

Министерство образования Республики Беларусь
Учреждение образования
Учреждение образования «Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

Дисциплина: Компьютерные системы и сети (КСиС)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе на
тему

**ПРОГРАММНОЕ СРЕДСТВО
«Чат-приложение через веб-сокеты»**

БГУИР 6-05-0612-01 116 ПЗ

Студент: гр. 351001 Руденя Д.А.

Руководитель: Шамына А. Ю.

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	5
1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ.....	6
1.1 Обзор аналогов	6
1.2 Постановка задачи.....	9
2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА	10
2.1 Структура программы.....	10
2.2 Проектирование интерфейса программного средства.....	10
2.2.1 Главное окно	10
2.3 Проектирование функционала программного средства	16
2.3.1 Функция обработки клиента.....	16
2.3.2 Функция принятия сообщений.....	17
2.3.3 Функция отправки сообщений	17
3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА	18
3.1 Функции для работы с сетью и API.....	18
3.1.1 Инициализация и подключение к серверу	18
3.1.2 Авторизация пользователя.....	18
3.1.3 Загрузка и скачивание вложений.....	18
3.1.4 Отправка и получение сообщений.....	19
3.2 Функции работы интерфейса	19
3.2.1 Добавление сообщения в чат	19
3.2.2 Обновление интерфейса.....	19
3.3 Функции для обработки данных	19
3.3.1 Преобразование данных.....	19
3.3.2 Работа с файлами	20
4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА.....	21
5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ	22
5.1 Интерфейс программного средства	22
5.1.1 Экран отправки сообщений.....	22
5.2 Управление программным средством	23
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....	25
ПРИЛОЖЕНИЕ А	26
ПРИЛОЖЕНИЕ Б.....	29

ВВЕДЕНИЕ

Чат-приложения играют ключевую роль в современном мире, предоставляя пользователям возможность мгновенного обмена сообщениями. Такие приложения используются как для личного общения, так и для рабочих нужд, обеспечивая удобство и оперативность коммуникации. Известные аналоги, такие как WhatsApp, Telegram и Slack, продемонстрировали, насколько востребованы подобные решения и какие возможности они предоставляют.

Технической основой многих из этих приложений является модель клиент-серверной архитектуры с использованием сетевых сокетов. Сокеты представляют собой мощный инструмент для реализации передачи данных между устройствами, что делает их незаменимыми для создания надежных и высокопроизводительных систем обмена сообщениями. Например, WhatsApp использует протоколы, основанные на сокетах, для обеспечения минимальной задержки при передаче сообщений.

Актуальность разработки собственного чат-приложения обусловлена как образовательной, так и практической ценностью. В процессе создания такого проекта изучаются ключевые аспекты сетевого программирования, такие как обработка соединений, протоколы передачи данных, многопоточность и управление ресурсами. Более того, реализация такого приложения позволяет получить опыт, применимый в реальных проектах, будь то корпоративные системы обмена сообщениями или новые социальные платформы.

В данной работе рассматривается разработка чат-приложения на языке C# и Dart с использованием веб-сокетов, фреймворков ASP.NET и Flutter, а также протокола HTTP. Основное внимание уделяется обработке сетевых соединений, обеспечению безопасности передачи данных и эффективному взаимодействию клиентов с сервером. Такой подход не только позволяет понять базовые принципы работы сетевых приложений, но и создает прочную основу для дальнейших исследований и усовершенствований в данной области.

1 АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Обзор аналогов

Telegram - это удобный и многофункциональный мессенджер, предназначенный для обмена текстовыми и мультимедийными сообщениями. Его минималистичный интерфейс интуитивно понятен, что делает Telegram идеальным выбором как для личного общения, так и для работы. Благодаря использованию облачных технологий, приложение обеспечивает доступ к переписке с любых устройств, синхронизируя данные в реальном времени. Telegram поддерживает создание групповых чатов и каналов, что делает его подходящим инструментом для общения в больших коллективах и распространения информации.

Более продвинутые возможности Telegram включают использование ботов для автоматизации задач, интеграцию с API для разработки собственных решений и функции редактирования отправленных сообщений. Поддержка отправки файлов большого размера, настройка приватности и системы двухфакторной аутентификации подчеркивают его универсальность и безопасность. Telegram также предоставляет мощные инструменты для настройки уведомлений, управления медиафайлами и организации информации.

Пользователь может настроить внешний вид Telegram с помощью тем, анимаций и выбора цветовых схем. Это делает приложение визуально приятным и персонализированным для каждого.

Несмотря на широкий набор функций, Telegram остаётся лёгким в освоении, однако может показаться избыточным для пользователей, которые ищут исключительно базовые возможности для обмена сообщениями. Для максимально комфортного использования требуется определённое время на изучение всех возможностей и настроек приложения.

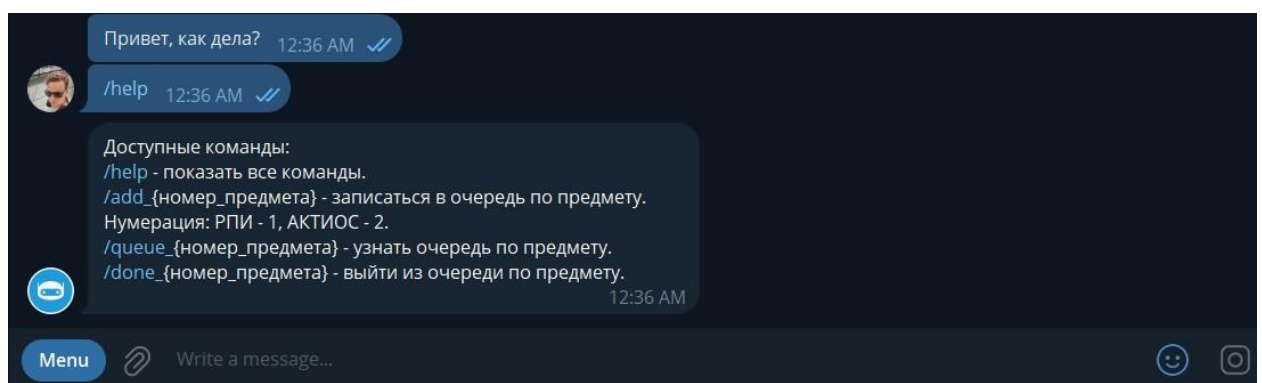


Рисунок 1.1 – Программное средство «Telegram»

WhatsApp - это популярный мессенджер, предлагающий простой и надежный способ общения. Интерфейс WhatsApp минималистичен и интуитивно понятен, что делает его удобным даже для неопытных пользователей. Основные функции включают обмен текстовыми сообщениями, голосовые и видеозвонки, а также отправку мультимедийных файлов, таких как изображения, видео и документы. Благодаря сквозному шифрованию, приложение обеспечивает высокий уровень безопасности, защищая переписку и звонки от несанкционированного доступа.

Для группового общения WhatsApp предлагает возможность создавать чаты с участием до 1024 человек и рассылать сообщения через списки рассылки. Функции резервного копирования данных в облако позволяют сохранять переписку и восстанавливать её при необходимости.

Дополнительные возможности включают настройку фонового изображения для чатов, создание личного статуса и использование реакции на сообщения. WhatsApp поддерживает голосовые сообщения и стикеры, что делает общение более живым и разнообразным. Пользователи могут также подключаться к WhatsApp Web или использовать десктопное приложение для работы с сообщениями на компьютере.

Несмотря на свои преимущества, WhatsApp имеет ограничения, такие как ограничение на размер отправляемых файлов (до 2 ГБ) и необходимость привязки к номеру телефона. Для пользователей, которым требуются более сложные функции автоматизации или интеграции, таких как в Telegram, WhatsApp может показаться менее функциональным. Однако его простота и надежность делают его универсальным инструментом для повседневного общения.



Рисунок 1.2 – Программное средство «WhatsApp»

Slack - это мощный инструмент для командной коммуникации, разработанный с акцентом на совместную работу и продуктивность. Интерфейс Slack организован вокруг рабочих пространств, которые можно настраивать под нужды конкретной команды или проекта. Основные функции включают обмен сообщениями в каналах, личных чатах, а также поддержку голосовых и видеозвонков. Slack упрощает управление общением, предоставляя удобные инструменты для организации обсуждений и поиска информации.

Одним из ключевых преимуществ Slack является возможность интеграции с широким спектром сторонних приложений и сервисов, таких как Google Drive, Trello и GitHub. Это позволяет автоматизировать процессы, управлять задачами и получать уведомления из других систем прямо в рабочем пространстве. Slack также поддерживает создание пользовательских ботов и автоматизацию через Workflow Builder, что делает его гибким инструментом для сложных проектов.

Для удобства работы Slack предлагает функции управления уведомлениями, такие как возможность настройки приоритетов и временного отключения звуков. Пользователи могут использовать встроенные фильтры для поиска сообщений, файлов и ссылок, а также закреплять важные материалы в каналах. Богатый выбор тем оформления и настройка интерфейса позволяют адаптировать приложение к личным предпочтениям.

Несмотря на широкий функционал, Slack может быть сложен для новичков из-за изобилия настроек и функций. Также бесплатная версия имеет ограничения, такие как ограниченный доступ к архиву сообщений и

ограничения на объем хранимых файлов. Тем не менее, для команд, нуждающихся в эффективной и масштабируемой системе коммуникации, Slack остаётся одним из лучших решений на рынке.

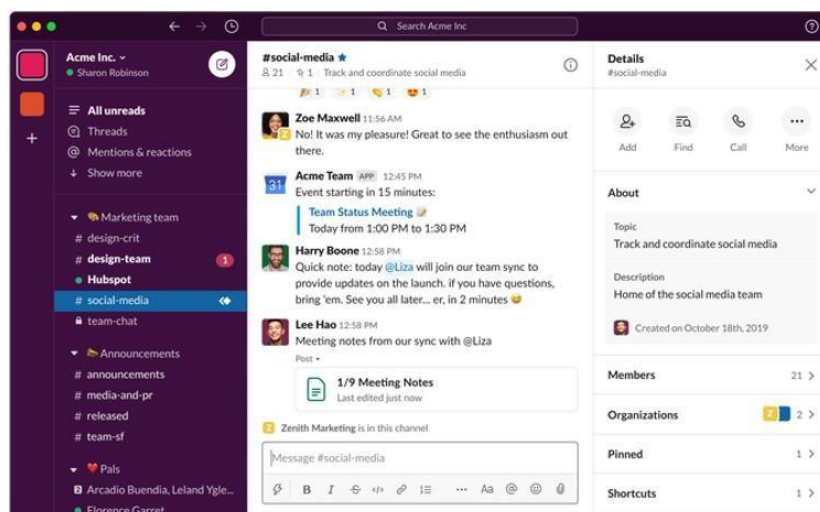


Рисунок 1.3 – Программное средство «Slack»

1.2 Постановка задачи

В рамках данного курсового проекта планируется разработка программного средства «Чат-приложение через веб-сокеты», адаптированного для запуска на современных смартфонах.

В процессе реализации будет разработан функционал для написания и отправки сообщений, логирования, регистрации, авторизации.

В Чат-приложении планируется реализовать следующие функции:

- Написание и отправка текстовых сообщений;
- Авторизация, регистрация, логирование;
- Работа с несколькими клиентами одновременно;
- Одновременный приём и отправка сообщений;
- Логика обработки сообщений сервером;
- Восстановление сообщений из логов;
- Отправку и сохранение файлов.

Для разработки серверной части программного средства будет использоваться язык программирования C# с использованием ASP.NET и для клиентской части Dart с использованием Flutter.

2 ПРОЕКТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

2.1 Структура программы

В процессе разработки чат-приложения было принято решение использовать следующую архитектуру:

Клиент – Настраивает работу отправляющих и принимающих сокетов, отправляет и принимает сообщения, отображая их пользователю, отвечает за регистрацию, авторизацию и подключение к серверу, позволяет пользователю выбирать получателя, также некоторые взаимодействия происходят через REST HTTP.

Сервер – Обработывает сообщения пользователей, попутно записывая их в базу данных и уведомляя получателей, обеспечивает проверки на корректность авторизации и регистрации пользователей, отвечает за отправку всей информации о пользователе, доступных контактов пользователю и статусов отправки, сохраняет веб-сокеты текущих пользователей.

Для клиента было выбрано сделать 5 экранов:

- Экран с чатами;
- Экран регистрации и авторизации;
- Экран обмена сообщениями;
- Экран с профилем пользователя;
- Экран с настройками.

2.2 Проектирование интерфейса программного средства

Для разработки интерфейса использовались стандартные виджеты Flutter.

2.2.1 Главное окно

Интерфейс приложения клиента изображен как рисунках 2.1, 2.2, 2.3, 2.4, 2.5.

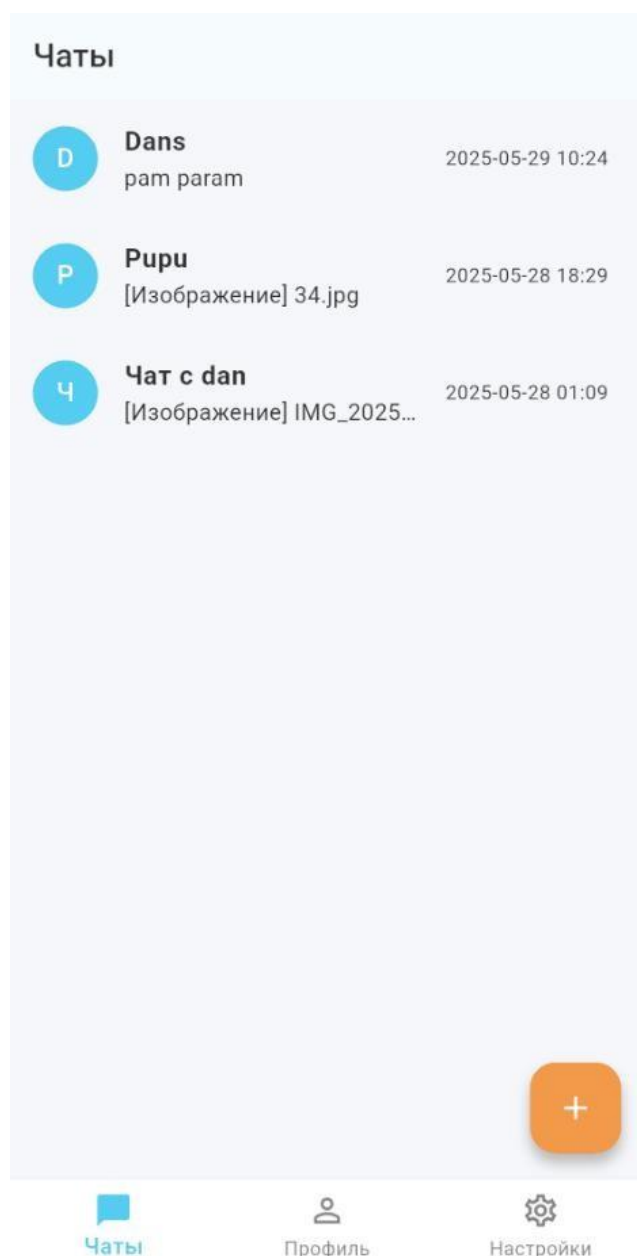
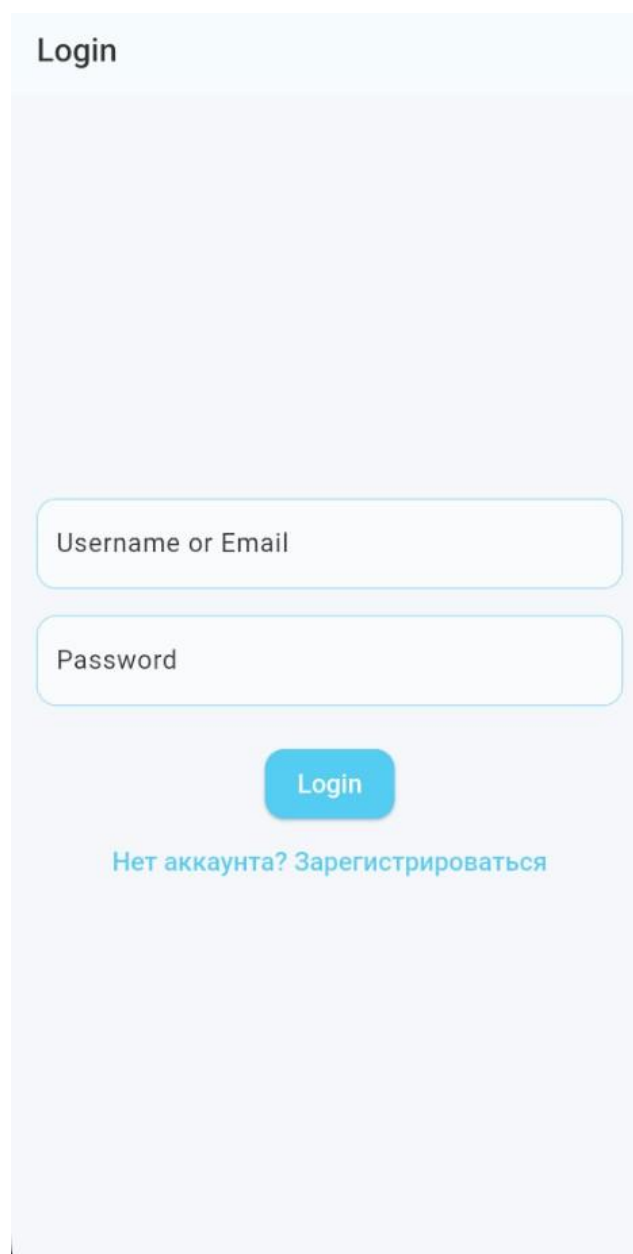


Рисунок 2.1 – Экран с чатами



The image shows a mobile application login screen. At the top, the word "Login" is displayed in a dark font. Below this, there are two input fields: the first is labeled "Username or Email" and the second is labeled "Password". Both fields have a light blue border. Below the password field is a blue button with the text "Login" in white. At the bottom of the screen, there is a link that says "Нет аккаунта? Зарегистрироваться" in a blue font.

Рисунок 2.2 – Экран регистрации и авторизации

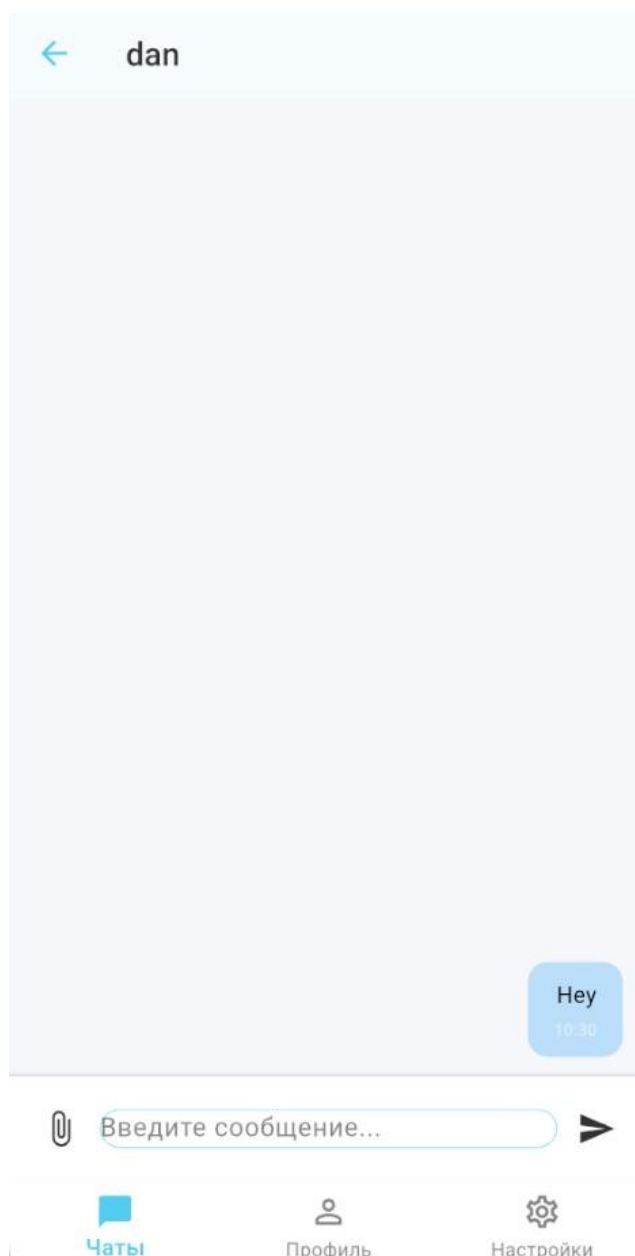


Рисунок 2.3 – Экран обмена сообщениями

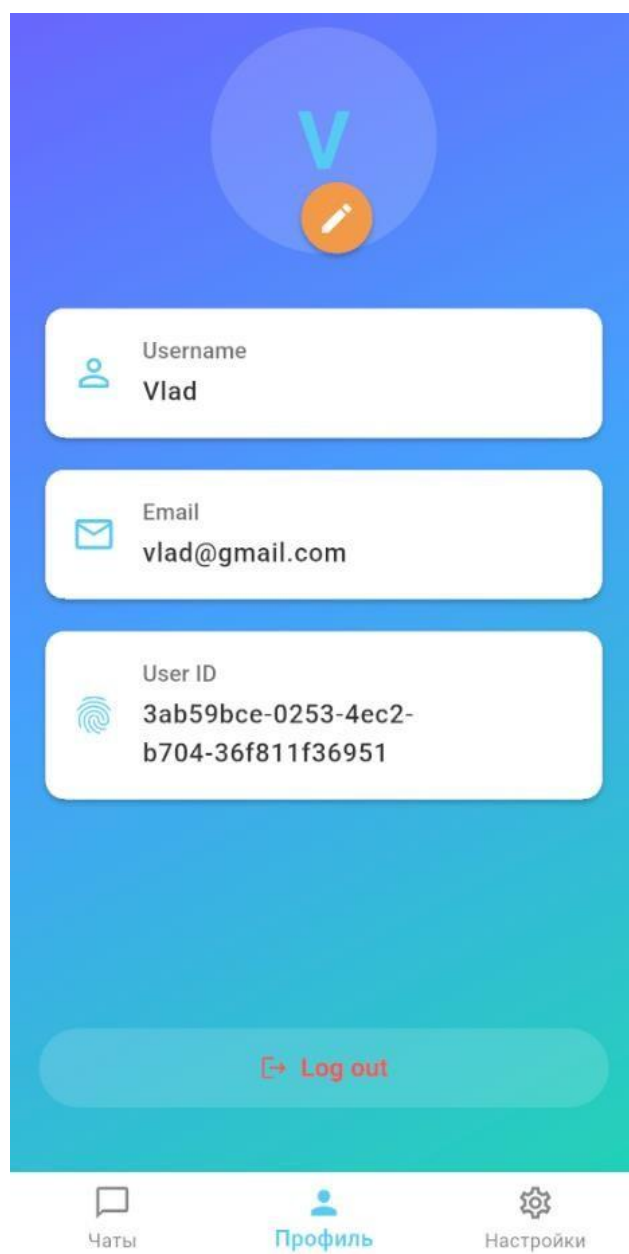


Рисунок 2.4 – Экран с профилем пользователя

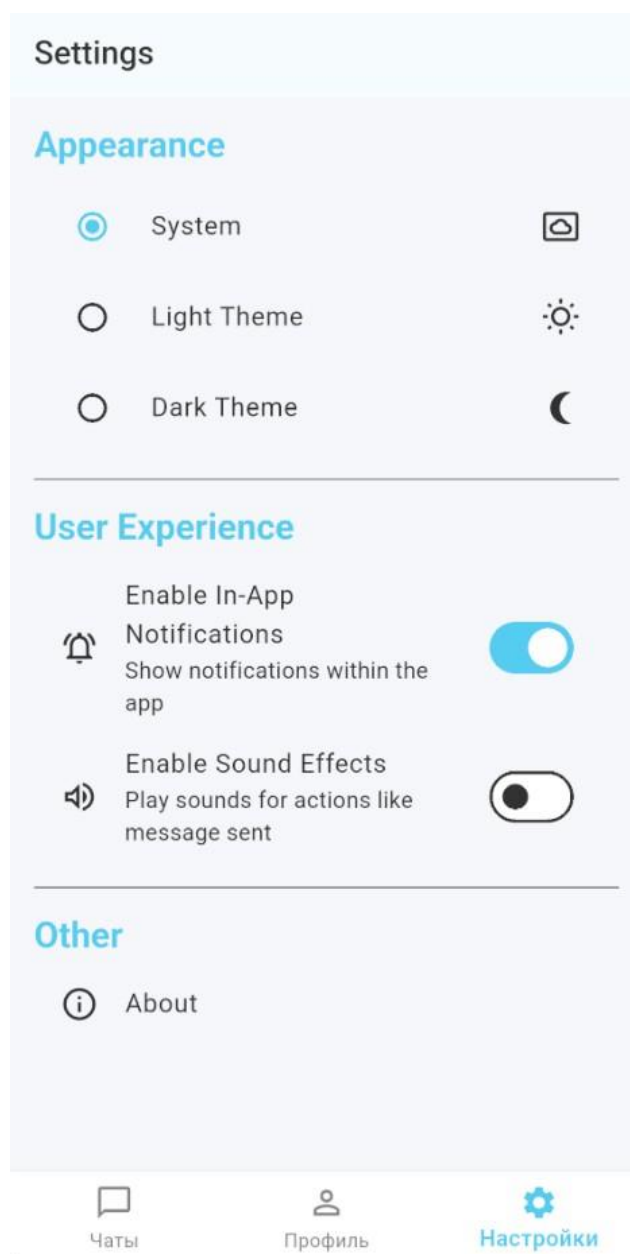


Рисунок 2.5 – Экран с настройками

Окно с чатами включает в себя компоненты отображения чатов, кнопку для создания нового чата или группы, навигационную панель снизу.

2.3 Проектирование функционала программного средства

Ключевым аспектом разработки чат-приложения является его функциональность, которая определяется набором алгоритмов, способом передачи данных, а также различными дополнительными возможностями, такими как логирование, регистрация, авторизация, работа с вложениями и т.д. С учетом этого программное средство должно предоставлять следующие функции:

Основные функции клиента:

- Отправка сообщений (через SignalR и REST API).
- Прием сообщений в реальном времени (SignalR).

Дополнительные функции клиента:

- Авторизация и регистрация пользователя через API.
- Ввод адреса сервера (или автоматическое определение для мобильных платформ).
- Выбор чата или собеседника.
- Восстановление истории сообщений (загрузка истории из базы данных через API).
- Отправка и скачивание вложений (файлов, изображений) с помощью presigned URL и MinIO.
- Отображение статуса "печатает", реакции на сообщения, уведомления о прочтении.

Основные функции сервера:

- Прием и пересылка сообщений между пользователями (SignalR Hub).
- Асинхронная обработка множества клиентов (SignalR, Web API).
- Хранение истории сообщений и пользователей (MongoDB/PostgreSQL).

Дополнительные функции сервера:

- Логирование действий пользователей и ошибок.
- Работа с файлами: генерация presigned URL для загрузки и скачивания вложений (MinIO).
- Проверка прав доступа к сообщениям и вложениям.
- Обработка регистрации и авторизации (JWT, хранение паролей в базе).

2.3.1 Функция обработки клиента

В Flutter-клиенте обработка пользователя реализуется через Блосархитектуру и взаимодействие с сервером по REST API и SignalR. После

авторизации пользователь может создавать чаты, отправлять и получать сообщения, а также работать с вложениями.

Пример инициализации соединения и авторизации:

```
final hubConnection = HubConnectionBuilder()
    .withUrl('${AppConfig.signalRBaseUrl}${AppConfig.chatHubUrl}', options:
...).build(); await hubConnection.start();
```

Схема приведена в приложении А.

2.3.2 Функция принятия сообщений

Клиент получает сообщения в реальном времени через SignalR. Все входящие сообщения отображаются в интерфейсе, форматируются с учетом времени и отправителя.

Пример обработки входящих сообщений:

```
hubConnection.on('ReceiveMessage', (arguments) {
    // arguments содержит данные сообщения
    setState(() {
        messages.add(MessageModel.fromJson(arguments[0]));
    });
});
```

Схема приведена в приложении А.

2.3.3 Функция отправки сообщений

Перед отправкой сообщения клиент проверяет, выбран ли чат/собеседник, и отправляет данные на сервер через SignalR или REST API. Сервер доставляет сообщение всем участникам чата.

Пример отправки сообщения:

```
await hubConnection.invoke('SendMessage', args: [chatId, messageText]);
```

Схема приведена в приложении А.

3 РАЗРАБОТКА ПРОГРАММНОГО СРЕДСТВА

3.1 Функции для работы с сетью и API

Одним из основных компонентов программного обеспечения является функционал работы с сетью, REST API и real-time соединением через SignalR. Ниже приведены основные функции, реализующие эту функциональность.

3.1.1 Инициализация и подключение к серверу

В Flutter-клиенте для обмена сообщениями и вложениями используется HTTP API и SignalR (WebSocket) для real-time событий.

Пример инициализации SignalR:

```
final hubConnection = HubConnectionBuilder()
  .withUrl('${AppConfig.signalRBaseUrl}${AppConfig.chatHubUrl}')
  .build(); await
hubConnection.start();
```

3.1.2 Авторизация пользователя

Авторизация реализована через REST API. После успешного входа пользователь получает JWT-токен, который используется для всех последующих запросов. Пример запроса авторизации:

```
final response = await dio.post(
  '${AppConfig.baseUrl}/auth/login', data:
  {'username': username, 'password': password},
);
final token = response.data['token']; dio.options.headers['Authorization']
= 'Bearer $token';
```

3.1.3 Загрузка и скачивание вложений

Для загрузки файлов используется presigned URL, получаемый с сервера. Клиент отправляет файл напрямую в MinIO, а для скачивания — получает временную ссылку и скачивает файл во временную папку устройства.

Пример скачивания и открытия вложения:

```
final response = await http.get(Uri.parse(url));
if (response.statusCode == 200) { final bytes
= response.bodyBytes; final dir = await
```



```

getTemporaryDirectory(); final filePath =
'${dir.path}/${att.fileName}'; final file =
File(filePath); await
file.writeAsBytes(bytes); await
OpenFile.open(filePath);
}

```

3.1.4 Отправка и получение сообщений

Отправка сообщений реализована через SignalR. Клиент вызывает серверный метод, сервер рассылает сообщение всем участникам чата.

Пример отправки сообщения:

await hubConnection.invoke('SendMessage', args: [chatId, messageText]); Обработка входящих сообщений:

```

hubConnection.on('ReceiveMessage', (arguments) { //
Добавить сообщение в локальный список и обновить UI
});

```

3.2 Функции работы интерфейса

3.2.1 Добавление сообщения в чат

```

void addMessageToChat(String message) {
  setState(() {
    messages.add(message);
  });
}

```

3.2.2 Обновление интерфейса

В Flutter обновление интерфейса происходит через setState или Bloc.

3.3 Функции для обработки данных

3.3.1 Преобразование данных

Для сериализации и десериализации данных используются модели с json_serializable/freeze.

Пример: factory MessageModel.fromJson(Map<String, dynamic> json) => _MessageModelFromJson(json);

3.3.2 Работа с файлами

Для работы с файлами используются пакеты `file_picker`, `path_provider`, `open_file`.

4 ТЕСТИРОВАНИЕ ПРОГРАММНОГО СРЕДСТВА

В процессе тестирования была выявлена проблема невозможности управления MinIO через локальную API. В админ-панель было невозможно зайти.

Для решения этой проблемы была выбрана более старая, но стабильная версия MinIO. Теперь доступ к bucket открыт серверу и клиентам, которые авторизовались.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Интерфейс программного средства

Качественно разработанный интерфейс способствует повышению удобства и интуитивности использования программного продукта. В этом программном средстве основное окно приложения выполнено в минималистичном стиле как у Telegram, что упрощает навигацию и работу пользователя с ним.

5.1.1 Экран отправки сообщений

Экран отправки сообщений — это основное рабочее пространство программы. Оно включает стандартные элементы: поле ввода сообщений, кнопка для отправки файла и поле с сообщениями, которые помогают пользователю общаться с другими пользователями:

Поле с сообщениями: содержит все посланные и принятые сообщения вместе с их временем прибытия и отправки, а также именем отправителя и реакциями на сообщения;

Поле ввода сообщений: содержит кнопку для отправки, кнопку для прикрепления файла и текстовое поле для ввода сообщения;

Основная часть окна — это поле с полосой прокрутки, позволяющей просматривать все сообщения. Внизу главного окна расположено поле ввода сообщения и кнопка для отправки.

Интерфейс экрана отправки сообщений изображен на рисунке 5.1.

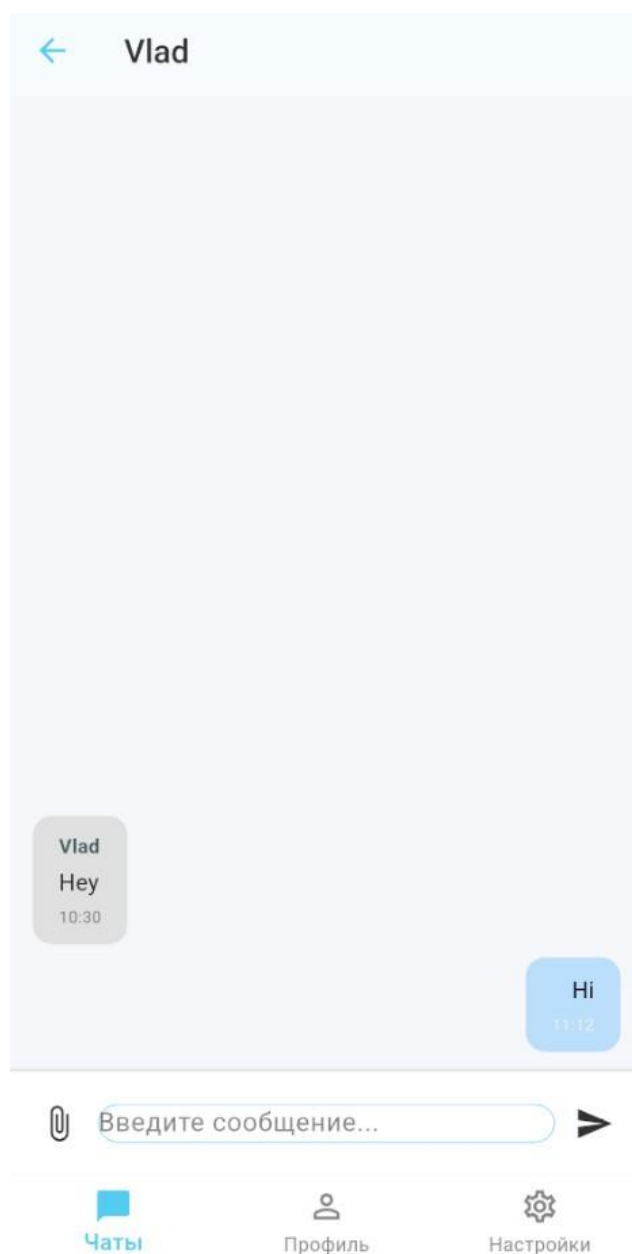


Рисунок 5.1 – Интерфейс экрана отправки сообщений

5.2 Управление программным средством

Перед работой с приложением требуется включить сервер, развернуть все БД через docker-compose, затем запустить клиент, подключится по IP к серверу, ввести логин и пароль либо зарегистрироваться. Перед отправкой сообщения требуется выбрать пользователя-получателя, далее ввести желаемое сообщение и нажать на кнопку для отправления. После этого сообщение придёт пользователю и отобразится у вас и у него со временем отправки в поле сообщений.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было разработано современное кроссплатформенное чат-приложение с использованием Flutter (Dart) для клиента и ASP.NET Core для серверной части. Приложение предоставляет пользователям основные функции обмена сообщениями, работы с вложениями и общения в реальном времени с другими пользователями. Решение оптимизировано для работы на мобильных устройствах (Android, iOS), а также поддерживает запуск на Web и Desktop, что обеспечивает широкую доступность и удобство использования.

Создание чат-приложения на основе современных технологий позволило реализовать все поставленные цели, а также углубить знания в области клиент-серверных архитектур, работы с REST API, WebSocket (SignalR), облачным хранилищем файлов (MinIO) и безопасной аутентификацией пользователей (JWT). Такой подход подчеркнул важность использования современных фреймворков и сервисов для обеспечения масштабируемости, безопасности и расширяемости приложения.

Реализованное чат-приложение включает следующие возможности:

1. Регистрация и авторизация пользователей через API;
2. Создание и удаление чатов, выбор собеседника;
3. Отправка и приём текстовых сообщений в реальном времени (SignalR);
4. Работа с вложениями: загрузка и скачивание файлов через MinIO с использованием presigned URL;
5. Реализация статусов "печатает", реакции на сообщения, уведомления о прочтении;
6. Восстановление истории сообщений из базы данных;
7. Логирование действий и ошибок на сервере;
8. Поддержка одновременной работы с несколькими пользователями и чатами.

В результате разработки было достигнуто понимание принципов построения современных кроссплатформенных приложений, организации взаимодействия между клиентом и сервером, а также интеграции с облачными сервисами хранения данных. Разработанное чат-приложение представляет собой надежное и производительное решение, подходящее для использования в корпоративной среде, образовательных учреждениях и других сферах, где требуется защищённый и удобный обмен сообщениями. Проект может служить основой для дальнейшего расширения и добавления новых функций, таких как каналы, интеграция с внешними сервисами, расширенные уведомления и др.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- [1] Документация Flutter: <https://docs.flutter.dev/>
- [2] Документация ASP.NET Core: <https://learn.microsoft.com/ru-ru/aspnet/core>
- [3] Документация SignalR: <https://learn.microsoft.com/ru-ru/aspnet/core/signalr>
- [4] Документация MinIO: <https://min.io/docs/minio/linux/index.html>
- [5] Мартин Р. Чистый код: создание, анализ и рефакторинг.
Библиотека программиста. - СПб.: Питер. 2018. – 464 с.
- [6] Орлов, С. А. Технологии разработки программного обеспечения: учеб.
Пособие. – СПб, 2003. – 321 с.

ПРИЛОЖЕНИЕ А
(Обязательное)

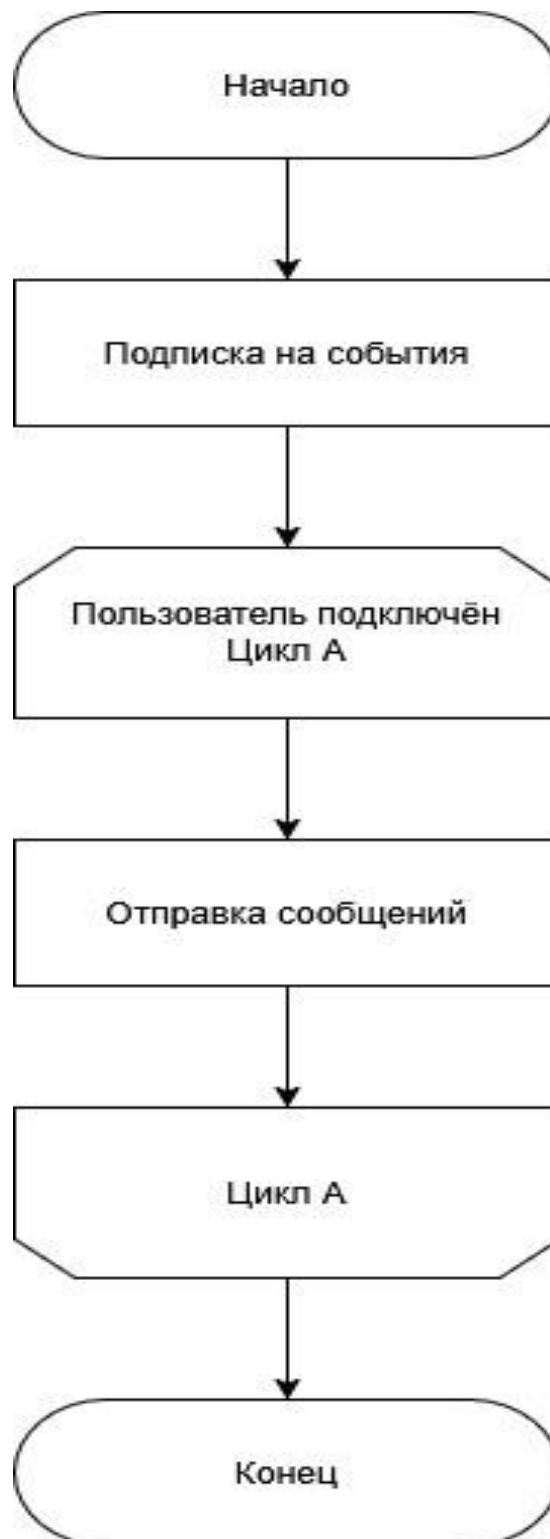


Схема инициализации



Схема принятия сообщений

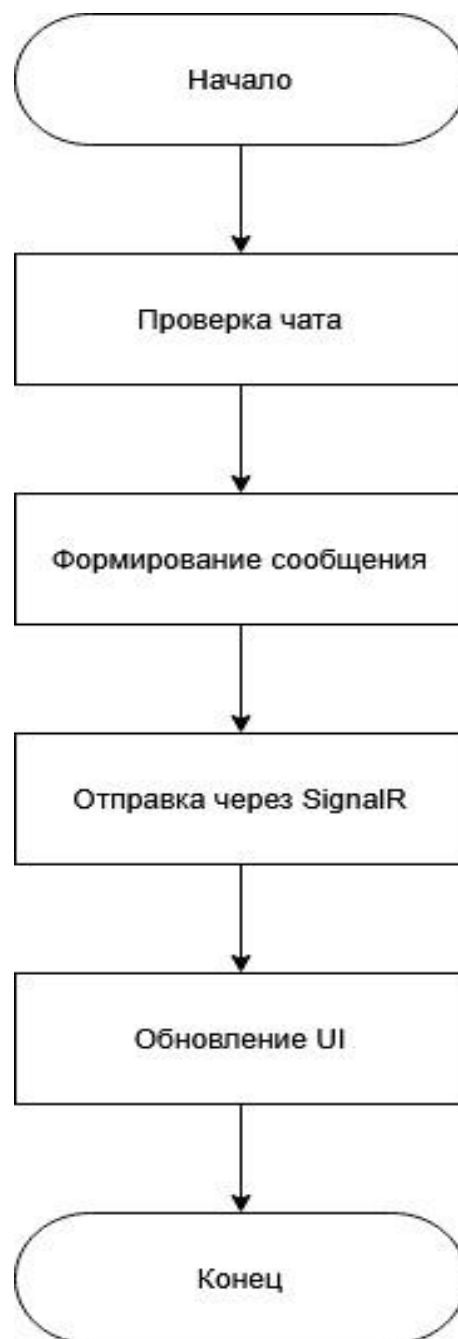


Схема отправки сообщений

ПРИЛОЖЕНИЕ Б

(Обязательное)

Текст программы Код

AttachmentService.cs

```
using AutoMapper; using
Microsoft.Extensions.Logging; using
SigmailServer.Application.DTOS; using
SigmailServer.Application.Services.Interfaces; using
Microsoft.AspNetCore.StaticFiles; using
Microsoft.Extensions.Options; using
SigmailServer.Domain.Enums; using
SigmailServer.Domain.Interfaces; using
SigmailServer.Domain.Models;

namespace SigmailServer.Application.Services;

public class AttachmentService : IAttachmentService
{
    private readonly IFileStorageRepository _fileStorageRepository;
    private readonly IMapper _mapper;      private readonly
ILogger<AttachmentService> _logger;      private readonly
IUnitOfWork _unitOfWork;
    private readonly AttachmentSettings _attachmentSettings; // Добавлено

    public AttachmentService(
        IFileStorageRepository fileStorageRepository,
        IMapper mapper,
        ILogger<AttachmentService> logger,
        IUnitOfWork unitOfWork,
        IOptions<AttachmentSettings> attachmentSettingsOptions) // Добавлено
    {
        _fileStorageRepository = fileStorageRepository;
        _mapper = mapper;
        _logger = logger;
        _unitOfWork = unitOfWork;
        _attachmentSettings = attachmentSettingsOptions.Value; // Добавлено
    }

    // Вариант 1: Клиент загружает файл на сервер, сервер пересылает в S3
    // Этот вариант менее предпочтителен для больших файлов из-за нагрузки на сервер.
    // Presigned URLs (Вариант 2) обычно лучше.
    public async Task<AttachmentDto> UploadAttachmentAsync(Guid uploaderId, Stream fileStream,
string fileName, string contentType, AttachmentType attachmentType) {
        _logger.LogInformation("User {UploaderId} uploading attachment {FileName} (server-side
proxy)", uploaderId, fileName);

        if (fileStream == null || fileStream.Length == 0)
        {
            throw new ArgumentException("File stream cannot be null or empty.",
nameof(fileStream));
        }
        if (string.IsNullOrEmpty(fileName))
        {
            throw new ArgumentException("File name cannot be empty.", nameof(fileName));
        }

        // Генерация уникального ключа файла          var
fileExtension = Path.GetExtension(fileName);
        var uniqueFileKey = $"{uploaderId}/{Guid.NewGuid()}fileExtension}";
```

```

        if (string.IsNullOrEmpty(contentType))
        {
            new FileExtensionContentTypeProvider().TryGetContentType(fileName, out
contentType);
            contentType ??= "application/octet-stream";
        }

        string fileUrlOrKey = await _fileStorageRepository.UploadFileAsync(fileStream,
uniqueFileKey, contentType); // Репозиторий должен вернуть ключ или URL

        // Здесь может быть создание записи об Attachment в БД (если они не вложены в Message)
        // SigmailServer.Domain.Models.Attachment attachmentEntity = new
SigmailServer.Domain.Models.Attachment
        // {
        //     FileKey = fileUrlOrKey, // или uniqueFileKey, если UploadFileAsync возвращает
только его
        //     FileName = fileName,
        //     ContentType = contentType,
        //     Type = attachmentType,
        //     Size = fileStream.Length, // Важно: если stream уже прочитан, Length может быть
неверным. Получать до чтения.
        //     // UploaderId = uploaderId, CreatedAt = DateTime.UtcNow, etc.
        // };
        // await _unitOfWork.Attachments.AddAsync(attachmentEntity); // Если есть такой
репозиторий
        // await _unitOfWork.CommitAsync();

        _logger.LogInformation("Attachment {FileName} uploaded by {UploaderId}, key:
{FileKey}", fileName, uploaderId, fileUrlOrKey);

        // Возвращаем DTO. PresignedUrl для скачивания можно сгенерировать сразу или по запросу.
var downloadUrl = await _fileStorageRepository.GeneratePresignedUrlAsync(fileUrlOrKey,
(TimeSpan.FromHours(1)));

        return new AttachmentDto
        {
            FileKey = fileUrlOrKey,
            FileName = fileName,
            ContentType = contentType,
            Type = attachmentType,
            Size = fileStream.Length, // Перепроверить получение размера
            PresignedUrl = downloadUrl
            // ThumbnailKey и другие поля по необходимости
        };
    }

    public async Task<UploadAttachmentResponseDto> GetPresignedUploadUrlAsync(Guid uploaderId,
string fileName, string contentType, long fileSize, AttachmentType attachmentType) {
        _logger.LogInformation("User {UploaderId} requesting presigned URL for upload:
{FileName}, Size: {FileSize}, ContentType: {ContentType}", uploaderId, fileName, fileSize,
contentType);

        if (string.IsNullOrEmpty(fileName)) throw new ArgumentException("File name is
required.");
        if (fileSize <= 0) throw new ArgumentException("File size must be greater than
zero.");

        // Валидация размера файла
        if (fileSize > _attachmentSettings.MaxFileSizeBytes)
        {
            throw new ArgumentException($"File size exceeds the maximum allowed limit of
{_attachmentSettings.MaxFileSizeMB} MB.");
        }
    }

```

```

        var fileExtension = Path.GetExtension(fileName).ToLowerInvariant();
        if (string.IsNullOrEmpty(contentType))
        {
            new FileExtensionContentTypeProvider().TryGetContentType(fileName, out
            contentType);
            contentType ??= "application/octet-stream"; // Fallback
        }

        // Валидация типа файла (по расширению и ContentType)
        bool extensionAllowed = _attachmentSettings.AllowedFileExtensions.Any(ext =>
        ext.Equals(fileExtension, StringComparison.OrdinalIgnoreCase));
        bool contentTypeAllowed = _attachmentSettings.AllowedContentTypes.Any(ct =>
        ct.Equals(contentType, StringComparison.OrdinalIgnoreCase));

        if ((_attachmentSettings.AllowedFileExtensions.Any() && !extensionAllowed) ||
        (_attachmentSettings.AllowedContentTypes.Any() && !contentTypeAllowed))
        {
            _logger.LogWarning("File type not allowed. FileName: {FileName}, Extension:
            {FileExtension}, ContentType: {ContentType}", fileName, fileExtension, contentType);
            throw new ArgumentException("File type not allowed.");
        }

        var uniqueFileKey = $"uploads/{uploaderId}/{Guid.NewGuid()}{fileExtension}";
        var presignedUrl = await
        _fileStorageRepository.GeneratePresignedUrlAsync(uniqueFileKey, TimeSpan.FromMinutes(15));
        // ВАЖНО: Не изменяйте host/port/scheme у presignedUrl!
        // Presigned URL должен быть сгенерирован сразу с правильным endpoint (реальный IP
        сервера MinIO/S3),
        // который задаётся в конфиге S3StorageSettings (например, http://192.168.1.100:9000)
        // Если вы работаете на реальном устройстве, endpoint должен быть доступен с телефона.

        _logger.LogInformation("Presigned URL generated for {FileKey}", uniqueFileKey);
        return new UploadAttachmentResponseDto
        {
            FileKey = uniqueFileKey,
            FileName = fileName,
            ContentType = contentType,
            Size = fileSize,
            Type = attachmentType,
            PresignedUploadUrl = presignedUrl
        };
    }

    public async Task<string> GetPresignedDownloadUrlAsync(string fileKey, Guid currentUserId
    /* Добавлено для проверки прав */)
    {
        _logger.LogInformation("User {CurrentUserId} requesting presigned download URL for
        file key {FileKey}", currentUserId, fileKey);
        if (string.IsNullOrEmpty(fileKey)) throw new ArgumentException("File key is
        required.");

        // Опциональная проверка существования файла (S3 вернет ошибку 404, если URL
        недействителен или файл удален)
        // bool fileExists = await _fileStorageRepository.FileExistsAsync(fileKey); //
        Потребуется метод в IFileStorageRepository
        // if (!fileExists) throw new KeyNotFoundException("File not found.");

        // Проверка прав доступа:
        // Связываем fileKey с сообщением, чтобы проверить, имеет ли currentUserId доступ к
        этому чату/сообщению.
        // Это упрощенная проверка. В реальности может потребоваться более сложная логика.
        var messageContainingFile = await
        _unitOfWork.Messages.GetByAttachmentKeyAsync(fileKey);
    }

```

```

        if (messageContainingFile
            != null)
        {
            var isMember = await
            _unitOfWork.Chats.IsUserMemberAsync(messageContainingFile.ChatId, currentUserId);
            if (!isMember)
            {
                _logger.LogWarning("User {CurrentUserId} does not have permission to access
                file {FileKey} via chat {ChatId}", currentUserId, fileKey, messageContainingFile.ChatId);
                throw new UnauthorizedAccessException("You do not have permission to access this file.");
            }
            else
            {
                // Если файл не привязан к сообщению (например, аватар пользователя), нужна другая
                // логика проверки.
                // TODO: Реализовать проверку для аватаров, если они хранятся через этот сервис.
                // Пример:
                // var user = await _unitOfWork.Users.GetByIdAsync(currentUserId); // Предположим,
                // мы хотим проверить, является ли это аватаром текущего пользователя
                // if (user == null || user.ProfileImageUrl != fileKey) {
                //     // Или это может быть аватар другого пользователя, если профили публичны
                //     var targetUser = await _unitOfWork.Users.FirstOrDefaultAsync(u => u.ProfileImageUrl ==
                //     fileKey); // нужен метод в репозитории
                //     if (targetUser == null) { // Файл не является известным аватаром
                //         _logger.LogWarning("Could not determine access rights for file key
                //         {FileKey} for user {CurrentUserId} (not a message attachment or known avatar). Denying
                //         access.", fileKey, currentUserId);
                //         throw new UnauthorizedAccessException("Cannot determine access rights
                //         for this file.");
                //     }
                //     // Если это аватар другого пользователя, и профили публичны, то доступ
                //     // можно разрешить.
                //     // Если нет, то проверяем, currentUserId == targetUserId
                // }
                _logger.LogWarning("Could not determine access rights for file key {FileKey} for
                user {CurrentUserId} (file not found in any message or other context). Denying access.",
                fileKey, currentUserId);
                throw new UnauthorizedAccessException("Cannot determine access rights for this
                file or file not found.");
            }
        }

        return await _fileStorageRepository.GeneratePresignedUrlAsync(fileKey,
            TimeSpan.FromHours(1));
    }

    public async Task DeleteAttachmentAsync(Guid currentUserId, string fileKey)
    {
        _logger.LogInformation("User {CurrentUserId} attempting to delete attachment with key
        {FileKey}", currentUserId, fileKey);
        if (string.IsNullOrWhiteSpace(fileKey)) throw new ArgumentException("File key is
        required.");

        // Проверка прав на удаление:
        // Пользователь может удалить файл, если он отправитель сообщения, к которому
        // прикреплен файл.
        // Админы/владельцы чата могут удалять сообщения (и, следовательно, файлы) в своих
        // чатах.
        // Ищем сообщение, содержащее этот fileKey
        // Нужен метод: Task<Message?> GetMessageByFileKeyAsync(string fileKey,
        // CancellationToken cancellationToken = default);
        Message? message = await _unitOfWork.Messages.GetByAttachmentKeyAsync(fileKey);

        if (message == null)
        {
            // TODO: Реализовать логику удаления для аватаров или других файлов, не связанных
            // с сообщениями, если это необходимо.
            // Пример:

```

```

        // var user = await _unitOfWork.Users.GetByIdAsync(currentUserId);
        // if (user != null && user.ProfileImageUrl == fileKey) {
        //     // Пользователь удаляет свой собственный аватар
        //     await _fileStorageRepository.DeleteFileAsync(fileKey);
        //     user.ProfileImageUrl = null; // или на URL по умолчанию
        //     await _unitOfWork.Users.UpdateAsync(user);
        //     await _unitOfWork.CommitAsync();
        //     _logger.LogInformation("User {CurrentUserId} deleted their avatar
{FileKey}.", currentUserId, fileKey);
        //     return;
        // }

        _logger.LogWarning("File {FileKey} not found in any message or no permission for
user {CurrentUserId} to delete (file not a message attachment or other context).", fileKey,
currentUserId);
        throw new UnauthorizedAccessException("You do not have permission to delete this
file or file not found.");
    }

    bool canDelete = false;
    if (message.SenderId == currentUserId)
    {
        canDelete = true;
    }
    else
    {
        var chatMember = await _unitOfWork.Chats.GetChatMemberAsync(message.ChatId,
currentUserId);
        if (chatMember != null && (chatMember.Role == ChatMemberRole.Admin ||
chatMember.Role == ChatMemberRole.Owner))
        {
            // TODO: Добавить более гранулярную проверку прав (может ли админ удалять
*любые* файлы в чате, или только файлы из удаляемых им сообщений)
            canDelete = true;
        }
    }

    if (!canDelete)
    {
        _logger.LogWarning("User {CurrentUserId} does not have permission to delete file
{FileKey} from message {MessageId}", currentUserId, fileKey, message.Id);
        throw new UnauthorizedAccessException("You do not have permission to delete this
file.");
    }

    await _fileStorageRepository.DeleteFileAsync(fileKey);
    _logger.LogInformation("File {FileKey} deleted from S3 by user {CurrentUserId}.",
fileKey, currentUserId);

    // Удаляем метаданные файла из сообщения в MongoDB
    message.Attachments.RemoveAll(a => a.FileKey == fileKey);
    await _unitOfWork.Messages.UpdateAsync(message); // UpdateAsync в MessageRepository
должен обновить весь документ
    _logger.LogInformation("Attachment metadata for {FileKey} removed from message
{MessageId}", fileKey, message.Id);
}
}

```

Код AuthService.cs

```

using System.Security.Authentication; using
AutoMapper;

```

```

using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging; using
SigmailServer.Application.DTOS;
using SigmailServer.Application.Services.Interfaces;
using SigmailServer.Domain.Interfaces; using
SigmailServer.Domain.Models;

namespace SigmailServer.Application.Services;

public class AuthService : IAuthService
{
    private readonly IUnitOfWork _unitOfWork;    private
readonly IMapper _mapper;    private readonly
IPasswordHasher _passwordHasher;    private readonly
IJwtTokenGenerator _jwtTokenGenerator;    private readonly
ILogger<AuthService> _logger;
    private readonly IConfiguration _configuration; // Для срока действия Refresh Token

    public AuthService(
IUnitOfWork unitOfWork,
IMapper mapper,
IPasswordHasher passwordHasher,
IJwtTokenGenerator jwtTokenGenerator,
ILogger<AuthService> logger,
IConfiguration configuration) // Добавлен IConfiguration
    {
        _unitOfWork = unitOfWork;
        _mapper = mapper;
        _passwordHasher = passwordHasher;
        _jwtTokenGenerator = jwtTokenGenerator;
        _logger = logger;
        _configuration = configuration;
    }

    public async Task<AuthResultDto> RegisterAsync(CreateUserDto dto)
    {
        _logger.LogInformation("Attempting to register user {Username}", dto.Username);

        if (string.IsNullOrEmpty(dto.Username) || string.IsNullOrEmpty(dto.Email) ||
string.IsNullOrEmpty(dto.Password))
        {
            throw new ArgumentException("Username, email, and password are required.");
        }

        if (await _unitOfWork.Users.GetByUsernameAsync(dto.Username) != null)
        {
            _logger.LogWarning("Registration failed: Username {Username} already exists.",
dto.Username);
            throw new ArgumentException("Username already exists.");
        }
        if (await _unitOfWork.Users.GetByEmailAsync(dto.Email) != null)
        {
            _logger.LogWarning("Registration failed: Email {Email} already exists.",
dto.Email);
            throw new ArgumentException("Email already exists.");
        }

        var passwordHash = _passwordHasher.HashPassword(dto.Password);
        var user = new User(dto.Username, dto.Email, passwordHash); // Конструктор User должен
это поддерживать

        await _unitOfWork.Users.AddAsync(user);
        // CommitAsync здесь может быть преждевременным, если генерация токена или установка
RT требуют новой транзакции
        // Однако, для простоты, оставим пока так. В идеале, RT сохраняется после успешной
генерации.
    }
}

```



```

        var (accessToken, accessTokenExpiration, refreshToken) =
            _jwtTokenGenerator.GenerateTokens(user);
        refresh_token_validity_in_days =
            _configuration.GetValue<int>("Jwt:RefreshTokenValidityInDays", 7);
        user.SetRefreshToken(refreshToken,
            DateTime.UtcNow.AddDays(refreshTokenValidityInDays));

        // Обновляем пользователя с RT. Если AddAsync уже вызвал SaveChanges (в MongoDB), то
        Update может быть не нужен,
        // но для EF Core (PostgreSQL) - нужен. UnitOfWork должен это разрулить.
        // await _unitOfWork.Users.UpdateAsync(user);
        await _unitOfWork.CommitAsync(); // Финальный коммит
        _logger.LogInformation("User {Username} registered
            successfully with ID {UserId}", dto.Username, user.Id);
        return new AuthResultDto
        {
            User = _mapper.Map<UserDto>(user),
            AccessToken = accessToken,
            AccessTokenExpiration = accessTokenExpiration,
            RefreshToken = refreshToken
        };
    }

    public async Task<AuthResultDto> LoginAsync(LoginDto dto)
    {
        _logger.LogInformation("Attempting to login user {UsernameOrEmail}",
            dto.UsernameOrEmail);
        if (string.IsNullOrEmpty(dto.UsernameOrEmail) ||
            string.IsNullOrEmpty(dto.Password))
        {
            throw new ArgumentException("Username/Email and password are required.");
        }

        var user = await _unitOfWork.Users.GetByUsernameAsync(dto.UsernameOrEmail)
            ?? await _unitOfWork.Users.GetByEmailAsync(dto.UsernameOrEmail);

        if (user == null || !_passwordHasher.VerifyPassword(user.PasswordHash, dto.Password))
        {
            _logger.LogWarning("Login failed for {UsernameOrEmail}: Invalid credentials",
                dto.UsernameOrEmail);
            throw new InvalidCredentialException("Invalid username
                or password.");
        }

        var (accessToken, accessTokenExpiration, refreshToken) =
            _jwtTokenGenerator.GenerateTokens(user);
        refresh_token_validity_in_days =
            _configuration.GetValue<int>("Jwt:RefreshTokenValidityInDays", 7);
        user.SetRefreshToken(refreshToken,
            DateTime.UtcNow.AddDays(refreshTokenValidityInDays));
        user.GoOnline(); // Обновляем статус

        await _unitOfWork.Users.UpdateAsync(user);
        await _unitOfWork.CommitAsync();

        _logger.LogInformation("User {UsernameOrEmail} (ID: {UserId}) logged in successfully",
            dto.UsernameOrEmail, user.Id);
        return new AuthResultDto
        {
            User = _mapper.Map<UserDto>(user),
            AccessToken = accessToken,
            AccessTokenExpiration = accessTokenExpiration,
            RefreshToken = refreshToken
        };
    }
}

```

```

    public async Task<AuthResultDto?> RefreshTokenAsync(RefreshTokenDto dto)
    {
        _logger.LogInformation("Attempting to refresh token");
        if (string.IsNullOrEmpty(dto.Token))
        {
            throw new ArgumentException("Refresh token is required.");
        }

        var user = await _unitOfWork.Users.GetByRefreshTokenAsync(dto.Token);

        if (user == null || user.RefreshTokenExpiryTime <= DateTime.UtcNow)
        {
            _logger.LogWarning("Refresh token failed: Invalid or expired token.");
            if (user != null) // Если токен был, но истек, чистим его
            {
                user.ClearRefreshToken();
                await _unitOfWork.Users.UpdateAsync(user);
            }
            await _unitOfWork.CommitAsync();
            return null; // Или выбросить исключение
        }

        var (newAccessToken, newAccessTokenExpiration, newRefreshToken) =
            _jwtTokenGenerator.GenerateTokens(user);
        var refreshTokenValidityInDays =
            _configuration.GetValue<int>("Jwt:RefreshTokenValidityInDays", 7);
        user.SetRefreshToken(newRefreshToken, DateTime.UtcNow.AddDays(refreshTokenValidityInDays));

        await _unitOfWork.Users.UpdateAsync(user);
        await _unitOfWork.CommitAsync();

        _logger.LogInformation("Token refreshed successfully for user {UserId}", user.Id);
        return new AuthResultDto
        {
            User = _mapper.Map<UserDto>(user),
            AccessToken = newAccessToken,
            AccessTokenExpiration = newAccessTokenExpiration,
            RefreshToken = newRefreshToken
        };
    }

    public async Task LogoutAsync(Guid userId, string refreshTokenToInvalidate)
    {
        _logger.LogInformation("User {UserId} logging out", userId);
        var user = await _unitOfWork.Users.GetByIdAsync(userId);
        if (user != null)
        {
            // Инвалидируем только если предоставленный RT совпадает с текущим у пользователя
            if (user.RefreshToken == refreshTokenToInvalidate)
            {
                user.ClearRefreshToken();
            }
            user.GoOffline(); // В любом случае ставим оффлайн
            await _unitOfWork.Users.UpdateAsync(user);
            await _unitOfWork.CommitAsync();
            _logger.LogInformation("User {UserId} processed logout. Refresh token cleared if matched.", userId);
        }
        else
        {
            _logger.LogWarning("Logout for user {UserId}: User not found.", userId);
        }
    }
}

```

Код ChatService.cs

```
using AutoMapper;
using Microsoft.Extensions.Logging; using
SigmilServer.Application.DTOs;
using SigmilServer.Application.Services.Interfaces;
using SigmilServer.Domain.Enums; using
SigmilServer.Domain.Interfaces; using
SigmilServer.Domain.Models;

namespace SigmilServer.Application.Services;

public class ChatService : IChatService
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IMessageRepository _messageRepository;
    private readonly IMapper _mapper;    private readonly
    IRealTimeNotifier _realTimeNotifier;    private readonly
    ILogger<ChatService> _logger;

    public ChatService(
    IUnitOfWork unitOfWork,
        IMessageRepository messageRepository,
        IMapper mapper,
        IRealTimeNotifier realTimeNotifier,
        ILogger<ChatService> logger)
    {
        _unitOfWork = unitOfWork;
        _messageRepository = messageRepository;
        _mapper = mapper;
        _realTimeNotifier = realTimeNotifier;
        _logger = logger;
    }

    public async Task<ChatDto> CreateChatAsync(Guid creatorId, CreateChatDto dto)
    {
        _logger.LogInformation("User {CreatorId} creating chat of type {ChatType} with name
        '{ChatName}'", creatorId, dto.Type, dto.Name);

        if (dto.Type == ChatType.Private)
        {
            if (dto.MemberIds == null || dto.MemberIds.Count != 1)
            {
                throw new ArgumentException("Private chat must be created with exactly one
                other member.");
            }
            Guid otherMemberId = dto.MemberIds.First();
            if (creatorId == otherMemberId)
            {
                throw new ArgumentException("Cannot create a private chat with yourself.");
            }

            // Проверяем, существует ли уже такой приватный чат
            var existingPrivateChat = await
            _unitOfWork.Chats.GetPrivateChatByMembersAsync(creatorId, otherMemberId);
            if (existingPrivateChat != null)
            {
                _logger.LogInformation("Private chat between {User1} and {User2} already
                exists (ID: {ChatId}). Returning existing.", creatorId, otherMemberId,
                existingPrivateChat.Id);
                // Нужно загрузить всю информацию для ChatDto
                return await MapChatToDtoAsync(existingPrivateChat, creatorId);
            }
        }
    }
}
```

```

        else // Group or Channel
        {
            if (string.IsNullOrEmpty(dto.Name))
            {
                throw new ArgumentException("Chat name is required for group or channel
chats.");
            }

            var chat = new Chat
            {
                Name = dto.Type == ChatType.Private ? null : dto.Name,
                Type = dto.Type,
                Description = dto.Description,
                CreatorId = creatorId,
                CreatedAt = DateTime.UtcNow,
                UpdatedAt = DateTime.UtcNow
                // AvatarUrl будет установлен позже, если нужно
            };

            await _unitOfWork.Chats.AddAsync(chat);
            // Важно: CommitAsync нужен здесь, чтобы chat.Id был сгенерирован перед добавлением
            // участников, если Id генерируется БД.
            // Если Guid генерируется в конструкторе Chat, то можно коммитить позже. Предположим,
            // Id генерируется при Add.
            await _unitOfWork.CommitAsync(); // Commit для получения Chat.Id

            await _unitOfWork.Chats.AddMemberAsync(chat.Id, creatorId, ChatMemberRole.Owner);

            var memberIdsToNotify = new HashSet<Guid> { creatorId };

            if (dto.MemberIds != null)
            {
                foreach (var memberId in dto.MemberIds.Distinct()) // Уникальные ID
                {
                    if (memberId == creatorId && dto.Type != ChatType.Private) continue; //
                    // Создатель уже добавлен как владелец

                    var userExists = await _unitOfWork.Users.GetByIdAsync(memberId) != null;
                    if (!userExists)
                    {
                        _logger.LogWarning("User with ID {MemberId} not found. Skipping for chat
{ChatId}.", memberId, chat.Id);
                        continue;
                    }
                    await _unitOfWork.Chats.AddMemberAsync(chat.Id, memberId,
ChatMemberRole.Member);
                    memberIdsToNotify.Add(memberId);
                }
            }

            await _unitOfWork.CommitAsync(); // Коммит для добавления участников

            var chatDto = await MapChatToDtoAsync(chat, creatorId);

            await _realTimeNotifier.NotifyChatCreatedAsync(memberIdsToNotify.ToList(), chatDto);
            _logger.LogInformation("Chat {ChatId} (Name: '{ChatName}') created successfully.", chat.Id,
chat.Name);
            return chatDto;
        }

        public async Task<ChatDto> GetChatByIdAsync(Guid chatId, Guid currentUserId)
        {
            _logger.LogInformation("User {CurrentUserId} requesting chat {ChatId}", currentUserId,
chatId);

```

```

        var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);
if (chat == null)
{
    _logger.LogWarning("Chat {ChatId} not found.", chatId);
return null;
}

        var isMember = await _unitOfWork.Chats.IsUserMemberAsync(chatId, currentUserId);
if (!isMember)
{
    // Для каналов можно разрешить просмотр не-участникам, если это публичный канал.
    // Пока что требуем членства для всех типов.
    _logger.LogWarning("User {CurrentUserId} is not a member of chat {ChatId}. Access
denied.", currentUserId, chatId);
    throw new UnauthorizedAccessException("User is
not a member of this chat.");
}

        return await MapChatToDtoAsync(chat, currentUserId);
}

        public async Task<IEnumerable<ChatDto>> GetUserChatsAsync(Guid userId, int page = 1, int
pageSize = 20)
{
    _logger.LogInformation("Requesting chats for user {UserId}, Page: {Page}, PageSize:
{PageSize}", userId, page, pageSize);
    // Пагинация должна быть реализована в репозитории GetUserChatsAsync
    var
chats = await _unitOfWork.Chats.GetUserChatsAsync(userId); // TODO: Добавить пагинацию в
IChatRepository

    var chatDtos = new List<ChatDto>();
    foreach (var chat in chats) // Убрана временная пагинация .Skip().Take()
    {
        chatDtos.Add(await MapChatToDtoAsync(chat, userId));
    }
    return chatDtos;
}

        private async Task<ChatDto> MapChatToDtoAsync(Chat chat, Guid currentUserId)
{
    var chatDto = _mapper.Map<ChatDto>(chat);
    if
(!string.IsNullOrEmpty(chat.LastMessageId))
    {
        var lastMessage = await _messageRepository.GetByIdAsync(chat.LastMessageId);
if (lastMessage != null)
        {
            var lastMessageDto = _mapper.Map<MessageDto>(lastMessage);
if (lastMessage.SenderId != Guid.Empty)
            {
                var sender = await _unitOfWork.Users.GetByIdAsync(lastMessage.SenderId);
lastMessageDto.Sender = _mapper.Map<UserSimpleDto>(sender);
            }
            chatDto.LastMessage = lastMessageDto;
        }
    }

    var members = await _unitOfWork.Chats.GetChatMembersAsync(chat.Id);
chatDto.Members = _mapper.Map<List<UserSimpleDto>>(members); // Мappings списка
пользователей
    chatDto.MemberCount = members.Count(); // Количество участников

    // Используем новый метод репозитория для подсчета непрочитанных сообщений
chatDto.UnreadCount = (int)await
_messageRepository.GetUnreadMessageCountForUserInChatAsync(chat.Id, currentUserId);
    return
chatDto;
}

```

```

        public async Task<ChatDto> UpdateChatDetailsAsync(Guid chatId, Guid currentUserId,
UpdateChatDto dto)
        {
            _logger.LogInformation("User {CurrentUserId} updating details for chat {ChatId} with
Name: \"{DtoName}\"", currentUserId, chatId, dto.Name);
            var chat = await
_unitOfWork.Chats.GetByIdAsync(chatId);
            if (chat == null)
            {
                _logger.LogWarning("Chat {ChatId} not found for update.", chatId);
                throw new KeyNotFoundException("Chat not found.");
            }

            if (chat.Type == ChatType.Private)
            {
                throw new InvalidOperationException("Cannot update details of a private chat
directly.");
            }

            var member = await _unitOfWork.Chats.GetChatMemberAsync(chatId, currentUserId);
            if (member == null || (member.Role != ChatMemberRole.Admin && member.Role !=
ChatMemberRole.Owner))
            {
                _logger.LogWarning("User {CurrentUserId} does not have permission to update chat
{ChatId} details.", currentUserId, chatId);
                throw new UnauthorizedAccessException("User does not have permission to update
chat details.");
            }

            bool updated = false;
            if (!string.IsNullOrEmpty(dto.Name) && chat.Name != dto.Name)
            {
                chat.Name =
dto.Name;
                updated =
true;
            }
            if (dto.Description != null && chat.Description != dto.Description)
            {
                chat.Description = dto.Description;
                updated = true;
            }
            var updatedChatDto = await MapChatToDtoAsync(chat, currentUserId); // Готовим DTO до
коммита

            if (updated)
            {
                chat.UpdatedAt = DateTime.UtcNow;
                await _unitOfWork.Chats.UpdateAsync(chat);
                await _unitOfWork.CommitAsync();
                _logger.LogInformation("Chat {ChatId} details updated.", chatId);

                var memberIds = (await _unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u =>
u.Id).ToList();
                if (memberIds.Any())
                {
                    await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(memberIds,
updatedChatDto);
                }
            }

            return updatedChatDto; // Возвращаем DTO с обновленными данными (или старыми, если не
было изменений)
        }

        public async Task AddMemberToChatAsync(Guid chatId, Guid currentUserId, Guid userIdToAdd)
        {
            _logger.LogInformation("User {CurrentUserId} adding user {UserIdToAdd} to chat
{ChatId}", currentUserId, userIdToAdd, chatId);
            var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);
            if
(chat == null) throw new KeyNotFoundException("Chat not found.");

```

```

        if (chat.Type == ChatType.Private) throw new InvalidOperationException("Cannot add
members to a private chat.");

        var currentUserMember = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
currentUserMember);
        if (currentUserMember == null || (currentUserMember.Role != ChatMemberRole.Admin &&
currentUserMember.Role != ChatMemberRole.Owner))
        {
            throw new UnauthorizedAccessException("User does not have permission to add
members.");
        }

        var userToAdd = await _unitOfWork.Users.GetByIdAsync(userIdToAdd);
        if (userToAdd == null) throw new KeyNotFoundException("User to add not found.");

        var isAlreadyMember = await _unitOfWork.Chats.IsUserMemberAsync(chatId, userIdToAdd);
        if (isAlreadyMember) throw new InvalidOperationException("User is already a member of this
chat.");

        await _unitOfWork.Chats.AddMemberAsync(chatId, userIdToAdd, ChatMemberRole.Member);
        await _unitOfWork.CommitAsync();
        _logger.LogInformation("User {UserIdToAdd} added to chat {ChatId} by user
{CurrentUserId}", userIdToAdd, chatId, currentUserMember);

        var memberIds = (await _unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u =>
u.Id).ToList();
        var addedUserDto = _mapper.Map<UserSimpleDto>(userToAdd);
        var chatDto = await MapChatToDtoAsync(chat, currentUserMember); // Для передачи
обновленного чата

        await _realTimeNotifier.NotifyMemberAddedToChatAsync(memberIds, chatDto, addedUserDto,
currentUserMember);
        public async Task RemoveMemberFromChatAsync(Guid chatId, Guid currentUserMember, Guid
userIdToRemove)
        {
            _logger.LogInformation("User {CurrentUserId} removing user {UserIdToRemove} from chat
{ChatId}", currentUserMember, userIdToRemove, chatId);
            var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);
            if
(chat == null) throw new KeyNotFoundException("Chat not found.");
            if (chat.Type == ChatType.Private) throw new InvalidOperationException("Cannot remove
members from a private chat.");

            var memberToRemove = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
userIdToRemove);
            if (memberToRemove == null) throw new InvalidOperationException("User to remove is not
a member of this chat.");

            // Логика прав на удаление:
            // Владелец может удалять кого угодно (кроме себя, для этого есть LeaveChat).
            // Админ может удалять обычных участников.
            // Пользователь может удалять сам себя (LeaveChat).
            // Нельзя удалить владельца, если ты не владелец.

            var currentUserMember = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
currentUserMember);
            if (currentUserMember == null) throw new UnauthorizedAccessException("Current user is
not a member or not found.");

            bool canRemove = false;
            if (currentUserMember == userIdToRemove) // Сам себя пользователь удаляет через LeaveChat
            {
                throw new InvalidOperationException("Use LeaveChat to remove yourself from the
chat.");
            }
            if (currentUserMember.Role == ChatMemberRole.Owner)

```

```

        {
            if (memberToRemove.Role == ChatMemberRole.Owner && currentUserId !=
                userIdToRemove) // Владелец пытается удалить другого владельца (не должно быть возможно)
            {
                // Если это единственный владелец, он не может быть удален таким образом.
                if (chat.Members.Count(m => m.Role == ChatMemberRole.Owner) <= 1 &&
                    memberToRemove.UserId == chat.CreatorId)
                {
                    throw new InvalidOperationException("Cannot remove the original owner and
only owner of the chat.");
                }
            }

            canRemove = true;
        }
        else if (currentUserMember.Role == ChatMemberRole.Admin && memberToRemove.Role ==
            ChatMemberRole.Member)
        {
            canRemove = true;
        }
        if (!canRemove)
        {
            throw new UnauthorizedAccessException("User does not have permission to remove
this member.");
        }

        await _unitOfWork.Chats.RemoveMemberAsync(chatId, userIdToRemove);
        await _unitOfWork.CommitAsync();
        _logger.LogInformation("User {UserIdToRemove} removed from chat {ChatId} by user
{CurrentUserId}", userIdToRemove, chatId, currentUserId);

        var chatDto = await MapChatToDtoAsync(chat, currentUserId); // Для передачи
обновленного чата
        var memberIdsToNotify = chat.Members.Where(m => m.UserId !=
            userIdToRemove).Select(m
=> m.UserId).ToList(); // Все, кроме удаленного
        memberIdsToNotify.Add(userIdToRemove); // Уведомляем и удаленного пользователя

        await _realTimeNotifier.NotifyMemberRemovedFromChatAsync(memberIdsToNotify, chatDto,
            userIdToRemove, currentUserId);
    }

    public async Task PromoteMemberToAdminAsync(Guid chatId, Guid currentUserId, Guid
        userIdToPromote)
    {
        _logger.LogInformation("User {CurrentUserId} promoting user {UserIdToPromote} to Admin
in chat {ChatId}", currentUserId, userIdToPromote, chatId);
        var chat = await
        _unitOfWork.Chats.GetByIdAsync(chatId);
        if (chat == null) throw new
        KeyNotFoundException("Chat not found.");
        if (chat.Type == ChatType.Private) throw new InvalidOperationException("Roles are not
        applicable in private chats.");

        var currentUserMember = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
            currentUserId);
        if (currentUserMember == null || currentUserMember.Role !=
            ChatMemberRole.Owner)
        {
            throw new UnauthorizedAccessException("Only the owner can promote members to
            admin.");
        }

        var memberToPromote = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
            userIdToPromote);
        if (memberToPromote == null) throw new KeyNotFoundException("User to promote not found
        in this chat.");
        if (memberToPromote.Role == ChatMemberRole.Admin) throw new
        InvalidOperationException("User is already an admin.");
    }

```



```

        if (memberToPromote.Role == ChatMemberRole.Owner) throw new
InvalidOperationException("Cannot change role of the owner.");

        await _unitOfWork.Chats.UpdateMemberRoleAsync(chatId, userIdToPromote,
ChatMemberRole.Admin);
        await _unitOfWork.CommitAsync();
        _logger.LogInformation("User {UserIdToPromote} promoted to Admin in chat {ChatId} by
user {CurrentUserId}", userIdToPromote, chatId, currentUserId);

        var memberIds = (await _unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u =>
u.Id).ToList();
        var chatDto = await MapChatToDtoAsync(chat, currentUserId);
        await _realTimeNotifier.NotifyChatMemberRoleChangedAsync(memberIds, chatDto,
userIdToPromote, ChatMemberRole.Admin, currentUserId);    }

    public async Task DemoteAdminToMemberAsync(Guid chatId, Guid currentUserId, Guid
adminUserIdToDemote)    {
        _logger.LogInformation("User {CurrentUserId} demoting admin {AdminUserIdToDemote} to
Member in chat {ChatId}", currentUserId, adminUserIdToDemote, chatId);
        var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);        if (chat ==
null) throw new KeyNotFoundException("Chat not found.");
        if (chat.Type == ChatType.Private) throw new InvalidOperationException("Roles are not
applicable in private chats.");

        var currentUserMember = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
currentUserId);        if (currentUserMember == null || currentUserMember.Role !=
ChatMemberRole.Owner)
        {
            throw new UnauthorizedAccessException("Only the owner can demote admins.");
        }

        var adminToDemote = await _unitOfWork.Chats.GetChatMemberAsync(chatId,
adminUserIdToDemote);
        if (adminToDemote == null) throw new KeyNotFoundException("Admin to demote not found
in this chat.");        if (adminToDemote.Role != ChatMemberRole.Admin) throw new
InvalidOperationException("User is not an admin.");

        await _unitOfWork.Chats.UpdateMemberRoleAsync(chatId, adminUserIdToDemote,
ChatMemberRole.Member);
        await _unitOfWork.CommitAsync();
        _logger.LogInformation("Admin {AdminUserIdToDemote} demoted to Member in chat {ChatId}
by user {CurrentUserId}", adminUserIdToDemote, chatId, currentUserId);

        var memberIds = (await _unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u =>
u.Id).ToList();
        var chatDto = await MapChatToDtoAsync(chat, currentUserId);
        await _realTimeNotifier.NotifyChatMemberRoleChangedAsync(memberIds, chatDto,
adminUserIdToDemote, ChatMemberRole.Member, currentUserId);    }

    public async Task LeaveChatAsync(Guid chatId, Guid currentUserId)
    {
        _logger.LogInformation("User {CurrentUserId} attempting to leave chat {ChatId}",
currentUserId, chatId);
        var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);        if
(chat == null) throw new KeyNotFoundException("Chat not found.");

        var memberToLeave = await _unitOfWork.Chats.GetChatMemberAsync(chatId, currentUserId);
        if (memberToLeave == null) throw new InvalidOperationException("User is not a member of this
chat.");

        bool isLastMember = chat.Members.Count() == 1;
        bool isOwnerLeaving = memberToLeave.Role == ChatMemberRole.Owner;

```

```

        if (isOwnerLeaving && isLastMember) // Владелец выходит из чата, и он последний
участник
        {
            _logger.LogInformation("Owner {CurrentUserId} is the last member leaving chat
{ChatId}. Deleting chat.", currentUserId, chatId);
            // Удаляем сообщения чата
            await _unitOfWork.Messages.DeleteMessagesByChatIdAsync(chatId); // Физическое
удаление сообщений
            // Удаляем сам чат (включая всех участников через каскадное удаление в БД)
            await _unitOfWork.Chats.DeleteAsync(chatId);
            await _unitOfWork.CommitAsync(); // Уведомлять некого, чат удален
            _logger.LogInformation("Chat {ChatId} and its messages deleted as the last member
(owner) left.", chatId);
            return; // Выход из метода, так как чат удален
        }
        else if (isOwnerLeaving && chat.Members.Count(m => m.Role == ChatMemberRole.Owner) ==
1)
        {
            // Владелец уходит, но есть другие участники. Нужно назначить нового владельца.
            var admins = chat.Members.Where(m => m.Role == ChatMemberRole.Admin && m.UserId !=
currentUserId).OrderBy(m => m.JoinedAt).ToList();
            var members = chat.Members.Where(m => m.Role == ChatMemberRole.Member && m.UserId
!= currentUserId).OrderBy(m => m.JoinedAt).ToList();

            Guid? newOwnerId = null;
            newOwnerId = admins.First().UserId;
            newOwnerId = members.First().UserId;

            if (admins.Any())
            else if (members.Any())

            if (newOwnerId.HasValue)
            {
                await _unitOfWork.Chats.UpdateMemberRoleAsync(chatId, newOwnerId.Value,
ChatMemberRole.Owner);
                _logger.LogInformation("User {NewOwnerId} promoted to Owner in chat {ChatId}
as previous owner left.", newOwnerId.Value, chatId);
            }
            else // Это не должно произойти, если isLastMember = false
            {
                _logger.LogWarning("Owner {CurrentUserId} leaving chat {ChatId}, but no
suitable member found to promote to new owner. This might lead to an orphaned chat.",
currentUserId, chatId);
                // Потенциально, можно просто удалить чат, если владелец уходит и некого
назначить.
                // Или запретить выход владельцу, если он единственный и есть другие
участники, не являющиеся админами.
                // Для простоты, пока разрешим выход, но это проблемная ситуация.
            }
        }
        // Удаляем участника из чата
        await _unitOfWork.Chats.RemoveMemberAsync(chatId, currentUserId);
        chat.UpdatedAt = DateTime.UtcNow; // Обновляем время изменения чата
        await _unitOfWork.Chats.UpdateAsync(chat); // Для обновления UpdatedAt
        await _unitOfWork.CommitAsync();
        _logger.LogInformation("User {CurrentUserId} left chat {ChatId}.", currentUserId,
chatId);

        var remainingMemberIds = (await
_unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u => u.Id).ToList();
        var chatDto = await MapChatToDtoAsync(chat, currentUserId); // currentUserId здесь уже
не участник, но нужен для контекста DTO

        if (remainingMemberIds.Any())
        {
            await _realTimeNotifier.NotifyMemberLeftChatAsync(remainingMemberIds, chatDto,
currentUserId);

```

```

    } } public async Task DeleteChatAsync(Guid chatId, Guid
currentUserId) // Добавлен метод {
    _logger.LogInformation("User {CurrentUserId} attempting to delete chat {ChatId}",
currentUserId, chatId);
    var chat = await _unitOfWork.Chats.GetByIdAsync(chatId); if
(chat == null) throw new KeyNotFoundException("Chat not found.");

    var member = await _unitOfWork.Chats.GetChatMemberAsync(chatId, currentUserId);
    // Только владелец может удалить чат
    if (member == null || member.Role != ChatMemberRole.Owner)
    {
        throw new UnauthorizedAccessException("Only the owner can delete the chat.");
    }

    // 1. Уведомить всех участников (кроме текущего пользователя, если он еще там), что
чат будет удален
    var memberIdsToNotify = chat.Members.Select(m => m.UserId).Where(id => id !=
currentUserId).ToList();
    if (memberIdsToNotify.Any())
    {
        // Нужен метод в IRealTimeNotifier, например, NotifyChatDeletedAsync(List<Guid>
userIds, Guid chatId)
        // await _realTimeNotifier.NotifyChatDeletedAsync(memberIdsToNotify, chatId);
    }

    // 2. Удалить все сообщения чата из MongoDB
    await _unitOfWork.Messages.DeleteMessagesByChatIdAsync(chatId);
    _logger.LogInformation("All messages for chat {ChatId} deleted from MongoDB.",
chatId);

    // 3. Удалить сам чат из PostgreSQL (ChatMembers удалятся каскадно)
    await _unitOfWork.Chats.DeleteAsync(chatId); await
_unitOfWork.CommitAsync();
    _logger.LogInformation("Chat {ChatId} and its members deleted from PostgreSQL by owner
{CurrentUserId}.", chatId, currentUserId);

    // TODO: Notify (если есть кого) что чат удален. Хотя тут никого. (Уже сделано выше
перед удалением)
}

public async Task<IEnumerable<UserSimpleDto>> GetChatMembersAsync(Guid chatId) // Убран
currentUserId, чтобы соответствовать интерфейсу
{
    _logger.LogInformation("Requesting members for chat {ChatId}", chatId);
    // Проверка прав на получение списка участников должна быть выполнена до вызова этого
метода,
    // если этот метод не предполагает предоставление общедоступной информации о составе
чата.
    var members = await _unitOfWork.Chats.GetChatMembersAsync(chatId);
    return _mapper.Map<IEnumerable<UserSimpleDto>>(members); }

public async Task<bool> IsUserMemberAsync(Guid chatId, Guid userId)
{
    _logger.LogDebug("Checking if user {UserId} is a member of chat {ChatId}", userId,
chatId); return await _unitOfWork.Chats.IsUserMemberAsync(chatId, userId); }
    public async Task<ChatDto> UpdateChatAvatarAsync(Guid chatId, Guid currentUserId, Stream
avatarStream, string contentType)
    {
        _logger.LogInformation("User {CurrentUserId} updating avatar for chat {ChatId}",
currentUserId, chatId);
        var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);
        if (chat == null)
        {

```

```

        _logger.LogWarning("Chat {ChatId} not found for avatar update.", chatId);
        throw new KeyNotFoundException("Chat not found.");
    }

    if (chat.Type == ChatType.Private)
    {
        throw new InvalidOperationException("Cannot update avatar of a private chat directly.");
    }

    var member = await _unitOfWork.Chats.GetChatMemberAsync(chatId, currentUserId);
    if (member == null || (member.Role != ChatMemberRole.Admin && member.Role != ChatMemberRole.Owner))
    {
        _logger.LogWarning("User {CurrentUserId} does not have permission to update chat {ChatId} avatar.", currentUserId, chatId);
        throw new UnauthorizedAccessException("User does not have permission to update chat avatar.");
    }

    // Генерируем имя файла для аватара чата
    string fileName =
        $"chat_avatar_{chatId}{Path.GetExtension(contentType.Split('/').Last())}"; // Простое имя файла

    // Используем IAttachmentService для загрузки. AttachmentType может быть специальным для аватаров.
    // Либо напрямую IFileStorageRepository, если не нужна запись в Attachment DTO/Entity
    // Для простоты, предположим, что AttachmentService.UploadAttachmentAsync подходит
    // или у вас есть специальный метод для загрузки общих файлов, не привязанных к сообщениям.
    // Здесь мы напрямую используем FileStorageRepository для загрузки и обновления URL в чате.

    var fileKey = await _unitOfWork.Files.UploadFileAsync(avatarStream, fileName,
        contentType);
    if (string.IsNullOrEmpty(fileKey))
    {
        _logger.LogError("Failed to upload avatar for chat {ChatId}. File key is null or empty.", chatId);
        throw new Exception("Avatar upload failed.");
    }

    // Удаляем старый аватар, если он был
    if (!string.IsNullOrEmpty(chat.AvatarUrl))
    {
        try
        {
            await _unitOfWork.Files.DeleteFileAsync(chat.AvatarUrl); // Предполагаем, что AvatarUrl хранит fileKey
        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to delete old avatar {OldFileKey} for chat {ChatId}.", chat.AvatarUrl, chatId);
            // Не прерываем операцию, если старый аватар не удалился
        }
    }

    chat.AvatarUrl = fileKey; // Сохраняем новый fileKey как AvatarUrl
    chat.UpdatedAt = DateTime.UtcNow;

    await _unitOfWork.Chats.UpdateAsync(chat);
    await _unitOfWork.CommitAsync();

    _logger.LogInformation("Avatar for chat {ChatId} updated. New avatar key: {FileKey}",
        chatId, fileKey);
    var chatDto = await MapChatToDtoAsync(chat,
        currentUserId);

    // Уведомление участников чата

```

```

        var memberIds = (await _unitOfWork.Chats.GetChatMembersAsync(chatId)).Select(u =>
u.Id).ToList();
        if (memberIds.Any())
        {
            // Предполагается, что NotifyChatDetailsUpdatedAsync уведомит об изменении DTO,
включая новый AvatarUrl
            await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(memberIds, chatDto);
        }

        return chatDto;
    }
}

```

Код ContactService.cs

```

using AutoMapper;
using Microsoft.Extensions.Logging; using
SigmilServer.Application.DTOS;
using SigmilServer.Application.Services.Interfaces;
using SigmilServer.Domain.Enums; using
SigmilServer.Domain.Interfaces; using
SigmilServer.Domain.Models;

namespace SigmilServer.Application.Services;

public class ContactService : IContactService
{
    private readonly IUnitOfWork _unitOfWork;
    private readonly IMapper _mapper;
    private readonly ILogger<ContactService> _logger;
    private readonly INotificationService _notificationService; // Для создания уведомлений

    public ContactService(
        IUnitOfWork unitOfWork,
        IMapper mapper,
        ILogger<ContactService> logger,
        INotificationService notificationService)
    {
        _unitOfWork = unitOfWork;
        _mapper = mapper;
        _logger = logger;
        _notificationService = notificationService;
    }

    public async Task SendContactRequestAsync(Guid requesterId, ContactRequestDto dto)
    {
        _logger.LogInformation("User {RequesterId} sending contact request to user
{TargetUserId}", requesterId, dto.TargetUserId);
        if (requesterId == dto.TargetUserId)
        {
            throw new ArgumentException("Cannot send contact request to yourself.");
        }

        var targetUser = await _unitOfWork.Users.GetByIdAsync(dto.TargetUserId);
        if (targetUser == null)
        {
            throw new KeyNotFoundException("Target user not found.");
        }

        // Проверяем существующие запросы в любом направлении
        var existingRequest =
await _unitOfWork.Contacts.GetContactRequestAsync(requesterId, dto.TargetUserId) ??
await

```

```

_unitOfWork.Contacts.GetContactRequestAsync(dto.TargetUserId, requesterId);

    if (existingRequest != null)
    {
        switch(existingRequest.Status)
        {
            case
ContactRequestStatus.Accepted:
                throw new InvalidOperationException("Users are already contacts.");
case ContactRequestStatus.Pending:
                if (existingRequest.UserId == requesterId) // Запрос уже отправлен этим
пользователем
                    throw new InvalidOperationException("Contact request already sent and
is pending.");
                else // Запрос ожидает ответа от текущего requesterId
                    throw new InvalidOperationException($"There is a pending request from
user {targetUser.Username}. Please respond to it.");
            case
ContactRequestStatus.Declined:
                // Можно разрешить повторный запрос после Decline, или требовать удаления
старой записи
                // Пока просто создадим новую, если Decline был от другого пользователя.
// Если Decline был от targetUser на запрос requesterId, то можно дать отправить снова.
                // Если Decline был от requesterId на запрос targetUser, то targetUser
должен отправить снова.
                // Для простоты: если есть Declined, предполагаем, что можно отправить
новый, старый будет проигнорирован или перезаписан при принятии.
                // Или, лучше, если запись Declined существует, то ее нужно сначала
"снять" или удалить.
                // Пока что, если Declined, разрешим отправить новый.
                _logger.LogInformation("Previous request between {User1} and {User2} was
declined. Allowing new request.", requesterId, dto.TargetUserId);
                break;
            case
ContactRequestStatus.Blocked:
                throw new InvalidOperationException("Cannot send request, one of the
users is blocked.");
        }
    }

    var contactRequest = new Contact
    {
        UserId = requesterId, // Тот, кто отправил запрос
        ContactUserId = dto.TargetUserId, // Кому запрос
        Status = ContactRequestStatus.Pending,
        RequestedAt = DateTime.UtcNow
    };
    await _unitOfWork.Contacts.AddAsync(contactRequest);
    await _unitOfWork.CommitAsync();
    _logger.LogInformation("Contact request from {RequesterId} to {TargetUserId} sent
successfully. ID: {ContactRequestId}", requesterId, dto.TargetUserId, contactRequest.Id);

    var requester = await _unitOfWork.Users.GetByIdAsync(requesterId);
    await _notificationService.CreateAndSendNotificationAsync(
        dto.TargetUserId,
        NotificationType.ContactRequestReceived, // TODO: Нужен
NotificationType.ContactRequestReceived
        $"User {requester?.Username ?? "Unknown"} sent you a contact request.",
        "New Contact Request",
        relatedEntityId: requesterId.ToString(), // ID отправителя запроса
        relatedEntityType: "UserContactRequest"
    );
}

    public async Task RespondToContactRequestAsync(Guid responderId,
RespondToContactRequestDto dto)

```

```

    {
        _logger.LogInformation("User {ResponderId} responding to contact request {RequestId}
with {Response}", responderId, dto.RequestId, dto.Response);
        var contactRequest = await _unitOfWork.Contacts.GetByIdAsync(dto.RequestId);
        if (contactRequest == null)
        {
            throw new KeyNotFoundException("Contact request not found.");
        }
        // Убеждаемся, что responderId - это тот, кому был адресован запрос (ContactUserId)
        if (contactRequest.ContactUserId != responderId)
        {
            throw new UnauthorizedAccessException("User cannot respond to this request.");
        }
        if (contactRequest.Status != ContactRequestStatus.Pending)
        {
            throw new InvalidOperationException($"Request is not pending.
Current status: {contactRequest.Status}.");
        }
        if (dto.Response != ContactRequestStatus.Accepted && dto.Response !=
ContactRequestStatus.Declined)
        {
            throw new ArgumentException("Invalid response status. Must be Accepted or
Declined.");
        }

        contactRequest.Status = dto.Response;
        contactRequest.RespondedAt = DateTime.UtcNow;
        _unitOfWork.Contacts.UpdateAsync(contactRequest);
        _unitOfWork.CommitAsync();
        _logger.LogInformation("Contact request {RequestId} (from {OriginalRequesterId} to
{ResponderId}) responded with {Response}",
            dto.RequestId, contactRequest.UserId, responderId, dto.Response);

        var responder = await _unitOfWork.Users.GetByIdAsync(responderId);
        string notificationMessage = dto.Response == ContactRequestStatus.Accepted
? $"User {responder?.Username ?? "Unknown"} accepted your contact request."
: $"User {responder?.Username ?? "Unknown"} declined your contact request.";
        NotificationType notificationType = dto.Response == ContactRequestStatus.Accepted
? NotificationType.ContactRequestAccepted
: NotificationType.ContactRequestDeclined;

        await _notificationService.CreateAndSendNotificationAsync(
contactRequest.UserId, // Уведомляем инициатора запроса
            notificationType, // TODO: Нужен NotificationType.ContactRequestResponded
            notificationMessage, "Contact Request Update",
            relatedEntityId: responderId.ToString(), // ID того, кто ответил
            relatedEntityType: "UserContactResponse"
        );
    }

    public async Task RemoveContactAsync(Guid currentUserId, Guid contactUserIdToRemove)
    {
        _logger.LogInformation("User {CurrentUserId} attempting to remove contact with user
{ContactUserIdToRemove}", currentUserId, contactUserIdToRemove);

        var contactEntry = await _unitOfWork.Contacts.GetContactRequestAsync(currentUserId,
contactUserIdToRemove) ??
            await
_unitOfWork.Contacts.GetContactRequestAsync(contactUserIdToRemove, currentUserId);

        if (contactEntry == null || contactEntry.Status != ContactRequestStatus.Accepted)
        {

```

```

        throw new KeyNotFoundException("No accepted contact relationship found to
remove.");
    }

    // Вместо удаления можно менять статус, например на "RemovedByInitiator" или
    "RemovedByTarget"
    // Или действительно удалять запись. Пока удаляем.
    await _unitOfWork.Contacts.DeleteAsync(contactEntry.Id);
    await _unitOfWork.CommitAsync();
    _logger.LogInformation("Contact between {CurrentUserId} and {ContactUserIdToRemove}
(Entry ID: {ContactEntryId}) removed.",
        currentUserId, contactUserIdToRemove, contactEntry.Id);

    // TODO: Optionally notify contactUserIdToRemove that they were removed.
    // Это может быть спорным моментом с точки зрения UX (тихое удаление или уведомление).
    var currentUser = await _unitOfWork.Users.GetByIdAsync(currentUserId);
    await _notificationService.CreateAndSendNotificationAsync(
        contactUserIdToRemove,
        NotificationType.ContactRemoved,
        $"User {currentUser?.Username ?? "Unknown"} has removed you from their contacts.",
        "Contact Removed",
        relatedEntityId: currentUserId.ToString(),
        relatedEntityType: "User"
    );
}

public async Task<IEnumerable<ContactDto>> GetUserContactsAsync(Guid userId,
ContactRequestStatus? statusFilter = ContactRequestStatus.Accepted) {
    _logger.LogInformation("Requesting contacts for user {UserId} with status filter
{StatusFilter}", userId, statusFilter);
    var contacts = await _unitOfWork.Contacts.GetUserContactsAsync(userId, statusFilter);

    var contactDtos = new List<ContactDto>();
    foreach(var contact in contacts)
    {
        // Определяем, кто в этой записи является "другим" пользователем
        var otherUserId = (contact.UserId == userId) ? contact.ContactUserId : contact.UserId;
        var contactUserDomain = await _unitOfWork.Users.GetByIdAsync(otherUserId);
        if
        (contactUserDomain != null)
        {
            contactDtos.Add(new
            ContactDto {
                ContactEntryId
                = contact.Id,
                User = _mapper.Map<UserDto>(contactUserDomain), // Это DTO
                пользователяконтакта
                Status = contact.Status,
                RequestedAt = contact.RequestedAt,
                RespondedAt = contact.RespondedAt
            });
        }
        else
        {
            _logger.LogWarning("Could not find user details for contact entry
{ContactEntryId}, other user ID {OtherUserId}", contact.Id, otherUserId);
        }
    }
    return contactDtos;
}

public async Task<IEnumerable<ContactDto>> GetPendingContactRequestsAsync(Guid userId)
{
    _logger.LogInformation("Requesting pending contact requests for user {UserId}
(requests sent TO this user)", userId);
    // Этот метод должен возвращать запросы, где `userId` является `ContactUserId`
    (получателем запроса)
    var requests = await

```



```

_unitOfWork.Contacts.GetPendingContactRequestsForUserAsync(userId);

    var requestDtos = new List<ContactDto>();
    foreach(var req in requests)
    {
        // В данном случае, req.UserId - это тот, кто отправил запрос пользователю
        `userId`
        var requesterUser = await _unitOfWork.Users.GetByIdAsync(req.UserId);
        if(requesterUser != null)
        {
            requestDtos.Add(new ContactDto {
                ContactEntryId = req.Id,
                User = _mapper.Map<UserDto>(requesterUser), // DTO пользователя,
                Status = req.Status, // Должен быть Pending
                RequestedAt = req.RequestedAt
            });
        }
        else
        {
            _logger.LogWarning("Could not find user details for requester ID
{RequesterId} in contact request {ContactRequestId}", req.UserId, req.Id);
        }
    }
    return requestDtos;
}
}

```

Код MessageReactionService.cs

```

using AutoMapper;
using SigmailServer.Application.DTOS;
using SigmailServer.Application.Services.Interfaces;
using SigmailServer.Domain.Interfaces; using
SigmailServer.Domain.Models;
using Microsoft.Extensions.Logging; // Для логирования using
SigmailServer.Domain.Exceptions; // <--- ВОЗВРАЩАЕМ USING

namespace SigmailServer.Application.Services;

public class MessageReactionService : IMessageReactionService
{
    private readonly IMessageRepository _messageRepository;    private readonly IUnitOfWork
_unitOfWork; // Для MongoDB это может быть "пустышка", но если есть транзакционность...
    private readonly IMapper _mapper;
    private readonly IRealTimeNotifier _realTimeNotifier; // Для уведомлений через SignalR
    private readonly ILogger<MessageReactionService> _logger;
    private readonly IChatRepository _chatRepository; // Для получения chatId

    public MessageReactionService(
        IMessageRepository messageRepository,
        IUnitOfWork unitOfWork,
        IMapper mapper,
        IRealTimeNotifier realTimeNotifier,
        ILogger<MessageReactionService> logger,
        IChatRepository chatRepository)
    {
        _messageRepository = messageRepository;
        _unitOfWork = unitOfWork;
        _mapper = mapper;
        _realTimeNotifier = realTimeNotifier;
        _logger = logger;
        _chatRepository = chatRepository;
    }
}

```

```

        public async Task<IEnumerable<ReactionDto>> AddReactionAsync(string messageId, Guid
        userId, string emoji)
        {
            _logger.LogInformation("Attempting to add reaction (emoji: {Emoji}, userId: {UserId})
            to message {MessageId}", emoji, userId, messageId);

            var message = await _messageRepository.GetByIdAsync(messageId);
            if (message == null)
            {
                _logger.LogWarning("AddReactionAsync: Message with ID {MessageId} not found.",
                messageId);
                throw new NotFoundException($"Message with ID {messageId} not
                found.");
            }

            var existingReaction = message.Reactions.FirstOrDefault(r => r.Emoji == emoji);

            if (existingReaction != null)
            {
                if (!existingReaction.UserIds.Contains(userId))
                {
                    existingReaction.UserIds.Add(userId);
                    existingReaction.LastReactedAt = DateTime.UtcNow;
                    _logger.LogInformation("User {UserId} added to existing reaction {Emoji} for
                    message {MessageId}", userId, emoji, messageId);
                }
            }
            else
            {
                _logger.LogInformation("User {UserId} already reacted with {Emoji} to message
                {MessageId}. No changes made.", userId, emoji, messageId);
                // Можно просто вернуть текущие реакции без изменений или считать это успехом
                // return _mapper.Map<IEnumerable<ReactionDto>>(message.Reactions); // Если ничего не
                // изменилось
            }
            else
            {
                message.Reactions.Add(new Reaction(emoji, userId));
                _logger.LogInformation("New reaction {Emoji} by user {UserId} added to message
                {MessageId}", emoji, userId, messageId);
            }

            await _messageRepository.UpdateAsync(message);
            // await _unitOfWork.SaveChangesAsync(); // Если бы это был EF Core для Message
            _logger.LogInformation("Message {MessageId} updated in repository with new reaction info.",
            messageId);

            var updatedReactionsDto = _mapper.Map<IEnumerable<ReactionDto>>(message.Reactions);

            // Уведомление клиентов через SignalR
            // Нам нужен ChatId, чтобы отправить уведомление в нужную группу
            // Предположим, что MessageModel содержит ChatId. Если нет, нужно его получить.
            // В текущей Message.cs ChatId есть.
            await _realTimeNotifier.NotifyMessageReactionsUpdatedAsync(message.ChatId.ToString(),
            messageId, updatedReactionsDto);
            _logger.LogInformation("SignalR notification sent for reaction update on message
            {MessageId} in chat {ChatId}", messageId, message.ChatId);

            return updatedReactionsDto;
        }

        public async Task<IEnumerable<ReactionDto>> RemoveReactionAsync(string messageId, Guid
        userId, string emoji)
        {
            _logger.LogInformation("Attempting to remove reaction (emoji: {Emoji}, userId:
            {UserId}) from message {MessageId}", emoji, userId, messageId);

            var message = await _messageRepository.GetByIdAsync(messageId);
            if (message == null)
            {

```

```

        _logger.LogWarning("RemoveReactionAsync: Message with ID {MessageId} not found.",
messageId);
        throw new NotFoundException($"Message with ID {messageId} not found.");
    }

    var reactionToRemove = message.Reactions.FirstOrDefault(r => r.Emoji == emoji);

    if (reactionToRemove != null)
    {
        bool userRemoved = reactionToRemove.UserIds.Remove(userId);
        if (userRemoved)
        {
            _logger.LogInformation("User {UserId}'s reaction {Emoji} removed from message
{MessageId}", userId, emoji, messageId);
            if (reactionToRemove.UserIds.Count == 0)
            {
                message.Reactions.Remove(reactionToRemove);
                _logger.LogInformation("Emoji reaction {Emoji} entirely removed from
message {MessageId} as no users are left.", emoji, messageId);
            }
            else
            {
                // Обновить LastReactedAt, если это важно (кто последний убрал - не так
важно, как кто последний добавил)
                // reactionToRemove.LastReactedAt = DateTime.UtcNow;
            }

            await
_messageRepository.UpdateAsync(message);
            // await _unitOfWork.SaveChangesAsync();
            _logger.LogInformation("Message {MessageId} updated in repository after
reaction removal.", messageId);

            var updatedReactionsDto =
_mapper.Map<IEnumerable<ReactionDto>>(message.Reactions);
            await
_realTimeNotifier.NotifyMessageReactionsUpdatedAsync(message.ChatId.ToString(), messageId,
updatedReactionsDto);
            _logger.LogInformation("SignalR notification sent for reaction removal on
message {MessageId} in chat {ChatId}", messageId, message.ChatId);
            return
updatedReactionsDto;
        }
        else
        {
            _logger.LogInformation("User {UserId} had not reacted with {Emoji} to message
{MessageId}. No changes made.", userId, emoji, messageId);
        }
    }
    else
    {
        _logger.LogInformation("Reaction with emoji {Emoji} not found on message
{MessageId}. No changes made.", emoji, messageId);
    }
    // Если ничего не изменилось, возвращаем текущее состояние реакций
    return _mapper.Map<IEnumerable<ReactionDto>>(message.Reactions);
}

```

Код MessageService.cs

```

using AutoMapper;
using Microsoft.Extensions.Logging; using
SigmilServer.Application.DTOS;
using SigmilServer.Application.Services.Interfaces;
using SigmilServer.Domain.Enums; using

```

```

SigmailServer.Domain.Interfaces; using
SigmailServer.Domain.Models;

namespace SigmailServer.Application.Services;

public class MessageService : IMessageService
{
    private readonly IMessageRepository _messageRepository;
    private readonly IUnitOfWork _unitOfWork; // Для IChatRepository, IUserRepository
    private readonly IMapper _mapper;      private readonly IRealTimeNotifier
    _realTimeNotifier;      private readonly ILogger<MessageService> _logger;      private
    readonly INotificationService _notificationService;      private readonly IUserService
    _userService;      private readonly IAttachmentService _attachmentService;

    public MessageService(
        IMessageRepository messageRepository,
        IUnitOfWork unitOfWork,
        IMapper mapper,
        IRealTimeNotifier realTimeNotifier,
        ILogger<MessageService> logger,
        INotificationService notificationService,
        IUserService userService,
        IAttachmentService attachmentService) // <--- add this
    {
        _messageRepository = messageRepository;
        _unitOfWork = unitOfWork;
        _mapper = mapper;
        _realTimeNotifier = realTimeNotifier;
        _logger = logger;
        _notificationService = notificationService;
        _userService = userService;
        _attachmentService = attachmentService;
    }

    public async Task<MessageDto?> SendMessageAsync(Guid senderId, CreateMessageDto dto)
    {
        _logger.LogInformation("User {SenderId} sending message to chat {ChatId}. Text:
        '{TextSnippet}'. ClientMsgId: {ClientMessageId}",
            senderId, dto.ChatId, dto.Text?.Substring(0, Math.Min(dto.Text.Length, 20)),
            dto.ClientMessageId);

        var chat = await _unitOfWork.Chats.GetByIdAsync(dto.ChatId);
        if (chat == null) throw new KeyNotFoundException($"Chat with ID {dto.ChatId} not
        found.");

        var isMember = await _unitOfWork.Chats.IsUserMemberAsync(dto.ChatId, senderId);
        if (!isMember) throw new UnauthorizedAccessException($"Sender {senderId} is not a member
        of chat {dto.ChatId}.");

        if (string.IsNullOrEmpty(dto.Text) && (dto.Attachments == null ||
        !dto.Attachments.Any()))
        {
            throw new ArgumentException("Message must have text or attachments.");
        }

        var message = new Message
        {
            // Id будет сгенерирован MongoDB автоматически или в конструкторе Message
            ChatId = dto.ChatId,
            SenderId = senderId,
            Text = dto.Text,
            Timestamp = DateTime.UtcNow,

```

```

        Status = MessageStatus.Sent, // Изначально Sent. Статус Delivered/Read обновляется
        позже.
        ForwardedFromMessageId = dto.ForwardedFromMessageId,
        // ForwardedFromUserId будет установлен, если ForwardedFromMessageId не null и
        сообщение переслано от другого пользователя
    };

    if (!string.IsNullOrEmpty(dto.ForwardedFromMessageId))
    {
        var originalMessage = await
        _messageRepository.GetByIdAsync(dto.ForwardedFromMessageId);
        if (originalMessage != null)
        {
            message.ForwardedFromUserId = originalMessage.SenderId;
        }
        else
        {
            _logger.LogWarning("Original message {ForwardedFromMessageId} for forwarding
            not found.", dto.ForwardedFromMessageId);
            // Можно либо бросить ошибку, либо продолжить без ForwardedFromUserId
        }
    }

    if (dto.Attachments != null && dto.Attachments.Any())
    {
        foreach (var attDto in dto.Attachments)
        {
            // Предполагается, что FileKey уже существует (файл загружен через
            AttachmentService, и клиент передал ключ)
            // Также предполагается, что CreateAttachmentDto содержит все необходимые поля
            для SigmailServer.Domain.Models.Attachment
            var attachment = _mapper.Map<Attachment>(attDto);
            message.Attachments.Add(attachment);
        }
    }

    await _messageRepository.AddAsync(message);
    // После добавления сообщения в MongoDB, message.Id должен быть заполнен.

    // Обновляем LastMessageId в чате (хранится в PostgreSQL)
    await _unitOfWork.Chats.UpdateLastMessageAsync(dto.ChatId, message.Id);
    await _unitOfWork.CommitAsync(); // Сохраняем изменения в PostgreSQL
    (Chat.LastMessageId)

    var messageDto = await MapMessageToDtoAsync(message, senderId); // Используем
    вспомогательный метод, передаем senderId как currentUserId

    // Уведомляем всех участников чата о новом сообщении
    var chatMembers = await _unitOfWork.Chats.GetChatMembersAsync(dto.ChatId);
    var memberIdsToNotify = chatMembers.Select(m => m.Id).ToList();

    foreach (var memberId in memberIdsToNotify)
    {
        // Отправляем real-time уведомление о новом сообщении
        if (messageDto != null)
        {
            await _realTimeNotifier.NotifyMessageReceivedAsync(memberId, messageDto);
        }
    }

    // Создаем персистентное уведомление для тех, кто не отправитель
    if (memberId != senderId)
    {
        var senderUser = await _unitOfWork.Users.GetByIdAsync(senderId);
    }

```

```

        var textSnippet = messageDto?.Text?.Substring(0,
Math.Min(messageDto.Text.Length, 50)) ?? "Attachment";
        await _notificationService.CreateAndSendNotificationAsync(
memberId,
            NotificationType.NewMessage,
            $"{{(senderUser?.Username ?? "Someone")}}: {{textSnippet}}",
            $"New message in {{(chat?.Name ?? "chat")}}",
message.Id,
            "Message"
        );
    }
}

// Теперь уведомим об обновлении самого чата (например, для обновления LastMessage в
списке чатов)
var updatedChatEntity = await _unitOfWork.Chats.GetByIdAsync(dto.ChatId);
if (updatedChatEntity != null)
{
    ChatDto chatDtoForUpdate = _mapper.Map<ChatDto>(updatedChatEntity);
    // Устанавливаем только что отправленное сообщение как последнее в
DTO для SignalR уведомления
    // так как updatedChatEntity.LastMessageId был только что обновлен этим
сообщением.
    chatDtoForUpdate.LastMessage = messageDto;
    // Также убедимся, что основные поля чата, которые могли измениться (как
UpdatedAt), присутствуют
    // AutoMapper должен был перенести updatedChatEntity.UpdatedAt в
chatDtoForUpdate.UpdatedAt
    // Если нет, то chatDtoForUpdate.UpdatedAt = updatedChatEntity.UpdatedAt;

    // Обновляем список участников в DTO, если он маппится и важен для клиента при
ChatDetailsUpdated
    // Это зависит от того, как настроен AutoMapper и что ожидает клиент от
ChatDetailsUpdated
    // var membersForDto = await _unitOfWork.Chats.GetChatMembersAsync(dto.ChatId);
    // chatDtoForUpdate.Members = _mapper.Map<List<UserSimpleDto>>(membersForDto);
    // chatDtoForUpdate.MemberCount = membersForDto.Count;
    // Если _mapper.Map<ChatDto>(updatedChatEntity) уже корректно заполняет members и
memberCount, то выше не нужно.

    if (updatedChatEntity.Members != null && updatedChatEntity.Members.Any())
    {
        // Используем маппинг из ChatMember в UserSimpleDto, который уже должен быть
настроен в ChatProfile
        // AutoMapper сможет спануть List<ChatMember> в List<UserSimpleDto>, если есть
маппинг ChatMember -> UserSimpleDto.
        chatDtoForUpdate.Members =
_mapper.Map<List<UserSimpleDto>>(updatedChatEntity.Members);
        _logger.LogInformation((Exception?)null, "MessageService: Populated
chatDtoForUpdate.Members with {Count} members for chat {ChatId}",
chatDtoForUpdate.Members.Count, chatDtoForUpdate.Id);
    }
    else
    {
        chatDtoForUpdate.Members = new List<UserSimpleDto>(); // Инициализируем пустым
списком, если нет участников
        _logger.LogWarning((Exception?)null, "MessageService:
updatedChatEntity.Members was null or empty for chat {ChatId}. Initialized
chatDtoForUpdate.Members as empty list.", chatDtoForUpdate.Id);
    }

    foreach (var memberId in memberIdsToNotify) // Используем тот же список участников
    {
        await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(new
List<Guid> { memberId }, chatDtoForUpdate); // Отправляем каждому индивидуально, если метод
принимает List
    }
}

```

```

        // Или если NotifyChatDetailsUpdatedAsync может принять IEnumerable<Guid> сразу
        // await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(memberIdsToNotify, chatDtoForUpdate);

        // Судя по интерфейсу IRealTimeNotifier, он принимает List<Guid> memberIds
    }

    // Если NotifyChatDetailsUpdatedAsync принимает IEnumerable или List:
    await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(memberIdsToNotify,
        chatDtoForUpdate);

    _logger.LogInformation("Chat {ChatId} details update notification sent to
{MemberCount} members.", dto.ChatId, memberIdsToNotify.Count);
    }
    else
    {
        _logger.LogWarning("Chat {ChatId} not found after update for sending ChatDetailsUpdated notification.", dto.ChatId);
    }

    _logger.LogInformation("Message {MessageId} sent by {SenderId} to chat {ChatId}.
Notified {MemberCount} members for new message.", message.Id, senderId, dto.ChatId,
memberIdsToNotify.Count);    return messageDto;
    }

    public async Task<MessageDto> CreateMessageWithAttachmentAsync(Guid senderId,
        CreateMessageWithAttachmentDto dto)
    {
        _logger.LogInformation("User {SenderId} creating message with attachment in chat
{ChatId}. FileKey: {FileKey}, FileName: {FileName}",
senderId, dto.ChatId, dto.FileKey, dto.FileName);

        var chat = await _unitOfWork.Chats.GetByIdAsync(dto.ChatId);
        if (chat == null) throw new KeyNotFoundException($"Chat with ID {dto.ChatId} not
found.");

        var isMember = await _unitOfWork.Chats.IsUserMemberAsync(dto.ChatId, senderId);
        if (!isMember) throw new UnauthorizedAccessException($"Sender {senderId} is not a member
of chat {dto.ChatId}.");

        if (string.IsNullOrEmpty(dto.FileKey) || string.IsNullOrEmpty(dto.FileName))
        {
            throw new ArgumentException("FileKey and FileName are required for messages with
attachments.");
        }

        var attachment = new Attachment
        {
            FileKey = dto.FileKey,
            FileName = dto.FileName,
            ContentType = dto.ContentType,
            Size = dto.FileSize,
            Type = dto.AttachmentType,
            Width = dto.Width,
            Height = dto.Height,
            ThumbnailKey = dto.ThumbnailKey
            // Другие поля Attachment, если они есть и приходят из DTO, можно добавить здесь
        };
    }

```

```

var message = new Message
{
    ChatId = dto.ChatId,
    SenderId = senderId,
    Timestamp = DateTime.UtcNow,
    Status = MessageStatus.Sent, // Изначально Sent
    Attachments = new List<Attachment> { attachment }
    // Text может быть null или пустым для сообщений только с вложениями
};

await _messageRepository.AddAsync(message);
await _unitOfWork.Chats.UpdateLastMessageAsync(dto.ChatId, message.Id);
await _unitOfWork.CommitAsync();

var messageDto = await MapMessageToDtoAsync(message, senderId);

// Уведомляем всех участников чата о новом сообщении
var chatMembers = await _unitOfWork.Chats.GetChatMembersAsync(dto.ChatId);
var memberIdsToNotify = chatMembers.Select(m => m.Id).ToList();

foreach (var memberId in memberIdsToNotify)
{
    if (messageDto != null)
    {
        await _realTimeNotifier.NotifyMessageReceivedAsync(memberId, messageDto);
    }

    if (memberId != senderId)
    {
        var senderUser = await
            _unitOfWork.Users.GetByIdAsync(senderId);
        var notificationText =
            $"{(senderUser?.Username ?? "Someone")} sent an attachment: {dto.FileName}";
        var notificationTitle = $"New attachment in {(chat?.Name ?? "chat")}";

        await _notificationService.CreateAndSendNotificationAsync(
            memberId,
            NotificationType.NewMessage, // Или можно создать
            NotificationType.NewAttachment, notificationText,
            notificationTitle, message.Id,
            "Message"
        );
    }
}

// Уведомляем об обновлении чата (LastMessage)
var updatedChatEntity = await _unitOfWork.Chats.GetByIdAsync(dto.ChatId);
if (updatedChatEntity != null)
{
    ChatDto chatDtoForUpdate = _mapper.Map<ChatDto>(updatedChatEntity);
    chatDtoForUpdate.LastMessage = messageDto;

    // --- НАЧАЛО БЛОКА ДЛЯ ВСТАВКИ/ЗАМЕНЫ ---
    if (updatedChatEntity.Members != null && updatedChatEntity.Members.Any())
    {
        // Используем маппинг из List<ChatMember> в List<UserSimpleDto>.
        // ChatProfile должен содержать маппинг ChatMember -> UserSimpleDto.
        chatDtoForUpdate.Members =
            _mapper.Map<List<UserSimpleDto>>(updatedChatEntity.Members);
        _logger.LogInformation((Exception?)null, "MessageService: Populated
chatDtoForUpdate.Members with {Count} members for chat {ChatId}",
chatDtoForUpdate.Members.Count, chatDtoForUpdate.Id);
    }
    else
    {
        chatDtoForUpdate.Members = new List<UserSimpleDto>();
        _logger.LogWarning((Exception?)null, "MessageService:

```



```

updatedChatEntity.Members was null or empty for chat {ChatId}. Initialized
chatDtoForUpdate.Members as empty list.", chatDtoForUpdate.Id);
    }
    // --- КОНЕЦ БЛОКА ДЛЯ ВСТАВКИ/ЗАМЕНЫ ---

    // Существующие отладочные логи (их можно оставить или убрать после исправления)
    _logger.LogInformation((Exception?)null, "PRE-NOTIFICATION LOG for Chat {ChatId}:
LastMessage.Id='{LastMessageId}', HasAttachments={HasAttachments},
MemberCountFromDto={MemberCount}",
chatDtoForUpdate.Id,
chatDtoForUpdate.LastMessage?.Id,
chatDtoForUpdate.LastMessage?.Attachments?.Any(),
chatDtoForUpdate.Members?.Count ?? -1);
    if
(chatDtoForUpdate.LastMessage?.Attachments != null)
    {
        foreach (var att in chatDtoForUpdate.LastMessage.Attachments)
        {
            _logger.LogInformation((Exception?)null, "PRE-NOTIFICATION LOG Attachment
for Chat {ChatId}: FileKey='{FileKey}', FileName='{FileName}'",
chatDtoForUpdate.Id, att.FileKey, att.FileName);
        }
    }
    // КОНЕЦ ДОБАВЛЕННЫХ ЛОГОВ

    // Код для NotifyChatDetailsUpdatedAsync аналогичен тому, что в SendMessageAsync
    await _realTimeNotifier.NotifyChatDetailsUpdatedAsync(memberIdsToNotify, chatDtoForUpdate);
    _logger.LogInformation("Chat {ChatId} details update notification sent to
{MemberCount} members after attachment message.", dto.ChatId, memberIdsToNotify.Count);
}

_logger.LogInformation("Message {MessageId} with attachment {FileKey} created by
{SenderId} in chat {ChatId}. Notified {MemberCount} members.",
message.Id, dto.FileKey, senderId, dto.ChatId, memberIdsToNotify.Count);

return messageDto;
}

private async Task<MessageDto> MapMessageToDtoAsync(Message? message, Guid currentUserId)
{
    if (message == null) return null;

    var dto = _mapper.Map<MessageDto>(message);
    dto.IsRead = message.ReadBy.Contains(currentUserId);

    if (message.SenderId != Guid.Empty)
    {
        var sender = await _unitOfWork.Users.GetByIdAsync(message.SenderId);
        if (sender != null)
        {
            dto.Sender = await _userService.MapUserToSimpleDtoWithAvatarUrlAsync(sender);
        }
    }
    // Populate PresignedUrl for each attachment
    if (dto.Attachments != null)
    {
        foreach (var attachment in dto.Attachments)
        {
            if (!string.IsNullOrEmpty(attachment.FileKey))
            {
                try
                {
                    attachment.PresignedUrl = await
_attachmentService.GetPresignedDownloadUrlAsync(attachment.FileKey, currentUserId);
                }
            }
        }
    }
}

```

```

        }
        catch (Exception ex)
        {
            _logger.LogWarning(ex, "Failed to generate presigned URL for
attachment {FileKey}", attachment.FileKey);
            attachment.PresignedUrl = null;
        }
    }
}

return dto;
}

private async Task<IEnumerable<MessageDto>> MapMessagesToDtoAsync(IEnumerable<Message> messages,
    Guid currentUserId)
{
    if (messages == null || !messages.Any()) return Enumerable.Empty<MessageDto>();
    var senderIds = messages.Select(m => m.SenderId).Where(id => id !=
Guid.Empty).Distinct().ToList();
    var sendersWithPresignedUrls = new Dictionary<Guid, UserSimpleDto>();
    if (senderIds.Any())
    {
        var senderUsers = await _unitOfWork.Users.GetManyByIdsAsync(senderIds);
        foreach (var user in senderUsers)
        {
            if (senderIds.Contains(user.Id))
            {
                sendersWithPresignedUrls[user.Id] = await
                _userService.MapUserToSimpleDtoWithAvatarUrlAsync(user);
            }
        }
        var dtos =
        new List<MessageDto>();
        foreach (var message in messages)
        {
            var dto = _mapper.Map<MessageDto>(message);
            dto.IsRead = message.ReadBy.Contains(currentUserId);
            if (message.SenderId != Guid.Empty &&
            sendersWithPresignedUrls.TryGetValue(message.SenderId, out var senderDto))
            {
                dto.Sender = senderDto;
            }
            // Populate PresignedUrl for each attachment
            if (dto.Attachments != null)
            {
                foreach (var attachment in dto.Attachments)
                {
                    if (!string.IsNullOrEmpty(attachment.FileKey))
                    {
                        try
                        {
                            attachment.PresignedUrl = await
                            _attachmentService.GetPresignedDownloadUrlAsync(attachment.FileKey, currentUserId);
                        }
                        catch (Exception ex)
                        {
                            _logger.LogWarning(ex, "Failed to generate presigned URL for
attachment {FileKey}", attachment.FileKey);
                            attachment.PresignedUrl = null;
                        }
                    }
                }
            }
            dtos.Add(dto);
        }
    }
}

```

```

    }
    return dtos;
}

public async Task<IEnumerable<MessageDto>> GetMessagesAsync(Guid chatId, Guid
currentUserId, int page = 1, int pageSize = 20)
{
    _logger.LogInformation("User {CurrentUserId} requesting messages for chat {ChatId},
Page: {Page}, PageSize: {PageSize}", currentUserId, chatId, page, pageSize);    var
isMember = await _unitOfWork.Chats.IsUserMemberAsync(chatId, currentUserId);    if
(!isMember) throw new UnauthorizedAccessException($"User {currentUserId} is not a member of
chat {chatId}.");

    var messages = await _messageRepository.GetByChatAsync(chatId, page, pageSize);
return await MapMessagesToDtoAsync(messages, currentUserId);    }

public async Task<MessageDto?> GetMessageByIdAsync(string id, Guid currentUserId)
{
    _logger.LogInformation("User {CurrentUserId} requesting message {MessageId}",
currentUserId, id);
    var message = await _messageRepository.GetByIdAsync(id);
if (message == null)
    {
        _logger.LogWarning("Message {MessageId} not found.", id);
return null;
    }

    var isMember = await _unitOfWork.Chats.IsUserMemberAsync(message.ChatId,
currentUserId);
    if (!isMember) throw new UnauthorizedAccessException($"User {currentUserId} is not a
member of the chat {message.ChatId} this message belongs to.");

    return await MapMessageToDtoAsync(message, currentUserId);
}

public async Task EditMessageAsync(string messageId, Guid editorUserId, string newText)
{
    _logger.LogInformation("User {EditorUserId} editing message {MessageId} with new text:
'{NewTextSnippet}'", editorUserId, messageId, newText.Substring(0, Math.Min(newText.Length,
20)));
    var message = await _messageRepository.GetByIdAsync(messageId);
    if (message == null) throw new KeyNotFoundException($"Message {messageId} not
found.");    if (message.SenderId != editorUserId) throw new
UnauthorizedAccessException($"User {editorUserId} cannot edit message {messageId} (not
sender).");

    // TODO: Проверить, не истекло ли время на редактирование (например, 24 часа)
// if ((DateTime.UtcNow - message.Timestamp).TotalHours > 24)
// {
//     throw new InvalidOperationException("Message can no longer be edited.");
// }
    if (message.IsDeleted) throw new InvalidOperationException("Cannot edit a deleted
message.");    if (message.Text == newText)
    {
        _logger.LogInformation("Message {MessageId} text is the same. No edit performed.",
messageId);
        return; // Нет изменений
    }

    message.Edit(newText); // Метод в доменной модели Message
await _messageRepository.UpdateAsync(message);

    var messageDto = await MapMessageToDtoAsync(message, editorUserId);

```

```

        var chatMembers = await _unitOfWork.Chats.GetChatMembersAsync(message.ChatId);
var memberIds = chatMembers.Select(m => m.Id).ToList();        if (messageDto != null)
    {
        await _realTimeNotifier.NotifyMessageEditedAsync(memberIds, messageDto);
    }
    _logger.LogInformation("Message {MessageId} edited successfully by {EditorUserId}.",
messageId, editorUserId);
}

public async Task DeleteMessageAsync(string messageId, Guid deleterUserId)
{
    _logger.LogInformation("User {DeleterUserId} attempting to delete message
{MessageId}", deleterUserId, messageId);
    var message = await _messageRepository.GetByIdAsync(messageId);
    if (message == null) throw new KeyNotFoundException($"Message {messageId} not
found.");        if (message.IsDeleted)
    {
        _logger.LogInformation("Message {MessageId} is already deleted.", messageId);
return;
    }

    // Проверка прав на удаление:
    // 1. Отправитель сообщения.
    // 2. Админ/Владелец чата (если есть такие права в бизнес-логике).
    var member = await _unitOfWork.Chats.GetChatMemberAsync(message.ChatId,
deleterUserId);        bool canDelete = message.SenderId == deleterUserId ||
        (member != null && (member.Role == ChatMemberRole.Admin || member.Role
== ChatMemberRole.Owner));
        // TODO: Добавить более гранулярные права для админов (может ли админ
удалять чужие сообщения).

    if (!canDelete)
    {
        throw new UnauthorizedAccessException($"User {deleterUserId} cannot delete message
{messageId}.");
    }

    message.SoftDelete(); // Метод в доменной модели Message
await _messageRepository.UpdateAsync(message);

    var chatMembers = await _unitOfWork.Chats.GetChatMembersAsync(message.ChatId);
var memberIds = chatMembers.Select(m => m.Id).ToList();
    await _realTimeNotifier.NotifyMessageDeletedAsync(memberIds, messageId,
message.ChatId);
    _logger.LogInformation("Message {MessageId} soft-deleted successfully by
{DeleterUserId}.", messageId, deleterUserId);
}

public async Task MarkMessageAsReadAsync(string messageId, Guid readerUserId, Guid chatId)
{
    var message = await _messageRepository.GetByIdAsync(messageId);
    if (message == null) throw new KeyNotFoundException($"Message with ID {messageId} not
found.");

    // Проверяем, что пользователь является участником чата
var chat = await _unitOfWork.Chats.GetByIdAsync(chatId);
    if (chat == null || !chat.Members.Any(m => m.UserId == readerUserId))
    {
        throw new UnauthorizedAccessException($"User {readerUserId} is not authorized to
mark messages as read in chat {chatId}.");
    }
}

```

```

        // Добавляем пользователя в список прочитавших, если его там еще нет
        if (!message.ReadBy.Contains(readerUserId))
        {
            message.ReadBy.Add(readerUserId);
            if (message.Status < MessageStatus.Read) // Обновляем статус сообщения, если оно
                было только Sent/Delivered
            {
                message.Status = MessageStatus.Read;
            }
            await _messageRepository.UpdateAsync(message);
            _logger.LogInformation("Message {MessageId} marked as read by user
{ReaderUserId}", messageId, readerUserId);

            // Уведомляем отправителя о том, что сообщение прочитано
            // Это комплексная логика, возможно, выходящая за рамки простого MarkAsRead
            // Уведомление других участников чата об обновлении статуса сообщения
            // Получаем актуальный список участников чата для уведомления
            var currentChatMembers = await _unitOfWork.Chats.GetChatMembersAsync(chatId);
            var memberUserIdsToNotify = currentChatMembers.Select(cm => cm.Id).ToList();
            // ИСПРАВЛЕН Вызов: аргументы и их порядок
            await _realTimeNotifier.NotifyMessageReadAsync(memberUserIdsToNotify, messageId,
            readerUserId, chatId);
        }
        else
        {
            _logger.LogInformation("Message {MessageId} was already read by user
{ReaderUserId}", messageId, readerUserId);
        }
    }

    public async Task MarkMessagesAsDeliveredAsync(IEnumerable<string> messageIds, Guid
    recipientUserId)
    {
        _logger.LogDebug("Marking messages as delivered to user {RecipientUserId}. Message
        IDs: {MessageIdsString}", recipientUserId, string.Join(",", messageIds));

        // Этот метод обычно вызывается, когда клиент пользователя подтверждает получение
        // сообщений.
        // Обновляем DeliveredTo в каждом сообщении и, возможно, общий статус сообщения.
        // IMessageRepository.MarkMessagesAsDeliveredToAsync должен эффективно обновить это в БД.
        await _messageRepository.MarkMessagesAsDeliveredToAsync(messageIds, recipientUserId);

        // После обновления, возможно, нужно уведомить отправителей этих сообщений,
        // что их сообщения доставлены этому recipientUserId. Это сложная логика уведомлений.
        // Для простоты, пока не будем отправлять real-time уведомления об этом событии обратно
        // отправителям.
        // Отправители могут получить обновленный статус сообщений при следующем запросе
        GetMessagesAsync.
        _logger.LogInformation("Messages ({MessageCount}) marked as delivered to
        {RecipientUserId}.", messageIds.Count(), recipientUserId);
    }
}

```

Код NotificationService.cs

```

using AutoMapper;
using Microsoft.Extensions.Logging; using
SigmailServer.Application.DTOS;
using SigmailServer.Application.Services.Interfaces;
using SigmailServer.Domain.Enums; using
SigmailServer.Domain.Interfaces; using
SigmailServer.Domain.Models;

namespace SigmailServer.Application.Services;

```

```

public class NotificationService : INotificationService
{
    private readonly INotificationRepository _notificationRepository;    private readonly
    IMapper _mapper;    private readonly IRealTimeNotifier _realTimeNotifier;    private
    readonly ILogger<NotificationService> _logger; private readonly IUnitOfWork _unitOfWork; //
    Может понадобиться для получения доп. инфо (User)

    public NotificationService(
        INotificationRepository notificationRepository,
        IMapper mapper,
        IRealTimeNotifier realTimeNotifier,
        ILogger<NotificationService> logger,
        IUnitOfWork unitOfWork)
    {
        _notificationRepository = notificationRepository;
        _mapper = mapper;
        _realTimeNotifier = realTimeNotifier;
        _logger = logger;
        _unitOfWork = unitOfWork;
    }

    public async Task<IEnumerable<NotificationDto>> GetUserNotificationsAsync(Guid userId,
    bool unreadOnly = false, int page = 1, int pageSize = 20)
    {
        _logger.LogInformation("Requesting notifications for user {UserId}, UnreadOnly:
    {UnreadOnly}, Page: {Page}, PageSize: {PageSize}", userId, unreadOnly, page, pageSize);
        // Пагинация должна быть реализована в репозитории GetForUserAsync
        var notifications = await _notificationRepository.GetForUserAsync(userId, unreadOnly);
        // TODO: Добавить пагинацию в репозиторий
        var pagedNotifications =
    notifications
        .OrderByDescending(n
    => n.CreatedAt)
        .Skip((page - 1) * pageSize)
        .Take(pageSize);
        return
    _mapper.Map<IEnumerable<NotificationDto>>(pagedNotifications);    }

    public async Task MarkNotificationAsReadAsync(string notificationId, Guid userId)
    {
        _logger.LogInformation("User {UserId} marking notification {NotificationId} as read",
    userId, notificationId);
        var notification = await _notificationRepository.GetByIdAsync(notificationId);
        if (notification == null)
        {
            _logger.LogWarning("Notification {NotificationId} not found.", notificationId);
            throw new KeyNotFoundException("Notification not found.");
        }
        if (notification.UserId != userId)
        {
            _logger.LogWarning("User {UserId} does not own notification {NotificationId}.",
    userId, notificationId);
            throw new UnauthorizedAccessException("User does not own
    this notification.");
        }

        if (!notification.IsRead)
        {
            await _notificationRepository.MarkAsReadAsync(notificationId); // Метод
            репозитория должен обновить IsRead = true
            _logger.LogInformation("Notification {NotificationId} marked as read for user
    {UserId}.", notificationId, userId);
            // Можно отправить real-time обновление клиенту, что уведомление прочитано (если
            UI это отображает)
            // await _realTimeNotifier.NotifyNotificationUpdatedAsync(userId,

```

```

_mapper.Map<NotificationDto>(await _notificationRepository.GetByIdAsync(notificationId)));
    }
else
    {
        _logger.LogInformation("Notification {NotificationId} was already read.",
notificationId);
    }
}

public async Task MarkAllUserNotificationsAsReadAsync(Guid userId)
{
    _logger.LogInformation("User {UserId} marking all their notifications as read",
userId);
    var unreadNotifications = (await _notificationRepository.GetForUserAsync(userId,
true)).ToList();
    if (!unreadNotifications.Any())
    {
        _logger.LogInformation("No unread notifications found for user {UserId}.",
userId);
        return;
    }

    foreach (var notification in unreadNotifications)
    {
        // Этот метод должен делать Update в БД
        await _notificationRepository.MarkAsReadAsync(notification.Id);
    }

    _logger.LogInformation("{Count} unread notifications for user {UserId} marked as
read.", unreadNotifications.Count, userId);
    // TODO: Потенциально отправить одно real-time событие, что все прочитано
}

public async Task CreateAndSendNotificationAsync(
    Guid recipientUserId,
    NotificationType type,
    string
message,
    string? title = null,
    string? relatedEntityId = null,
    string? relatedEntityType = null)
    {
        _logger.LogInformation("Creating notification for user {RecipientUserId}, Type:
{Type}, Title: '{Title}', Message: '{MessageSnippet}'",
recipientUserId, type, title, message.Substring(0, Math.Min(message.Length, 30)));

        if (string.IsNullOrEmpty(message))
        {
            _logger.LogError("Cannot create notification with empty message for user
{RecipientUserId}", recipientUserId);
            throw new ArgumentException("Notification message cannot be empty.");
        }

        var user = await _unitOfWork.Users.GetByIdAsync(recipientUserId);
        if(user == null)
        {
            _logger.LogWarning("Recipient user {RecipientUserId} not found. Notification not
created.", recipientUserId);
            return; // Или бросить исключение, если получатель
обязателен
        }

        var notification = new Notification
        {
            // Id генерируется в конструкторе/MongoDB
            UserId = recipientUserId,
            Type = type,
            Title = title,
            Message = message,
            RelatedEntityId = relatedEntityId,
            RelatedEntityType = relatedEntityType,

```

```

        IsRead = false,
        CreatedAt = DateTime.UtcNow
    };
    await _notificationRepository.AddAsync(notification);
    _logger.LogInformation("Notification {NotificationId} created for user {RecipientUserId}. Sending real-time update.", notification.Id, recipientUserId);
    await
    _realTimeNotifier.NotifyNewNotificationAsync(recipientUserId,
    _mapper.Map<NotificationDto>(notification));    }

    public async Task DeleteNotificationAsync(string notificationId, Guid userId)
    {
        _logger.LogInformation("User {UserId} attempting to delete notification {NotificationId}", userId, notificationId);
        var notification = await _notificationRepository.GetByIdAsync(notificationId);
        if (notification == null)
        {
            _logger.LogWarning("Notification {NotificationId} for deletion not found.", notificationId);
            throw new KeyNotFoundException("Notification not found.");
        }
        if (notification.UserId != userId)
        {
            _logger.LogWarning("User {UserId} does not own notification {NotificationId}. Cannot delete.", userId, notificationId);
            throw new UnauthorizedAccessException("User does not own this notification.");
        }

        // Предполагаем, что в INotificationRepository есть метод DeleteAsync(string id)
        // Если нет, и используется DeleteOldNotificationsAsync, то логика другая.
        // Добавим в IMessageRepository такой же метод DeleteAsync(string id) для консистентности
        // await _notificationRepository.DeleteAsync(notificationId); // Если такой метод есть
        _logger.LogWarning("DeleteNotificationAsync called. Assumed INotificationRepository has a DeleteAsync(string id) method. If not, implement or adjust. This is a placeholder.");
        // Заглушка, так как интерфейс INotificationRepository не объявлял DeleteAsync(id)
        // Если бы был, то:
        // await _notificationRepository.DeleteAsync(notificationId);
        // _logger.LogInformation("Notification {NotificationId} deleted by user {UserId}.", notificationId, userId);
        await Task.CompletedTask; // Удалите эту заглушку при наличии метода
    }
}

```

Код UserService.cs

```

using AutoMapper;
using Microsoft.Extensions.Logging; using
SigmailServer.Domain.Interfaces; using
SigmailServer.Application.DTOS;
using SigmailServer.Application.Services.Interfaces;
using SigmailServer.Domain.Models; using BCrypt.Net;
using SigmailServer.Domain.Enums; using
System.Threading;

namespace SigmailServer.Application.Services;

public class UserService : IUserService
{
    private readonly IUnitOfWork _unitOfWork;    private readonly
    IMapper _mapper;    private readonly IRealTimeNotifier
    _realTimeNotifier;    private readonly ILogger<UserService>
    _logger;    private readonly INotificationService

```



```

_notificationService;    private readonly IFileStorageRepository
_fileStorageRepository;

    public UserService(
IUnitOfWork unitOfWork,
        IMapper mapper,
        IRealTimeNotifier realTimeNotifier,
        ILogger<UserService> logger,
        INotificationService notificationService,
        IFileStorageRepository fileStorageRepository)
    {
        _unitOfWork = unitOfWork;
        _mapper = mapper;
        _realTimeNotifier = realTimeNotifier;
        _logger = logger;
        _notificationService = notificationService;
        _fileStorageRepository = fileStorageRepository;
    }

    private async Task<UserDto> MapUserToDtoWithAvatarUrlAsync(User user)
    {
        var userDto = _mapper.Map<UserDto>(user);
        if (!string.IsNullOrEmpty(userDto.ProfileImageUrl))
        {
            try
            {
                // ProfileImageUrl хранит ключ файла.
                // Генерируем presigned URL для доступа к файлу.
                // TimeSpan можно вынести в конфигурацию, если нужно.
                var presignedUrl = await
                _fileStorageRepository.GeneratePresignedUrlAsync(userDto.ProfileImageUrl,
                TimeSpan.FromHours(1));
                userDto.ProfileImageUrl =
                presignedUrl;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Failed to get presigned URL for avatar {AvatarKey} for
                user {UserId}. Returning key itself.", userDto.ProfileImageUrl, user.Id);
                // Оставляем ключ как fallback. Клиент не сможет загрузить, но и не должен
                унасть.
            }
            return userDto;
        }
    }

    public async Task<UserSimpleDto?>
    MapUserToSimpleDtoWithAvatarUrlAsync(User user)
    {
        if (user == null)
        {
            _logger.LogWarning("MapUserToSimpleDtoWithAvatarUrlAsync called with null user.");
            return null; // Возвращаем null, если пользователь null
        }

        var userSimpleDto = _mapper.Map<UserSimpleDto>(user);
        if (!string.IsNullOrEmpty(userSimpleDto.ProfileImageUrl))
        {
            try
            {
                var presignedUrl = await
                _fileStorageRepository.GeneratePresignedUrlAsync(userSimpleDto.ProfileImageUrl,
                TimeSpan.FromHours(1));
                userSimpleDto.ProfileImageUrl = presignedUrl;
            }
            catch (Exception ex)
            {
                _logger.LogError(ex, "Failed to get presigned URL for avatar {AvatarKey} for
                user {UserId} (SimpleDto). Returning key itself.", userSimpleDto.ProfileImageUrl, user.Id);
            }
            return userSimpleDto;
        }
    }

```

```

        private async Task<IEnumerable<UserDto>> MapUsersToDtoWithAvatarUrlAsync(IEnumerable<User>
users)
        {
            var userDtos = new List<UserDto>();
            foreach (var user in users)
            {
                userDtos.Add(await MapUserToDtoWithAvatarUrlAsync(user));
            }
            return userDtos;
        }
        private
        async Task<IEnumerable<UserSimpleDto>>
        MapUsersToSimpleDtoWithAvatarUrlAsync(IEnumerable<User> users)
        {
            var userSimpleDtos = new List<UserSimpleDto>();
            foreach (var user in users)
            {
                userSimpleDtos.Add(await MapUserToSimpleDtoWithAvatarUrlAsync(user));
            }
            return userSimpleDtos;
        }

        public async Task<UserDto?> GetByIdAsync(Guid id)
        {
            _logger.LogInformation("Requesting user by ID {UserId}", id);
            var user = await _unitOfWork.Users.GetByIdAsync(id);
            if (user == null)
            {
                _logger.LogWarning("User with ID {UserId} not found.", id);
                return null;
            }
            // return _mapper.Map<UserDto>(user);
            return await MapUserToDtoWithAvatarUrlAsync(user);
        }

        public async Task UpdateOnlineStatusAsync(Guid id, bool isOnline, string? deviceToken =
null)
        {
            _logger.LogInformation("Updating online status for user {UserId}: IsOnline={IsOnline},
DeviceToken={DeviceToken}", id, isOnline, deviceToken != null ? "provided" : "not provided");
            var user = await _unitOfWork.Users.GetByIdAsync(id);
            if (user == null)
            {
                _logger.LogWarning("User {UserId} not found for status update.", id);
                // Consider throwing KeyNotFoundException if user must exist
                return;
            }

            if (isOnline) user.GoOnline(deviceToken);
            else user.GoOffline();

            await _unitOfWork.Users.UpdateAsync(user);
            await _unitOfWork.CommitAsync();
            _logger.LogInformation("Online status updated for user {UserId} to
IsOnline={IsOnline}", id, user.IsOnline);

            // Уведомление контактов об изменении статуса
            var contacts = await _unitOfWork.Contacts.GetUserContactsAsync(id,
ContactRequestStatus.Accepted);
            var contactUserIds = contacts
                .Select(c => c.UserId == id ? c.ContactUserId : c.UserId)
                .Distinct()
                .ToList();
            if
            (contactUserIds.Any())
            {
                // Можно было бы дополнительно отфильтровать contactUserIds, чтобы уведомлять
                только тех, кто онлайн,
                // но это усложнит (нужно будет делать запрос к Users), и сам клиент решит, как
                ему обработать это.
                await

```

```

_realTimeNotifier.NotifyUserStatusChangedAsync(contactUserIds, id, user.IsOnline,
user.LastSeen);
    _logger.LogInformation("Notified {Count} contacts about status change for user
{UserId}", contactUserIds.Count, id);
    }
}

public async Task<IEnumerable<UserDto>> GetOnlineUsersAsync()
{
    _logger.LogInformation("Requesting all online users");
    // Внимание: этот метод может быть очень ресурсоемким, если пользователей много.
    // Рассмотрите альтернативы, если это не для админ-панели.
    var users = await _unitOfWork.Users.GetAllAsync(); // Предполагается, что GetAllAsync
существует var onlineUsers = users.Where(u => u.IsOnline);
    // return _mapper.Map<IEnumerable<UserDto>>(onlineUsers);
    return await MapUsersToDtoWithAvatarUrlAsync(onlineUsers);
}

public async Task<UserDto?> GetByUsernameAsync(string username)
{
    _logger.LogInformation("Requesting user by username {Username}", username);
    if (string.IsNullOrEmpty(username))
    {
        // Consider throwing ArgumentException if username cannot be empty
        _logger.LogWarning("Requested username was null or whitespace.");
        return null;
    }
    var user = await _unitOfWork.Users.GetByUsernameAsync(username);
    if (user == null)
    {
        _logger.LogWarning("User with username {Username} not found.", username);
        return null;
    }
    // return _mapper.Map<UserDto>(user);
    return await MapUserToDtoWithAvatarUrlAsync(user);
}

public async Task<IEnumerable<UserDto>> SearchUsersAsync(string searchTerm, Guid
currentUserId)
{
    _logger.LogInformation("User {CurrentUserId} searching for users with term
'{SearchTerm}'", currentUserId, searchTerm);
    if (string.IsNullOrEmpty(searchTerm))
    {
        return Enumerable.Empty<UserDto>();
    }
    // IUserRepository.FindUsersAsync должен быть реализован для поиска по username, email
и т.д.
    // и не должен возвращать текущего пользователя.
    var users = await _unitOfWork.Users.FindUsersAsync(searchTerm, 20,
CancellationToken.None);

    // Исключаем текущего пользователя из результатов поиска
    var foundUsers = users.Where(u => u.Id != currentUserId);
    // return _mapper.Map<IEnumerable<UserDto>>(foundUsers);
    return await MapUsersToDtoWithAvatarUrlAsync(foundUsers);
}

public async Task UpdateUserProfileAsync(Guid userId, UpdateUserProfileDto dto)
{
    _logger.LogInformation("User {UserId} updating profile with DTO:
Username={DtoUsername}, Email={DtoEmail}", userId, dto.Username, dto.Email);
    var user = await _unitOfWork.Users.GetByIdAsync(userId); if (user ==
null)
    {
        _logger.LogWarning("User {UserId} not found for profile update.", userId);
        throw new KeyNotFoundException("User not found.");
    }
}

```

```

        bool changed = false;
        string? oldUsername = user.Username; // Сохраняем старое имя для уведомлений
        string? oldEmail = user.Email; // Сохраняем старый email

        // Обновляем имя пользователя, если оно предоставлено и отличается
        if (!string.IsNullOrWhiteSpace(dto.Username) && dto.Username != user.Username)
        {
            var existingUserByUsername =
await
_unitOfWork.Users.GetByUsernameAsync(dto.Username);
            if (existingUserByUsername != null && existingUserByUsername.Id != userId)
            {
                throw new ArgumentException("Username already taken.");
            }
            // user.Username = dto.Username; // Будет установлено через UpdateProfile
            changed = true;
        }
        // Обновляем email, если он предоставлен и отличается
        if (!string.IsNullOrWhiteSpace(dto.Email) && dto.Email != user.Email)
        {
            var existingUserByEmail = await _unitOfWork.Users.GetByEmailAsync(dto.Email);
            if (existingUserByEmail != null && existingUserByEmail.Id != userId)
            {
                throw new ArgumentException("Email already taken.");
            }
            // user.Email = dto.Email; // Будет установлено через UpdateProfile
            changed = true;
        }
        // Обновляем Bio, если оно предоставлено и отличается (может быть null или пустым)
        if (dto.Bio != user.Bio)
        {
            // user.Bio = dto.Bio; // Будет установлено через UpdateProfile
            changed = true;
        }

        if(changed)
        {
            // Используем метод модели User для обновления полей
            user.UpdateProfile(
                string.IsNullOrWhiteSpace(dto.Username) ? oldUsername : dto.Username,
                string.IsNullOrWhiteSpace(dto.Email) ? oldEmail : dto.Email,
                dto.Bio,
                user.ProfileImageUrl // ProfileImageUrl здесь не меняем, он обновляется
отдельно
            );

            await _unitOfWork.Users.UpdateAsync(user);
            await _unitOfWork.CommitAsync();
            _logger.LogInformation("Profile for user {UserId} updated.", userId);

            // Уведомление контактов, если изменилось имя пользователя
            if (!string.IsNullOrWhiteSpace(dto.Username) && dto.Username != oldUsername)
            {
                var contacts = await _unitOfWork.Contacts.GetUserContactsAsync(userId,
ContactRequestStatus.Accepted);
                var contactUserIdsToNotify = contacts
                    .Select(c => c.UserId == userId ? c.ContactUserId : c.UserId)
                    .Distinct();

                foreach (var contactUserId in contactUserIdsToNotify)
                {
                    // TODO: Пересмотреть текст уведомления, чтобы он был более общим, если и
email меняется
                    await _notificationService.CreateAndSendNotificationAsync(
                        contactUserId,

```

```

        NotificationType.UserProfileUpdated,
        $"User '{oldUsername}' is now known as '{user.Username}'.",
        "User Profile Updated",
        user.Id.ToString(),
relatedEntityId:
relatedEntityType: "User"
    );
}

_logger.LogInformation("Notified {Count} contacts about username change for
user {UserId}", contactUserIdsToNotify.Count(), user.Id);
    // Также можно отправить real-time уведомление об изменении профиля, если это
нужно
    // await
_realTimeNotifier.NotifyUserProfileChangedAsync(contactUserIdsToNotify, user.Id, user.Username,
user.ProfileImageUrl); // Нужен метод в IRealTimeNotifier
}
else
{
    _logger.LogInformation("No changes detected for user {UserId} profile.", user.Id);
} } public async Task UpdateUserAvatarAsync(Guid userId, string avatarFileKey)
{
    _logger.LogInformation("User {UserId} updating avatar with new file key
{AvatarFileKey}", user.Id, avatarFileKey);
    var user = await _unitOfWork.Users.GetByIdAsync(userId);
    if (user == null)
    {
        _logger.LogWarning("User {UserId} not found for avatar update.", user.Id);
        // Если файл уже загружен в S3, а пользователь не найден, это проблема.
        // Возможно, стоит удалить "бесхозный" файл из S3.
        // Пока просто выбрасываем исключение. throw new
        KeyNotFoundException("User not found.");
    }

    string? oldAvatarKey = user.ProfileImageUrl;

    user.ProfileImageUrl = avatarFileKey; // Обновляем URL/ключ аватара

    await _unitOfWork.Users.UpdateAsync(user);
    await _unitOfWork.CommitAsync();
    _logger.LogInformation("Avatar updated for user {UserId}. New key: {NewKey}, Old key:
{OldKey}", user.Id, avatarFileKey, oldAvatarKey ?? "N/A");

    // Удаляем старый аватар из S3, если он был и он не является каким-то placeholder'ом
по умолчанию
    if (!string.IsNullOrEmpty(oldAvatarKey) && oldAvatarKey !=
avatarFileKey)
    {
        try
        {
            _logger.LogInformation("Attempting to delete old avatar {OldAvatarKey} for
user {UserId}", oldAvatarKey, user.Id);
            await
            _fileStorageRepository.DeleteFileAsync(oldAvatarKey);
            _logger.LogInformation("Successfully deleted old avatar {OldAvatarKey} for
user {UserId}", oldAvatarKey, user.Id);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to delete old avatar {OldAvatarKey} for user
{UserId} from S3. This needs to be handled manually or by a cleanup job.", oldAvatarKey,
user.Id);
        }
    }
    // Уведомляем контакты/других пользователей об обновлении аватара через SignalR
    try
    {
        var contacts = await _unitOfWork.Contacts.GetUserContactsAsync(userId,
ContactRequestStatus.Accepted);
        var contactUserIdsToNotify = contacts
        .Select(c => c.UserId == userId ? c.ContactUserId : c.UserId)
        .Distinct()

```

```

        .ToList(); // ToList() чтобы избежать многократного выполнения запроса

        if (contactUserIdsToNotify.Any())
        {
            string? newAvatarFullUrl = null;
            if (!string.IsNullOrEmpty(user.ProfileImageUrl)) // user.ProfileImageUrl
            {
                try
                {
                    newAvatarFullUrl = await
_ fileStorageRepository.GeneratePresignedUrlAsync(user.ProfileImageUrl, TimeSpan.FromHours(1));
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, "Failed to generate presigned URL for new avatar
{AvatarKey} for user {UserId} during notification. Avatar URL will be sent as key.",
user.ProfileImageUrl, userId);
                    newAvatarFullUrl = user.ProfileImageUrl; // Отправляем ключ как
fallback, если генерация URL не удалась
                }
            }
            else
            {
                newAvatarFullUrl = string.Empty; // Если ключа нет, отправляем пустую
строку
            }

            // Убедитесь, что NotifyUserAvatarUpdatedAsync существует в IRealTimeNotifier и
SignalRNotifier
            await _realTimeNotifier.NotifyUserAvatarUpdatedAsync(contactUserIdsToNotify,
userId, newAvatarFullUrl);
            _logger.LogInformation("Notified {Count} contacts about avatar update for user
{UserId}", contactUserIdsToNotify.Count, userId);
        }

        catch (Exception ex)
        {
            _logger.LogError(ex, "Failed to notify contacts about avatar update for user
{UserId}.", userId);
            // Не прерываем выполнение, так как основная операция (обновление аватара) уже
завершена.
        }
    }

    public async Task<IEnumerable<UserSimpleDto>> SearchUsersAsync(string searchTerm, int
limit = 20)
    {
        _logger.LogInformation("Searching for users (simple DTO) with term '{SearchTerm}' and
limit {Limit}", searchTerm, limit);
        if (string.IsNullOrEmpty(searchTerm))
        {
            return Enumerable.Empty<UserSimpleDto>();
        }
        // Пока для простоты оставим без этого

        var users = await _unitOfWork.Users.FindUsersAsync(searchTerm.ToLower(), limit,
CancellationToken.None); // Заменено SearchUsersByUsernameAsync на FindUsersAsync и добавлен
CancellationToken.None
        // return _mapper.Map<IEnumerable<UserSimpleDto>>(users);
        return await MapUsersToSimpleDtoWithAvatarUrlAsync(users);
    }

    public async Task ChangePasswordAsync(Guid userId, string oldPassword, string newPassword)
    {
        _logger.LogInformation("User {UserId} attempting to change password.", userId);
        var user = await _unitOfWork.Users.GetByIdAsync(userId);
        if (user == null)
        {
            _logger.LogWarning("User {UserId} not found for password change.", userId);
            throw new KeyNotFoundException("User not found.");
        }
    }

```

```

        if (!BCrypt.Net.BCrypt.Verify(oldPassword, user.PasswordHash))
    {
        _logger.LogWarning("Invalid old password for user {UserId}.", userId);
        throw new ArgumentException("Invalid old password.");
    }

    var newPasswordHash = BCrypt.Net.BCrypt.HashPassword(newPassword);
    user.UpdatePassword(newPasswordHash); // Предполагается, что в User.cs есть метод
UpdatePassword

    await _unitOfWork.Users.UpdateAsync(user);
    await _unitOfWork.CommitAsync();
    _logger.LogInformation("Password changed successfully for user {UserId}.", userId);
}
}

```