if (!database\_.Open(std::move(connection\_params))) {

std::terminate();

}

}

void SearcherBot::InitBot() {

SetCommands();

bot\_.getEvents().onAnyMessage([this](const MessagePtr& message) {

try {

database\_.CreateLog(message->from->id, db::MESSAGE\_TYPE::USER\_MESSAGE,

message->text);

} catch (...) {

}

});

bot\_.getEvents().onCommand("start", [this](const MessagePtr& message) {

on\_start\_command(message);

});

bot\_.getEvents().onCommand("help", [this](const MessagePtr& message) {

on\_help\_command(message);

});

bot\_.getEvents().onCommand("feedback", [this](const MessagePtr& message) {

on\_feedback\_command(message);

});

bot\_.getEvents().onNonCommandMessage([this](const MessagePtr& message) {

on\_non\_command\_message(message);

});

bot\_.getEvents().onCallbackQuery(

[this](const TgBot::CallbackQuery::Ptr& callback\_query) {

auto& data = callback\_query->data;

if (StringTools::startsWith(data, "page\_")) {

on\_change\_page(callback\_query);

} else if (data == "left") {

on\_left\_rotate(callback\_query);

} else if (data == "right") {

on\_right\_rotate(callback\_query);

}

});

}

void SearcherBot::StartLongPoll() {

TgBot::TgLongPoll long\_poll{bot\_};

while (true) try {

long\_poll.start();

} catch (std::exception& exception) {

std::cout << exception.what() << std::endl;

}

}

void SearcherBot::SetCommands() {

std::vector<TgBot::BotCommand::Ptr> commands;

auto CreateCommand = [](const std::string& command,

const std::string& description) {

auto bot\_command = std::make\_shared<TgBot::BotCommand>();

bot\_command->command = command;

bot\_command->description = description;

return bot\_command;

};

commands.push\_back(CreateCommand("help", "Отобразить помощь"));

commands.push\_back(CreateCommand("feedback", "Оставить отзыв"));

bot\_.getApi().setMyCommands(commands);

}

void SearcherBot::on\_non\_command\_message(const MessagePtr& message) {

if (CreatingFirstAdmin(message)) {

return;

}

if (bot\_.TryNextStep(message)) {

return;

}

db::User::ROLE role = database\_.GetRole(message->from->id);

bool is\_invoke{false};

switch (role) {

case db::User::ROLE::UNREGISTER:

SendMessage(message->chat->id, "Ожидайте. Ваша заявка обрабатывается.");

bot\_.getApi().sendSticker(message->chat->id,

"CAACAgIAAxkBAAECjKZg5qowYcTG\_rwGkTq\_V5Sf2s-"

"jYwACLAADJHFiGsUg5gPvePzkIAQ");

is\_invoke = true;

break;

case db::User::ROLE::NONE:

if (message->text == "📝 Оставить заявку") {

SubmitYourApplication(message);

} else {

SendMessage(message->chat->id,

"Здравствуйте, " + message->from->firstName +

". Оставьте заявку для регистрации",

ui::RegistrationMarkup());

}

is\_invoke = true;

break;

case db::User::ROLE::ADMIN:

is\_invoke = InvokeCommand(message, admin\_commands\_) ||

InvokeCommand(message, user\_commands\_);

break;

case db::User::ROLE::USER:

if (!IsRentExpired(message->from->id)) {

SendMessage(message->chat->id,

"Подписка закончилась, продлите аренду.");

return;

}

is\_invoke = InvokeCommand(message, user\_commands\_);

break;

}

if (!is\_invoke) {

SendMessage(message->chat->id, "Неизвестная команда", GetStartMarkup(role));

bot\_.getApi().sendSticker(message->chat->id,

"CAACAgIAAxkBAAECgdtg23a5ZyJnJC2aAqufM7LsBYm6"

"8AACMwADJHFiGkNsrGPTFqfgIAQ");

}

}

void SearcherBot::InsertUser(UserID telegram\_user\_id, db::User::ROLE role,

const std::string& first\_name,

const std::string& second\_name,

const std::string& username) {

if (database\_.GetRole(telegram\_user\_id) != db::User::ROLE::NONE) {

return;

}

db::User user;

user.user\_id = telegram\_user\_id;

user.name = first\_name;

user.surname = second\_name;

user.username = username;

user.is\_admin = (role == db::User::ROLE::ADMIN);

user.is\_info\_valid = true;

database\_.InsertUser(user);

}

bool SearcherBot::CreatingFirstAdmin(const MessagePtr& message) {

if (!database\_.IsFirstUser()) {

return false;

}

if (message->text == "🥇 Добавить меня") {

auto& user = message->from;

InsertUser(user->id, db::User::ROLE::ADMIN, user->firstName, user->lastName,

user->username);

SendMessage(message->chat->id, "Поздравляю, Вы первый администратор.",

ui::AdminStartMarkup());

} else {

SendMessage(message->chat->id,

"Здравствуй, " + message->from->firstName +

". Если Вы первый администратор, то добавьте себя.",

ui::FirstAdminMarkup());

}

return true;

}

void SearcherBot::SubmitYourApplication(const MessagePtr& message) {

if (database\_.GetRole(message->from->id) == db::User::ROLE::NONE) {

auto user = message->from;

db::Application application;

application.user\_id = user->id;

application.name = user->firstName;

application.surname = user->lastName;

application.username = user->username;

database\_.SubmitApplication(application);

NotifyAdmins(application);

SendMessage(message->from->id, "Заявка успешно оставлена.");

} else {

SendMessage(message->from->id, "Вы уже оставили заявку.");

}

}

void SearcherBot::NotifyAdmins(const db::Application& application) {

for (const db::Admin& admin : database\_.Admins()) {

SendMessage(admin.user\_id,

"Новая заявка!"

"\nID: " +

std::to\_string(application.user\_id) + "\nИмя: " +

application.name + "\nФамилия: " + application.surname +

"\nUsername: " + application.username +

"\nВремя: " + application.submit\_date);

}

}

bool SearcherBot::InvokeCommand(

const MessagePtr& message,

const std::unordered\_map<std::string, SearcherBot::Step>& commands) {

auto it = commands.find(message->text);

if (it == commands.end()) {

return false;

}

it->second(message);

return true;

}

TgBot::ReplyKeyboardMarkup::Ptr SearcherBot::GetStartMarkup(

db::User::ROLE role) {

switch (role) {

case db::User::ROLE::ADMIN:

return ui::AdminStartMarkup();

case db::User::ROLE::USER:

return ui::UserStartMarkup();

case db::User::ROLE::NONE:

case db::User::ROLE::UNREGISTER:

default:

return ui::RegistrationMarkup();

}

}

bool SearcherBot::AcceptApplication(std::int32\_t telegram\_user\_id,

db::User::ROLE role) {

auto application\_optional = database\_.DeleteApplication(telegram\_user\_id);

if (!application\_optional.has\_value()) {

return false;

}

db::Application application = std::move(application\_optional.value());

InsertUser(telegram\_user\_id, role, application.name, application.surname,

application.username);

SendMessage(telegram\_user\_id, "Добро пожаловать!", GetStartMarkup(role));

bot\_.getApi().sendSticker(telegram\_user\_id,

"CAACAgIAAxkBAAECjV9g5wMDUOGkgaTwvii1ePsWCHvSXwACLg"

"ADJHFiGojoNkNqQEMUIAQ");

return true;

}

bool SearcherBot::DeleteUser(UserID telegram\_user\_id) {

if (database\_.DeleteUser(telegram\_user\_id)) {

SendMessage(telegram\_user\_id, "Вы были удалены.", ui::RegistrationMarkup());

bot\_.getApi().sendSticker(telegram\_user\_id,

"CAACAgIAAxkBAAECjWFg5xV0DUxab0VMQEqG7AABG4dN9rMA"

"AioAAyRxYhpL1dVk2ohqnCAE");

return true;

}

return false;

}

bool SearcherBot::RenewLease(UserID telegram\_user\_id, int32\_t number\_of\_days) {

std::string new\_date = database\_.RenewLeast(telegram\_user\_id, number\_of\_days);

if (!new\_date.empty()) {

SendMessage(telegram\_user\_id, "Изменение срока аренды -> " + new\_date);

}

return !new\_date.empty();

}

std::string SearcherBot::ToString(const db::User& user) {

std::string user\_format\_info;

if (user.is\_admin) {

user\_format\_info += "👑 Администратор 👑";

}

user\_format\_info.append("\nID: ").append(std::to\_string(user.user\_id));

user\_format\_info.append("\nФамилия: ").append(user.surname);

user\_format\_info.append("\nИмя: ").append(user.name);

user\_format\_info.append("\nUsername: ").append(user.username);

user\_format\_info.append("\nВалидные данные: ")

.append(user.is\_info\_valid ? "Да" : "Нет");

if (user.is\_admin) {

user\_format\_info += "\nДата регистрации: ";

} else {

user\_format\_info += "\nДата окончания аренды: ";

}

user\_format\_info += user.rent\_to;

return user\_format\_info;

}

void SearcherBot::on\_applications\_list(const SearcherBot::MessagePtr& message) {

std::vector<db::Application> applications = database\_.Applications();

if (applications.empty()) {

SendMessage(message->chat->id, "Новых заявок нет.");

return;

}

for (std::size\_t i = 1; const db::Application& application : applications) {

SendMessage(message->chat->id,

"Заявка №" + std::to\_string(i) + std::string{"\nID: "} +

std::to\_string(application.user\_id) + "\nИмя: " +

application.name + "\nФамилия: " + application.surname +

"\nUsername: " + application.username +

"\nДата подачи заявки: " + application.submit\_date);

++i;

}

}

void SearcherBot::on\_insert\_admin(const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id, "Введите ID администратора:");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

bool is\_ok;

try {

is\_ok =

AcceptApplication(std::stoi(message->text), db::User::ROLE::ADMIN);

} catch (std::invalid\_argument& exception) {

SendMessage(message->chat->id, "Аргумент не является числом");

return;

}

if (is\_ok) {

SendMessage(message->chat->id, "Заявка принята.");

} else {

SendMessage(message->chat->id, "Заявка не принята.");

}

});

}

void SearcherBot::on\_insert\_user(const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id, "Введите ID пользователя:");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

bool is\_ok;

try {

is\_ok = AcceptApplication(std::stoi(message->text), db::User::ROLE::USER);

} catch (std::invalid\_argument& exception) {

SendMessage(message->chat->id, "Аргумент не является числом");

return;

}

if (is\_ok) {

SendMessage(message->chat->id, "Заявка принята.");

} else {

SendMessage(message->chat->id, "Заявка не принята.");

}

});

}

void SearcherBot::on\_users\_list(const SearcherBot::MessagePtr& message) {

std::vector<db::User> users = database\_.Users();

for (const db::User& user : users) {

SendMessage(message->chat->id, ToString(user));

}

}

void SearcherBot::on\_delete\_user(const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id, "Введите ID пользователя:");

auto delete\_user = [this](const MessagePtr& message) {

try {

bool is\_ok = DeleteUser(std::stoi(message->text));

SendMessage(message->chat->id, is\_ok ? "Пользователь успешно удален."

: "Пользователь не удален.");

} catch (std::invalid\_argument& exception) {

SendMessage(message->chat->id, "Вы ввели не число.");

return;

}

};

bot\_.RegistryNextStep(message->chat->id, delete\_user);

}

void SearcherBot::on\_renew\_lease(const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id, "Введите ID пользователя:");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

std::int32\_t user\_id;

try {

user\_id = std::stoi(message->text);

} catch (std::invalid\_argument&) {

SendMessage(message->chat->id, "Вы ввели не число.");

return;

}

SendMessage(message->chat->id, "Введите количество дней:");

bot\_.RegistryNextStep(

message->chat->id, [this, user\_id](const MessagePtr& message) {

std::int32\_t number\_of\_days;

try {

number\_of\_days = std::stoi(message->text);

} catch (std::invalid\_argument&) {

SendMessage(message->from->id, "Вы ввели не число");

return;

}

if (RenewLease(user\_id, number\_of\_days)) {

SendMessage(message->chat->id, "Аренда успешно обновлена.");

} else {

SendMessage(message->chat->id, "Ошибка. Аренда не обновлена.");

}

});

});

}

void SearcherBot::on\_back\_to\_start\_markup(

const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id,

message->from->firstName + ", выберите действие.",

ui::AdminStartMarkup());

}

void SearcherBot::InitAdminCommands() {

admin\_commands\_ = {{"📣📣📣 Групповая рассылка 📣📣📣",

[this](const MessagePtr& message) {

on\_broadcast(message);

}},

{"👨‍💻 ‍Панель пользователя",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Выберите действие.",

ui::UserStartMarkupForAdmin());

}},

{"🔓 Панель администратора",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Доступ открыт.",

ui::AdminMarkup());

}},

{"📋 Список заявок",

[this](const MessagePtr& message) {

on\_applications\_list(message);

}},

{"➕ Добавить администратора",

[this](const MessagePtr& message) {

on\_insert\_admin(message);

}},

{"➕ Добавить пользователя",

[this](const MessagePtr& message) {

on\_insert\_user(message);

}},

{"📚 Список пользователей",

[this](const MessagePtr& message) {

on\_users\_list(message);

}},

{"➖ Удалить пользователя",

[this](const MessagePtr& message) {

on\_delete\_user(message);

}},

{"📆 Продлить аренду",

[this](const MessagePtr& message) {

on\_renew\_lease(message);

}},

{"🔙 На главную",

[this](const MessagePtr& message) {

on\_back\_to\_start\_markup(message);

}},

{"🔙 Выбор функции", [this](const MessagePtr& message) {

SendMessage(message->chat->id, "Выберите действие.",

ui::UserStartMarkupForAdmin());

}}};

}

void SearcherBot::InitUserCommands() {

user\_commands\_ = {{"➖ Удалить ключевую фразу для подписки",

[this](const MessagePtr& message) {

on\_remove\_key\_phrase\_for\_subscribe(message);

}},

{"➕ Добавить ключевую фразу для подписки",

[this](const MessagePtr& message) {

on\_append\_key\_phrase\_for\_subscribe(message);

}},

{"➖ Удалить ключевую фразу для канала",

[this](const MessagePtr& message) {

on\_remove\_key\_phrase\_for\_channel(message);

}},

{"➕ Добавить ключевую фразу для канала",

[this](const MessagePtr& message) {

on\_append\_key\_phrase\_for\_channel(message);

}},

{"📗 Просмотреть подписки",

[this](const MessagePtr& message) {

on\_show\_subscribes(message);

}},

{"➖ Удалить подписку",

[this](const MessagePtr& message) {

on\_delete\_subscribe(message);

}},

{"➕ Добавить подписку",

[this](const MessagePtr& message) {

on\_insert\_subscribe(message);

}},

{"🔙 Выбор функции",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Выберите действие.",

ui::UserStartMarkup());

}},

{"📜 Подписки",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Выберите действие.",

ui::UserSubscribeMarkup());

}},

{"🔍 Найти",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Введите запрос");

bot\_.RegistryNextStep(message->chat->id,

[this](const MessagePtr& message) {

on\_search\_messages(message);

});

}},

{"📒 Список каналов",

[this](const MessagePtr& message) {

on\_channels\_for\_search\_show(message);

}},

{"🔕 Выйти из канала",

[this](const MessagePtr& message) {

on\_leave\_from\_channel(message);

}},

{"📣 Добавить публичный канал",

[this](const MessagePtr& message) {

on\_insert\_channel\_for\_search(message);

}},

{"🔎 Поиск",

[this](const MessagePtr& message) {

SendMessage(message->chat->id, "Выберите действие.",

ui::UserSearchingMarkup());

}},

{"🔙 На главную", [this](const MessagePtr& message) {

db::User::ROLE role =

database\_.GetRole(message->from->id);

SendMessage(message->chat->id, "Вы на главной странице.",

GetStartMarkup(role));

}}};

}

bool SearcherBot::InsertChannelForSearch(SearcherBot::UserID telegram\_user\_id,

SearcherBot::ChatID chat\_id) {

database\_.InsertChannel(chat\_id);

return database\_.InsertChatForSearch(telegram\_user\_id, chat\_id);

}

std::optional<std::string> SearcherBot::ErasePublicChatForSearch(

UserID telegram\_user\_id, const std::string& channel\_username) {

std::optional<std::pair<std::int64\_t, std::string>> chat =

engine\_.SearchPublicChat(channel\_username);

if (!chat.has\_value()) {

return std::nullopt;

}

auto [chat\_id, title] = chat.value();

if (!database\_.DeleteChatForSearch(telegram\_user\_id, chat\_id)) {

return std::nullopt;

}

return title;

}

void SearcherBot::on\_channels\_for\_search\_show(

const SearcherBot::MessagePtr& message) {

std::vector<db::ChatID> chat\_ids =

database\_.GetChannelsForSearch(message->from->id);

if (chat\_ids.empty()) {

SendMessage(message->chat->id, "Список каналов для поиска пуст.");

return;

}

std::string result\_message{"Список каналов для поиска:\n"};

std::int64\_t i{};

for (ChatID chat\_id : chat\_ids) {

auto chat = engine\_.GetChatInfo(chat\_id);

if (!chat) {

// TODO: delete chat from database

continue;

}

result\_message.append(std::to\_string(++i))

.append(") ")

.append(chat->title\_)

.append("\n");

}

SendMessage(message->chat->id, result\_message);

}

void SearcherBot::on\_leave\_from\_channel(

const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id,

"Введите название канала, например, @KirillDanilchuk\_channel");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

std::optional<std::string> title\_ =

ErasePublicChatForSearch(message->from->id, message->text);

if (title\_.has\_value()) {

SendMessage(message->chat->id, "Канал: \"" + title\_.value() +

"\" удален из каналов для поиска.");

} else {

SendMessage(message->chat->id, "Не удалось удалить канал.");

}

});

}

void SearcherBot::on\_insert\_channel\_for\_search(

const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id,

"Введите название публичного канала, "

"например, @KirillDanilchuk\_channel");

bot\_.RegistryNextStep(message->from->id, [this](const MessagePtr& message) {

for (auto& chat\_username : Split(message->text, " ")) {

std::optional<std::pair<ChatID, std::string>> chat{

engine\_.SearchPublicChat(chat\_username)};

if (!chat.has\_value()) {

SendMessage(message->chat->id, "Не удалось найти канал");

continue;

}

bool is\_ok =

InsertChannelForSearch(message->from->id, chat.value().first);

if (is\_ok) {

SendMessage(message->chat->id,

"Канал: \"" + chat.value().second +

"\" успешно добавлен в список для поиска.");

} else {

SendMessage(message->chat->id,

"Вы уже ищите в канале: \"" + chat.value().second + '.');

}

}

});

}

void SearcherBot::on\_search\_messages(const SearcherBot::MessagePtr& message) {

const std::string& text = message->text;

std::vector<std::pair<std::string, std::int64\_t>> links =

GetMessagesLinks(message->from->id, text);

if (links.empty()) {

SendMessage(message->chat->id, "Ничего не нашлось");

bot\_.getApi().sendSticker(message->chat->id,

"CAACAgIAAxkBAAECgdlg23ag37xfkl61J4-"

"cz3CS7ZftwwACJwADJHFiGg-k3QShSuM3IAQ");

return;

}

auto inline\_markup = ui::CreateInlineKeyboardMarkup(links.size());

SendMessage(message->chat->id,

text + " : 1 из " + std::to\_string(links.size()) + " : " +

links.front().first,

inline\_markup);

}

std::vector<std::pair<std::string, std::int64\_t>> SearcherBot::GetMessagesLinks(

UserID telegram\_user\_id, const std::string& query) {

std::vector<std::pair<std::string, std::int64\_t>> links; //<link, count\_view>

std::vector<db::ChatID> chat\_ids =

database\_.GetChannelsForSearch(telegram\_user\_id);

for (ChatID chat\_id : chat\_ids) {

for (auto& message : engine\_.GetMessagesFromChat(chat\_id, query)) {

auto link = engine\_.GetMessageLink(message);

if (link) {

links.emplace\_back(engine\_.GetMessageLink(message)->link\_,

message->interaction\_info\_

? message->interaction\_info\_->view\_count\_

: 0);

}

}

}

std::sort(links.begin(), links.end(),

[](const std::pair<std::string, std::int64\_t>& a,

const std::pair<std::string, std::int64\_t>& b) {

return a.second > b.second;

});

return links;

}

namespace {

std::pair<std::string, size\_t> GetData(TgBot::Message::Ptr& message) {

constexpr auto separator = " : ";

auto args = Split(message->text, separator);

std::string query = std::move(args[0]);

std::string part\_two = std::move(args[1]);

size\_t current\_page = std::stoll(Split(part\_two, " из ")[0]) -

1; // -1 because for user I add +1

return {std::move(query), current\_page};

}

} // namespace

void SearcherBot::on\_right\_rotate(

const TgBot::CallbackQuery::Ptr& callback\_query) {

auto& message = callback\_query->message;

auto& reply\_buttons = message->replyMarkup->inlineKeyboard[0];

auto [query, \_] = GetData(message);

auto links = GetMessagesLinks(message->chat->id, query);

for (size\_t i = 0; i < BUTTONS\_COUNT; ++i) {

size\_t current\_button\_number = std::stoi(reply\_buttons[i]->text) - 1;

size\_t next\_button\_number =

(current\_button\_number + BUTTONS\_COUNT) % links.size();

reply\_buttons[i]->text = std::to\_string(next\_button\_number + 1);

reply\_buttons[i]->callbackData =

"page\_" + std::to\_string(next\_button\_number);

}

EditMessage(message->chat->id, message->messageId, message->text,

message->replyMarkup);

}

void SearcherBot::EditMessage(int64\_t chat\_id, int32\_t message\_id,

const std::string& message,

TgBot::GenericReply::Ptr keyboard) {

bot\_.getApi().editMessageText(message, chat\_id, message\_id, "", "", false,

std::move(keyboard));

}

void SearcherBot::on\_left\_rotate(

const TgBot::CallbackQuery::Ptr& callback\_query) {

auto& message = callback\_query->message;

auto& reply\_buttons = message->replyMarkup->inlineKeyboard[0];

auto [query, \_] = GetData(message);

auto links = GetMessagesLinks(message->chat->id, query);

for (size\_t i = 0; i < BUTTONS\_COUNT; ++i) {

size\_t current\_button\_number = std::stoi(reply\_buttons[i]->text) - 1;

long long next\_button\_number = current\_button\_number - BUTTONS\_COUNT;

if (next\_button\_number < 0) {

next\_button\_number += links.size();

}

reply\_buttons[i]->text = std::to\_string(next\_button\_number + 1);

reply\_buttons[i]->callbackData =

"page\_" + std::to\_string(next\_button\_number);

}

EditMessage(message->chat->id, message->messageId, message->text,

message->replyMarkup);

}

void SearcherBot::on\_change\_page(

const TgBot::CallbackQuery::Ptr& callback\_query) {

size\_t next\_page = std::stoll(Split(callback\_query->data, "\_")[1]);

SetCurrentPage(callback\_query->message, next\_page);

}

void SearcherBot::SetCurrentPage(TgBot::Message::Ptr message, size\_t page) {

auto [query, \_] = GetData(message);

auto links = GetMessagesLinks(message->chat->id, query);

if (links.empty()) {

return;

}

auto current\_link = links[page];

bot\_.getApi().editMessageText(query + " : " + std::to\_string(page + 1) +

" из " + std::to\_string(links.size()) +

" : " + current\_link.first,

message->chat->id, message->messageId, "", "",

false, message->replyMarkup);

}

void SearcherBot::on\_insert\_subscribe(const MessagePtr& message) {

SendMessage(message->chat->id, "Какой канал слушать?");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

auto observer\_chat = engine\_.SearchPublicChat(message->text);

if (!observer\_chat) {

SendMessage(message->chat->id, "Такого канала не существует.");

return;

}

SendMessage(message->chat->id, "В какой канал публиковать?");

bot\_.RegistryNextStep(

message->chat->id,

[observer\_chat, observer\_chat\_username = message->text,

this](const MessagePtr& message) {

auto receiver\_chat = engine\_.SearchPublicChat(message->text);

if (!receiver\_chat) {

SendMessage(message->chat->id, "Такого канала не существует.");

return;

}

bool was\_join = engine\_.JoinPublicChannel(message->text) &&

engine\_.JoinPublicChannel(observer\_chat\_username);

auto [is\_ok, error\_message] = InsertSubscribe(

message->from->id, observer\_chat->first, receiver\_chat->first);

if (is\_ok && was\_join) {

SendMessage(message->chat->id, "Подписка оформлена.");

} else {

SendMessage(message->chat->id,

"Не удалось оформить подписку. " + error\_message);

}

});

});

}

std::pair<bool, std::string> SearcherBot::InsertSubscribe(

SearcherBot::UserID telegram\_user\_id, SearcherBot::ChatID observer,

SearcherBot::ChatID receiver) {

if (!IsAdmin(telegram\_user\_id, receiver)) {

return {false,

"Не достаточно прав. Добавьте бота администратором в канал."};

}

if (database\_.InsertChannel(observer) && database\_.InsertChannel(receiver) &&

database\_.CreateSubscribe(telegram\_user\_id, observer, receiver)) {

return {true, {}};

} else {

return {false, "Данная подписка уже существует."};

}

}

void SearcherBot::InitEngine() {

engine\_.SetNewMessageHandler([this](td\_api::updateNewMessage& new\_message) {

NewMessageHandler(new\_message);

});

std::thread([this]() {

engine\_.RunFetchLoop();

}).detach();

}

void SearcherBot::NewMessageHandler(td\_api::updateNewMessage& new\_message) {

auto chat = engine\_.GetChatInfo(new\_message.message\_->chat\_id\_);

if (!engine\_.IsChannel(chat)) {

return;

}

ChatID observer = new\_message.message\_->chat\_id\_;

std::vector<db::Subscribe> subscribes = database\_.Subscribes(observer);

std::string message\_text = engine\_.GetText(new\_message.message\_);

for (db::Subscribe subscribe : subscribes) {

auto receiver\_id = subscribe.receiver;

auto telegram\_user\_id = subscribe.user\_id;

if (!IsRentExpired(telegram\_user\_id)) {

continue;

}

auto& forward\_info = new\_message.message\_->forward\_info\_;

if (forward\_info && forward\_info->from\_chat\_id\_ == receiver\_id) {

continue;

}

db::Channel channel = database\_.GetChannel(receiver\_id);

std::vector<std::string> key\_phrases\_for\_channel =

std::move(channel.key\_phrases);

std::vector<std::string> key\_phrases\_for\_subscribe =

std::move(subscribe.key\_phrases);

// Append key phrases for subscribe to key phrases for channel

key\_phrases\_for\_channel.insert(

key\_phrases\_for\_channel.end(),

std::make\_move\_iterator(key\_phrases\_for\_subscribe.begin()),

std::make\_move\_iterator(key\_phrases\_for\_subscribe.end()));

if (key\_phrases\_for\_channel.empty() ||

Check(message\_text, key\_phrases\_for\_channel)) {

engine\_.ForwardMessage(new\_message.message\_, receiver\_id);

}

}

}

void SearcherBot::on\_delete\_subscribe(const MessagePtr& message) {

SendMessage(message->chat->id, "От какого канала отвязать подписку?");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

auto chat\_info = engine\_.SearchPublicChat(message->text);

if (!chat\_info.has\_value()) {

SendMessage(message->chat->id, "Канала не существует.");

return;

}

std::int64\_t receiver\_id{chat\_info.value().first};

if (!IsAdmin(message->from->id, receiver\_id)) {

SendMessage(message->chat->id, "Ошибка, Вы не администратор канала.");

return;

}

SendMessage(message->chat->id, "Какой канал отписать?");

bot\_.RegistryNextStep(

message->chat->id, [this, receiver\_id](const MessagePtr& message) {

auto chat\_info = engine\_.SearchPublicChat(message->text);

if (!chat\_info.has\_value()) {

SendMessage(message->chat->id, "Канала не существует.");

return;

}

std::int64\_t observer\_id{chat\_info.value().first};

if (!database\_.DeleteSubscribe(message->from->id, observer\_id,

receiver\_id)) {

SendMessage(message->chat->id, "Данной подписки не существовало.");

} else {

SendMessage(message->chat->id, "Подписка успешно удалена.");

}

});

});

}

void SearcherBot::on\_show\_subscribes(const SearcherBot::MessagePtr& message) {

auto append\_formatted\_receiver =

[this](std::string& result, const std::string& title, ChatID chat\_id,

size\_t receiver\_number) {

result.append("Канал ")

.append(std::to\_string(receiver\_number))

.append("[")

.append(std::to\_string(chat\_id))

.append("]")

.append(": ")

.append(title)

.append("\n");

db::Channel channel = database\_.GetChannel(chat\_id);

std::vector<std::string> key\_phrases = std::move(channel.key\_phrases);

if (key\_phrases.empty()) {

return;

}

result.append("Ключевые фразы: \n");

for (size\_t i = 0; i < key\_phrases.size() - 1; ++i) {

result.append("\"").append(key\_phrases[i]).append("\", ");

}

result.append("\"").append(key\_phrases.back()).append("\"\n");

};

auto append\_formatted\_observer =

[](std::string& result, const std::string& title, ChatID observer\_id,

const std::vector<std::string> key\_phrases, size\_t receiver\_number,

size\_t observer\_number) {

result.append("\t- ")

.append(std::to\_string(receiver\_number))

.append(".")

.append(std::to\_string(observer\_number))

.append(" ")

.append(title)

.append("[ ")

.append(std::to\_string(observer\_id))

.append("]")

.append("\n");

if (key\_phrases.empty()) {

return;

}

result.append("Ключевые фразы: \n");

for (size\_t i = 0; i < key\_phrases.size() - 1; ++i) {

result.append("\"").append(key\_phrases[i]).append("\", ");

}

result.append("\"").append(key\_phrases.back()).append("\"\n");

};

// Code begin here

std::vector<db::ChatID> receiver\_ids =

database\_.ReceiversOfUser(message->from->id);

if (receiver\_ids.empty()) {

SendMessage(message->chat->id, "Список чатов пуст.");

return;

}

std::string result{"📃 Список чатов 📃\n"};

size\_t receiver\_count{0};

for (db::ChatID receiver\_id : receiver\_ids) {

auto receiver\_info = engine\_.GetChatInfo(receiver\_id);

if (!receiver\_info) {

database\_.DeleteChannel(receiver\_id);

continue;

}

append\_formatted\_receiver(result, receiver\_info->title\_, receiver\_id,

++receiver\_count);

result.append("Прослушивает:\n");

std::vector<db::Subscribe> subscribes =

database\_.SubscribesOfReceiver(receiver\_id);

size\_t observer\_count{0};

for (const db::Subscribe& subscribe : subscribes) {

auto observer\_info = engine\_.GetChatInfo(subscribe.observer);

if (!observer\_info) {

database\_.DeleteChannel(subscribe.observer);

continue;

}

append\_formatted\_observer(result, observer\_info->title\_,

subscribe.observer, subscribe.key\_phrases,

receiver\_count, ++observer\_count);

}

result.append("\n");

}

SendMessage(message->chat->id, result);

}

bool SearcherBot::HasAccessToKeyPhrases(UserID telegram\_user\_id,

ChatID receiver) {

return database\_.ExistsReceiverFromUser(telegram\_user\_id, receiver);

}

void SearcherBot::on\_append\_key\_phrase\_for\_channel(

const SearcherBot::MessagePtr& message) {

ShowReceivers(message->from->id);

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

ChatID channel\_id;

try {

channel\_id = std::stoll(message->text);

} catch (const std::invalid\_argument& exception) {

SendMessage(message->chat->id, "Вы ввели не число.");

return;

}

if (!HasAccessToKeyPhrases(message->from->id, channel\_id)) {

SendMessage(message->chat->id, "У вас нет доступа к данному чату.");

return;

}

SendMessage(message->chat->id, "Введите ключевую фразу");

bot\_.RegistryNextStep(

message->chat->id, [this, channel\_id](const MessagePtr& message) {

database\_.AppendKeyPhraseForChannel(channel\_id, message->text);

SendMessage(message->chat->id, "Ключевая фраза успешно добавлена");

});

});

}

void SearcherBot::on\_remove\_key\_phrase\_for\_channel(

const SearcherBot::MessagePtr& message) {

ShowReceivers(message->from->id);

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

try {

ChatID channel\_id = std::stoll(message->text);

if (HasAccessToKeyPhrases(message->from->id, channel\_id)) {

SendMessage(message->chat->id, "Введите ключевую фразу");

bot\_.RegistryNextStep(

message->chat->id, [this, channel\_id](const MessagePtr& message) {

database\_.RemoveKeyPhraseFromChannel(channel\_id, message->text);

SendMessage(message->chat->id, "Ключевая фраза успешно удалена");

});

}

} catch (const std::invalid\_argument& exception) {

SendMessage(message->chat->id, "Вы ввели не число.");

}

});

}

void SearcherBot::ShowReceivers(UserID telegram\_user\_id) {

std::vector<db::ChatID> receiver\_ids =

database\_.ReceiversOfUser(telegram\_user\_id);

if (receiver\_ids.empty()) {

SendMessage(telegram\_user\_id, "Список каналов пуст. Добавьте подписку.");

return;

}

std::string result{"Список публикующих каналов.\n"};

size\_t i{0};

for (db::ChatID receiver\_id : receiver\_ids) {

auto receiver\_info = engine\_.GetChatInfo(receiver\_id);

result.append(std::to\_string(++i) + ". " + receiver\_info->title\_)

.append(" [" + std::to\_string(receiver\_id) + "]\n");

}

SendMessage(telegram\_user\_id, result);

SendMessage(telegram\_user\_id, "Введите ID публикующего канала.");

}

void SearcherBot::on\_feedback\_command(const SearcherBot::MessagePtr& message) {

SendMessage(message->chat->id,

"Введите Ваш отзыв (пожелание, жалобу, замечание и т.д.) и об "

"этом узнает команда разработки.");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

constexpr UserID KIRILL\_DANILCHUK\_ID = 579488735;

SendMessage(KIRILL\_DANILCHUK\_ID, "Отзыв:");

bot\_.getApi().forwardMessage(KIRILL\_DANILCHUK\_ID, message->chat->id,

message->messageId);

});

}

void SearcherBot::on\_broadcast(const MessagePtr& message) {

SendMessage(message->chat->id, "Введите сообщение для рассылки");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

for (const db::User& user : database\_.Users()) {

bot\_.getApi().forwardMessage(user.user\_id, message->chat->id,

message->messageId);

}

});

}

void SearcherBot::on\_start\_command(const SearcherBot::MessagePtr& message) {

db::User::ROLE role = database\_.GetRole(message->from->id);

SendMessage(message->chat->id, "Выберите действие", GetStartMarkup(role));

}

void SearcherBot::on\_help\_command(const SearcherBot::MessagePtr& message) {

db::User::ROLE role = database\_.GetRole(message->from->id);

TgBot::InputFile::Ptr input\_file = DocForRole(role);

if (!input\_file) {

SendMessage(

message->chat->id,

"Документация доступна только зарегистрированным пользователям.");

return;

}

input\_file->fileName = "Документация.pdf";

std::string caption{"Документация для "};

if (role == db::User::ROLE::USER) {

caption += "пользователя.";

} else if (role == db::User::ROLE::ADMIN) {

caption += "администратора.";

}

auto thumb = TgBot::InputFile::fromFile("docs/cat.jpg", "image/jpeg");

bot\_.getApi().sendDocument(message->chat->id, input\_file, thumb, caption);

}

TgBot::InputFile::Ptr SearcherBot::DocForRole(db::User::ROLE role) {

switch (role) {

case db::User::ROLE::ADMIN:

return TgBot::InputFile::fromFile("docs/admin.pdf", "application/pdf");

case db::User::ROLE::USER:

return TgBot::InputFile::fromFile("docs/user.pdf", "application/pdf");

default:

return nullptr;

}

}

bool SearcherBot::IsRentExpired(SearcherBot::UserID telegram\_user\_id) {

if (database\_.GetRole(telegram\_user\_id) == db::User::ROLE::ADMIN) {

return true;

}

return database\_.IsRentExpired(telegram\_user\_id);

}

bool SearcherBot::IsAdmin(SearcherBot::UserID telegram\_user\_id,

SearcherBot::ChatID chat\_id) {

auto administrators = engine\_.GetAdministrators(chat\_id);

if (!administrators) {

return false;

}

auto it = std::find\_if(administrators->administrators\_.begin(),

administrators->administrators\_.end(),

[telegram\_user\_id](auto& administrator) {

return administrator->user\_id\_ == telegram\_user\_id;

});

return it != administrators->administrators\_.end();

}

void SearcherBot::on\_append\_key\_phrase\_for\_subscribe(

const MessagePtr& message) {

SendMessage(

message->chat->id,

"Введите ID канала откуда слушают и куда публикуют через пробел.");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

std::vector<std::string> args\_str = Split(message->text, " ");

if (args\_str.size() != 2) {

SendMessage(message->from->id, "Вы ввели не два параметра.");

return;

}

ChatID observer\_id;

ChatID receiver\_id;

try {

observer\_id = std::stoll(args\_str.front());

receiver\_id = std::stoll(args\_str.back());

} catch (std::invalid\_argument& exception) {

SendMessage(message->from->id, "Вы ввели не число.");

return;

}

db::SERIAL relations\_of\_channel\_id =

database\_.GetRelationId(message->from->id, observer\_id, receiver\_id);

if (relations\_of\_channel\_id == 0) {

SendMessage(message->from->id, "У Вас нет доступа к данной подписке");

return;

}

SendMessage(message->chat->id, "Введите ключевую фразу");

bot\_.RegistryNextStep(message->chat->id, [this, relations\_of\_channel\_id](

const MessagePtr& message) {

database\_.AppendKeyPhraseForSubscribe(relations\_of\_channel\_id,

message->text);

SendMessage(message->chat->id, "Ключевая фраза для подписки добавлена.");

});

});

}

void SearcherBot::on\_remove\_key\_phrase\_for\_subscribe(

const SearcherBot::MessagePtr& message) {

SendMessage(

message->chat->id,

"Введите ID канала откуда слушают и куда публикуют через пробел.");

bot\_.RegistryNextStep(message->chat->id, [this](const MessagePtr& message) {

std::vector<std::string> args\_str = Split(message->text, " ");

if (args\_str.size() != 2) {

SendMessage(message->from->id, "Вы ввели не два параметра.");

return;

}

ChatID observer\_id;

ChatID receiver\_id;

try {

observer\_id = std::stoll(args\_str.front());

receiver\_id = std::stoll(args\_str.back());

} catch (std::invalid\_argument& exception) {

SendMessage(message->from->id, "Вы ввели не число.");

return;

}

db::SERIAL relations\_of\_channel\_id =

database\_.GetRelationId(message->from->id, observer\_id, receiver\_id);

if (relations\_of\_channel\_id == 0) {

SendMessage(message->from->id, "У Вас нет доступа к данной подписке");

return;

}

SendMessage(message->chat->id, "Введите ключевую фразу");

bot\_.RegistryNextStep(message->chat->id, [this, relations\_of\_channel\_id](

const MessagePtr& message) {

database\_.RemoveKeyPhraseForSubscribe(relations\_of\_channel\_id,

message->text);

SendMessage(message->chat->id, "Ключевая фраза для подписки удалена.");

});

});

}

void SearcherBot::SendMessage(SearcherBot::ChatID chat\_id,

const std::string& message,

TgBot::GenericReply::Ptr reply\_markup) {

bot\_.getApi().sendMessage(chat\_id, message, false, 0,

std::move(reply\_markup));

database\_.CreateLog(chat\_id, db::MESSAGE\_TYPE::BOT\_MESSAGE, message);

}

//

// Created by Kirill Danilchuk on 20/06/2021.

//

#include <olestem/stemming/russian\_stem.h>

#include <algorithm>

#include <boost/algorithm/string.hpp>

#include <boost/algorithm/string/split.hpp>

#include <codecvt>

#include <locale>

#include <utility.hpp>

std::vector<std::string> Split(const std::string& string,

const std::string& separator) {

std::vector<std::string> result;

std::string\_view view{string};

while (true) {

std::size\_t pos = view.find(separator);

if (pos == std::string::npos) {

if (!view.empty()) {

result.push\_back(static\_cast<std::string>(view));

}

break;

}

result.push\_back(static\_cast<std::string>(view.substr(0, pos)));

view.remove\_prefix(pos + separator.size());

}

return result;

}

std::vector<std::string> WordSplit(const std::string& text) {

std::vector<std::string> result;

boost::algorithm::split(result, text, boost::is\_any\_of(" \t\n!\"#$%&'()\*+,-./:;<=>?@[\\]^\_`{|}~"),

boost::token\_compress\_on);

result.erase(std::remove(result.begin(), result.end(), ""), result.end());

return result;

}

void StemWords(std::vector<std::string>& words) {

stemming::russian\_stem<> StemRussian;

using converter\_type = std::codecvt\_utf8<wchar\_t>;

std::wstring\_convert<converter\_type, wchar\_t> convert;

for (std::string& word : words) {

std::wstring wide\_word = convert.from\_bytes(word);

StemRussian(wide\_word);

word = convert.to\_bytes(wide\_word);

}

}

std::vector<std::string> SplitAndStem(const std::string& text) {

std::vector<std::string> result = WordSplit(text);

StemWords(result);

return result;

}

bool IsSubset(std::vector<std::string> subset, std::vector<std::string> set) {

std::sort(subset.begin(), subset.end());

std::sort(set.begin(), set.end());

std::vector<std::string> intersection;

std::set\_intersection(subset.begin(), subset.end(), set.begin(), set.end(),

std::back\_inserter(intersection));

return intersection.size() == subset.size();

}

void ToLower(std::vector<std::string>& strings) {

for (std::string& string : strings) {

using converter\_type = std::codecvt\_utf8<wchar\_t>;

std::wstring\_convert<converter\_type, wchar\_t> convert;

std::wstring lower\_string = boost::algorithm::to\_lower\_copy(

convert.from\_bytes(string), std::locale("ru\_RU"));

string = convert.to\_bytes(lower\_string);

}

}

std::vector<std::string> SplitAndStemInLowerCase(const std::string& text) {

std::vector<std::string> result = SplitAndStem(text);

ToLower(result);

return result;

}

bool Check(const std::string& text,

const std::vector<std::string>& key\_phrases) {

std::vector<std::string> text\_words = SplitAndStemInLowerCase(text);

for (const std::string& key\_phrase : key\_phrases) {

std::vector<std::string> key\_words = SplitAndStemInLowerCase(key\_phrase);

if (IsSubset(key\_words, text\_words)) {

return true;

}

}

return false;

}

cmake\_minimum\_required(VERSION 3.16)

project(test)

set(CMAKE\_CXX\_STANDARD 20)

file(GLOB\_RECURSE SOURCE\_FILES \*.cpp

${CMAKE\_CURRENT\_SOURCE\_DIR}/../src/utility.cpp)

add\_executable(${PROJECT\_NAME} ${SOURCE\_FILES})

include\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/../include)

find\_package(GTest REQUIRED)

target\_link\_libraries(${PROJECT\_NAME} PUBLIC GTest::GTest GTest::Main)

target\_include\_directories(${PROJECT\_NAME} PUBLIC ${GTEST\_INCLUDE\_DIR})

#add\_subdirectory(${CMAKE\_CURRENT\_SOURCE\_DIR}/ThirdParty/OleanderStemmingLibrary)

include\_directories(${PROJECT\_NAME} PRIVATE ${CMAKE\_CURRENT\_SOURCE\_DIR}/ThirdParty/OleanderStemmingLibrary/include)

target\_link\_libraries(${PROJECT\_NAME} PRIVATE oleanderstemminglibrary)

//

// Created by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 20/06/2021.

//

#include <gtest/gtest.h>

#include <utility.hpp>

TEST(Split, any\_cases) {

{

auto result = Split("/добавь\_пользователя 12312313", " ");

ASSERT\_EQ(result.size(), 2);

EXPECT\_EQ(result[0], "/добавь\_пользователя");

EXPECT\_EQ(result[1], "12312313");

}

{

auto result = Split("/Файл : 2 из 34 : /http", " : ");

ASSERT\_EQ(result.size(), 3);

EXPECT\_EQ(result[0], "/Файл");

EXPECT\_EQ(result[1], "2 из 34");

EXPECT\_EQ(result[2], "/http");

}

{

auto result = Split("/Файл : ", " : ");

ASSERT\_EQ(result.size(), 1);

EXPECT\_EQ(result[0], "/Файл");

}

{

auto result = Split("/Файл : : ", " : ");

ASSERT\_EQ(result.size(), 2);

EXPECT\_EQ(result[0], "/Файл");

EXPECT\_EQ(result[1], "");

}

}

TEST(Check, true\_case) {

EXPECT\_TRUE(Check(

"Типы файлов в Linux | Уроки по Linux - Ravesli\n"

"При навигации по файловой системе в Linux, вы обязательно столкнетесь с "

"различными типами файлов. Наиболее часто используемые и очевидные типы "

"— это обычные файлы и каталоги. Однако помимо их в Linux также "

"существуют еще 5 специальных типов.",

{"файл обязательно"}));

EXPECT\_TRUE(Check(

"Типы файлов в Linux | Уроки по Linux - Ravesli\n"

"При навигации по файловой системе в Linux, вы обязательно столкнетесь с "

"различными типами файлов. Наиболее часто используемые и очевидные типы "

"— это обычные файлы и каталоги. Однако помимо их в Linux также "

"существуют еще 5 специальных типов.",

{"файлы"}));

EXPECT\_TRUE(Check(

"Типы файлов в Linux | Уроки по Linux - Ravesli\n"

"При навигации по файловой системе в Linux, вы обязательно столкнетесь с "

"различными типами файлов. Наиболее часто используемые и очевидные типы "

"— это обычные файлы и каталоги. Однако помимо их в Linux также "

"существуют еще 5 специальных типов.",

{"файлов"}));

;

}

TEST(Check, false\_case) {

EXPECT\_FALSE(Check(

"Типы файлов в Linux | Уроки по Linux - Ravesli\n"

"При навигации по файловой системе в Linux, вы обязательно столкнетесь с "

"различными типами файлов. Наиболее часто используемые и очевидные типы "

"— это обычные файлы и каталоги. Однако помимо их в Linux также "

"существуют еще 5 специальных типов.",

{"файл каракули"}));

EXPECT\_FALSE(Check(

"Типы файлов в Linux | Уроки по Linux - Ravesli\n"

"При навигации по файловой системе в Linux, вы обязательно столкнетесь с "

"различными типами файлов. Наиболее часто используемые и очевидные типы "

"— это обычные файлы и каталоги. Однако помимо их в Linux также "

"существуют еще 5 специальных типов.",

{"каракули"}));

EXPECT\_TRUE(

Check("Корреспондента египетского телеканала в прямом эфире сбил скутер, "

"когда он рассказывал о тяжелой ситуации на дорогах",

{"ОИ"})); //Steam(ОИ) == О

}

//

// Create by Kirill Danilchuk <kirill.danilchuk01@gmail.com> on 06/20/2021 21:40

//

#include "SearcherBot.hpp"

int main() {

SearcherBot bot;

return 0;

}