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.

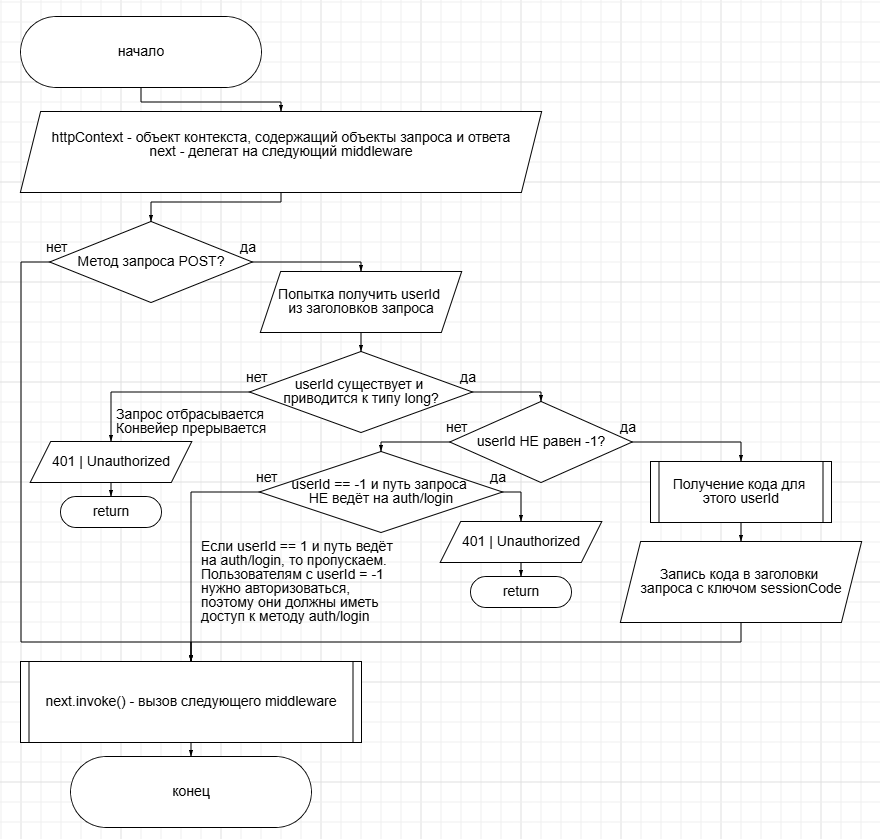


Рисунок 2 – схема работы промежуточного слоя авторизации

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