Министерство науки и высшего образования Российской Федерации ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «ОРЕНБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Институт математики и информационных технологий

Кафедра компьютерной безопасности и математического обеспечения информационных систем

**КУРСОВАЯ РАБОТА**

По языкам программирования

**Проектирование и разработка системы логирования событий в мессенджере Telegram**

ОГУ 10.05.01 3025 914 ОО

Руководитель

К.Р. Джукашев

« » 20 г.

Студент группы 23КБ(с)РЗПО-1

А.А. Коновалов

« » 20 г.

Оренбург 2025

Утверждаю

Заведующий кафедрой КБМОИС

                           И.В. Влацкая

«        »                         2024 г.

# ЗАДАНИЕ

**на выполнение курсовой работы**

студенту Коновалову Андрею Алексеевичу

специальности 10.05.01 Компьютерная безопасность

по дисциплине «Языки программирования»

1 Тема работы Проектирование и разработка системы логирования событий в мессенджере Telegram

2 Срок сдачи студентом проекта (работы) «    »   \_\_\_\_\_\_\_      2024 г.

3 Цель и задачи работы Разработать программный продукт для сохранения, упорядочивания и отображения истории событий аккаунта Telegram.

Задачи:

1. Разработать структуру хранения логов.
2. Разработать серверную часть системы логирования
3. Спроектировать и разработать интерфейсную часть с использованием технологии WebApp.
4. Реализовать авторизацию пользователя(ей) в аккаунт Telegram через веб-интерфейс.
5. Обеспечить защиту передаваемых данных посредством протокола https.

4 Исходные данные к проекту (работе): Учебная и научная литература содержащая:

- теоретические и практические сведения о платформе разработки .NET и языке C#;

- теоретические и практические сведение о языке разработки JavaScript;

- информацию об основах криптографии и о системах сквозного шифрования;

- документация Telegram Bot Api и Telegram Api

5 Перечень вопросов, подлежащих разработке:

1. Освоить на практике разработку с использованием языков C# и JavaScript.
2. Изучить методы Telegram Api, позволяющие осуществить логирование событий.
3. Изучить методы Telegram Bot Api, позволяющие встроить WebApp в бота Telegram.
4. Разработать и реализовать программный продукт, обеспечивающий сохранение, упорядочивания и отображения истории событий аккаунта Telegram.

Дата выдачи и получения задания

Руководитель «       »                      2024 г.                      К.Р. Джукашев

подпись инициалы фамилия

Студент «       »                      2024 г.                      А.А. Коновалов

подпись инициалы фамилия

# Аннотация

Курсовая работа посвящена созданию серверного приложения, цель которого записывать все события, происходящие на аккаунте Телеграм, а также разработку веб-интерфейса для просмотра сохранённых записей и управлением конфигурацией сохранения. Основные задачи включают разработку серверного приложения, которое отвечает за сохранение или «логирование» событий, Телеграм бота для управления серверным приложением прямо из мессенджера, а также веб-интерфейса встроенного в Телеграм бота.

Работа содержит … листов текста, …

# Содержание

# Введение

Мессенджер телеграм предлагает функционал удаления сообщений или диалогов для всех собеседников в неограниченное время, в неограниченном количестве. Таким образом, люди могут сами ограничивать использование какой-либо информации собеседниками, если посчитают это нужным.

Но также есть люди, которые используют данный функционал во вред. Например, мошенники, которые не хотят раскрывать свои методики обмана людей, и избавляются от улик удаляя сообщения, или искажая историю сообщений.

В таких случаях было бы прекрасно иметь под рукой инструмент, который позволит сохранять и просматривать историю переписки, со всеми её изменениями.

# Обозначения и сокращения

**Пользователь** – Телеграм аккаунт, с которого происходит взаимодействие с веб-приложением.

**Аккаунт** – Телеграм аккаунт, с которым происходит взаимодействие.

**Сервер** – серверная часть приложения или “бэкенд”.

**Бот** – Телеграм бот с помощью которого пользователь будет открывать веб-приложение.

**Логи** – журнал. Под журналом имеется в виду структура хранения записей в базе данных. Так же может означать и записи о работе приложения.

**Лог** – запись в журнале. Под записью имеется в виду строка в одной из таблиц базы данных. Так же может означать запись о работе приложения.

**Цель** – Чат аккаунта, чьи события будут сохраняться в логи.

# Техническое задание

Разработать программный продукт интегрированный в мессенджер Телеграм по средствам Telegram API и Telegram Bot Api.

* Создать Телеграм бота.

Управление должно быть реализовано с использованием веб-приложения или Web-App. Через это приложение можно будет ввести свои данные для входа в аккаунт Телеграм и отслеживать события, происходящие на этом аккаунте через это веб-приложение. Один пользователь должен иметь возможность добавить несколько аккаунтов.

* Разработать веб-сайт для мобильного формата.
* Создать кнопку в Телеграм боте для открытия сайта как веб-приложения внутри Телеграм.

Авторизация и аутентификация. Определять от какого пользователя пришёл запрос и проверять не самозванец ли этот пользователь.

* Создание сессии
* Проверка подписи

Сервер должен запускать сессии пользовательских аккаунтов и логировать (запоминать) события, пришедшие на них. Логи должны быть дополняемыми. *Пример: пришло сообщение – лог; сообщение было отредактировано – новый лог со ссылкой на старый (родительский) лог.*

* Разработать систему хранения событий.
* Написать приложение для логирования событий.
* Написать веб-сервер (бэкенд) для предоставления информации (логов) для веб-приложения.
* Реализовать поддержку изменения конфигурации «на горячую» через веб-приложение. В конфигурацию входят: аккаунты для отслеживания, изменение логируемых чатов. Должна быть возможность редактировать эти параметры без перезапуска приложения.

# Выбор методов, способов и средств разработки

Серверное приложение будет разработано на платформе ASP.NET8.0. Благодаря веб-серверу можно будет легко взаимодействовать с серверной частью приложения. ASP.NET предоставляет удобные способы реализации REST API.

Для реализации получения событий с Телеграм аккаунта будет использоваться библиотека WTelegram для C#. Её код находится в открытом доступе и её использование бесплатно. Библиотека предоставляет удобный интерфейс для взаимодействия с Telegram API.

Для работы с базой данных будет использоваться Entity Framework, позволяющий удобно работать с данными из БД в представлении обычных объектов классов.

Взаимодействие пользователя с программой будет в веб-приложении, встроенном в Телеграм бота. Веб-приложение будет написано с использованием библиотеки React для JavaScript.

Веб-приложение будет развёрнуто на GitHub pages. Это бесплатный и простой способ развёртывания веб-страницы с SSL сертификатом.

Серверная инфраструктура (приложение и СУБД) будет упакована в docker-compose, что позволит быстро разворачивать его на любом сервере.

# Описание алгоритмов

## Использование технологии WebApp

Открывающееся в Телеграме веб-приложение это по сути своей обычный сайт. Отличие лишь в том, что подключена библиотека, которая выполняет функции авторизации и вызовом некоторого функционала Телеграм. Подробнее о функционале можно узнать в официальной документации Telegram mini apps.

Авторизация через Телеграм позволит не реализовывать свою авторизацию, позволит не хранить хеши паролей и т.д. в таблице БД с пользователями мы будет просто хранить *user\_id* для каждого пользователя, который взаимодействовал с ботом.

В подключаемой библиотеке для WebApp есть поле *initData*. Это поле хранит строку с данными о пользователе, откуда и когда он открыл веб-приложение и хеш этой *initData*. Этот хеш используется для аутентификации данных. Благодаря нему, посторонний не сможет воспользоваться нашим API.

Для того, чтобы проверить пришедшую на бэкенд *initData*, нужно выполнить следующие шаги:

* Получить *data\_check\_string* путём сортировки всех полей полученных данных по ключу и записи их в виде:

<Ключ[1]>=<Значение[1]>

<Ключ[2]>=<Значение[2]>

…

<Ключ[n]>=<Значение[n]>

* Получить хеш для *secretKey* по алгоритму HMAC-SHA256, где ключ — это константная строка *“WebApp”*, а хешируемые данные – секретный API ключ Телеграм бота.
* Получить финальный хеш по алгоритму HMAC-SHA-256, где ключ это *secretKey*, а данные это *dataCheckString*.

Финальный хеш нужно сравнить с тем, что находится в полученной *initData*. При несовпадении этих хешей, отбрасываем запрос с этой *initData*. Также при использовании WebApp доступен различный функционал интеграции с телеграм ботом, но в этом проекте он не потребуется.

## Технология http и https

**HTTP (HyperText Transfer Protocol)** — это протокол прикладного уровня, который определяет формат обмена данными между клиентом и сервером. Он используется для передачи нешифрованной информации: текста, изображений, мультимедиа. По умолчанию HTTP использует порт 80 и не обеспечивает защиту данных — информация передается в открытом виде, что делает ее уязвимой для перехвата или подмены.

HTTP-сообщение (запрос или ответ) состоит из трех основных частей: стартовой строки, заголовков и тела. Они разделяются символами переноса строки (\r\n). Формат зависит от типа сообщения — запрос от клиента или ответ от сервера.

<Стартовая строка>

<Заголовки>

<Пустая строка>

<Тело> (опционально)

**Стартовая строка** содержит метод запроса, URI (путь к ресурсу) и версию HTTP. **Заголовки** - параметры, описывающие запрос: тип данных, браузер, куки и т.д. **Тело** используется для передачи данных. Например: сервер возвращает html разметку страницы.

**HTTPS (HyperText Transfer Protocol Secure)** — это расширение HTTP с поддержкой шифрования для защиты данных. Он использует криптографические протоколы SSL/TLS, которые шифруют трафик между клиентом и сервером. Таким образом осуществляется защита данных от чтения третьими лицами и предотвращает изменение данных. Для работы HTTPS требуется SSL-сертификат, выпущенный доверенным центром сертификации (CA), который подтверждает подлинность сервера. Перед тем как установить защищённое соединение, браузер запрашивает этот документ и обращается к центру сертификации, чтобы подтвердить легальность документа. Если он действителен, то браузер считает этот сайт безопасным и позволяет обмен данными. Этот процесс называется рукопожатием (handshake): клиент и сервер согласовывают параметры шифрования, проверяют сертификат, генерируют общий сеансовый ключ. Сеансовый ключ служит одновременно как для шифрования, так и для дешифрования передаваемых данных. Только после этого передаются зашифрованные данные.

## Технология CORS

CORS — это механизм, позволяющий безопасно выполнять междоменные HTTP-запросы. Он работает через взаимодействие браузера и сервера с использованием специфичных заголовков. Ниже описаны ключевые алгоритмы и этапы работы CORS.

Сначала идёт определение типа запроса: браузер проверяет, относится ли он к категории «простых». Простым считается запрос, который использует методы GET, HEAD или POST, содержит только стандартные заголовки (Accept, Content-Type с разрешенными значениями *application/x-www-form-urlencoded*, *multipart/form-data*, *text/plain*) и не включает пользовательские заголовки. Если запрос не соответствует этим критериям, браузер автоматически отправляет предварительный запрос типа OPTIONS для проверки разрешений сервера.

Предварительный запрос (preflight) содержит метаданные: заголовок *Origin* с доменом клиента, *Access-Control-Request-Method* с методом основного запроса и *Access-Control-Request-Headers* со списком пользовательских заголовков. Сервер должен ответить заголовками *Access-Control-Allow-Origin* (разрешенные домены), *Access-Control-Allow-Methods* (список методов), *Access-Control-Allow-Headers* (разрешенные заголовки) и *Access-Control-Max-Age* (время кэширования ответа). Если сервер подтверждает права доступа, браузер выполняет основной запрос. В противном случае запрос блокируется с ошибкой CORS.

## Что такое REST API

**API** — это набор правил, протоколов и инструментов, которые позволяют различным программным компонентам взаимодействовать друг с другом. Он определяет, как и какими методами одна программа может запрашивать данные или функционал у другой, не вникая в её внутреннюю реализацию. API выступает посредником между системами, упрощая интеграцию и обмен информацией.

**REST (Representational State Transfer)**: основан на HTTP-методах (GET, POST, PUT, DELETE). Данные передаются в формате JSON или XML.

## Вид пакета REST API

Первым определением для REST API является определением вида пакета. Вид пакета определяет то, в каком виде серверная часть возвращает данные и как их обрабатывать.

Пакет данных для общения между клиентом и сервером всегда будет включать в себя поля *signature* и *ts*, в независимости от того, кто отправил этот пакет. Пакет может содержать прочие поля в зависимости от вызываемого метода на сервере. Каждый класс, представляющий пакет данных наследуется от базового класса Backend.Tools.HttpDataBase. Этот класс предоставляет два обязательных поля *ts* и *signature* и методы для работы с пакетом:

Чтобы базовый пакет мог спокойно подписать данные, все пользовательские классы, которые могут содержаться в пакете, должны наследоваться от базового класса Backend.Tools.ParsebleToDictionaryBase. Этот класс предоставляет реализацию метода ToDict, который все свойства класса записывает в структуру Dictionary<string, object?>. Также доступно переопределение этого метода. Этот метод понадобиться при подписи пакета, подробнее в одном из следующих разделов. В этом же методе некоторые типы данных преобразуются в особый вид. Например:

* Тип DateTime, конвертируется в тип long, означающий unix time формат с миллисекундами.
* Если значение свойства приводится к Backend.Tools.ParsebleToDictionaryBase, то в результат добавляется результат вызова ToDict для значения этого свойства.
* Если значение свойства – список с элементами, которые приводятся к Backend.Tools.ParsebleToDictionaryBase, то для каждого элемента вызывается ToDict и массив результатов вызовов ToDict добавляется в результат
* Остальные типы в результат добавляются «как есть».

## Использование сервисов в ASP.NET

Сервисы в ASP.NET это мощнейший инструмент, который предоставляет возможность написания сервисов, работающих на фоне или использования DI (Dependency Injection). Регистрация сервисов происходит с помощью методов AddTransient (что сервис создается каждый раз, когда его запрашивают), AddScoped (сервис создаются единожды для каждого запроса), AddSingleton (единственный экземпляр на всё время работы приложения) или AddHostedService (запуск фоновой задачи).

Для реализации сервиса, работающего на фоне, необходимо создать класс, который наследуется от базового класса BackgourndService или от интерфейса IHostedService. Разница в уровне абстракции:

IHostedService – это низкоуровневый интерфейс, который требует ручной реализации двух методов: StartAsync(CancellationToken) – запускает службу при старте приложения и StopAsync(CancellationToken) – корректно останавливает службу при завершении работы.

BackgroundService – это надстройка над IHostedService, которая упрощает создание фоновых задач. Он предоставляет один единственный метод ExecuteAsync, где размещается основная логика, а завершается задача при переключении CancellationToken.

BackgorundService удобнее, т.к. нет необходимости следить за потоком выполнения. Пока метод ExecuteAsync не завершён – сервис запущен, значит ExecuteAsync и есть поток выполнения.

В программе присутствуют два сервиса: CoreService (для выполнения незначительных задач) и LoggingService (выделенный в отдельный сервис функционал логирования событий от телеграм).

## Алгоритм использования Entity Framework Core в приложении

Entity Framework Core (EF Core) — это объектно-реляционный маппер (ORM), упрощающий взаимодействие с базами данных через абстракции C#. Для работы с определёнными БД нужно установить NuGet пакеты Microsoft.EntityFrameworkCore и провайдера для выбранной СУБД. Для данного проекта выбрана база данных MySql и провайдером был выбран Pomelo.EntityFrameworkCore.MySql.

Работа с данными осуществляется через классы, определяющие строение БД. EntityFramework способен сам создать таблицы и зависимости между ними, нужно только описать сущности и их связи.

Работать с данными можно с помощью контекста данных, наследующегося от класса DbContext. В нём определяются свойства типа DbSet<T>, где тип T это ранее упомянутые сущности. В этом же классе контексте настраивается подключение к БД.

EFCore поддерживает миграции. Миграции в EFCore — это механизм версионирования и синхронизации структуры базы данных с моделью данных приложения. Они автоматически генерируют код для преобразования текущей схемы БД в новое состояние, заданное изменениями в классах сущностей. Миграции позволяют применять, откатывать и отслеживать эволюцию базы данных в рамках жизненного цикла приложения.

Выборку данных можно осуществлять с помощью LINQ, а механизм отслеживания изменений ChangeTracker предоставляет удобное изменение данных.

## Авторизация и аутентификация

Авторизация будет осуществляться по заголовку *userId* в запросе к серверу, а аутентификация по подписи пакета. Подписью пакета назовём результат хеш функции от данных, преобразованных в строку определённого вида, по специальному ключу. Этот ключ будем называть кодом сессии, или *sessionCode*.

Получить этот код клиент может при отправке запроса на метод API auth/login с данными *initData*. При отправке запроса, сервер зарегистрирует пользователя и сгенерирует специальный код сессии, который будет активен ограниченное время. Сбросить время жизни можно вызовом любого метода или же вызовом специального метода auth/ping. Важно, при вызове auth/login, в заголовках должен присутствовать *userId* со значением (-1), для обозначения того, что клиент ещё не авторизован. В случае успеха, сервер отдаст ответ, в котором будет находиться *sessionCode* и объект *me*. В объекте *me* будет находиться *userId* – идентификатор пользователя телеграм, который открыл веб приложение. Следующие запросы должны содержать в заголовках *userId* с полученными идентификатором пользователя.

Аутентификация будет осуществляться через подпись отправляемых данных.

Как сказано ранее, для подписи нужно преобразовать данные в строку определённого вида:

userId:{userId}\_ts:{ts}\_data.arrays.0=firstElementOfArray&data.arrays.1=secondElementsOfArray&data.someData=someValue&data.someObject.keyInObject=valueFromObject…

Как видно, сначала всегда будет идти идентификатор пользователя (*userId*) и время создания пакета (*ts*). Назовём эту часть строки заголовком. Разделителем заголовков является нижнее подчёркивание. После каждого параметра заголовка ставится нижнее подчёркивание, даже если после последнего параметра нет других данных. После заголовка начинаются сами данные. Данные записаны в формате search params, т.е. так, как бы мы их отправляли GET запросом.

Ключ=Значение

Разделителем является символ амперсанда. Каждый ключ содержит вложенную структуру к этим данным. Т.е. указываются все имена ключей объектов и номера индексов массивов, по которым мы можем получить данные. Разделителем ключей является точка. Самый первый (корневой) ключ именуется как *data*. Таким образом мы получаем общий ключ для наших данных, который мы записываем в строку. Повторяем операцию для каждых данных и получаем строку, для которой нужно применить хеш функцию.

Ключом для хеш функции выступает код сессии. Результат хеш функции кодируем в HEX и это и есть наша подпись.

Чтобы сторона, принимающая пакет, знала о подписи, её нужно записать в поле *signature*. Таким образом, принимающая сторона сможет проверить подпись, самостоятельно подписав пакет и сравнив локально полученную подпись, с той, что пришла вместе с пакетом. Если подписи совпадают – аутентификация пройдена и данные валидны.

## Сопоставление запросов и кодов сессий

Для упрощения получения актуального кода сессии для пользователя на стороне сервера, чтобы в дальнейшем проверять пришедшие пакеты и подписывать новые, используется инструмент ASP.NET называемый Middleware.

Middleware представляет собой делегат принимающий HttpContext – контекст запроса, делегат на следующий middleware и возвращает Task. Таким образом, создаётся конвейер обработки запросов и откликов, где каждый компонент:

* Определяет, нужно ли передать запрос следующему компоненту в конвейере
* Может выполнять работу как до, так и после следующего компонента в конвейере

На рисунке 1 приведена схема работы конвейера middleware.



Рисунок 1 – наглядная схема работы конвейера Middleware

Таким образом, использование этого инструмента даёт возможность написать свой собственный промежуточный слой, который в свою очередь будет внедрять актуальный код сессии в заголовок запроса, или отбрасывать запрос при несоблюдении правил. Схема такого промежуточного слоя представлена на рисунке 2.

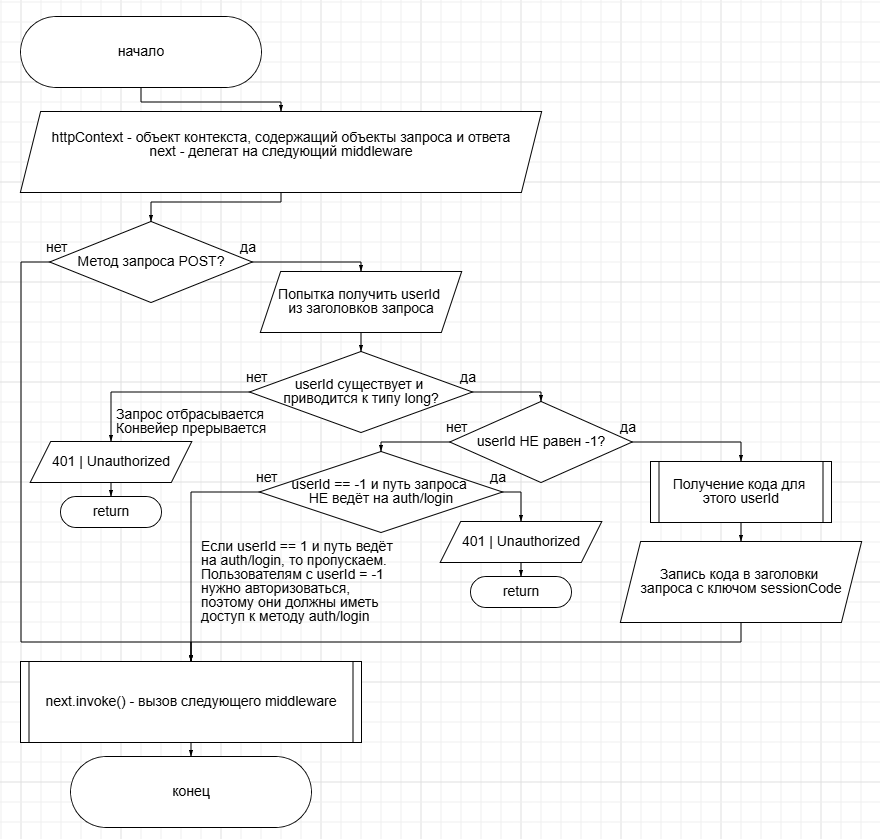


Рисунок 2 – схема работы промежуточного слоя авторизации

Все методы, отличные от POST пропускаются без проверок, т.к. методы API на сервере работают только через POST. Если в заголовках запроса отсутствует *userId*, то такой запрос отбрасывается. Если *userId* имеет значение   
(-1), но запрос не нацелен на auth/login – запрос отбрасывается. В иных случаях – правила соблюдены и запрос пропускается.

## Управление аккаунтами Telegram

Чтобы подключиться к серверам телеграм, используется библиотека wiz0u/WTelegram. Для использования методов клиента, нужно создать объект класса Client. В библиотеке WTelegram есть несколько способов авторизации.

1. Подключиться к серверам телеграм и последовательно передавать в объект клиента данные (номер телефона, код верификации и т.д.), тогда в конструктор класса Client, нужно передать API\_ID, API\_HASH и путь до директории хранения сессий. API\_ID и API\_HASH являются идентификаторами приложения. Они создаются на сайте <https://my.telegram.org/auth> и привязываются к аккаунту. Эти два параметра необходимы для использования серверов телеграм.
2. Передать в конструктор функцию обратного вызова, которая будет возвращать запрашиваемые клиентом значения. Среди запрашиваемых значений могут быть API\_ID, API\_HASH, номер телефона или путь до директории хранения сессий. Удобно использовать этот способ для уже сохранённых сессий, вход в которые был осуществлён ранее.

ASP.NET предоставляет возможность запускать собственные сервисы. Благодаря этому, предоставляется возможность написания сервиса, который будет следить за запущенными клиентами Telegram.

За всеми запущенными аккаунтами будет следить AccountManager. Каждый аккаунт находится в объекте класса LoadedAccount, а в нём в свою очередь определены свойства *LastTrigger*, которое хранит время последнего обращения к этому клиенту и *InUse*, которое при значении true говорит о том, что клиент находится в работе/занят. Эти поля нужны для регуляции запущенных клиентов. Если клиент запущен некоторое время, и за это время ни разу не использовался, то его лучше завершить и выгрузить из памяти.

## Взаимодействие пользователя с программой

Для взаимодействия пользователя с системой логирования предоставлен интерфейс в виде WebApp (далее приложение).

При первом открытии приложения пользователя встретит окно входа в аккаунт. Оно представляет собой поле ввода для номера телефона. После отправки номера телефона, на телеграм аккаунт с этим номером приходит код, а страница приложения запрашивает уже его. При установленном облачном пароле на аккаунте, запрашивается и он. Таким образом, пользователь создал сессию аккаунта на сервере, и теперь сервер может производить манипуляции с этим аккаунтом.

При повторном открытии приложения пользователя встречает список аккаунтов, в которые он авторизовался на сервере. При выборе аккаунта, появляется список диалогов доступных для отслеживания. Справа от диалога находится чекбокс. При нажатии на него, этот диалог помечается как «цель». С этого момента, все пришедшие события на этот аккаунт для этого диалога будут сохранены.

Просмотреть сохранённые события можно при нажатии на диалог. Открывается страница, похожая на обычный чат. Справа сообщения, отправленные от самого аккаунта, а слева от других участников. Если событие о сообщении пришло единожды, т.е. оно не изменялось и не удалялось, то никаких визуальных пометок оно в себе содержать не будет. Если же, сообщение было как-то изменено или удалено, в нижней части сообщения будет кнопка открытия истории. Под историей понимается история событий, связанных с этим сообщением. При нажатии на кнопку откроется список всех событий с этим сообщением.

Чтобы сервер понимал, какие чаты аккаунта нужно отслеживать, разработана система «целей». После добавления аккаунта, при клике на него открывается список диалогов аккаунта. Слева от аватара диалога, находится чекбокс. Если чекбокс выбран, то этот чат отслеживается. При изменении состояния чекбокса, отправляется запрос на сервер на метод *target/updateTarget* с данными о состоянии чекбокса, и в зависимости от состояния чекбокса, сервер помечает данный диалог как цель и начинает отслеживать, или же наоборот – перестаёт отслеживать.

## Разработка на React

React — это популярная JavaScript-библиотека с открытым исходным кодом, для построения пользовательских интерфейсов (UI). Она ориентирована на компонентный подход, где каждый элемент интерфейса (кнопка, форма, страница) представляет собой независимый, переиспользуемый компонент с собственной логикой и состоянием. Это позволяет разработчикам создавать сложные приложения, комбинируя простые блоки, что упрощает поддержку и масштабирование кода.

Одной из ключевых особенностей React является использование виртуального DOM (Document Object Model). Вместо прямого взаимодействия с реальным DOM, React создает его легковесную копию в памяти. При изменении состояния компонента библиотека сравнивает виртуальный DOM с актуальным и обновляет только те части интерфейса, которые действительно изменились. Это значительно повышает производительность, особенно в приложениях с динамическим контентом, таких как соцсети или дашборды аналитики.

Для использования библиотеки, необходимо установить его через пакетный менеджер npm, который в свою очередь поставляется вместе с Node JS. Создаётся пустое приложение по умолчанию через программу **Create React App**, в которое уже можно добавлять новые элементы, страницы и т.д.

После создания проекта черези npx create-react-app my-app генерируются папки и файлы:

my-app/

├── node\_modules/ # Зависимости проекта (npm/yarn)

├── public/ # Статические файлы

│ ├── index.html # Главный HTML-шаблон

│ └── favicon.ico # Иконка сайта

├── src/ # Исходный код приложения

│ ├── App.js # Главный компонент приложения

│ ├── App.css # Стили для App.js

│ ├── index.js # Точка входа (подключение React к DOM)

│ └── index.css # Глобальные стили

├── .gitignore # Игнорируемые файлы для Git

├── package.json # Метаданные и скрипты проекта

└── README.md # Описание проекта

**App.js** — корневой компонент, куда добавляются другие компоненты (например, хедер, страницы).

**index.js** — рендерит компонент <App /> в элемент DOM с id="root".

Для удобства код организуют в тематические папки. Пример:

src/

├── api/ # Файлы для работы с API (запросы к серверу)

├── assets/ # Статические ресурсы: шрифты, SVG, изображения

├── components/ # Переиспользуемые UI-компоненты (кнопки, формы)

│ ├── Button/

│ │ ├── Button.jsx

│ │ └── Button.module.css

│ └── Header.jsx

├── context/ # Контексты React (глобальное состояние)

├── hooks/ # Кастомные хуки (например, useFetch)

├── pages/ # Страницы приложения (роуты)

│ ├── Home.jsx

│ └── Profile.jsx

├── routes/ # Конфигурация маршрутизации (React Router)

├── styles/ # Глобальные стили, темы, переменные CSS

├── utils/ # Вспомогательные функции (форматирование дат, валидация)

├── App.jsx # Главный компонент

└── index.js # Точка входа

Глобальные стили хранятсяв index.css или styles/global.css. Стили компонентов рядом с компонентом. (как в примере выше).

# Описание программы

## Модули и **файлы программного продукта**

### База данных

MySql – реляционная база данных, которая необходима для хранения пользовательских данных и состояния программы. Довольно надёжна, быстрое чтения и проста в установке и конфигурации, поэтому выбрана именно она.

### Серверная часть

TelegramUpdatesLoggerBackend – приложение которое логирует события и предоставляет API для управления логированием.

Таблица 1 – содержание решения TelegramUpdatesLoggerBackend

|  |  |  |
| --- | --- | --- |
| **Название модуля** | **Содержание и назначение** | **Видимость** |
| Config | Предоставляет конфигурацию, заданную при запуске. | Виден всем |
| Backend | ASP.NET приложение, предоставляющее REST API и логирование событий Telegram. | Главный. Не виден никому. |
| Core | Предоставляет всю логику логирования. (Запускается из Backend) | Виден для Backend |
| Database | Предоставляет интерфейс работы с БД по средствам EFCore. | Виден для Core и Backend |
| Docker-compose | Оркестратор контейнеров Docker. Нужен для удобного развёртывания на сервере. | Главный. Не виден никому. |

### Клиентская часть

telegram-logger-app – веб-приложение с использованием библиотеки React, которое предоставляет интерфейс взаимодействия с приложением логирования.

Таблица 2 – компоненты telegram-logger-app использующие стили

|  |  |
| --- | --- |
| **Название компонента** | **Назначение** |
| BackButton | Элемент управления «Назад» для возврата на предыдущую страницу. |
| Loading | Объект загрузки страницы. |
| AccountItem | Объект аккаунта. |
| DialogItem | Объект диалога (переписка) в аккаунте. |
| MessageItem | Объект сообщения. |
| HistoryLog | Объект истории изменения конкретного сообщения. |

В приложении telegram-logger-app существуют несколько страниц для отображения данных. Страницы приведены в таблице 3.

Таблица 3 – страницы telegram-logger-app

|  |  |
| --- | --- |
| **Название страницы** | **Назначение** |
| LoadingPage | Страница загрузки данных, с заголовком, текстом и анимацией. |
| NewAccountPage | Страница добавления аккаунта. |
| ErrorPage | Страница ошибки с описанием ошибки. |
| AccountsPage | Страница аккаунтов. На этой странице отображены добавленные диалоги. |
| DialogsPage | Страница диалогов конкретного аккаунта. На этой странице отображены все диалоги, доступные для отслеживания с выбранного аккаунта. |
| MessagesPage | Страница сообщений для конкретного диалога. На этой страницы отображены все сохранённые сообщения. |
| ChangeHistoryPage | Страница изменения сообщения. На этой страницы отображены все изменения конкретного сообщения. |

## Устройство серверной части

### Хранение конфигурации

Для хранения конфигурации используется класс ProgramConfig. Он предоставляет get свойства с информацией. Информация загружается статическим конструктором из переменных окружения, а переменные окружения в свою очередь загружает оркестратор контейнеров docker-compose из .env файла.

### Описание устройства базы данных

Для создания структуры БД использован фреймворк Entity Framework Core. Этот фреймворк позволяет представлять данные из БД в виде сущностей, которые описаны классами. Для работы с БД, создаётся класс, наследуемый от DbContext и называемый контекстом. Создаётся класс, называемый сущностью, в котором свойствами описываются столбцы БД. Чтобы создать таблицу в БД, надо в контекст добавить свойство типа DbSet<T>, где T принимает сущность.

В перегрузке метода OnConfiguring в классе контексте описывается подключение к БД. В перегрузке метода OnModelCreating – правила создания таблиц. Эта перегрузка используется, когда нужно уточнить правила создания таблиц, если EF Core не может самостоятельно их определить или возник конфликт, или же если нужно определить собственные связи между таблицами.

На таблице 4 представлено описание сущностей БД.

Таблица 4 – описание сущностей БД

|  |  |
| --- | --- |
| **Сущность БД** | **Назначение** |
| User | Объект пользователя. Нужен для разграничения прав. |
| Session | Объект сессии. Нужен для того, чтобы удерживать открытую сессию. Часть авторизации. |
| Account | Объект аккаунта. Нужен для учёта аккаунтов, к которым у приложения есть доступ. |
| AccountLog | Объект лога аккаунта. Логом выступает информация о том, когда к аккаунту был получен доступ, когда он запущен и т.д. |
| LoggingTarget | Объект цели для отслеживания. Нужен, чтобы приложение понимало, какие диалоги нужно отслеживать. |
| UpdateLog | Объект лога события. Она содержит базовую информацию для лога, идентификатор, цель. |
| UpdateMessageLog | Объект лога события сообщения. Наследуется от UpdateLog и содержит дополнительную информацию о событии, текст сообщения, отправитель и т.д. |
| UpdateDeleteMessageLog | Объект лога события удаления сообщения. Наследуется от UpdateLog и содержит идентификатор удалённого сообщения |

Данные сущности образуют таблицы БД Users, Sessions, Accounts, AccountHistory, Targets, Updates, UpdatesMessage, UpdatesDeleteMessage соответственно.

### Устройство пакета REST API

На таблице 5 представлены классы, необходимые для работы представления данных REST API.

Таблица 5 – описание классов для работы с пакетами REST API

|  |  |  |
| --- | --- | --- |
| **Название класса** | **Назначение** | **Родитель** |
| ParsebleToDictoinaryBase | Базовый класс для данных, которые могут попасть в пакет REST API. Предоставляет реализацию конвертации в словарь. | - |

Продолжение таблицы 5

|  |  |  |
| --- | --- | --- |
| HttpDataBase | Базовый класс для пакета REST API. Предоставляет реализации подписи пакета, проверки подписи. | ParsebleToDictoinaryBase |
| HttpEmtpyData | Пакет с пустыми данными. | HttpDataBase |
| HttpErrorData | Пакет с данными об ошибке. С полями статус кода и описанием ошибки. | HttpDataBase |

Все пакеты данных должны наследоваться от класса HttpDataBase. Все классы, которые содержаться в данных, должны наследоваться от ParsebleToDictionaryBase. Таким образом достигается правильное преобразование данных в строку для подписи, а значит, подпись осуществляется корректно.

На таблице 6 приведены методы базового класса пакета. Пришедший пакет необходимо проверить методом Verify, а новый подписать перед отправкой методом Sign. Метод ToObjectResult служит для приведения к типу ObjectResult.

Таблица 6 – методы базового класса HttpDataBase

|  |  |
| --- | --- |
| **Название метода** | **Назначение** |
| ToObjectResult() | Приведение к словарю, а затем к типу ObjectResult |
| Sign(long userId, string sessionCode) | Подписать пакет |
| Verify(long userId, string sessionCode) | Проверить подпись пакета |

Единственный метод класса ParsebleToDictionaryBase: ToDict служит для преобразования данных в словарь. При том, имена свойств преобразуются в lowerCamelCase, а для некоторых типов данных преобразование осуществляется определёнными образом.

Если свойство имеет тип данных DateTime, то его нужно преобразовать во временную метку в миллисекундах Unix timestamp. Эта временная метка указывает кол-во миллисекунд, прошедших с момента 1 января 1970 года.

Если свойство имеет тип даных ParsebleToDictionaryBase, то для этого свойства нужно вызвать метод ToDict и уже эти преобразованные данные добавить в результат.

Если свойство имеет тип данных IEnumerable< ParsebleToDictionaryBase > (перечисление объектов типа ParsebleToDictionaryBase), то для каждого объекта нужно вызвать ToDict и полученных массив преобразованных данных добавить в результат.

И если ни одно из предыдущих условий не сработало, то данные можно добавлять сразу в результат без преобразования.

### Подпись пакета

На таблице 7 приведено описание приватных методов класса HttpDataBase.

Таблица 7 – приватные методы класса HttpDataBase

|  |  |
| --- | --- |
| **Название метода** | **Назначение** |
| GetSortedParams | Используется для получения отсортированных по ключу данных, где каждый ключ имеет вложенную структуру (см. **Авторизация и аутентификация**) |
| DefaulTransform | Статический метод, который рекурсивно разворачивает вложенную структуру словаря и помещает данные в сортированный словарь. |

Приведением данных к виду, который можно подписать, занимается метод GetSortedParams. Этот метод получает пакет в виде словаря, с помощью ранее рассмотренного метода ToDict, далее эти данные трансформируются с помощью метода DefaultTransform.

DefaultTransform проходит по каждому элементу исходного словаря, и если данные представляют собой словарь, то рекурсивно вызывает себя уже для этого словаря. При этом DefaultTransform одним из аргументов принимает также строку, содержащую предыдущий (родительский) ключ. Таким образом, второй вызов DefaultTransform перед каждым ключом словаря будет дописывать родительский ключ в результат.

Если элементом словаря является список, то этот список преобразуется в словарь, где ключи — это индексы списка. Далее, по аналогии, для полученного словаря также рекурсивно вызывается DefaultTransform.

Далее, для каждого элемента выполняется попытка десериализовать этот элемент из Json. При неудаче, элемент записывается «как есть».

Таким образом, после сворачивания всех рекурсий метод GetSortedParams возвращает данные, готовые для подписи.

Для подписи, данные преобразуются в строку в виде searchParams, с помощью LINQ запроса, а для валидации подписи, из полученных данных убираются поля signature и ts, выполняется хеширование и сравнение хешей. При несовпадении хешей выбрасывается исключение. Хеширование выполняется алгоритмом HMAC.SHA256, где ключом выступает строка sessionCode.

### Методы REST API

Сервер имеет три основных раздела методов: Auth, Account, Target. Описание всех методов каждого раздела приведены в таблице 8

Таблица 8 – Методы API

|  |  |  |
| --- | --- | --- |
| **Раздел** | **Метод** | **Назначение** |
| auth | logIn | Генерирует для этого пользователя специальный sessionCode по переданной initData |

Продолжение таблицы 8

|  |  |  |
| --- | --- | --- |
| auth | ping | Продлевает время жизни sessionCode |
| logOut | Метод не определён |
| account | newAccount | Добавление нового аккаунта. Ввод номера телефона |
| setCode | Добавление нового аккаунта. Ввод кода |
| setPassword | Добавление нового аккаунта. Ввод облачного пароля |
| getMyAccounts | Получение всех добавленных аккаунтов |
| getDialogs | Получение диалогов, доступных для отслеживания с аккаунта |
| target | updateTarget | Метод обновления цели. Вкл/Выкл. |
| getSavedLogs | Получение сохранённых сообщений для цели. |

### Сервис CoreService

CoreService назван так, потому что является главным сервисом, который запускает сущности, наследуемые от интерфейса IWorker. Таким образом Серверная часть выделяет только один фоновый сервис для неограниченного кол-ва исполнителей.

Интерфейс IWorker объявляет метод   
Task Handle(ApplicationContext context), где в качестве аргумента передаётся контекст базы данных, а возвращаемым типом является Task, механизма асинхронности C#. Таким образом, метод может поддерживать выполнение асинхронных задач, например для записи в базу данных.

Основными исполнителями являются LoadedAccountsWorker и SessionWorker. Эти исполнители занимаются отслеживанием запущенных аккаунтов и отслеживанием времени жизни сессий.

### Сервис LoggingService

LoggingService занимается отслеживанием приходящих обновлений для аккаунтов и выделен в отдельный, независимый фоновый сервис для надёжности.

LoggingService имеет метод Update для обновления конфигурации «на горячую». Таким образом, после каждого вызова target/updateTarget вызывается метод Update у LoggingService. В этом методе из БД выбираются все **последние** логи обновлений с группировкой по **PeerId**, а затем, из этой выборки выбираются только те, что включены с группировкой для каждого отдельного аккаунта. Такой порядок действий необходим для упорядочивания данных именно в таком формате. Далее метод запускает аккаунты и устанавливает для них обработчик обновлений. Если аккаунт был запущен ранее и логгеры уже установлены, то устанавливать для аккаунта обработчик обновлений заново не нужно, а логгеры выставляются заново. Каждый логгер представляет класс UpdatesLogger, который хранит цель логирования, аккаунт и экземпляр запущенного аккаунта. При получении обновления для аккаунта, из словаря всех логгеров, по ключу представляющий собой идентификатор аккаунта в БД, выбирается список всех логгеров этого аккаунта. Далее вызывается метод GetLogger находящийся у LoggingService, который из этого списка логгеров выбирает логгер для PeerId к которому пришло обновления. Если логгер для этого PeerId отсутствуют, то обновление пропускается.

Если пришедшее обновление это обновление о удалении сообщения, то для того, чтобы определить логгер, нужно достать из БД тот PeerId которому принадлежит пришедший messageId. Это необходимо т.к. обновление о удалении сообщения не содержит данных о PeerId.

Экземпляры класса UpdatesLogger накапливают сообщения внутри себя, а с помощью вызова метода Save сохраняют накопленные логги в БД. Именно этот метод вызывает сервис LoggingService каждую итерацию своей работы.

## Устройство клиентской части

### Отправка запросов к серверу

Методы каждого раздела API располагается в соответствующем файле по пути src/Tools/Api/. Все эти разделы собраны в одном месте в файле src/Tools/Api.js. Подпись пакетов реализована в файле src/Tools/signature.js, аналогично тому, как она реализована в серверной части.

### Отрисовка страниц

В главном файле приложения src/App.js реализован метод drawPage, который принимает компонент для отрисовки, и меняет состояние главной страницы на только что принятый компонент. Таким образом, механизм библиотеки React перерисовывает страницу с использованием нового компонента.

По умолчанию, первой страницей при загрузке является страница загрузки.

# Приложение А

**Backend/Program.cs**

using Backend.Controllers.Auth;

using Backend.Controllers.Cors;

using Backend.Services;

using Database;

using Microsoft.EntityFrameworkCore;

namespace Backend

{

public class Program

{

public static readonly string ApiVer = "v1";

public static void Main(string[] args)

{

new ApplicationContext().Database.Migrate();

var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddDbContext<ApplicationContext>();

builder.Services.AddHostedService<CoreService>();

builder.Services.AddHostedService<LoggingService>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

var app = builder.Build();

app.UsePathBase("/api/" + ApiVer);

app.UseHttpsRedirection();

app.Use(CorsController.InsertHeaders); // Custom cors

// Только так у меня получилось протащить заголовки,

// разрешающие CORS, через прокси keenDNS.

// При использовании CORS через Services, OPTIONS

// запросы не проходили из-за чего браузер блокировал

// ответы.

app.Use(AuthController.CustomAuthorization);

app.MapControllers();

app.Run();

}

}

}

**Backend/Dockerfile**

# См. статью по ссылке https://aka.ms/customizecontainer, чтобы узнать как настроить контейнер отладки и как Visual Studio использует этот Dockerfile для создания образов для ускорения отладки.

# Этот этап используется при запуске из VS в быстром режиме (по умолчанию для конфигурации отладки)

FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base

USER root

WORKDIR /app

EXPOSE 8080

EXPOSE 8081

# Этот этап используется для сборки проекта службы

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build

ARG BUILD\_CONFIGURATION=Release

WORKDIR /src

COPY ["Core/Backend.csproj", "Core/"]

RUN dotnet restore "./Core/Backend.csproj"

COPY . .

WORKDIR "/src/Core"

RUN dotnet build "./Backend.csproj" -c $BUILD\_CONFIGURATION -o /app/build

# Этот этап используется для публикации проекта службы, который будет скопирован на последний этап

FROM build AS publish

ARG BUILD\_CONFIGURATION=Release

RUN dotnet publish "./Backend.csproj" -c $BUILD\_CONFIGURATION -o /app/publish /p:UseAppHost=false

# Этот этап используется в рабочей среде или при запуске из VS в обычном режиме (по умолчанию, когда конфигурация отладки не используется)

FROM base AS final

WORKDIR /app

COPY --from=publish /app/publish .

ENTRYPOINT ["dotnet", "Backend.dll"]

**Backend/Properties/launchSettings.json**

{

"profiles": {

"http": {

"commandName": "Project",

"launchUrl": "swagger",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

},

"dotnetRunMessages": true,

"applicationUrl": "http://localhost:5050"

},

"https": {

"commandName": "Project",

"launchUrl": "swagger",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

},

"dotnetRunMessages": true,

"applicationUrl": "https://localhost:7093;http://localhost:5050"

},

"IIS Express": {

"commandName": "IISExpress",

"launchUrl": "swagger",

"environmentVariables": {

"ASPNETCORE\_ENVIRONMENT": "Development"

}

},

"Container (Dockerfile)": {

"commandName": "Docker",

"environmentVariables": {

"ASPNETCORE\_HTTPS\_PORTS": "8081",

"ASPNETCORE\_HTTP\_PORTS": "8080"

},

"publishAllPorts": true,

"useSSL": true

}

},

"$schema": "http://json.schemastore.org/launchsettings.json",

"iisSettings": {

"windowsAuthentication": false,

"anonymousAuthentication": true,

"iisExpress": {

"applicationUrl": "http://localhost:42919",

"sslPort": 44385

}

}

}

**Backend/Controllers/Accounts/AccountController.cs**

using Backend.Controllers.Acccount.Logic;

using Backend.Controllers.Acccount.Logic.Types;

using Backend.Controllers.Acccount.Requests;

using Backend.Controllers.Acccount.Responses;

using Backend.Tools;

using Database;

using Microsoft.AspNetCore.Mvc;

namespace Backend.Controllers.Acccount

{

[Route("account")]

[ApiController]

public class AccountController(ILogger<AccountController> logger) : ControllerBase

{

[HttpPost("newAccount")]

public async Task<IActionResult> NewAccount(

ApplicationContext context,

[FromBody] NewAccountRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode)

{

ArgumentNullException.ThrowIfNull(args.phone);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

var status = await AccountManager.NewAccount(context, args.phone, userId);

var response = new NewAccountResponse()

{

ownerId = userId,

phone = args.phone,

status = status,

};

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception in account/newAccount: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "Something wrong", userId, sessionCode);

}

}

[HttpPost("setCode")]

public async Task<IActionResult> SetCode(

ApplicationContext context,

[FromBody] SetCodeRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode)

{

ArgumentNullException.ThrowIfNull(args.phone);

ArgumentNullException.ThrowIfNull(args.code);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

var status = await AccountManager.NewAccountSetCode(context, args.phone, args.code);

var response = new NewAccountResponse()

{

ownerId = userId,

phone = args.phone,

status = status,

};

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception in account/setCode: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "Something wrong", userId, sessionCode);

}

}

[HttpPost("setPassword")]

public async Task<IActionResult> SetPassword(

ApplicationContext context,

[FromBody] SetPasswordRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode)

{

ArgumentNullException.ThrowIfNull(args.phone);

ArgumentNullException.ThrowIfNull(args.password);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

var status = await AccountManager

.NewAccountSetCloudPassword(context, args.phone, args.password);

var response = new NewAccountResponse()

{

ownerId = userId,

phone = args.phone,

status = status,

};

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception in account/setPassword: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "Something wrong", userId, sessionCode);

}

}

[HttpPost("getMyAccounts")]

public async Task<IActionResult> GetMyAccounts(

ApplicationContext context,

[FromBody] HttpEmptyData args,

[FromHeader] long userId,

[FromHeader] string sessionCode)

{

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

var result = await AccountManager.GetAccounts(context, userId);

var response = new GetMyAccountsResponse() { accounts = result };

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception in account/getMyAccounts: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "Something wrong", userId, sessionCode);

}

}

[HttpPost("getDialogs")]

public async Task<IActionResult> GetDialogs(

ApplicationContext context,

[FromBody] GetDialogsRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode,

CancellationToken cancellationToken)

{

ArgumentNullException.ThrowIfNull(args.phoneNumber);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

List<DialogInfo> result = await AccountManager

.GetDialogs(context, userId, args.phoneNumber, args.offsetId, args.limit, cancellationToken);

var response = new GetDialogsResponse() { dialogs = result };

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception in account/getDialogs: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "Something wrong", userId, sessionCode);

}

}

}

}

**Backend/Controllers/Account/Logic/AccountManager.cs**

using Backend.Controllers.Acccount.Logic.Types;

using Database;

using Core.Workers;

using Config;

using Core.Workers.Types;

using Database.Enum;

using TL;

using Microsoft.EntityFrameworkCore;

namespace Backend.Controllers.Acccount.Logic

{

public static class AccountManager

{

public static async Task<string> NewAccount(ApplicationContext context, string phone, long ownerId)

=> await LoadedAccountsWorker.OpenNewAccount(context, phone, ownerId);

public static async Task<string> NewAccountSetCode(ApplicationContext context, string phone, string code)

=> await LoadedAccountsWorker.SetCodeToNewAccount(context, phone, code);

public static async Task<string> NewAccountSetCloudPassword(ApplicationContext context, string phone, string password)

=> await LoadedAccountsWorker.SetCloudPasswordToNewAccount(context, phone, password);

public static async Task<List<AccountInfo>> GetAccounts(ApplicationContext context, long userId)

{

List<AccountInfo> result = [];

var accounts = context.Accounts.Where(a => a.OwnerId == userId);

foreach (var acc in accounts)

{

var account = await LoadedAccountsWorker.Get(userId, acc.PhoneNumber);

if (account.Status == LoadedAccount.Statuses.Logged)

{

result.Add(new()

{

PhoneNumber = account.PhoneNumber,

PhotoUrl = ProgramConfig.Path.Static + await LoadedAccountsWorker.DownloadAvatar(account.Client, account.Client.User),

Title = account.Client.User.first\_name + " " + account.Client.User.last\_name,

Username = account.Client.User.username

});

}

}

return result;

}

public static async Task<List<DialogInfo>> GetDialogs(

ApplicationContext context, long userId, string phone,

int offsetId, int limit, CancellationToken cancellationToken)

{

var account = await LoadedAccountsWorker.Get(userId, phone);

var targets = await context.Targets

.Include(t => t.FromAccount)

.Where(t => t.FromAccount.PhoneNumber == account.PhoneNumber)

.GroupBy(t => t.PeerId)

.Select(g => new

{

PeerId = g.Key,

LastTargetLog = g.Where(t => t.Id == g.Max(t => t.Id)),

})

.ToListAsync(cancellationToken);

InputPeer? offsetPeer = null;

DateTime offsetDate = default;

List<DialogInfo> result = [];

const int maxLimit = 100;

int prevGet = maxLimit;

while (result.Count < limit && prevGet >= maxLimit)

{

var dialogs = await account.Client.Messages\_GetDialogs(limit: maxLimit, offset\_id: offsetId, offset\_peer: offsetPeer, offset\_date: offsetDate);

prevGet = dialogs.Dialogs.Length;

foreach (var dialog in dialogs.Dialogs)

{

string topMessage = "<empty>";

var msgOfDialog = dialogs.Messages.FirstOrDefault(m => m.Peer.ID == dialog.Peer.ID);

switch (msgOfDialog)

{

case Message msg:

topMessage = msg.message;

break;

case MessageService msgService:

try

{

topMessage = msgService.action switch

{

MessageActionPinMessage => "Сообщение закреплено",

\_ => "Неизвестное действие"

};

}

catch { topMessage = "msg service exception"; }

break;

}

DialogInfo? tmp = dialogs.UserOrChat(dialog) switch

{

User user => new(user, await LoadedAccountsWorker.DownloadAvatar(account.Client, user), topMessage, dialog.TopMessage),

Chat chat => new(chat, await LoadedAccountsWorker.DownloadAvatar(account.Client, chat), topMessage, dialog.TopMessage),

Channel channel => new(channel, await LoadedAccountsWorker.DownloadAvatar(account.Client, channel), topMessage, dialog.TopMessage),

\_ => null,

};

if (tmp?.GetInputPeer() != null)

{

offsetPeer = tmp.GetInputPeer();

offsetDate = dialogs.Messages.First(d => d.ID == dialog.TopMessage).Date;

offsetId = dialog.TopMessage;

}

if (tmp != null && !tmp.IsBot() && !tmp.IsChannel())

{

tmp.IsTarget = targets.FirstOrDefault(g => g.PeerId == tmp.PeerId)?.LastTargetLog.First().Status == LoggingTargetStatus.Enabled;

result.Add(tmp);

}

if (result.Count >= limit) break;

}

}

return result;

}

}

}

**Backend/Controllers/Acccount/Logic/Types/AccountInfo.cs**

using Backend.Tools;

namespace Backend.Controllers.Acccount.Logic.Types

{

public class AccountInfo : ParsebleToDictionaryBase

{

public required string PhoneNumber { get; set; }

public required string Title { get; set; }

public string? Username { get; set; }

public required string PhotoUrl { get; set; }

}

}

**Backend/Controllers/Acccount/Logic/Types/DialogInfo.cs**

using Backend.Tools;

using Config;

using TL;

namespace Backend.Controllers.Acccount.Logic.Types

{

public class DialogInfo : ParsebleToDictionaryBase

{

public const int MAX\_MESSAGE\_PREVIEW = 50;

public DialogInfo(User reference, string photoFilename, string topMessage, int topMessageId)

{

PeerId = reference.id;

AccessHash = reference.access\_hash.ToString();

Title = reference.first\_name + " " + reference.last\_name;

TopMessage = topMessage.Length > MAX\_MESSAGE\_PREVIEW ? topMessage[..(MAX\_MESSAGE\_PREVIEW - 3)] + "..." : topMessage;

TopMessageId = topMessageId;

DialogType = reference.IsBot ? Types.Bot : Types.User;

\_isBot = reference.IsBot;

PhotoUrl = ProgramConfig.Path.Static + photoFilename;

\_inputPeer = reference.ToInputPeer();

}

public DialogInfo(Chat reference, string photoFilename, string topMessage, int topMessageId)

{

PeerId = reference.id;

AccessHash = null;

Title = reference.Title;

TopMessage = topMessage.Length > MAX\_MESSAGE\_PREVIEW ? topMessage[..(MAX\_MESSAGE\_PREVIEW - 3)] + "..." : topMessage;

TopMessageId = topMessageId;

DialogType = Types.Chat;

PhotoUrl = ProgramConfig.Path.Static + photoFilename;

\_inputPeer = reference.ToInputPeer();

}

public DialogInfo(Channel reference, string photoFilename, string topMessage, int topMessageId)

{

PeerId = reference.id;

AccessHash = reference.access\_hash.ToString();

Title = reference.Title;

TopMessage = topMessage.Length > MAX\_MESSAGE\_PREVIEW ? topMessage[..(MAX\_MESSAGE\_PREVIEW - 3)] + "..." : topMessage;

TopMessageId = topMessageId;

DialogType = Types.Channel;

PhotoUrl = ProgramConfig.Path.Static + photoFilename;

\_inputPeer = reference.ToInputPeer();

\_isChannel = true;

}

public long PeerId { get; set; }

public string? AccessHash { get; set; }

public string Title { get; set; }

public string TopMessage { get; set; }

public int TopMessageId { get; set; }

public bool IsTarget { get; set; }

public string DialogType { get; set; }

public string PhotoUrl { get; set; }

private bool \_isBot = false;

private bool \_isChannel = false;

private InputPeer? \_inputPeer = null;

public struct Types

{

public const string User = "User";

public const string Chat = "Chat";

public const string Channel = "Channel";

public const string Bot = "User|Bot";

}

public bool IsBot() => \_isBot;

public bool IsChannel() => \_isChannel;

public InputPeer? GetInputPeer() => \_inputPeer;

public override string ToString()

{

return $"({DialogType}, {PeerId}) | {Title} {TopMessage}";

}

}

}

**Backend/Controllers/Acccount/Requests/GetDialogsRequest.cs**

#pragma warning disable IDE1006

using Backend.Tools;

namespace Backend.Controllers.Acccount.Requests

{

public class GetDialogsRequest : HttpDataBase

{

public string? phoneNumber { get; set; }

public int offsetId { get; set; } = 0;

public int limit { get; set; } = 10;

}

}

**Backend/Controllers/Acccount/Requests/NewAccountRequest.cs**

#pragma warning disable IDE1006

namespace Backend.Controllers.Acccount.Requests

{

public class NewAccountRequest: Tools.HttpDataBase

{

public string? phone { get; set; }

}

}

**Backend/Controllers/Acccount/Requests/SetCodeRequest.cs**

#pragma warning disable IDE1006

namespace Backend.Controllers.Acccount.Requests

{

public class SetCodeRequest: Tools.HttpDataBase

{

public string? phone { get; set; }

public string? code { get; set; }

}

}

**Backend/Controllers/Acccount/Requests/SetPasswordRequest.cs**

#pragma warning disable IDE1006

namespace Backend.Controllers.Acccount.Requests

{

public class SetPasswordRequest: Tools.HttpDataBase

{

public string? phone { get; set; }

public string? password { get; set; }

}

}

**Backend/Controllers/Auth/AuthController.cs**

using Backend.Controllers.Auth.Logic;

using Backend.Controllers.Auth.Requests;

using Backend.Controllers.Auth.Response;

using Backend.Tools;

using Config;

using Core.Workers;

using Database;

using Microsoft.AspNetCore.Mvc;

using Microsoft.EntityFrameworkCore;

using Microsoft.Extensions.Primitives;

namespace Backend.Controllers.Auth

{

[Route("auth")]

[ApiController]

public class AuthController(ILogger<AuthController> logger ) : ControllerBase

{

[HttpPost("logIn")]

public async Task<IActionResult> LogIn([FromBody] LogInRequest args, ApplicationContext context)

{

ArgumentNullException.ThrowIfNull(args.initData);

var initData = new InitData(args.initData);

ArgumentNullException.ThrowIfNull(initData.Hash);

ArgumentNullException.ThrowIfNull(initData.User);

if (initData.User == null || initData.AuthDate == null)

return BadRequest("Wrong initData!");

if (ProgramConfig.DEV == false &&

initData.AuthDate!.Value.AddSeconds(10) < DateTime.UtcNow)

return BadRequest("initData is too old!");

try

{

await UserManager.RegisterOrUpdate(context, initData.User);

string newSession = await SessionWorker.OpenNew(context, initData.Hash, initData.User.Id);

await context.SaveChangesAsync();

var response = new LogInResponse()

{

sessionCode = newSession,

me = initData.User,

accountCount = await context.Accounts

.CountAsync(a =>

a.Owner.Id == initData.User.Id &&

a.Status == Database.Enum.AccountStatus.Active)

};

response.Sign(initData.User.Id, newSession);

return response.ToObjectResult();

}

catch (Exception ex)

{

logger.LogError("Unexpected exception when auth/logIn: {ex}", ex);

return HttpErrorData.Create();

}

}

[HttpPost("ping")]

public IActionResult Ping([FromBody] HttpEmptyData args, [FromHeader] long userId, [FromHeader] string sessionCode)

{

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

return new HttpEmptyData(userId, sessionCode).ToObjectResult();

}

public IActionResult LogOut([FromHeader] long userId, [FromHeader] string session)

{

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.NotImplemented, "NotImplemented", userId, session);

}

// Это middleware

public static async Task CustomAuthorization(HttpContext context, Func<Task> next)

{

if (context.Request.Method == "POST")

{

if (!context.Request.Headers.TryGetValue("userId", out StringValues tmp) ||

!long.TryParse(tmp.First(), out long userId))

{

context.Response.StatusCode = 401;

await context.Response.WriteAsync("Unauthorized");

return;

}

if (userId != -1)

{

context.Request.Headers.TryAdd("sessionCode", SessionWorker.GetCodeByUser(userId));

}

else if (userId == -1 && (context.Request.Path.Value == null || !context.Request.Path.Value.EndsWith("auth/logIn")))

{

context.Response.StatusCode = 401;

await context.Response.WriteAsync("Unauthorized");

return;

}

}

await next.Invoke();

}

}

}

**Backend/Controllers/Auth/Logic/UserManager.cs**

using Backend.Tools.Structs;

using Database;

using Microsoft.EntityFrameworkCore;

namespace Backend.Controllers.Auth.Logic

{

public static class UserManager

{

public static async Task RegisterOrUpdate(ApplicationContext context, User userInfo)

{

var user = await context.Users

.FirstOrDefaultAsync(u => u.Id == userInfo.Id);

if (user == null)

{

context.Users.Add(new()

{

Id = userInfo.Id,

FirstName = userInfo.FirstName,

LastName = userInfo.LastName,

LanguageCode = userInfo.LanguageCode,

Username = userInfo.Username

});

}

else

{

user.FirstName = userInfo.FirstName;

user.LastName = userInfo.LastName;

user.Username = userInfo.Username;

}

}

}

}

**Backend/Controllers/Auth/Requests/LogInRequest.cs**

#pragma warning disable IDE1006

using Backend.Tools;

namespace Backend.Controllers.Auth.Requests

{

public class LogInRequest : HttpDataBase

{

public string? initData { get; set; }

}

}

**Backend/Controllers/Auth/Response/LogInResponse.cs**

#pragma warning disable IDE1006

using Backend.Tools;

using Backend.Tools.Structs;

namespace Backend.Controllers.Auth.Response

{

public class LogInResponse : HttpDataBase

{

public required string sessionCode { get; set; }

public required User me { get; set; }

public int accountCount { get; set; }

}

}

**Backend/Controllers/Cors/CorsController.cs**

using Microsoft.AspNetCore.Mvc;

// Custom cors

// Только так у меня получилось протащить заголовки,

// разрешающие CORS, через прокси keenDNS.

// При использовании CORS через Services, OPTIONS

// запросы не проходили из-за чего браузер блокировал

// ответы.

namespace Backend.Controllers.Cors

{

[Route("{section}")]

[ApiController]

public class CorsController : ControllerBase

{

[HttpOptions("{methodName}")]

public void HandleOptions(string section, string methodName)

{

Response.StatusCode = 200;

Response.Headers.TryAdd("Access-Control-Allow-Methods", "POST");

Response.Headers.TryAdd("Access-Control-Allow-Headers", "Content-type, userId");

Response.Headers.TryAdd("Access-Control-Allow-Origin", "\*");

Response.Headers.TryAdd("Access-Control-Max-Age", "5");

}

public static async Task InsertHeaders(HttpContext context, Func<Task> next)

{

context.Response.Headers.TryAdd("Access-Control-Allow-Headers", "Content-type, userId");

context.Response.Headers.TryAdd("Access-Control-Allow-Origin", "\*");

await next.Invoke();

}

}

}

**Backend/Controllers/Static/StaticController.cs**

using Config;

using Microsoft.AspNetCore.Mvc;

namespace Backend.Controllers.Static

{

[Route("static")]

public class StaticController : Controller

{

[HttpGet("{filename}")]

public async Task Index(string filename)

{

if (System.IO.File.Exists(ProgramConfig.TelegramApiAuth.DownloadsDir + filename))

{

await Response.SendFileAsync(ProgramConfig.TelegramApiAuth.DownloadsDir + filename);

}

else NotFound();

}

}

}

**Backend/Controllers/Target/** **TargetController.cs**

using Backend.Controllers.Target.Logic;

using Backend.Controllers.Target.Logic.Types;

using Backend.Controllers.Target.Requests;

using Backend.Controllers.Target.Response;

using Backend.Tools;

using Database;

using Microsoft.AspNetCore.Mvc;

namespace Backend.Controllers.Target

{

[Route("target")]

[ApiController]

public class TargetController(ILogger<TargetController> logger) : ControllerBase

{

[HttpPost("updateTarget")]

public async Task<IActionResult> UpdateTarget(

ApplicationContext context,

[FromBody] UpdateTargetRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode,

CancellationToken cancellationToken)

{

ArgumentNullException.ThrowIfNull(args.phoneNumber);

ArgumentNullException.ThrowIfNull(args.enable);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

await TargetManager.SetTarget(context, userId, args.phoneNumber, args.peerId, args.accessHash, args.enable ?? false, cancellationToken);

var response = new TargetStatusResponse()

{

peerId = args.peerId,

enable = args.enable ?? false,

};

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (ArgumentException ex)

{

logger.LogWarning("User {userId} send request with wrong arguments un target/updateTarget: {ex}", userId, ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.BadRequest, ex.Message, userId, sessionCode);

}

catch (Exception ex)

{

logger.LogError("Unexpected exception when target/updateTarget: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "SomethingWrong", userId, sessionCode);

}

}

[HttpPost("getSavedLogs")]

public async Task<IActionResult> GetSavedLogs(

ApplicationContext context,

[FromBody] GetSavedLogsRequest args,

[FromHeader] long userId,

[FromHeader] string sessionCode,

CancellationToken cancellationToken

)

{

ArgumentNullException.ThrowIfNull(args.phoneNumber);

try { args.Verify(userId, sessionCode); } catch { return Unauthorized(); }

try

{

List<LogInfo> result = await TargetManager

.GetLogsHistory(context, args.phoneNumber, args.peerId, args.offsetId, args.limit, cancellationToken);

var response = new GetSavedLogsResponse()

{

logs = result

};

response.Sign(userId, sessionCode);

return response.ToObjectResult();

}

catch (ArgumentException ex)

{

logger.LogWarning("User {userId} send request with wrong arguments un target/getSavedLogs: {ex}", userId, ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.BadRequest, ex.Message, userId, sessionCode);

}

catch (Exception ex)

{

logger.LogError("Unexpected exception when target/getSavedLogs: {ex}", ex);

return HttpErrorData.CreateAndSign(System.Net.HttpStatusCode.InternalServerError, "SomethingWrong", userId, sessionCode);

}

}

}

}

**Backend/Controllers/Target/Logic/** **TargetManager.cs**

using Backend.Controllers.Target.Logic.Types;

using Database;

using Database.Entities;

using Database.Enum;

using Microsoft.EntityFrameworkCore;

namespace Backend.Controllers.Target.Logic

{

public class TargetManager

{

public static async Task SetTarget(

ApplicationContext context,

long userId, string phone,

long peerId, string? accessHash,

bool enable,

CancellationToken cancellationToken)

{

var account = context.Accounts.FirstOrDefault(a => a.OwnerId == userId && a.PhoneNumber == phone)

?? throw new KeyNotFoundException($"Account with phone {phone} not found for user {userId}");

var target = await context.Targets

.OrderByDescending(t => t.Time)

.FirstOrDefaultAsync(t => t.PeerId == peerId && t.FromAccount.PhoneNumber == phone, cancellationToken: cancellationToken);

if (target == null && enable == false) return;

var newTargetLog = new LoggingTarget()

{

AccessHash = accessHash != null ? long.Parse(accessHash) : null,

FromAccount = account,

PeerId = peerId,

Type = LoggingTargetType.Messages,

Status = enable ? LoggingTargetStatus.Enabled : LoggingTargetStatus.Disabled,

PrevTarget = target,

};

await context.Targets.AddAsync(newTargetLog, cancellationToken: cancellationToken);

await context.SaveChangesAsync(cancellationToken);

await Core.Services.LoggingService.Instance.Update(cancellationToken);

}

public static async Task<List<LogInfo>> GetLogsHistory(

ApplicationContext context,

string phoneNumber, long peerId,

int offsetId, int limit,

CancellationToken cancellationToken)

{

if (offsetId == 0) offsetId = int.MaxValue;

List<LogInfo> result = [];

var messageUpdatesQuery = context.UpdatesMessage

.Include(l => l.LoggingTarget)

.Include(l => l.LoggingTarget.PrevTarget)

.Include(l => l.LoggingTarget.FromAccount)

.Where(l =>

l.LoggingTarget.FromAccount.PhoneNumber == phoneNumber &&

l.LoggingTarget.PeerId == peerId &&

l.MessageId < offsetId)

.GroupBy(l => l.MessageId)

.Select(g => new

{

g.Key,

List = g.OrderByDescending(l => l.Id).ToList(),

})

.OrderByDescending(l => l.Key)

.Take(limit);

var deletedMessages = await context.UpdatesDeleteMessage

.Where(dm => messageUpdatesQuery.Any(g => g.Key == dm.MessageId))

.ToListAsync(cancellationToken);

var messageUpdates = await messageUpdatesQuery.ToListAsync(cancellationToken);

messageUpdates.ForEach(g =>

{

var tmp = g.List.Select(LogInfo.FromMessageLog);

var deleted = deletedMessages.FirstOrDefault(l => l.MessageId == g.Key);

LogInfo main;

if (deleted != null)

main = LogInfo.FromDeletedMessageLog(deleted);

else

main = tmp.OrderByDescending(l => l.LogTime).First();

main.PrevChanges = [.. tmp.Where(l => l.DbId != main.DbId)];

result.Add(main);

});

List<LogInfo> targets = [];

messageUpdates.ForEach(mu =>

{

targets.AddRange(mu.List.Select(l => LogInfo.FromTarget(l.LoggingTarget)));

targets.AddRange(mu.List

.Where(l => l.LoggingTarget.PrevTarget != null)

.Select(l => LogInfo.FromTarget(l.LoggingTarget.PrevTarget!)));

});

result.AddRange(targets.DistinctBy(li => li.DbId));

return [.. result.Select(l => new

{

Time = (l.Type == LogInfo.Types.Specifications) ?

l.LogTime :

(l.MessageDate == null ?

l.PrevChanges?.FirstOrDefault(pl => pl.MessageDate != null)?.MessageDate ?? l.LogTime :

l.MessageDate),

LogObj = l

}).OrderByDescending(l => l.Time).Select(l => l.LogObj).Take(limit)];

}

}

}

**Backend/Controllers/Target/Logic/Types/LogInfo.cs**

using Backend.Tools;

using Database.Entities;

using Database.Enum;

namespace Backend.Controllers.Target.Logic.Types

{

public class LogInfo : ParsebleToDictionaryBase

{

public long DbId { get; set; }

public required string Type { get; set; }

public required string Message { get; set; }

public required string MessageEntities { get; set; }

public DateTime? MessageDate { get; set; }

public int? MessageId { get; set; }

public long FromId { get; set; }

public DateTime LogTime { get; set; }

public List<LogInfo>? PrevChanges { get; set; } = [];

public long? PrevId { get; set; }

public struct Types

{

public const string Message = "Message";

public const string MessageDeleted = "MessageDeleted";

public const string Specifications = "Specifications";

}

public static LogInfo FromTarget(LoggingTarget tl)

{

return new LogInfo

{

DbId = tl.Id,

FromId = -1,

LogTime = tl.Time,

Message = tl.Type switch

{

LoggingTargetType.Messages => "Отслеживание сообщений: ",

\_ => "Unknown"

} + tl.Status switch

{

LoggingTargetStatus.Enabled => "включено",

LoggingTargetStatus.Disabled => "выключено",

LoggingTargetStatus.Deleted => "больше недоступно (чат удалён)",

LoggingTargetStatus.Banned => "больше недоступно: (чат недоступен или в чёрном списке)",

LoggingTargetStatus.Failure => "нарушено. Ошибка: " + tl.Error,

\_ => "unknown action"

},

MessageDate = null,

MessageEntities = "{}",

MessageId = null,

PrevId = null,

Type = LogInfo.Types.Specifications,

};

}

public static LogInfo FromMessageLog(UpdateMessageLog uml)

{

return new LogInfo

{

DbId = uml.Id,

FromId = uml.FromId,

LogTime = uml.Time,

Message = uml.Text,

MessageEntities = uml.TextEntities,

MessageDate = uml.MsgDate,

MessageId = uml.MessageId,

PrevId = uml.PrevEditId,

Type = LogInfo.Types.Message,

};

}

public static LogInfo FromDeletedMessageLog(UpdateDeleteMessageLog udml)

{

return new LogInfo

{

DbId = udml.Id,

FromId = -1,

LogTime = udml.Time,

Message = "",

MessageEntities = "",

MessageDate = null,

MessageId = udml.MessageId,

PrevId = null,

Type = LogInfo.Types.MessageDeleted,

};

}

}

}

**Backend/Controllers/Target/Requests/GetSavedLogsRequest.cs**

#pragma warning disable IDE1006

using Backend.Tools;

namespace Backend.Controllers.Target.Requests

{

public class GetSavedLogsRequest : HttpDataBase

{

public string? phoneNumber { get; set; }

public long peerId { get; set; }

public int offsetId { get; set; } = 0;

public int limit { get; set; } = 10;

}

}

**Backend/Controllers/Target/Requests/UpdateTargetRequest.cs**

#pragma warning disable IDE1006

using Backend.Tools;

namespace Backend.Controllers.Target.Requests

{

public class UpdateTargetRequest : HttpDataBase

{

public string? phoneNumber { get; set; }

public long peerId { get; set; }

public string? accessHash { get; set; }

public bool? enable { get; set; }

}

}

**Backend/Controllers/Target/Response/GetSavedLogsResponse.cs**

#pragma warning disable IDE1006

using Backend.Controllers.Target.Logic.Types;

using Backend.Tools;

namespace Backend.Controllers.Target.Response

{

public class GetSavedLogsResponse : HttpDataBase // TODO ... Проверить генерацию подписи с этими данными

{

public required List<LogInfo> logs { get; set; }

}

}

**Backend/Controllers/Target/Response/TargetStatusResponse.cs**

using Backend.Tools;

namespace Backend.Controllers.Target.Response

{

public class TargetStatusResponse : HttpDataBase

{

public long peerId { get; set; }

public bool enable { get; set; }

}

}

**Backend/Services/CoreService.cs**

namespace Backend.Services

{

public class CoreService : BackgroundService

{

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

=> await Core.Services.CoreService.Instance.ExecuteAsync(stoppingToken);

}

}

**Backend/Services/LoggingService.cs**

namespace Backend.Services

{

public class LoggingService : BackgroundService

{

protected override async Task ExecuteAsync(CancellationToken stoppingToken)

=> await Core.Services.LoggingService.Instance.ExecuteAsync(stoppingToken);

}

}

**Backend/Tools/HttpDataBase.cs**

#pragma warning disable IDE1006

using Microsoft.AspNetCore.Mvc;

using System.Collections;

using System.Security.Cryptography;

using System.Text;

using System.Text.Json;

namespace Backend.Tools

{

public abstract class HttpDataBase : ParsebleToDictionaryBase

{

public string? signature { get; set; }

public long ts { get; set; }

public ObjectResult ToObjectResult()

{

return new ObjectResult(ToDict());

}

public void Sign(long userId, string sessionCode)

{

var sortedParams = GetSortedParams();

ts = TimeStampConvertor.DatetimeToLong(DateTime.UtcNow);

string dataStr = $"userId:{userId}\_ts:{ts}\_";

dataStr += string.Join("&", sortedParams.Select(item => $"data.{item.Key}={item.Value}"));

signature = Convert.ToHexString(HMACSHA256.HashData(

Encoding.UTF8.GetBytes(sessionCode),

Encoding.UTF8.GetBytes(dataStr)))!;

}

public void Verify(long userId, string sessionCode)

{

if (sessionCode == "\_") throw new InvalidOperationException(nameof(sessionCode) + " is empty");

if (signature == null) throw new InvalidOperationException(nameof(signature) + " is null");

var sortedParams = GetSortedParams();

sortedParams.Remove("signature");

sortedParams.Remove("ts");

string dataStr = $"userId:{userId}\_ts:{ts}\_";

dataStr += string.Join("&", sortedParams.Select(item => $"data.{item.Key}={item.Value}"));

string tmp = Convert.ToHexString(HMACSHA256.HashData(

Encoding.UTF8.GetBytes(sessionCode),

Encoding.UTF8.GetBytes(dataStr)))!;

if (!signature.Equals(tmp, StringComparison.OrdinalIgnoreCase)) throw new UnauthorizedAccessException("Signature do not match!");

}

// В GetSortedParams надо ключи заполнять с именем корневого объекта.

// Пример: { root: { child: "value" } } -> { root.child : "value" }

protected virtual SortedDictionary<string, object> GetSortedParams()

{

var data = ToDict();

var ret = new SortedDictionary<string, object>();

DefaulTransform(data, "", ref ret);

return ret;

}

static void DefaulTransform(

Dictionary<string, object> source,

string parentKey,

ref SortedDictionary<string, object> result)

{

foreach (var item in source)

{

string currentKey = parentKey != string.Empty ? $"{parentKey}.{item.Key}" : item.Key;

if (item.Value is Dictionary<string, object> dictionary)

DefaulTransform(dictionary, currentKey, ref result);

else if (item.Value is IList objects)

{

var dict = new Dictionary<string, object>();

int index = 0;

foreach (var obj in objects)

{

dict.Add(index++.ToString(), obj);

}

DefaulTransform(dict, currentKey, ref result);

}

else

{

try

{

var tmp = ((JsonElement)item.Value).Deserialize<Dictionary<string, object>>() ?? [];

DefaulTransform(tmp, currentKey, ref result);

}

catch

{

result.Add(currentKey, item.Value);

}

}

}

}

}

}

**Backend/Tools/HttpEmptyData.cs**

namespace Backend.Tools

{

public class HttpEmptyData : HttpDataBase

{

public HttpEmptyData() { }

public HttpEmptyData(long userId, string sessionCode)

{

Sign(userId, sessionCode);

}

}

}

**Backend/Tools/HttpErrorData.cs**

using Microsoft.AspNetCore.Mvc;

using System.Net;

namespace Backend.Tools

{

public class HttpErrorData : HttpDataBase

{

public HttpStatusCode statusCode { get; set; }

public required string error { get; set; }

public static ObjectResult Create(

HttpStatusCode code = HttpStatusCode.InternalServerError, string error = "Something wrong")

=> new(new HttpErrorData() { statusCode = code, error = error });

public static ObjectResult CreateAndSign(

HttpStatusCode code,

string error,

long userId,

string sessionCode)

{

var result = new HttpErrorData() { statusCode = code, error = error };

result.Sign(userId, sessionCode);

return new ObjectResult(result);

}

}

}

**Backend/Tools/InitData.cs**

using Backend.Tools.Structs;

using Config;

using System.Data;

using System.Security.Cryptography;

using System.Text;

using System.Text.Json;

using System.Web;

namespace Backend.Tools

{

public class InitData

{

public User? User { get; private set; } = null;

public DateTime? AuthDate { get; private set; } = null;

public string? Hash { get; private set; } = null;

public InitData(string initData)

{

var data = HttpUtility.ParseQueryString(initData);

var sortedData = new SortedDictionary<string, string>(

data.AllKeys

.Where(k => k != null)

.ToDictionary(k => k ?? "", k => data[k] ?? ""),

StringComparer.Ordinal);

string dataCheckString = string.Join(

"\n",

sortedData

.Where(pair => pair.Key != "hash")

.Select(pair => $"{pair.Key}={pair.Value}"));

var secretKey = HMACSHA256.HashData(

Encoding.UTF8.GetBytes("WebAppData"),

Encoding.UTF8.GetBytes(ProgramConfig.TelegramBotKey));

string reference = data.Get("hash") ?? "";

string resultHash = Convert.ToHexString(

HMACSHA256.HashData(

secretKey,

Encoding.UTF8.GetBytes(dataCheckString)));

if (!resultHash.Equals(reference, StringComparison.OrdinalIgnoreCase))

throw new UnauthorizedAccessException("Wrong hash in initData");

var tmpUser = JsonSerializer.Deserialize<Dictionary<string, JsonElement>>(data.Get("user") ?? "{}");

if (tmpUser != null)

User = new User(tmpUser);

if (int.TryParse(data.Get("auth\_date"), out int auth\_date))

AuthDate = TimeStampConvertor.IntToDatetime(auth\_date);

Hash = data.Get("hash");

}

}

}

**Backend/Tools/ParsebleToDictionaryBase.cs**

using System.ComponentModel.DataAnnotations.Schema;

using System.Reflection;

namespace Backend.Tools

{

public class ParsebleToDictionaryBase

{

public virtual Dictionary<string, object> ToDict()

{

var ret = new Dictionary<string, object>();

var properties = GetType().GetProperties().Where(p => p.GetCustomAttribute<NotMappedAttribute>() == null);

foreach (var property in properties)

{

object? value = property.GetValue(this);

string name = char.ToLowerInvariant(property.Name[0]) + property.Name[1..];

if (value != null)

if (value is DateTime time)

ret.Add(name, TimeStampConvertor.DatetimeToLong(time));

else if (value is ParsebleToDictionaryBase parseble)

ret.Add(name, parseble.ToDict());

else if (value is IEnumerable<ParsebleToDictionaryBase> values)

ret.Add(name, values.Select(v => v.ToDict()).ToArray());

else

ret.Add(name, value);

}

return ret;

}

}

}

**Config/ProgramConfig.cs**

using Config.Entities;

namespace Config

{

public static class ProgramConfig

{

public const int LoggingSaveSec = 5;

public static string TelegramBotKey { get; }

public static DatabaseAuth DatabaseAuth { get; }

public static TelegramApiAuth TelegramApiAuth { get; }

public static PathCollection Path { get; }

public static bool DEV { get; }

static ProgramConfig()

{

TelegramBotKey = Environment.GetEnvironmentVariable("TELEGRAM\_BOT\_KEY") ?? "";

DatabaseAuth = new(

"telegram-logger-mysql",

"root",

Environment.GetEnvironmentVariable("MYSQL\_ROOT\_PASSWORD") ?? "root",

"TelegramLogger",

Environment.GetEnvironmentVariable("MYSQL\_VERSION") ?? "1.0.0");

TelegramApiAuth = new(

int.Parse(Environment.GetEnvironmentVariable("API\_ID") ?? "0"),

Environment.GetEnvironmentVariable("API\_HASH") ?? "");

DEV = Convert.ToBoolean(Environment.GetEnvironmentVariable("DEV"));

Path = new(Environment.GetEnvironmentVariable("BACKEND\_FOREIGN\_URL") ?? "~");

}

}

}

**Config/Entities/DatabaseAuth.cs**

namespace Config.Entities

{

public class DatabaseAuth

{

public string? Host { get; private set; }

public string? User { get; private set; }

public string? Password { get; private set; }

public string? Database { get; private set; }

public DatabaseAuth(string Host, string User, string Password, string Database, string Version)

{

this.Host = Host;

this.User = User;

this.Password = Password;

this.Database = Database;

var splited = Version.Split('.');

}

public string ConnectionString

{

get

{

return $"server={Host};" +

$"user={User};" +

$"password={Password};" +

$"database={Database}";

}

}

}

}

**Config/Entities/PathCollection.cs**

namespace Config.Entities

{

public class PathCollection(string foreignUrl)

{

public string Static { get => foreignUrl + "static/"; }

}

}

**Config/Entities/TelegramApiAuth.cs**

namespace Config.Entities

{

public class TelegramApiAuth

{

public TelegramApiAuth(int apiId, string apiHash)

{

ApiId = apiId;

ApiHash = apiHash;

}

public int ApiId { get; set; }

public string ApiHash { get; set; }

public string SessionsDir { get; set; } = "/app/sessions/";

public string DownloadsDir { get; set; } = "/app/downloads/";

}

}

**Core/Interfaces/IWorker.cs**

using Database;

namespace Core.Interfaces

{

public interface IWorker

{

public Task Handle(ApplicationContext context);

}

}

**Core/Services/CoreService.cs**

using Core.Interfaces;

using Database;

using System.Reflection;

namespace Core.Services

{

public class CoreService

{

static CoreService? instance = null;

private readonly static object x = new();

public static CoreService Instance

{

get

{

lock (x)

{

instance ??= new CoreService();

return instance;

}

}

}

public CoreService()

{

if (instance != null)

throw new InvalidOperationException("Is a singleton. For get instance use a CoreService.Instance property");

}

readonly PeriodicTimer \_timer = new(TimeSpan.FromSeconds(1));

public async Task ExecuteAsync(CancellationToken stoppingToken)

{

IWorker[] workers = Assembly

.GetExecutingAssembly()

.GetTypes().Where(t =>

t.IsAssignableTo(typeof(IWorker)) &&

t.IsInterface == false)

.Select(t => (IWorker)Activator.CreateInstance(t)!)

.ToArray()!;

Task? lastTask = null;

try

{

while (await \_timer.WaitForNextTickAsync(stoppingToken))

{

var context = new ApplicationContext();

foreach (IWorker worker in workers)

{

try

{

lastTask = worker.Handle(context);

await lastTask;

}

catch (Exception ex)

{

Console.Error.WriteLine($"Critical error in CoreService: {ex}");

}

}

lastTask = null;

}

}

catch (OperationCanceledException) { if (lastTask != null) await lastTask; }

}

}

}

**Core/Services/LoggingService.cs**

using Config;

using Core.Services.Types;

using Core.Workers;

using Core.Workers.Types;

using Database;

using Microsoft.EntityFrameworkCore;

using TL;

namespace Core.Services

{

public class LoggingService

{

static LoggingService? instance = null;

private static readonly object x = new();

public static LoggingService Instance

{

get

{

lock (x)

{

instance ??= new LoggingService();

return instance;

}

}

}

public LoggingService()

{

if (instance != null)

throw new InvalidOperationException("Is a singleton. For get instance use a LoggingService.Instance property");

}

public bool InWork { get; private set; } = false;

readonly Dictionary<long, List<UpdatesLogger>> Loggers = [];

public async Task Update(CancellationToken cancellationToken)

{

if (InWork == false) return;

using var context = new ApplicationContext();

var enabledTargetsLogs = await context.Targets

.AsNoTracking()

.Include(t => t.FromAccount)

.GroupBy(t => t.PeerId)

.Select(g => g.Where(t => t.Id == g.Max(t => t.Id)))

.ToListAsync(cancellationToken);

var enabledTargets = enabledTargetsLogs

.Select(etl => etl.First())

.Where(etl => etl.Status == Database.Enum.LoggingTargetStatus.Enabled)

.GroupBy(ge => ge.FromAccount.PhoneNumber)

.Select(g => new

{

Account = g.FirstOrDefault()!.FromAccount,

Targets = g.Select(ge => new

{

Target = ge,

InputPeer = GetInputPeer(ge.PeerId, ge.AccessHash)

})

})

.ToList();

foreach (var targets in enabledTargets)

{

Loggers.TryGetValue(targets.Account.Id, out List<UpdatesLogger>? loggers);

LoadedAccount? lacc = null;

loggers ??= [];

if (loggers.Count == 0)

{

loggers = [];

lacc = await LoadedAccountsWorker.Get(targets.Account.OwnerId, targets.Account.PhoneNumber);

lacc.Client.WithUpdateManager((update) => UpdateHandler(update, targets.Account.Id));

lacc.OnRestarted += () =>

{

Console.WriteLine("Account restarted. Set update handler again...");

lacc.Client.WithUpdateManager(

(update) => UpdateHandler(update, targets.Account.Id));

Console.WriteLine("Account restarted. Set update handler again... done");

};

Loggers.TryAdd(targets.Account.Id, loggers);

}

else

{

lacc = loggers.First().LoadedAccount;

loggers.Clear();

}

foreach (var target in targets.Targets)

{

var tmp = new UpdatesLogger(lacc, targets.Account, target.Target, target.InputPeer);

loggers.Add(tmp);

}

}

}

public async Task ExecuteAsync(CancellationToken stoppingToken)

{

var timer = new PeriodicTimer(TimeSpan.FromSeconds(ProgramConfig.LoggingSaveSec));

Task[]? tasks = null;

try

{

await StartAsync(stoppingToken);

while (await timer.WaitForNextTickAsync(stoppingToken))

{

foreach (var item in Loggers)

{

tasks = [.. item.Value.Select(l => l.Save())];

await WaitAllAsync(tasks);

}

}

}

catch (OperationCanceledException)

{

await WaitAllAsync(tasks);

await StopAsync();

}

}

async Task StartAsync(CancellationToken cancellationToken)

{

InWork = true;

await Update(cancellationToken);

}

Task StopAsync()

{

InWork = false;

foreach (var item in Loggers)

{

item.Value.ForEach(l => l.Dispose());

}

return Task.CompletedTask;

}

async Task UpdateHandler(Update update, long fromAccountId)

{

Loggers.TryGetValue(fromAccountId, out List<UpdatesLogger>? loggers);

if (loggers == null) return;

Task? logTask = null;

switch (update)

{

case UpdateNewMessage unm: logTask = GetLogger(loggers, unm.message.Peer.ID)?.HandleNewMessage(unm); break;

case UpdateEditMessage uem: logTask = GetLogger(loggers, uem.message.Peer.ID)?.HandleEditMessage(uem); break;

case UpdateDeleteMessages udm:

{

using var context = new ApplicationContext();

var peerId = (await context.UpdatesMessage

.AsNoTracking()

.Include(u => u.LoggingTarget)

.FirstOrDefaultAsync(u => u.MessageId == udm.messages.First()))

?.LoggingTarget.PeerId ?? -1;

if (peerId != -1)

logTask = GetLogger(loggers, peerId)?.HandleDeleteMessages(udm);

}

break;

}

if (logTask != null) await logTask;

}

static UpdatesLogger? GetLogger(List<UpdatesLogger> loggers, long peerId)

=> loggers.FirstOrDefault(l => l.PeerId == peerId);

public static InputPeer GetInputPeer(long peerId, long? accessHash)

{

if (peerId > 0) return new InputPeerUser(peerId, accessHash ?? 0);

else if (accessHash == null) return new InputPeerChat(peerId);

else return new InputPeerChannel(peerId, accessHash ?? 0);

}

static async Task WaitAllAsync(Task[]? tasks)

{

if (tasks == null) return;

foreach (var task in tasks)

await task;

}

}

}

**Core/Services/Types/UpdatesLogger.cs**

using Core.Workers.Types;

using Database;

using Database.Entities;

using Microsoft.EntityFrameworkCore;

using System.Text.Json;

using TL;

namespace Core.Services.Types

{

class UpdatesLogger : IDisposable

{

readonly List<UpdateLog> UpdatesLogs = [];

public UpdatesLogger(LoadedAccount lacc, Account fromAccount, LoggingTarget target, InputPeer peer)

{

\_lacc = lacc;

\_lacc.InUse = true;

\_account = fromAccount;

\_target = target;

\_peer = peer;

}

readonly LoggingTarget \_target;

readonly Account \_account;

readonly LoadedAccount \_lacc;

readonly InputPeer \_peer;

public long AccountId { get => \_account.Id; }

public long PeerId { get => \_peer.ID; }

public InputPeer InputPeer { get => \_peer; }

public LoadedAccount LoadedAccount { get => \_lacc; }

public async Task Save()

{

UpdateLog[] logsToSave;

lock (UpdatesLogs)

{

logsToSave = [.. UpdatesLogs];

UpdatesLogs.Clear();

}

if (logsToSave.Length == 0) return;

using var context = new ApplicationContext();

try

{

context.Targets.Attach(\_target);

}

catch (Exception ex)

{

Console.Error.WriteLine(ex.ToString());

}

try

{

context.Updates.AttachRange([.. logsToSave.OfType<UpdateMessageLog>().Where(l => l.PrevEdit != null).Select(l => l.PrevEdit!)]);

}

catch (Exception ex)

{

Console.Error.WriteLine(ex.ToString());

}

await context.Updates.AddRangeAsync([.. logsToSave]);

await context.SaveChangesAsync();

}

public void Dispose()

{

Save().Wait();

\_lacc.InUse = false;

}

public Task HandleNewMessage(UpdateNewMessage update)

{

switch (update.message)

{

case Message msg:

UpdatesLogs.Add(new UpdateMessageLog()

{

LoggingTarget = \_target,

Text = msg.message,

TextEntities = JsonSerializer.Serialize(msg.entities),

MessageId = msg.id,

FromId = msg.From == null ? -1 : msg.From.ID,

PrevEdit = null,

Time = DateTime.UtcNow,

MsgDate = msg.date

});

break;

default: Console.WriteLine("Unknown type of the message"); break;

}

return Task.CompletedTask;

}

public async Task HandleEditMessage(UpdateEditMessage update)

{

switch (update.message)

{

case Message msg:

UpdatesLogs.Add(new UpdateMessageLog()

{

LoggingTarget = \_target,

Text = msg.message,

TextEntities = JsonSerializer.Serialize(msg.entities),

MessageId = msg.id,

FromId = msg.From == null ? -1 : msg.From.ID,

PrevEdit = await GetPrevEdit(msg.id),

Time = DateTime.UtcNow,

MsgDate = msg.edit\_date

});

break;

default: Console.WriteLine("Unknown type of the message"); break;

}

}

public Task HandleDeleteMessages(UpdateDeleteMessages update)

{

var sharedTime = DateTime.UtcNow;

foreach (var msgId in update.messages)

{

UpdatesLogs.Add(new UpdateDeleteMessageLog()

{

LoggingTarget = \_target,

MessageId = msgId,

Time = sharedTime,

});

}

return Task.CompletedTask;

}

async Task<UpdateMessageLog?> GetPrevEdit(int messageId)

{

var prevEdit = UpdatesLogs.OfType<UpdateMessageLog>()

.OrderBy(l => l.Time).FirstOrDefault(l => l.MessageId == messageId);

if (prevEdit == null)

{

using var context = new ApplicationContext();

prevEdit = await context.UpdatesMessage

.AsNoTracking()

.Where(l =>

l.LoggingTargetId == \_target.Id &&

l.LoggingTarget.PeerId == PeerId)

.OrderByDescending(l => l.Time)

.FirstOrDefaultAsync(l => l.MessageId == messageId);

}

return prevEdit;

}

}

}

**Core/Tools/LongExtension.cs**

using System.Text;

namespace Core.Tools

{

public static class LongExtension

{

static readonly char[] URL\_SAFE\_ALPHABET = [

'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',

's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',

'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '1', '2',

'3', '4', '5', '6', '7', '8', '9', '0', '-', '\_'];

public static string ToEncodedString(this long userId)

{

StringBuilder ret = new();

long offset = userId % URL\_SAFE\_ALPHABET.Length;

while (userId > 0)

{

ret.Append(URL\_SAFE\_ALPHABET[offset + userId % 10]);

userId /= 10;

}

return ret.ToString();

}

}

}

**Backend/Tools/TimeStampConvertor.cs**

namespace Backend.Tools

{

public static class TimeStampConvertor

{

public readonly static DateTime \_dt1970;

static TimeStampConvertor()

{

\_dt1970 = new DateTime(1970, 1, 1, 0, 0, 0, 0, DateTimeKind.Utc);

}

public static DateTime IntToDatetime(int unixSeconds)

{

ArgumentOutOfRangeException.ThrowIfNegative(unixSeconds);

// Unix timestamp is seconds past epoch

return \_dt1970.AddSeconds(unixSeconds).ToUniversalTime();

}

public static DateTime LongToDatetime(long unixMilliseconds)

{

ArgumentOutOfRangeException.ThrowIfNegative(unixMilliseconds);

return \_dt1970.AddMilliseconds(unixMilliseconds).ToUniversalTime();

}

public static DateTime Diff(DateTime arg1, DateTime arg2)

{

return \_dt1970 + (arg1 - arg2);

}

public static long DatetimeToLong(DateTime time)

{

var unix = time.Subtract(\_dt1970);

return Convert.ToInt64(unix.TotalMilliseconds);

}

public static string ToTextRange(TimeSpan timeSpan)

{

int weeks = timeSpan.Days / 7;

if (timeSpan.TotalNanoseconds < 0) return "";

if (weeks > 0)

{

var residue = weeks % 10;

if (weeks == 1 || weeks > 20 && residue == 1)

return $" на {weeks} неделю";

else if ((weeks < 10 || weeks > 20) && residue > 1 && residue < 5)

return $" на {weeks} недели";

else

return $" на {weeks} недель";

}

else

{

if (timeSpan.Days > 0)

{

var residue = timeSpan.Days % 10;

if (timeSpan.Days == 1 || timeSpan.Days > 20 && residue == 1)

return $" на {timeSpan.Days} день";

else if ((timeSpan.Days < 10 || timeSpan.Days > 20) && residue > 1 && residue < 5)

return $" на {timeSpan.Days} дня";

else

return $" на {timeSpan.Days} дней";

}

else

{

if (timeSpan.Hours > 0)

{

var residue = timeSpan.Hours % 10;

if (timeSpan.Hours == 1 || timeSpan.Hours > 20 && residue == 1)

return $" на {timeSpan.Hours} час";

else if ((timeSpan.Hours < 10 || timeSpan.Hours > 20) && residue > 1 && residue < 5)

return $" на {timeSpan.Hours} часа";

else

return $" на {timeSpan.Hours} часов";

}

else

{

if (timeSpan.Minutes > 0)

{

var residue = timeSpan.Minutes % 10;

if (timeSpan.Minutes == 1 || timeSpan.Minutes > 20 && residue == 1)

return $" на {timeSpan.Minutes} минуту";

else if (residue > 1 && residue < 5)

return $" на {timeSpan.Minutes} минуты";

else

return $" на {timeSpan.Minutes} минут";

}

else

return "";

}

}

}

}

}

}

**Core/Workers/LoadedAccountsWorker.cs**

using Config;

using Core.Interfaces;

using Database;

using Microsoft.EntityFrameworkCore;

using WTelegram;

using TL;

using User = Database.Entities.User;

using Core.Workers.Types;

using System.Net.Sockets;

namespace Core.Workers

{

public class LoadedAccountsWorker : IWorker

{

static List<LoadedAccount> LoadedAccounts = [];

static bool NoInternetConnection = false;

async Task IWorker.Handle(ApplicationContext context)

{

if (NoInternetConnection && !await CheckInternetConnection()) return; // если интернет пропал, останавливаем работу

var triggerDate = DateTime.UtcNow.AddMinutes(-5);

List<LoadedAccount> actual = [];

foreach (var loadedAccount in LoadedAccounts)

if (loadedAccount.InUse == false &&

loadedAccount.LastTrigger < triggerDate)

await loadedAccount.Client.DisposeAsync();

else

{

if (loadedAccount.Client.Disconnected || loadedAccount.Client.User == null)

await Reconnect(loadedAccount);

actual.Add(loadedAccount);

}

LoadedAccounts = actual;

}

static Client GetNewClient(long userId, string phoneNumber)

=> new(ProgramConfig.TelegramApiAuth.ApiId,

ProgramConfig.TelegramApiAuth.ApiHash,

ProgramConfig.TelegramApiAuth.SessionsDir + $"{userId}\_{phoneNumber}");

static Client GetRegisteredClient(long userId, string phoneNumber)

=> new((what) => what switch

{

"api\_id" => ProgramConfig.TelegramApiAuth.ApiId.ToString(),

"api\_hash" => ProgramConfig.TelegramApiAuth.ApiHash,

"phone\_number" => phoneNumber,

"session\_pathname" => ProgramConfig.TelegramApiAuth.SessionsDir + $"{userId}\_{phoneNumber}",

\_ => null

});

static async Task<LoadedAccount> StarterNew(long userId, string phoneNumber)

{

var lAcc = await Starter(GetNewClient, userId, phoneNumber, true);

await lAcc.Client.ConnectAsync();

lAcc.Status = LoadedAccount.Statuses.Unknown;

return lAcc;

}

static async Task<LoadedAccount> StarterExist(long userId, string phoneNumber, bool canUpdate = false)

{

var lAcc = await Starter(GetRegisteredClient, userId, phoneNumber, canUpdate);

try

{

var task = lAcc.Client.LoginUserIfNeeded();

task.Wait(10\_000);

}

catch (Exception ex)

{

Console.WriteLine(ex.Message);

}

if (canUpdate || lAcc.Status == "")

{

lAcc.Status = lAcc.Client.User == null ? LoadedAccount.Statuses.Unknown : LoadedAccount.Statuses.Logged;

}

return lAcc;

}

static async Task<LoadedAccount> Starter(Func<long, string, Client> getClient, long userId, string phoneNumber, bool canUpdate)

{

LoadedAccount? lAcc = LoadedAccounts.FirstOrDefault(la => la.PhoneNumber == phoneNumber);

if (lAcc == null)

{

lAcc = new LoadedAccount()

{

Client = getClient(userId, phoneNumber),

PhoneNumber = phoneNumber,

OwnerId = userId,

};

LoadedAccounts.Add(lAcc);

}

else if (canUpdate)

{

await lAcc.Client.DisposeAsync();

lAcc.Client = getClient(userId, phoneNumber);

}

return lAcc;

}

static async Task Reconnect(LoadedAccount lacc)

{

if (!await CheckInternetConnection()) return;

try

{

await lacc.Client.DisposeAsync();

lacc.Client = GetRegisteredClient(lacc.OwnerId, lacc.PhoneNumber);

await lacc.Reconnect();

}

catch (SocketException)

{

await CheckInternetConnection();

}

catch (Exception ex)

{

Console.Error.WriteLine(ex.ToString());

}

}

/// <summary>

///

/// </summary>

/// <param name="phoneNumber"></param>

/// <param name="userId"></param>

/// <returns>Status</returns>

/// <exception cref="AccessViolationException"></exception>

/// <exception cref="ArgumentException"></exception>

public static async Task<string> OpenNewAccount(

ApplicationContext context,

string phoneNumber,

long userId)

{

LoadedAccount? lAcc = null;

var dbAccount = await context.Accounts

.Include(a => a.AccountLogs)

.FirstOrDefaultAsync(a => a.PhoneNumber == phoneNumber);

if (dbAccount != null)

{

if (dbAccount.OwnerId != userId)

throw new AccessViolationException(

"You cannot log in to this account because it belongs to another user");

lAcc = await StarterExist(userId, phoneNumber);

lAcc.InUse = true;

if (lAcc.Status == LoadedAccount.Statuses.Logged) // Аккаунт существует, повторный вход не нужен.

{

if (dbAccount.Status != Database.Enum.AccountStatus.Active)

{

await AccountDbLog(

context,

dbAccount,

Database.Enum.AccountStatus.Active,

Database.Enum.AccountLogType.Login,

"Login when try login repeate");

}

lAcc.InUse = false;

return lAcc.Status;

}

else

{

await AccountDbLog(

context,

dbAccount,

Database.Enum.AccountStatus.Broken,

Database.Enum.AccountLogType.Broke,

$"Can't start registered account at {DateTime.UtcNow}");

}

}

lAcc = await StarterNew(userId, phoneNumber);

var user = User.CreateAndAttach(context, userId);

if (dbAccount == null)

{

dbAccount = new()

{

Owner = user,

PhoneNumber = phoneNumber,

};

await context.Accounts.AddAsync(dbAccount);

}

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Opening, Database.Enum.AccountLogType.OpenNew, "", user);

lAcc.IdInDb = dbAccount.Id;

try

{

lAcc.Status = await lAcc.Client.Login(phoneNumber);

}

catch (Exception ex)

{

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Broken, Database.Enum.AccountLogType.Broke, ex.Message[..512], user);

}

if (lAcc.Client.User != null)

{

await AccountDbLog(

context,

dbAccount,

Database.Enum.AccountStatus.Active,

Database.Enum.AccountLogType.Login,

"Login after opening session");

lAcc.Status = LoadedAccount.Statuses.Logged;

}

else if (lAcc.Status != LoadedAccount.Statuses.Code && lAcc.Client.User == null)

{

string errorMessage = $"CurrentStatus: {lAcc.Status} but expected \"{LoadedAccount.Statuses.Code}\"";

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Broken, Database.Enum.AccountLogType.Broke, errorMessage, user);

lAcc.InUse = false;

throw new ArgumentException(errorMessage);

}

lAcc.Trigger();

lAcc.InUse = false;

return lAcc.Status;

}

/// <summary>

///

/// </summary>

/// <param name="phoneNumber"></param>

/// <param name="code"></param>

/// <returns>Status</returns>

/// <exception cref="ArgumentNullException"></exception>

/// <exception cref="InvalidOperationException"></exception>

public static async Task<string> SetCodeToNewAccount(

ApplicationContext context,

string phoneNumber,

string code)

{

LoadedAccount? lAcc = LoadedAccounts.FirstOrDefault(la => la.PhoneNumber == phoneNumber)

?? throw new ArgumentNullException(

$"Can't find account by phone number {phoneNumber}");

lAcc.InUse = true;

if (lAcc.Status != LoadedAccount.Statuses.Code)

{

lAcc.InUse = false;

throw new InvalidOperationException($"Status must be \"{LoadedAccount.Statuses.Code}\"");

}

var dbAccount = await context.Accounts

.Include(a => a.AccountLogs)

.FirstOrDefaultAsync(a => a.PhoneNumber == phoneNumber);

if (dbAccount == null)

{

lAcc.InUse = false;

throw new ArgumentNullException($"Can't find account in DB by phone number {phoneNumber}");

}

User user = User.CreateAndAttach(context, lAcc.OwnerId);

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Opening, Database.Enum.AccountLogType.SetCode, "", user);

await context.SaveChangesAsync();

try

{

lAcc.Status = await lAcc.Client.Login(code);

}

catch (Exception ex)

{

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Broken, Database.Enum.AccountLogType.Broke, ex.Message[..512], user);

}

if (lAcc.Client.User != null)

{

await AccountDbLog(

context,

dbAccount,

Database.Enum.AccountStatus.Active,

Database.Enum.AccountLogType.Login,

"Login after set code");

lAcc.Status = LoadedAccount.Statuses.Logged;

}

lAcc.Trigger();

lAcc.InUse = false;

return lAcc.Status;

}

/// <summary>

///

/// </summary>

/// <param name="context"></param>

/// <param name="phoneNumber"></param>

/// <param name="password"></param>

/// <returns>Status</returns>

/// <exception cref="ArgumentNullException"></exception>

/// <exception cref="InvalidOperationException"></exception>

public static async Task<string> SetCloudPasswordToNewAccount(

ApplicationContext context,

string phoneNumber,

string password)

{

LoadedAccount? lAcc = LoadedAccounts.FirstOrDefault(la => la.PhoneNumber == phoneNumber)

?? throw new ArgumentNullException($"Can't find account by phone number {phoneNumber}");

lAcc.InUse = true;

if (lAcc.Status != LoadedAccount.Statuses.Password)

{

lAcc.InUse = false;

throw new InvalidOperationException($"Status must be \"{LoadedAccount.Statuses.Password}\"");

}

var dbAccount = await context.Accounts

.Include(a => a.AccountLogs)

.FirstOrDefaultAsync(a => a.PhoneNumber == phoneNumber);

if (dbAccount == null)

{

lAcc.InUse = false;

throw new ArgumentNullException($"Can't find account in DB by phone number {phoneNumber}");

}

User user = User.CreateAndAttach(context, lAcc.OwnerId);

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Opening, Database.Enum.AccountLogType.SetPass, "", user);

try

{

lAcc.Status = await lAcc.Client.Login(password);

}

catch (Exception ex)

{

await AccountDbLog(context, dbAccount, Database.Enum.AccountStatus.Broken, Database.Enum.AccountLogType.Broke, ex.Message[..512], user);

}

if (lAcc.Client.User != null)

{

await AccountDbLog(

context,

dbAccount,

Database.Enum.AccountStatus.Active,

Database.Enum.AccountLogType.Login,

"Login after set password");

lAcc.Status = LoadedAccount.Statuses.Logged;

}

lAcc.Trigger();

lAcc.InUse = false;

return lAcc.Status;

}

/// <summary>

///

/// </summary>

/// <param name="client"></param>

/// <param name="peer"></param>

/// <returns>Photo's filename</returns>

public static async Task<string> DownloadAvatar(Client client, IPeerInfo peer)

{

string filename = "avatar\_" + peer.ID;

using var fileStream = new FileStream(ProgramConfig.TelegramApiAuth.DownloadsDir + filename, FileMode.Create, FileAccess.ReadWrite);

var photoInfo = await client.DownloadProfilePhotoAsync(peer, fileStream);

fileStream.Close();

return filename;

}

public static async Task<LoadedAccount> Get(long userId, string phoneNumber)

{

LoadedAccount? lAcc = LoadedAccounts.FirstOrDefault(la => la.PhoneNumber == phoneNumber && la.OwnerId == userId);

lAcc ??= await StarterExist(userId, phoneNumber);

lAcc.Trigger();

return lAcc;

}

/// <summary>

/// This method writes to the database

/// </summary>

/// <param name="context"></param>

/// <param name="dbAccount"></param>

/// <param name="generalStatus"></param>

/// <param name="logType"></param>

/// <param name="logDescription"></param>

/// <param name="ByUser"></param>

/// <returns></returns>

static async Task AccountDbLog(

ApplicationContext context,

Database.Entities.Account dbAccount,

Database.Enum.AccountStatus generalStatus,

Database.Enum.AccountLogType logType,

string logDescription,

User? ByUser = null)

{

dbAccount.Status = generalStatus;

dbAccount.AccountLogs ??= [];

dbAccount.AccountLogs.Add(new()

{

Account = dbAccount,

Type = logType,

Time = DateTime.UtcNow,

ByUser = ByUser,

Description = logDescription,

});

await context.SaveChangesAsync();

}

readonly static HttpClient \_checkClient = new();

static async Task<bool> CheckInternetConnection()

{

try

{

await \_checkClient.GetAsync("https://google.com");

if (NoInternetConnection)

{

NoInternetConnection = false;

Console.WriteLine("Internet available");

}

return true;

}

catch

{

if (NoInternetConnection == false)

{

Console.WriteLine("Internet not available");

NoInternetConnection = true;

}

return false;

}

}

}

}

**Core/Workers/SessionWorker.cs**

using Core.Interfaces;

using Core.Tools;

using Database;

using Database.Entities;

using Microsoft.EntityFrameworkCore;

using System.Security.Cryptography;

using System.Text;

namespace Core.Workers

{

public class SessionWorker : IWorker

{

static List<Types.Session> Sessions = [];

async Task IWorker.Handle(ApplicationContext context)

{

bool needSave = false;

List<Types.Session> actual = [];

foreach (Types.Session session in Sessions)

{

if (session.LastTrigger.AddSeconds(15) <= DateTime.UtcNow)

{

var dbEntity = await context.Sessions.FirstOrDefaultAsync(s => s.Id == session.Id);

if (dbEntity != null)

{

dbEntity.DiedAt = DateTime.UtcNow;

dbEntity.Status = Database.Enum.SessionStatus.Died;

needSave = true;

}

}

else actual.Add(session);

}

if (needSave)

{

Sessions = actual;

await context.SaveChangesAsync();

}

}

public static async Task<string> OpenNew(ApplicationContext context, string hash, long userId)

{

byte[] sessionCode = HMACSHA256.HashData(

Encoding.UTF8.GetBytes(userId.ToEncodedString()),

Encoding.UTF8.GetBytes(hash));

Session? currentSession = await context.Sessions

.FirstOrDefaultAsync(s =>

s.ToUser.Id == userId &&

s.Status == Database.Enum.SessionStatus.Active);

if (currentSession != null)

{

currentSession.Status = Database.Enum.SessionStatus.Revoked;

currentSession.DiedAt = DateTime.UtcNow;

if (Sessions.Any(s => s.Id == currentSession.Id))

Sessions.Remove(Sessions.First(s => s.Id == currentSession.Id));

}

User user = User.CreateAndAttachOrUpdate(context, userId);

var createdAt = DateTime.UtcNow;

var newSession = new Session()

{

CreatedAt = createdAt,

Status = Database.Enum.SessionStatus.Active,

ToUser = user,

DiedAt = null,

Code = Convert.ToBase64String(

HMACSHA256.HashData(

sessionCode,

Encoding.UTF8.GetBytes(createdAt.ToString("dd.MM.yyyy\_HH:mm:ffffff")))),

};

await context.Sessions.AddAsync(newSession);

await context.SaveChangesAsync();

Sessions.Add(new(newSession));

return newSession.Code;

}

public static string GetCodeByUser(long userId)

{

var tmp = Sessions.FirstOrDefault(s => s.ToUserId == userId);

if (tmp == null)

return "\_";

tmp.Trigger();

return tmp.Code ?? "\_";

}

}

}

**Core/Workers/Types/LoadedAccount.cs**

using WTelegram;

namespace Core.Workers.Types

{

public class LoadedAccount

{

public required Client Client { get; set; }

public required string PhoneNumber { get; set; } // Uses as unique id

public string Status { get; set; } = "";

public DateTime LoginTime { get; set; } = DateTime.UtcNow;

public DateTime LastTrigger { get; set; } = DateTime.UtcNow;

public required long OwnerId { get; set; }

public bool InUse { get; set; } = false;

public long IdInDb { get; set; } = 0;

public event Action? OnRestarted;

public DateTime \_lastRestart = DateTime.MinValue;

public async Task Reconnect()

{

if (\_lastRestart.AddSeconds(10) > DateTime.UtcNow)

return;

await Client.LoginUserIfNeeded();

if (Client.User != null)

{

OnRestarted?.Invoke();

Status = Statuses.Logged;

}

}

public void Trigger()

{

LastTrigger = DateTime.UtcNow;

}

public readonly struct Statuses

{

public static readonly string Unknown = "Unknown";

public static readonly string Logged = "Logged in";

public static readonly string Code = "verification\_code";

public static readonly string Password = "password";

}

}

}

**Core/Workers/Types/Session.cs**

namespace Core.Workers.Types

{

class Session

{

public Session(Database.Entities.Session source)

{

Id = source.Id;

Code = source.Code;

ToUserId = source.ToUser.Id;

Trigger();

}

public ulong Id { get; set; }

public string Code { get; set; }

public long ToUserId { get; set; }

public DateTime LastTrigger { get; private set; }

public void Trigger()

{

LastTrigger = DateTime.UtcNow;

}

}

}

**Database/ApplicationContext.cs**

using Config;

using Database.Entities;

using Microsoft.EntityFrameworkCore;

namespace Database

{

public class ApplicationContext : DbContext

{

public DbSet<User> Users { get; set; }

public DbSet<Session> Sessions { get; set; }

public DbSet<Account> Accounts { get; set; }

public DbSet<AccountLog> AccountHistory { get; set; }

public DbSet<LoggingTarget> Targets { get; set; }

public DbSet<UpdateLog> Updates { get; set; }

public DbSet<UpdateMessageLog> UpdatesMessage { get; set; }

public DbSet<UpdateDeleteMessageLog> UpdatesDeleteMessage { get; set; }

protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)

{

optionsBuilder.UseMySql(

ProgramConfig.DatabaseAuth.ConnectionString,

new MySqlServerVersion("1.0"), mySqlOptionsAction =>

{

mySqlOptionsAction.EnableRetryOnFailure(

maxRetryCount: 3,

maxRetryDelay: TimeSpan.FromSeconds(5),

errorNumbersToAdd: null);

});

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

modelBuilder.Entity<UpdateMessageLog>().ToTable("MessageUpdates");

modelBuilder.Entity<UpdateDeleteMessageLog>().ToTable("DeleteMessageUpdates");

}

}

}

**Database/Entities/Account.cs**

using Database.Enum;

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace Database.Entities

{

[Index(nameof(PhoneNumber))]

public class Account

{

public long Id { get; set; }

[NotMapped]

public long TelegramId { get; set; } // reserved

[MaxLength(50)]

public required string PhoneNumber { get; set; }

public required User Owner { get; set; }

public long OwnerId { get; set; }

public AccountStatus Status { get; set; }

public List<AccountLog>? AccountLogs { get; set; }

public DateTime CreatedAt { get; set; } = DateTime.UtcNow;

}

}

**Database/Entities/AccountLog.cs**

using Database.Enum;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

namespace Database.Entities

{

public class AccountLog

{

public long Id { get; set; }

public AccountLogType Type { get; set; }

[MaxLength(512)]

public string? Description { get; set; }

public required Account Account { get; set; }

public User? ByUser { get; set; }

public DateTime Time { get; set; } = DateTime.UtcNow;

[NotMapped]

[MaxLength(512)]

public string? RawData { get; set; } // reserved

}

}

**Database/Entities/LoggingTarget.cs**

using Database.Enum;

using Microsoft.EntityFrameworkCore;

using System.ComponentModel.DataAnnotations;

namespace Database.Entities

{

[Index(nameof(PeerId))]

public class LoggingTarget

{

public long Id { get; set; }

public long FromAccountId { get; set; }

public required Account FromAccount { get; set; }

public required long PeerId { get; set; }

public required long? AccessHash { get; set; }

public DateTime Time { get; set; } = DateTime.UtcNow;

public LoggingTargetType Type { get; set; }

public LoggingTargetStatus Status { get; set; }

public List<UpdateLog> Logs { get; set; } = [];

[MaxLength(100)]

public string? Error { get; set; } = null; // not null when status is Failure (-1)

public long? PrevTargetId { get; set; } = null;

public LoggingTarget? PrevTarget { get; set; } = null;

}

}

**Database/Entities/Session.cs**

using Database.Enum;

namespace Database.Entities

{

public class Session

{

public ulong Id { get; set; }

public required string Code { get; set; }

public long ToUserId { get; set; }

public required User ToUser { get; set; }

public DateTime CreatedAt { get; set; }

public DateTime? DiedAt { get; set; }

public SessionStatus Status { get; set; }

}

}

**Database/Entities/UpdateDeleteMessageLog.cs**

using System.ComponentModel.DataAnnotations.Schema;

namespace Database.Entities

{

public class UpdateDeleteMessageLog : UpdateLog

{

public int MessageId { get; set; }

}

}

**Database/Entities/UpdateLog.cs**

namespace Database.Entities

{

public class UpdateLog

{

public long Id { get; set; }

//public UpdateType Type { get; set; } // Не понадобится (скорее всего)

public DateTime Time { get; set; }

public long LoggingTargetId { get; set; }

public required LoggingTarget LoggingTarget { get; set; }

}

}

**Database/Entities/UpdateMessageLog.cs**

using System.ComponentModel.DataAnnotations;

namespace Database.Entities

{

public class UpdateMessageLog : UpdateLog

{

public int MessageId { get; set; }

[MaxLength(4096)]

public required string Text { get; set; }

public required string TextEntities { get; set; } // https://core.telegram.org/type/MessageEntity

public long FromId { get; set; }

public long? PrevEditId { get; set; }

public UpdateMessageLog? PrevEdit { get; set; } // Если сообщение изменено,

// то это поле указывает на предыдущую версию сообщения

public DateTime? MsgDate { get; set; }

// public MediaType? MediaType {get; set;}

// [MaxLength(256)]

// public string? MediaUrl {get; set;}

}

}

**Database/Entities/User.cs**

using System.ComponentModel.DataAnnotations;

namespace Database.Entities

{

public class User

{

public required long Id { get; set; } // Вместо AUTO\_INCREMENT будут айдишники из телеграма.

[MaxLength(64)]

public required string FirstName { get; set; }

[MaxLength(64)]

public string? LastName { get; set; }

[MaxLength(32)]

public string? Username { get; set; }

[MaxLength(10)]

public string? LanguageCode { get; set; }

public static User CreateAndAttach(ApplicationContext context, long userId)

{

var user = new User() { Id = userId, FirstName = "" };

context.Users.Attach(user);

return user;

}

public static User CreateAndAttachOrUpdate(ApplicationContext context, long userId)

{

var user = new User() { Id = userId, FirstName = "" };

try

{

context.Users.Attach(user);

}

catch

{

context.Users.Update(user);

context.Users.Attach(user);

}

return user;

}

}

}

**Database/Enum/AccountLogType.cs**

namespace Database.Enum

{

public enum AccountLogType

{

None = 0,

OpenNew = 1,

SetCode = 2,

SetPass = 3,

Remove = 4,

Login = 10,

Logout = 11,

Connect = 12,

Disconnect = 13,

Broke = 20,

}

}

**Database/Enum/AccountStatus.cs**

namespace Database.Enum

{

public enum AccountStatus

{

None = 0,

Active = 1,

Broken = 2,

Closed = 3,

Opening = 10,

}

}

**Database/Enum/LoggingTargetStatus.cs**

namespace Database.Enum

{

public enum LoggingTargetStatus

{

Failure = -1,

Unknown = 0,

Enabled = 1,

Disabled = 2,

Deleted = 10,

Banned = 11,

}

}

**Database/Enum/LoggingTargetType.cs**

namespace Database.Enum

{

public enum LoggingTargetType

{

Unknown = 0,

Messages = 1,

}

}

**Database/Enum/SessionStatus.cs**

namespace Database.Enum

{

public enum SessionStatus

{

Died = -1,

None = 0,

Active = 1,

Revoked = 2,

}

}

**Database/Enum/UpdateType.cs**

namespace Database.Enum

{

public enum UpdateType

{

Unknown = 0,

// 1 - 99 - messages

NewMessage = 1,

EditMessage = 2,

DeleteMessage = 3,

OnlineStatus = 100

}

}