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

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

# 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.

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, но сейчас нас интересует именно авторизация.

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.

Звучит довольно просто, но как этим пользоваться? В подключаемой библиотеке есть поле *initData*. Это поле храни 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\_stirng 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.

## Вид пакета

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:

Таблица 1 - Базовый функционал пакета

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

Пришедший пакет необходимо проверить Verify, а новый подписать перед отправкой Sign.

Чтобы базовый пакет мог спокойно подписать данные, все пользовательские классы 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.

Стандартная реализация метода 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);

if (value != null)

if (value is DateTime time)

ret.Add(property.Name, TimeStampConvertor.DatetimeToLong(time));

else if (value is ParsebleToDictionaryBase parseble)

ret.Add(property.Name, parseble.ToDict());

else if (value is IEnumerable<ParsebleToDictionaryBase> values)

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

else

ret.Add(property.Name, value);

}

return ret;

Мы получаем тип объекта this, и у него получаем все свойства, кроме тех что указаны с атрибутом NotMappedAttribute. Это позволит нам помечать некоторые свойства так, чтобы метод ToDict их игнорировал. Далее, мы проходимся по списку свойств и записываем их в структуру Dictionary<string, object>, но для некоторых типов данных нужен собственных подход.

Например, тип DateTime, мы конвертируем в unix формат с миллисекундами. Конвертация выполняется с помощью нехитрой манипуляции:

var unix = time.Subtract(\_dt1970);

return Convert.ToInt64(unix.TotalMilliseconds);

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

Далее, если значение свойства приводится к Backend.Tools.ParsebleToDictionaryBase, то мы добавляем результат вызова ToDict для этого значения.

Если значение свойства – список с элементами, которые приводятся к Backend.Tools.ParsebleToDictionaryBase, то вызываем для каждого ToDict и добавляем массив из результатов вызовов ToDict.

Для остальных типов, добавляем «как есть».

## Авторизация и аутентификация

Авторизация будет осуществляться по заголовку userId в запросе к серверу, а аутентификация в свою очередь 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 апи 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 мы видим, 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.

Для получения преобразованных данных будем использовать закрытый статический метод DefaultTransform из HttpDataBase.

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);

}

}

}

}

Мы видим, что метод проходится по исходным данным и для каждого ключа добавляет перед ним ключ родителя, если он есть, а потом сравнивает тип значения.

Если тип – это Dictionary<string, object>, то вызываем рекурсивно DefaultTransform для этого значения, а в аргумент родительского ключа передаём текущий ключ (который в свою очередь тоже может состоять из комбинации ключей родителя и текущего элемента).

Если тип – это список, то создаём новый словарь, куда копируем все элементы из массива, но в качестве ключа словаря используем индексы массива. После, также вызываем рекурсивно DefaultTransform для только что составленного словаря.

Для других типов, пытаемся привести значение к словарю с помощью приведения к типу JsonElement и вызова для него метода Deserialize< Dictionary<string, object>>. В случае неудачи – перед нами примитивный тип, просто добавляем в результат без преобразований.

После сворачивания всех рекурсий получаем словарь, в котором ключи отсортированы и имеют вложенную структуру через точку. Но т.к. этот метод статический, то для его использования нам понадобиться другой метод.

Этот метод предоставляет удобный интерфейс для получения данных с отсортированными ключами, имеющие вложенную структуру.

protected virtual SortedDictionary<string, object> GetSortedParams()

{

var data = ToDict();

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

DefaulTransform(data, "", ref ret);

return ret;

}

Теперь, когда мы имеем данные, с правильными ключами, можем приступать к подписи:

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)))!;

}

Получаем все отсортированные ключи, записываем время создания пакета, формируем заголовок, а после, к нему добавляем данные. Здесь используется LINQ выражение. sortedParams представляет собой словарь. Метод Select позволяет пройтись по всем элементам словаря и из каждого элемента получить строку. В результате выполнения мы получим перечисление всех строк (IEnumerable<string>), которые в свою очередь отправим в статический метод string.Join, который сконкатенирует все строких из перечисления через разделитель. Таким образом мы получим строку для хеширования. После этого, с помощью метода HMACSHA256.HashData мы получаем хеш этой строки по ключу sessionCode. Стоит обратить внимание, что хеш функция работает с массивом байтов, поэтому нам нужно преобразовать нашу строку и код сессии в массив байтов. Для этого воспользуемся методом Encoding.UTF8.GetBytes. Хеш функция также возвращает результатом массив байтов, поэтому необходимо закодировать его в HEX строку, с помощью метода Convert.ToHexString. Полученную HEX строку записываем в поле signature. Пакет подписан.

Для валидации подписи пакета используется такая же схема, не считая незначительных изменений.

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 этом методе перед началом работы обработаны сразу два частных случая. Первый – если код сессии представляет собой один единственный символ – нижнее подчёркивание. Такое бывает, если код сессии истёк, а клиент не запросил новый.

Второй – если signature в полученном пакете отсутствует – методу нечего проверять.

Далее похожая схема как и в подписи. Получаем отсортированные ключи, убираем оттуда signature и ts (они не представляют собой данные). userId и ts записываем в заголовок, и добавляем к заголовкам строку с данными. Получаем результат хеш функции, и только что полученный хеш сравниваем с тем, что пришёл вместе с пакетом. signature.Equals(tmp, StringComparison.OrdinalIgnoreCase). Здесь StringComparison.OrdinalIgnoreCase используется для сравнения строк без учёта регистра, т.к. допускается запись HEX строки как в нижнем, так и в верхнем регистре. Если хеши не совпадают, то выбрасывается исключение throw new UnauthorizedAccessException("Signature do not match!")

## Добавление аккаунта

Чтобы подключиться к серверам телеграм, я использую библиотеку WTelegram. Для использования методов клиента, нужно создать объект класса Client. В библиотеке WTelegram есть несколько способов авторизации.

1. Можно 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. Можно 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.

Будем использовать оба способа в разных ситуациях. Первый для входа в новую сессию, а второй – для запуска уже существующих сессий.

При создании нового аккаунта, последовательно

## Время жизни аккаунта

За всеми запущенными аккаунтами будет следить AccountManager. Каждый аккаунт находится в объекте класса LoadedAccount, а в нём в свою очередь определены

## Назначение цели для отслеживания

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.