Міністерство освіти і науки України
Криворізький національний університет
Кафедра моделювання та програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА
на здобуття ступеня вищої освіти бакалавра
зі спеціальності 121 – Інженерія програмного забезпечення

На тему: РОЗРОБКА МЕСЕНДЖЕРА З ІНТЕГРАЦІЄЮ AI ДЛЯ ВИЯВЛЕННЯ СПАМУ ТА ТОКСИЧНИХ ПОВІДОМЛЕНЬ У ЧАТАХ

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань

Студент гр. ІПЗ-21-1 _____ /Д. С. Бут /

Керівник кваліфікаційної роботи _____ / І. А. Котов /

Завідувач кафедри _____ / А. М. Стрюк /

Кривий Ріг
2025

Криворізький національний університет
Факультет: Інформаційних технологій
Кафедра: Моделювання та програмного забезпечення
Ступінь вищої освіти: бакалавр
Спеціальність: 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
А. М. Стрюк
«___» _____ 20__р.

ЗАВДАННЯ
на кваліфікаційну роботу
студенту групи ІПЗ-21-1 Бут Дениса Сергійовича

1. Тема: РОЗРОБКА МЕСЕНДЖЕРА З ІНТЕГРАЦІЄЮ AI ДЛЯ ВИЯВЛЕННЯ СПАМУ ТА ТОКСИЧНИХ ПОВІДОМЛЕНЬ У ЧАТАХ затверджено наказом по КНУ № 199с від «10» квітня 2025 р.
2. Термін подання студентом закінченої роботи: «15» червня 2025 р.
3. Вихідні дані по роботі: озроблюваний програмний комплекс має забезпечити безпечне спілкування користувачів у чатах за допомогою автоматичного виявлення спаму та токсичних повідомлень на основі технологій штучного інтелекту. Система повинна інтегрувати клієнт-серверну архітектуру з модулем фільтрації повідомлень у режимі реального часу.
4. Зміст пояснювальної записки: виконати аналіз предметної області, архітектурне та алгоритмічне проектування, реалізацію месенджера з інтегрованим AI-модулем, тестування, модерацію повідомлень.
5. Перелік ілюстративного матеріалу: блок схеми алгоритмів, діаграми архітектури, знімки інтерфейсів, знімки прикладів.

Календарный план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Огляд літератури за тематикою, аналіз існуючих месенджерів та методів виявлення токсичності	20.03.2025 – 26.03.2025
2	Підготовка матеріалів першого розділу: постановка задачі, аналіз проблеми спаму і токсичності	27.03.2025 – 02.04.2025
3	Підготовка матеріалів другого розділу: розробка моделей і архітектури месенджера	03.04.2025 – 09.04.2025
4	Розробка алгоритмів та схеми взаємодії з Perspective API для модерації повідомлень	10.04.2025 – 14.04.2025
5	Підготовка матеріалів третього розділу: реалізація клієнт-серверної архітектури та AI-модулів	15.04.2025 – 30.04.2025
6	Тестування месенджера, інтеграція підсистеми модерації, підготовка інтерфейсу користувача	01.05.2025 – 08.05.2025
7	Оформлення пояснювальної записки та підготовка до захисту	08.05.2025 – 15.06.2025

Дата видачі завдання: «20» березня 2025 р.

Студент _____ / Д. С. Бут /

Керівник роботи _____ / І. А. Котов /

РЕФЕРАТ

МЕСЕНДЖЕР, ШТУЧНИЙ ІНТЕЛЕКТ, ТОКСИЧНІСТЬ, СПАМ, PERSPECTIVE API

Пояснювальна записка: 63 с., 2 табл., 39 рис., 2 дод., 21 джерела.

Метою кваліфікаційної роботи є розробка сучасного клієнт-серверного месенджера з інтеграцією підсистеми штучного інтелекту для виявлення спаму та токсичних повідомлень у чатах.

Основним завданням при розробці системи стало поєднання інструментів реального часу з можливістю автоматичної модерації вмісту повідомлень. Для досягнення поставленої мети було проведено аналіз популярних месенджерів та методів боротьби з небажаним контентом. Оцінено роль штучного інтелекту в сучасних комунікаційних платформах.

У ході роботи створено архітектуру клієнт-серверного застосунку з використанням Angular, Node.js, Firebase, а також інтегровано Perspective API – хмарний сервіс для аналізу текстів на наявність токсичності та спаму. Повідомлення проходять модерацію перед збереженням у базі даних. За результатами аналізу їм присвоюється статус: «ok» або «blocked».

Отриманий програмний комплекс дозволяє здійснювати безпечне спілкування в режимі реального часу, автоматично виявляючи небажаний контент. Система легко масштабується, підтримує гнучке розширення та демонструє ефективність під час інтеграції зі сторонніми AI-рішеннями.

ABSTRACT

MESSENGER, ARTIFICIAL INTELLIGENCE, TOXICITY, SPAM, PERSPECTIVE API

Explanatory note: 63 pages, 2 tables, 39 figures, 2 appendices, 21 sources.

The aim of this qualification work is to develop a modern client-server messenger with an integrated artificial intelligence subsystem for detecting spam and toxic messages in chats.

The main task during development was to combine real-time communication tools with the ability to automatically moderate message content. To achieve this goal, an analysis of popular messengers and methods of dealing with undesirable content was conducted. The role of artificial intelligence in modern communication platforms was also evaluated.

During the course of the work, a client-server application architecture was developed using Angular, Node.js, and Firebase, and the Perspective API – a cloud-based service for analyzing text for toxicity and spam – was integrated. Messages are moderated before being saved to the database. Based on the analysis results, they are assigned a status: "ok" or "blocked."

The resulting software solution enables secure real-time communication, automatically detecting inappropriate content. The system is easily scalable, supports flexible extension, and demonstrates efficiency when integrated with third-party AI solutions.

Зміст

Вступ

У сучасному цифровому світі месенджери стали невід'ємною частиною повсякденного життя, забезпечуючи миттєвий зв'язок між людьми на відстані. Від особистого спілкування до бізнес-процесів, від обміну новинами до координації соціальних ініціатив – ці платформи об'єднують мільйони, стираючи кордони та часові зони. Однак із зростанням їхньої популярності зростає й темна сторона цифрової взаємодії: потоки текстових і мультимедійних повідомлень все частіше стають каналом для поширення спаму, маніпуляцій і образливого контенту, що загрожує як індивідуальній безпеці, так і здоров'ю соціальних мереж.

Кожен день користувачі стикаються з низкою викликів: автоматизовані розсилки, фейкові новини, агресивні коментарі чи навіть координація злочинних дій через чати. Ці загрози не лише порушують довіру до цифрового спілкування, але й створюють постійний тиск на розробників, які мають знаходити рівновагу між свободою слова, конфіденційністю та захистом суспільства.

На тлі цих проблем методи штучного інтелекту постають як ключовий інструмент прогресу. Вони пропонують інноваційні підходи до аналізу контенту, вчасно виявляючи небезпечні шаблони, фільтруючи шум і створюючи безпечніший простір для спілкування. Технології машинного навчання, обробки природної мови та комп'ютерного зору вже сьогодні вбудовані в основу функціонування месенджерів, трансформуючи їх із пасивних каналів передачі інформації в розумні системи, здатні навчатися та адаптуватися.

Проте шлях до ідеального балансу є складним. Суворі вимоги до шифрування даних, різноманітність мовних і культурних контекстів, а також постійна еволюція методів зловмисників ставлять нові питання перед розробниками та дослідниками. Як забезпечити ефективність алгоритмів, не порушуючи прав користувачів? Чи можна створити універсальні рішення для виявлення токсичності в умовах культурного розмаїття?

Ця робота досліджує, як штучний інтелект переосмислює роль месенджерів у сучасному світі, які механізми лежать в основі цих технологій і які виклики залишаються невирішеними. Аналізуючи взаємодію інновацій із соціальними

потребами, ця робота прагне показати, що майбутнє цифрової комунікації – це поєднання технічної досконалості, етичної відповідальності та глибокого розуміння людських цінностей.

1. ПОСТАНОВКА ЗАВДАННЯ РОЗРОБКИ МЕСЕНДЖЕРА З ІНТЕГРАЦІЄЮ AI

1.1 Загальний порівняльний аналіз принципів функціонування та функцій сучасних месенджерів

Месенджери виступають не лише інструментом швидкого обміну повідомленнями, а й багатофункціональними платформами з інтуїтивним інтерфейсом та розширеними сервісами. Вони застосовуються не тільки для приватного спілкування, але й інтегруються в бізнес-процеси, забезпечуючи соціальну координацію та навіть керують пристроями Інтернету речей. Ретельне вивчення архітектури й функціональних можливостей месенджерів дозволить виокремити основні відмінності між рішеннями та окреслити тренди подальшого розвитку.

Для подальшого аналізу обрано такі популярні на даний момент месенджери як: Telegram, Facebook Messenger, Signal, WhatsApp, Discord. Усі вони мають значну аудиторію, що робить їх репрезентативними з точки зору дослідження архітектурних рішень та функціональних можливостей:

- Telegram який в 2025 році налічує близько 900 мільйонів користувачів щомісяця [1];
- Facebook Messenger має більш ніж 1 мільярд користувачів що місяця [1];
- Signal набирає популярність серед користувачів та налічує приблизно 2,7 мільйонів завантажень щомісяця [2];
- WhatsApp зараз є лідером серед месенджерів та приблизно 16,3% опитаних людей вважають його улюбленим месенджером [3];
- Discord демонструє 154 мільйони користувачів щомісячно.

Такий набір дозволяє порівняти рішення з точки зору масштабованості, безпеки, API та інтеграційних можливостей, а також прослідкувати, як різниця в розмірах аудиторії впливає на підходи до архітектури та розвитку нових сервісів.

Таблиця 1.1 – Загальний порівняльний аналіз

Критерії	Telegram	Facebook	Signal	WhatsApp	Discord
Архітектура	Клієнт-сервер+ MTProto	Клієнт-сервер (GraphQL, WebSocket)	Клієнт-сервер (REST/HTTPS)	Клієнт-сервер (Erlang, Protobuf-протокол)	Мікросервіси на WebSocket+HTTP API
Шифрування	E2EE в “Секретних чатах”, TLS для груп	Опціональні секретні бесіди	E2EE за замовчуванням	E2EE за замовчуванням	TLS для передачі, без E2EE за замовчуванням
Синхронізація	Хмарне сховище повідомлень	Сервер з клонуванням історії	Локальне сховище, ключі на пристрої	Локальні резервні копії	Синхронізація через сервер сховища
Комунікація	Приватні, групові, канали, стікери	Приватні, групові, історії, реакції	Приватні, групові, мінімум метаданих	Приватні, групові, голосові	Приватні, серверні канали, голосові чати
Додаткові інструменти	Бот-платформ, API , платіжні канали	Боти, ігри, платіжні сервіси	Немає	Business API	Slash-команди, інтеграції з ігровими серверами
Конфіденційність	Захист від DDoS, мінімум метаданих у каналах	Збір метаданих, опції приватності	Мінімальні метадані, відкритий код сервера	E2EE за замовчування, мінімум метаданих, двофакторна аутентицікація	Ролі/права, опції “невидимки”

Проведемо детальний аналіз таблиці яка була наведена вище (див. таб. 1.1), що допоможе виокремити сильні й слабкі сторони кожної платформи.

Архітектура:

Усі представлені месенджери побудовані за клієнт-серверною моделлю, але зі значними відмінностями. Telegram використовує власний протокол MTProto, що дає змогу масштабувати кластер серверів і знижувати затримки, а також спрощує розгортання нових центрів обробки даних [4]. Facebook Messenger будує API на GraphQL і WebSocket, що додає гнучкості фронтенду й мінімізує надмірні запити [5]. Signal дотримується класичного REST-over-HTTPS, акцентуючи увагу на безпеці й простоті [6]. WhatsApp, написаний на Erlang із Protobuf-повідомленнями, демонструє високу надійність і стійкість до перевантажень [8]. Discord застосовує мікро сервісну архітектуру з WebSocket даний підхід оптимізований для низько латентної передачі голосу й відео [7].

Шифрування:

- Signal і WhatsApp забезпечують E2EE на всіх типах чатів за замовчуванням, використовуючи Signal Protocol, що гарантує, що розшифрувати повідомлення може тільки відправник і адресат [6];
- Telegram пропонує E2EE лише в “секретних чатах”, тоді як групові розмови захищені TLS-з’єднанням, але сервер має доступ до метаданих і змісту [4];
- Facebook Messenger дає опцію “секретних бесід” з E2EE на базі X25519, проте в звичайних чатах шифрування закінчується на транспортному рівні [5];
- Discord використовує лише TLS-шифрування при передачі даних, без наскрізного шифрування повідомлень, що робить його менш придатним для конфіденційного спілкування [7].

Синхронізація:

Зручність мультипристроїв залежить від обраного підходу до зберігання історії:

- Telegram реалізує хмарне сховище, тож історія доступна миттєво на будь-якому підключеному пристрої без додаткових налаштувань;
- Facebook Messenger “клонує” історію на сервері, але іноді виникають затримки при відновленні старих повідомлень, особливо в масивних групах;

- Signal зберігає все локально й підтримує синхронізацію лише через комп’ютер-клієнт, що підвищує безпеку, але ускладнює міграцію даних між телефонами;
- WhatsApp допускає локальні резервні копії на Google Drive або iCloud і має обмежену бета-версію мульти-девайс, де історія синхронізується через тимчасове посередництво серверів WhatsApp;
- Discord синхронізує всі дані через центральний сервер, включно зі статусами й голосовими каналами, що дозволяє миттєве переключення між ПК, веб-клієнтом і мобільним додатком.

Комунікація:

- Telegram додає до стандартних чатів канали, публічні групи й стікери, а також прив’язку бота та кастомні реакції;
- Facebook Messenger – це чати, спільні історії (“Stories”) і реакції на повідомлення, а також інтеграція з соцмережею Facebook;
- Signal обмежується текстом, стікерами й медіа файлами, без каналів чи “Stories”, щоб зменшити площу атаки для зловмисників;
- WhatsApp надає статуси (аналог “Stories”), голосові та відео-дзвінки, а також групові конференції до 32 учасників;
- Discord орієнтований на геймерську аудиторію: крім текстових і групових чатів, він створює постійні сервери з каналами (текстовими й голосовими), а відео-дзвінки та демонстрація екрану стають частиною стандартного набору.

Додаткові інструменти:

- Telegram має одну з найпотужніших бот-платформ, відкритий API, інтегровані платіжні канали та кастомні клавіатури;
- Facebook Messenger підтримує ігри, інтеграцію з платіжними сервісами Meta і потужний SDK для розробників бізнес-рішень;
- Signal свідомо вирізав додаткові сервіси, щоб зберегти легкість і надійність клієнта;

- WhatsApp надає Business API для автоматизації повідомлень комерційним клієнтам і каталоги товарів прямо в чаті;
- Discord відрізняється slash-командами, інтеграціями з ігровими платформами (Steam, Twitch).

Конфіденційність

- Signal – еталон щодо конфіденційності: мінімальні метадані, відкритий код і повна E2EE [6];
- WhatsApp теж має E2EE за замовчуванням, але збирає більше метаданих для аналітики й реклами, пропонуючи двофакторну аутентифікацію [8];
- Telegram зберігає повідомлення в зашифрованому вигляді на своїх серверах, але не відкриває код серверної частини й збирає метадані для балансування навантаження [4];
- Facebook Messenger має найбільше з п’яти обсяг даних про користувачів, адже інтегрований у соцмережу Facebook і частково використовує ці дані для таргетингу реклами [5];
- Discord збирає стандартні логи активності й передає їх на сервер, але не пропонує наскрізного шифрування, натомість надаючи гнучкі налаштування прав доступу на рівні серверів і каналів [7].

З огляду видно, що поступове підвищення вимог до безпеки призводить то того, що наскрізне шифрування стає базовою опцією тоді як транспортне TLS-шифрування лишається мінімумом для не “секретних” сценаріїв. Мультипристроєва синхронізація все частіше реалізовується через хмарне сховище, забезпечуючи миттєвий доступ до історії з будь-якого девайсу. Розвиток екосистеми сприяє швидкому впровадженню нових сервісів. Функціональний набір поєднує текстові й групові чати, мультимедіа та дзвінки. Ці загальні тенденції визначають, які архітектурні та функціональні рішення обиратимуть розробники нових месенджерів.

1.2 Аналіз проблеми спаму та токсичних повідомлень у чатах сучасних месенджерів

У міру еволюції засобів цифрового спілкування – від простих SMS до багатофункціональних месенджерів із підтримкою мультимедіа, бот-платформ і мультипристроєвої синхронізації – зростають і можливості для зловмисників. Нові канали комунікації стають майданчиком не лише для корисного обміну інформацією, а й поширення спаму, фішингових або шахрайських схем та токсичних повідомлень. Сучасні месенджери завдяки широкому колу функцій та відкритим API не лише спрощують щоденне спілкування, але й відкривають додаткові вектори атак, що вимагають від розробників і дослідників застосування ефективних методів виявлення та нейтралізації шкідливого контенту.

Нижче наведені основні категорії небезпечних або зловмисних повідомлень:

- Спам

  Небажані розсилки рекламного чи інформаційного характеру, які зазвичай відволікають, завантажують мережу й можуть містити посилання на шкідливі ресурси;

  За даними з Norton, серед найпоширеніших рекламних спам-розсилок у WhatsApp – інвестиційні повідомлення з обіцянками швидкого прибутку, які вводять користувачів в оману та можуть призвести до фінансових втрат [16];

- Фішинг (smishing)

  Повідомлення, що маскують під довірені (банк, родичі тощо), щоб виманити облікові дані, PIN-коди. У Великій Британії з 2023 по 2025 рік шахраї через WhatsApp маскувалися під синів і доньок жертв, надсилаючи повідомлення про нібито втрату телефону і прохання терміново перерахувати гроші за оренду житла. Внаслідок таких “smishing”-атак у країні було втрачено понад 226 тисяч [17];

- Скам (scam)

  Складні схеми обману із метою фінансової вигоди: інвестиційні шахрайства, “романтичні” афери, шахрайство з крипто валютою тощо.

У 2024 році жінка з Великої Британії витратила понад 22,8 тисячі на нібито романтичні стосунки з “Крісом Брауном”, підтверджені AI-генерованими відео та голосовими повідомленнями в WhatsApp і Telegram. Це призвело до емоційного та фінансового виснаження жертви [18];

- Поширення шкідливого програмного забезпечення.

  Повідомлення містять посилання або вкладення з вірусами, шпигунським ПЗ, банківськими троянами і т.д.

  Amesty International задокументувала випадки доставки шпигунського ПЗ Pegasus через Viber до журналістів організації BIRN у Сербії, що поставило під загрозу безпеку та свободу слова незалежних медіа [19];

- Токсичні повідомлення та мова ненависті

  Образи, приниження, заклики до насильства або дискримінація певних груп за ознаками раси, релігії, статі. З жовтня 2023 року кількість повідомлень із закликами до ненависті й насильства в Telegram зросла на 433%, зокрема проти різних етнічних релігійних спільнот, що загрожує ескалацією міжгрупових конфліктів [20];

- Кібербулінг

  Систематичне переслідування, приниження чи залякування особи в онлайн-чатах, що часто призводить до психологічних травм або суїцидальних думок.

  Дослідження NIH свідчить, що підлітки, які зазнали кібербулінгу, у 2-3 рази частіше мають суїцидальні думки й спроби [21];

- Дезінформація та радикалізація

  Розповсюдження неправдивих новин або екстремістських наративів із метою маніпуляції свідомістю користувачів або підготовки насильницьких дій. Закриті Telegram канали використовуються для поширення екстремістських ідей і координації

діяльності радикальних груп, що може призвести до радикалізації та терактів [22].

Підсумовуючи, аби протидіяти наростаючим загрозам спаму, фішингу, шахрайства і токсичних повідомлень у месенджерах, необхідно поєднати інноваційні технічні рішення, чіткі регуляторні рамки та активну просвітницьку роботу. Такий багаторівневий підхід забезпечить як оперативне виявлення загроз, так і зменшення їхнього соціального та економічного впливу.

1.3 Аналіз месенджерів із функціями виявлення спаму і токсичних повідомлень

Сучасні месенджери активно розвивають інструменти для боротьби зі спамом, проте функції автоматичного виявлення токсичних повідомлень залишаються рідкістю. Більшість платформ орієнтовані на базову безпеку (шифрування, автентифікацію) та фільтрацію спаму через алгоритми машинного навчання, але ігнорують складність контекстної токсичності. Розглянемо функції наявні в месенджерах які були представлені раніше.

Telegram:

З самого початку позиціонував себе як максимально відкрита та розширювана платформа з відкритим API.

- Анти спам у групах і каналах: У супер групах та каналах Telegram Автоматично підключає “Анти спам-бот” від офіційної команди, який заздалегідь блокує акаунти з ознаками масових розсилок чи підозрілою поведінкою такою як: надто часті повідомлення, велика кількість нових учасників за короткий час. Адміністратори можуть гнучко налаштувати пороги чутливості та додавати власні списки заблокованих слів і посилань [9];
- Скарги та тимчасові обмеження: Користувачі можуть поскаржитись на будь-яке повідомлення і в разі підтвердження порушення акаунт порушника обмежується на певний час або зовсім видаляється;

- Токсичність через стороні боти: Оскільки через шифрування клієнтів не можливо централізовано аналізувати всю кореспонденцію, модератори груп використовують стороні рішення у вигляді ботів на базі Google Perspective API або власні ML-моделі для детектування образливої лексики, ненормативної лексики чи закликів до насилля.

З наведеної вище інформації видно, що Telegram має базовий функціонал для виявлення спам-повідомлень і опцій для детекції токсичного контенту, однак для його ефективного використання потрібні додаткові налаштування та інтеграція сторонніх рішень, а за замовчуванням ці інструменти не є надто ефективними і більшість з них не працює в особистих чатах.

Facebook Messenger:

Як частина екосистеми Meta, Messenger поєднує серверне сканування з прозорими звітами.

- Машине навчання на сервері: Повідомлення проходять попередню фільтрацію на наявність шкідливих URL, фішингових схем і матеріалів дитячої порнографії, після чого підозрілі випадки передаються на модераторам Meta [10];
- Сповіщення про безпеку (Safety Alerts): У зашифрованих чатах модель аналізує метадані такі як: частоту повідомлень, різкий ріст аудиторії. Після чого оперативно попереджає користувача про потенційні атаки без розкриття вмісту [11].

Facebook Messenger реалізує потужні серверні механізми машинного навчання для попереднього сканування повідомлень на шкідливі URL та неприпустимий контент, доповнені сповіщеннями про безпеку в зашифрованих чатах. Проте, через орієнтацію на аналіз метаданих у зашифрованих бесідах та високу ступінь автоматизації іноді трапляються помилкові блокування, що вимагає подальшого вдосконалення алгоритмів для кращого балансу між безпекою та зручністю користувачів .

Signal:

Віддає пріоритет приватності, тому його інструменти відрізняються від інших платформ.

- Message Requests і CAPTCHA: Нові відправники потрапляють у розділ “Запит на повідомлення” доки їх не схвалить отримувач. При ознаках масової розсилки система може вимагати CAPTCHA перед надсиланням [12];
- Анонімні репорти. Кнопка “Заблокувати й повідомити про спам” передає лише номер відправника й анонімний хеш повідомлення, що дає змогу реагувати на масові скарги, не розкриваючи текст на сервері

Через принципи повного шифрування Signal не може впровадити жодних автоматичних фільтрів токсичності чи семантичного аналізу повідомлень. Відтак ефективність боротьби з образливим чи ворожим контентом залежить виключно від поведінкових сигналів і активності кінцевих користувачів.

WhatsApp:

WhatsApp поєднує інфраструктуру Meta з власними клієнтськими механізмами безпеки.

- Превентивне блокування акаунтів: Спам-акаунти автоматично обмежуються або видаляються до надсилання перших повідомлень [13];
- Сповіщення про підозрілі посилання: Якщо URL містить нетипові комбінації символів, WhatsApp маркує його як “Підозрілий” та показує попередження перед переходом;
- Аналітика обсягу та швидкості: Алгоритми враховують шаблоність текстів і швидкість надсилання, що дозволяє виявляти бот-мережі.

WhatsApp використовує підхід без розшифровки вмісту повідомлень, аналізуючи метадані реєстрації та поведінкові сигнали для превентивного блокування спамерських акаунтів та координації масових розсилок. Функція Block Unknown Account Messages додатково обмежує надходження великої кількості повідомлень від невідомих контактів, яку можна активувати в налаштуваннях приватності. На рівні клієнта WhatsApp відображає індикатор “Підозріле посилання” для URL із нетиповими комбінаціями символів, що допомагає попереджати користувача про

потенційні фішингові посилання Через повне шифрування WhatsApp не сканує текст чи зображення на предмет токсичності або образливих висловлювань. Виявлення таких форм шкідливого контенту можливе лише через звіти користувачів або за допомогою зовнішніх NLP-рішень і ботів.

Discord:

Discord пропонує одні з просунутіших інструментів модерації для спільноти.

- AutoMod із шаблонами: Вбудований AutoMod включає готові фільтри (“мовна ворожнеча”, “сексуальний контент”, “зловмисні посилання”), які можна активувати одним кліком [14];
- Користувацькі ключові слова та дії: Адміністратори створюють власні списки слів і вирішують, блокувати повідомлення, давати тайм-аут або сигналізувати модераторам;
- Реактивна аналітика рейдів: При масовому приєднанні користувачів система автоматично підвищує рівень модерації й застосовує “рейд режим” [15].

Загалом, Discord забезпечує найширший набір вбудованих засобів для виявлення спаму та токсичності, але для досягнення оптимальної ефективності модерації користувачам доведеться інвестувати час у конфігурування правил та моніторинг спрацьовувань.

З наведених вище прикладів можна з впевненістю сказати що сучасні месенджери ефективно борються зі спамом, але токсичність залишається “сірою зоною”. Тому поки що користувачам доводиться покладатися на власну обережність та сторонні інструменти.

1.4 Визначення ролі методів штучного інтелекту у функціонуванні сучасних месенджерів

Штучний інтелект (ШІ) вже став невід’ємною частиною сучасних месенджерів, трансформуючи їх із простих засобів комунікації у розумні платформи з передовими функціями безпеки, персоналізації та автоматизації. Його роль виходить за межі

базового аналізу даних – ШІ формує ядро системи, що забезпечують ефективність, зручність і захист мільярдів користувачів.

Однією з ключових сфер впливу ШІ є боротьба зі спамом та шкідливим контентом. Завдяки машинному навчанню алгоритми аналізують метадані, такі як частота повідомлень, геолокація або IP-адреси, щоб виявляти бот-мережі. Наприклад, WhatsApp щодня автоматично блокує понад 2.5 мільйонів акаунтів, які демонструють підозрілу активність. Комп'ютерний зір дозволяє розпізнавати неприйнятні зображення та відео: Telegram використовує нейромережі для виявлення екстремістських матеріалів навіть у зашифрованих чатах. Discord, у свою чергу, інтегрує динамічні фільтри AutoMod, які адаптують правила модерації, враховуючи історію порушень у конкретній спільноті. Ці технології не лише зменшують навантаження на модераторів, але й запобігають масовому поширенню шкідливих матеріалів.

Другим важливим напрямком є персоналізація користувацького досвіду. Завдяки NLP чат-боти, такі як ті, що використовуються у Facebook Messenger, розуміють складні запити користувачів і надають релевантні відповіді. Трансформери на кшталт GPT-3 або BERT дозволяють системам аналізувати контекст повідомлень і пропонувати відповідні стікери чи емодзі, як це реалізовано в Telegram та Viber. Аналіз настроїв (Sentiment Analysis) допомагає платформам на кшталт Slack оцінювати емоційний стан користувачів, що особливо корисне для корпоративних команд, які прагнуть покращити комунікацію.

Також одним із перспективних напрямків є автоматизація та оптимізація рутинних процесів дедалі активніше покладається на генеративний ШІ: інтеграції ChatGPT у Discord чи Slack генерують звіти, перекладають фрагменти тексту або навіть підтримують розмову від імені користувача. Прогностичні моделі оцінюють ризики фішингу чи шахрайства (Signal аналізує аномалії у запитах на зв'язок) і в разі потреби попереджають користувача про загрозу ще до її реалізації.

Месенджери стають інклюзивними завдяки ШІ. Skype Translator використовує рекурентні нейромережі для перекладу мов у реальному часі, руйнуючи мовні бар'єри. WhatsApp тестує функцію голосового транскрибування, яка конвертує аудіо

у текст для користувачів із порушеннями слуху, також схожі функції є у Telegram та Viber в преміум підписках. Такі інновації роблять цифрове спілкування доступним для мільйонів людей з різними потребами.

Технологічне підґрунтя цих функцій складають трансформери для глибокої роботи з текстом, згорткові нейромережі для аналізу зображень і відео, а також федеративне навчання для он-девайс обробки даних без відправлення персональної інформації на сервер. Алгоритми кластеризації дозволяють виявляти великі спам-кампанії за схожими шаблонами поведінки мільйонів користувачів.

Разом з тим впровадження ШІ створює низку викликів. Конфлікт між приватністю та безпекою перш за все проявляється в end-to-end шифруванні, яке обмежує центральний аналіз повідомлень і змушує переходити до он-девайс рішень або автономних ботів. Культурна упередженість моделей – наприклад, недостатнє розпізнавання місцевих сленгових виразів – може призводити до ігнорування частини токсичних повідомлень. А помилкові спрацьовування (у 2023 році Meta визнала до 15 % фальшивих блокувань у Messenger) потребують тоншого налаштування порогів та механізмів апеляції.

Перспективи розвитку галузі пов’язані з мультимодальними ШІ-системами, які поєднують аналіз тексту, аудіо та відео для глибшого розуміння контексту (наприклад, для виявлення сарказму за інтонаціями), децентралізованими моделями на базі блокчейну для збереження конфіденційності, а також запровадженням етичних стандартів прозорості.

Отже, ШІ перетворив месенджери на інтелектуальні екосистеми, але їх розвиток вимагає подолання технічних та етичних бар’єрів. Ключ до успіху – гнучкий підхід, де інновації поєднуються з повагою до приватності та культурного різноманіття. Наприклад, впровадження он-девайс мультимодальних моделей може стати компромісом між безпекою та конфіденційністю, а співпраця з локальними експертами допоможе подолати мовні упередження. Майбутнє месенджерів – це платформи, де ШІ не лише фільтрує загрози, але й покращує комунікацію, роблячи її безпечнішою та інклюзивною для мільярдів користувачів.

1.5 Постановка завдання розробки месенджера з інтеграцією AI для виявлення спаму та токсичних повідомлень у чатах

Данна дипломна робота ставить завдання створити сучасний клієнт-серверний месенджер, здатний не лише забезпечувати надійний і зручний обмін текстовими повідомленнями в режимі реального часу, а й автоматично ідентифікувати спам-розсилки і токсичні висловлювання за допомогою AI-модулів. Це дозволить підвищити якість комунікації, захистити користувачів від небажаного контенту та покращити загальне враження від використання додатку.

Для досягнення даної мети, необхідно сформувати та вирішити наступні задачі:

- Дослідити популярні месенджери з вбудованими системами модерації контенту;
- Оцінити сучасні AI-бібліотеки та сервіси для класифікації тексту;
- Визначити критерії якості роботи AI-модуля (точність, швидкодія, рівень помилкових спрацювань);
- Розробити загальну блок-діаграму клієнт-серверної взаємодії;
- Створення API-інтерфейсу для взаємодії основного серверу з AI-сервером;
- Створення UI-компонентів, що відображають статус повідомлення (підозріле/безпечне);
- Проведення функціональних тестів всіх модулів на відповідність поставленим вимогам.

Таким чином виконання цих задач забезпечить побудову повнофункціонального месенджера з інтегрованим AI-фільтром, що гарантує швидке й точне виявлення небажаного контенту та створить комфортне середовище для спілкування користувачів.

2. РОЗРОБКА МАТЕМАТИЧНИХ МОДЕЛЕЙ І АЛГОРИТМІВ ФУНКЦІОНУВАННЯ МЕСЕНДЖЕРА З ІНТЕГРАЦІЄЮ AI

2.1 Розробка математичних моделей семантичної оцінки токсичних повідомлень

Проаналізувавши основні сучасні підходи до вирішення задачі семантичної оцінки токсичних повідомлень, можна виділити три основні напрями:

- використання готових хмарних сервісів для аналізу тексту;
- застосування відкритих або ліцензійних моделей штучного інтелекту, які розгортаються та запускаються безпосередньо на власних серверах;
- комбінування власних простих правил (blacklist та keyword-фільтрація) із сучасними моделями машинного навчання.

До найбільш популярних хмарних сервісів належать Perspective API (Google), Azure Content Moderator (Microsoft), Amazon Comprehend (AWS) та інші спеціалізовані API для модерації тексту (наприклад, DeepAI Moderation API, OpenAI Moderation API). Такі сервіси забезпечують швидку інтеграцію, автоматичне масштабування, а також постійне оновлення моделей, однак мають ряд обмежень – наприклад, ліміти запитів, залежність від стабільності зовнішнього сервісу, а також недостатню підтримку для мало розповсюджених мов. Альтернативний підхід – тренування та розгортання власної моделі на основі відкритих моделей (наприклад, Detoxify, HateBERT, RuBERT, українськомовні BERT-моделі) із подальшим розміщенням на власних ресурсах. Це дозволяє краще адаптувати фільтр під конкретну задачу, контролювати доступність сервісу, уникнути передачі конфіденційних даних зовнішнім провайдерам, але вимагає суттєвих обчислювальних ресурсів і компетенцій у сфері NLP та MLOps.

У даному рішенні для семантичної оцінки токсичних повідомлень було обрано Perspective API – хмарний сервіс, який надається компанією Google Jigsaw. Perspective API аналізує текстові повідомлення й повертає оцінки різних атрибутів, таких як токсичність (TOXICITY), образливість (INSULT), ненависницькі висловлювання (IDENTITY_ATTACK), спам (SPAM), ненормативна лексика (PROFANITY) тощо.

Кожен атрибут визначається як ймовірність (від 0 до 1), що текст відповідає цій характеристиці.

Вибір Perspective API обумовлено рядом факторів:

- висока точність аналізу англомовних текстів завдяки використанню сучасних моделей;
- можливість перевірки кількох негативних атрибутів одним запитом;
- простота інтеграції через REST API та наявність клієнтських бібліотек;
- постійна підтримка та оновлення сервісу з боку Google.

Інтеграція Perspective API у систему реалізована за класичною схемою “клієнт-сервер”, що забезпечує централізований контроль і обробку користувацьких повідомлень із подальшим застосуванням сервісу машинного аналізу тексту. Такий підхід дозволяє не лише автоматизувати перевірку контенту на токсичність і небажані прояви, а й забезпечити масштабованість і гнучкість у подальшій розробці системи. Процес обробки повідомлень складається з кількох основних етапів:

- Надсилання повідомлення користувачем: Кінцевий користувач створює текстове повідомлення через клієнтське застосування і надсилає його на сервер. На цьому етапі здійснюється базова валідація вмісту;
- Обробка повідомлення на сервері: Сервер отримує повідомлення й формує структурований запит до зовнішнього сервісу Perspective API, вказуючи, які саме атрибути необхідно проаналізувати. Запит містить сам текст повідомлення та технічні параметри, необхідні для обробки;
- Аналіз тексту через Perspective API: Perspective API, розташований у хмарі Google Jigsaw, отримує запит і проводить машинний аналіз вмісту повідомлення. В результаті API повертає набір числових оцінок для кожного з атрибутів – наприклад, рівень токсичності, ймовірність спаму, ознаки образливості тощо. Кожен показник виражається у вигляді числа від 0 до 1, що відображає ймовірність відповідної характеристики у тексті;
- Оцінка результатів і прийняття рішення сервером: Сервер отримує результати аналізу й порівнює їх із заздалегідь визначеними пороговими

значеннями. Наприклад, якщо рівень токсичності перевищує встановлений поріг, повідомлення вважається неприйнятним для публікації й відмічається як потенційно небезпечне;

- Збереження повідомлення та інформування користувача: Після перевірки повідомлення зберігається у базі даних (разом із результатами аналізу та службовою інформацією). Сервер надсилає повідомлення всім користувачам чату.

Такий підхід дозволяє централізовано контролювати якість і безпеку контенту у системі, оперативно реагувати на потенційно шкідливі чи образливі повідомлення, а також акумулювати статистичну інформацію для подальшого аналізу та вдосконалення алгоритмів модерації. Детальніше із послідовністю інтеграції компонентів можна ознайомитись, переглянувши рисунок 2.1.

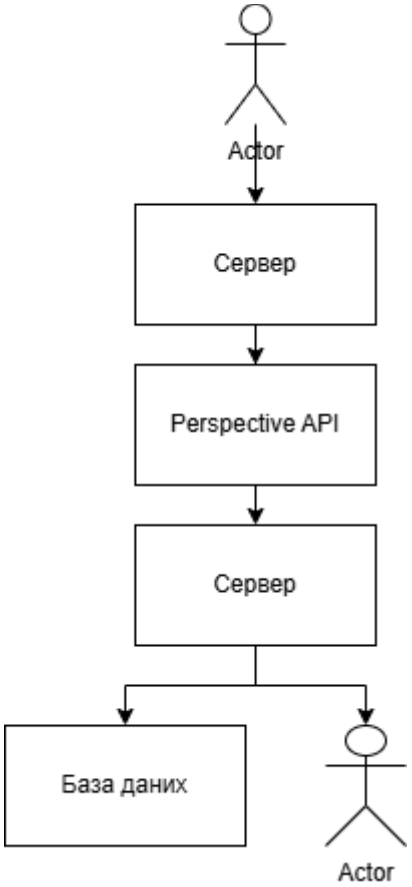

Рисунок 2.1 – Діаграма інтеграції

На основі цього аналізу для реалізації системи автоматизованої модерації повідомлень обрано інтеграцію із хмарним сервісом Perspective API, який забезпечує високу точність, зручність у використанні та підтримку сучасних моделей для

виявлення токсичності та інших небажаних атрибутів тексту. Представлена архітектура інтеграції дозволяє ефективно здійснювати фільтрацію контенту, оперативно реагувати на порушення й накопичувати релевантну статистику для подальшого вдосконалення системи. Такий підхід поєднує гнучкість масштабування із простотою впровадження й відповідає сучасним вимогам до якості онлайн-комунікацій.

2.2 Розробка структурної моделі програмного комплексу месенджера з інтеграцією штучного інтелекту

Для забезпечення надійної, безпечної та зручної комунікації між користувачами сучасних месенджерів дедалі важливішим стає впровадження механізмів автоматизованої модерації повідомлень, зокрема із застосуванням засобів штучного інтелекту. Структурна модель програмного комплексу месенджера з інтеграцією AI передбачає розподіл функціональності між кількома основними компонентами, кожен з яких виконує свою роль у загальному процесі обробки та модерації текстових повідомлень. Такий підхід дозволяє ефективно масштабувати систему, розмежовувати відповідальність між модулями, спрощувати супровід та розширення функціональності програмного комплексу.

Основні структурні компоненти програмного комплексу месенджера:

- Клієнтське застосування – відповідає за взаємодію з користувачем, реєстрацію, авторизацію, перегляд та надсилання повідомлень, відображення стану чатів у реальному часі;
- Серверна частина – реалізує бізнес-логіку месенджера, забезпечує передачу повідомлень між користувачами, перевірку повідомлень на токсичність та зберігання даних;
- База даних – містить інформацію про користувачів (email, пароль, uuid, логін), чати (id чату, учасники), а також повідомлення, які структуровано зберігаються окремо для кожного чату;
- Модуль інтеграції з Perspective API – здійснює автоматизовану перевірку повідомлень на токсичність шляхом взаємодії з зовнішнім сервісом AI;

- Зовнішній AI-сервіс – аналізує текст повідомлень на предмет токсичності, спаму, образливості та інших негативних характеристик, повертаючи серверу числову оцінку кожного з параметрів.

Взаємодія основних компонентів месенджера відбувається наступним чином. Усі дії користувача – такі як реєстрація, авторизація, надсилання чи отримання повідомлень – здійснюються через клієнтське застосування. Взаємодія між клієнтською та серверною частинами організована за допомогою WebSocket-з'єднання, для чого використовується бібліотека Socket.IO. Коли користувач надсилає повідомлення, клієнт передає його на сервер, де перед подальшою доставкою іншим учасникам чату відбувається попередня перевірка вмісту.

На сервері впроваджено модуль інтеграції зі штучним інтелектом: відповідна функція у коді передає текст повідомлення на аналіз до зовнішнього AI-сервісу – Perspective API. Perspective API здійснює машинний аналіз вмісту, визначаючи ймовірність наявності токсичності, спаму чи інших небажаних ознак, та повертає серверу числові оцінки відповідних атрибутів. Сервер порівнює отримані результати з встановленими пороговими значеннями й приймає рішення щодо подальшої долі повідомлення.

База даних Firebase у цій архітектурі використовується для зберігання облікових записів користувачів, структури чатів, а також усіх дозволених повідомлень із результатами їхньої перевірки. Такий підхід забезпечує централізований, автоматизований контроль якості та безпеки контенту в системі. Для детальнішого ознайомлення з структурою дивіться рисунок 2.2.

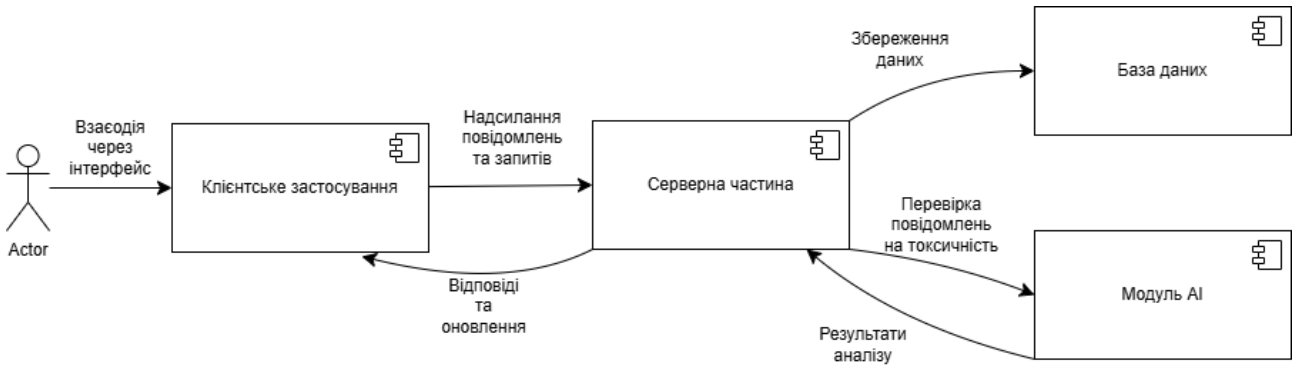

Рисунок 2.2 – Структурна діаграма компонентів

Запропонована структурна модель програмного комплексу забезпечує реалізацію автоматизованої модерації текстових повідомлень. Це підвищує рівень безпеки та якості комунікації між користувачами. Завдяки чіткому розподілу функціональних компонентів, зокрема виділенню модуля інтеграції зі штучним інтелектом, система є масштабованою, гнучкою та зручною в супроводі. Така архітектура спрощує впровадження нових функцій, адаптацію до сучасних вимог та гарантує захист від небажаного контенту в онлайн-чатах.

2.3 Розробка функціональної моделі роботи месенджера для виявлення спаму та токсичних повідомлень у чатах

Функціональна модель месенджера визначає базові сценарії використання системи, ключові операції та послідовність дій користувача під час взаємодії з програмним комплексом. Вона служить основою для проектування архітектури, тестування логіки та подальшого масштабування. Модель інтегрує як стандартні функції спілкування, так і сучасні механізми автоматизації, такі як модерація контенту.

На рисунку 2.3 представлено діаграму випадків використання, яка охоплює основні функціональні можливості, доступні користувачеві:

- Реєстрація та автентифікація: Користувач вводить email і пароль, система перевіряє їхню валідність та унікальність. Після успішної реєстрації генерується унікальний ідентифікатор і зберігається в базі даних. Для автентифікації використовується JWT-токен, що забезпечує безпеку сесії;
- Створення та перегляд чатів: Користувач може створити приватний чат з будь-яким знайденим користувачем;
- Надсилання та отримання повідомлень: Повідомлення надсилається через WebSocket-з’єднання що забезпечує миттєву доставку;
- Пошук користувачів та чатів: Алгоритм пошуку дозволяє швидко знаходити користувачів за частково введеним іменем або email.;
- Перегляд історії повідомлень: Користувач може переглядати історію будь-якого чату в якому брав участь.

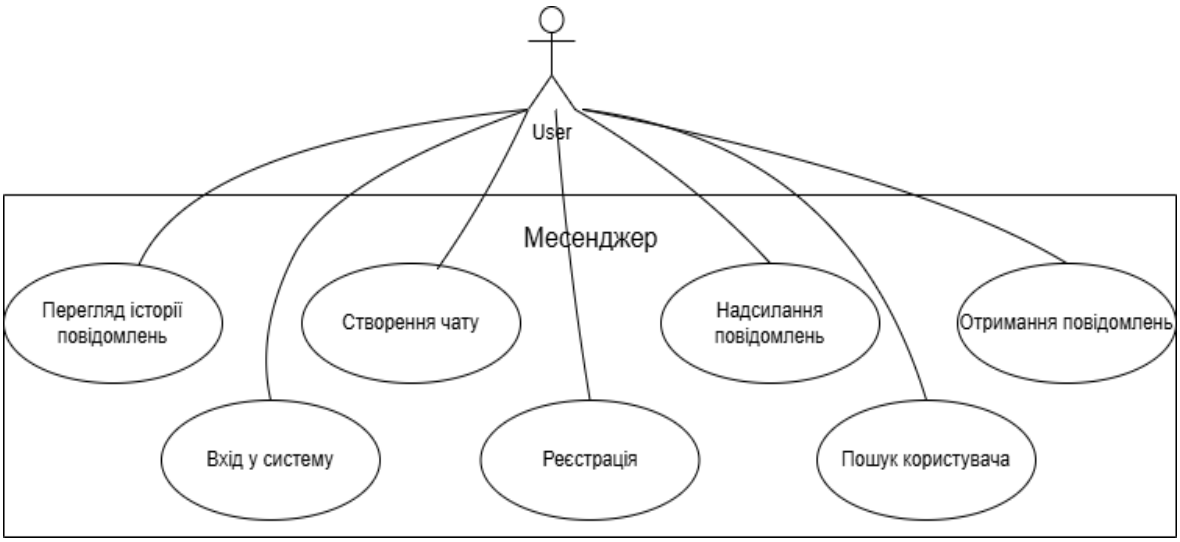

Рисунок 2.3 – Функціональна модель взаємодії користувача з месенджером

Рисунок 2.4 ілюструє функціональну схему програмного комплексу месенджера у вигляді діаграми діяльності (Activity Diagram). На схемі детально описано основні кроки користувача:

- Початок роботи з додатком;
- Проходження процедури авторизації/реєстрації;
- Взаємодія з головним меню, що містить список чатів;
- Можливість здійснення пошуку чату чи користувача;
- Вибір чату та подальше надсилання повідомлення.

У разі спроби пошуку чату або користувача, система перевіряє наявність відповідного об’єкта. Якщо чат або користувача не знайдено, користувач отримує відповідне повідомлення та повертається до головного меню.

В іншому випадку, після вибору чату, користувач може ввести та надіслати повідомлення, яке передається на сервер для подальшої обробки, зокрема для перевірки на наявність спаму чи токсичного контенту.

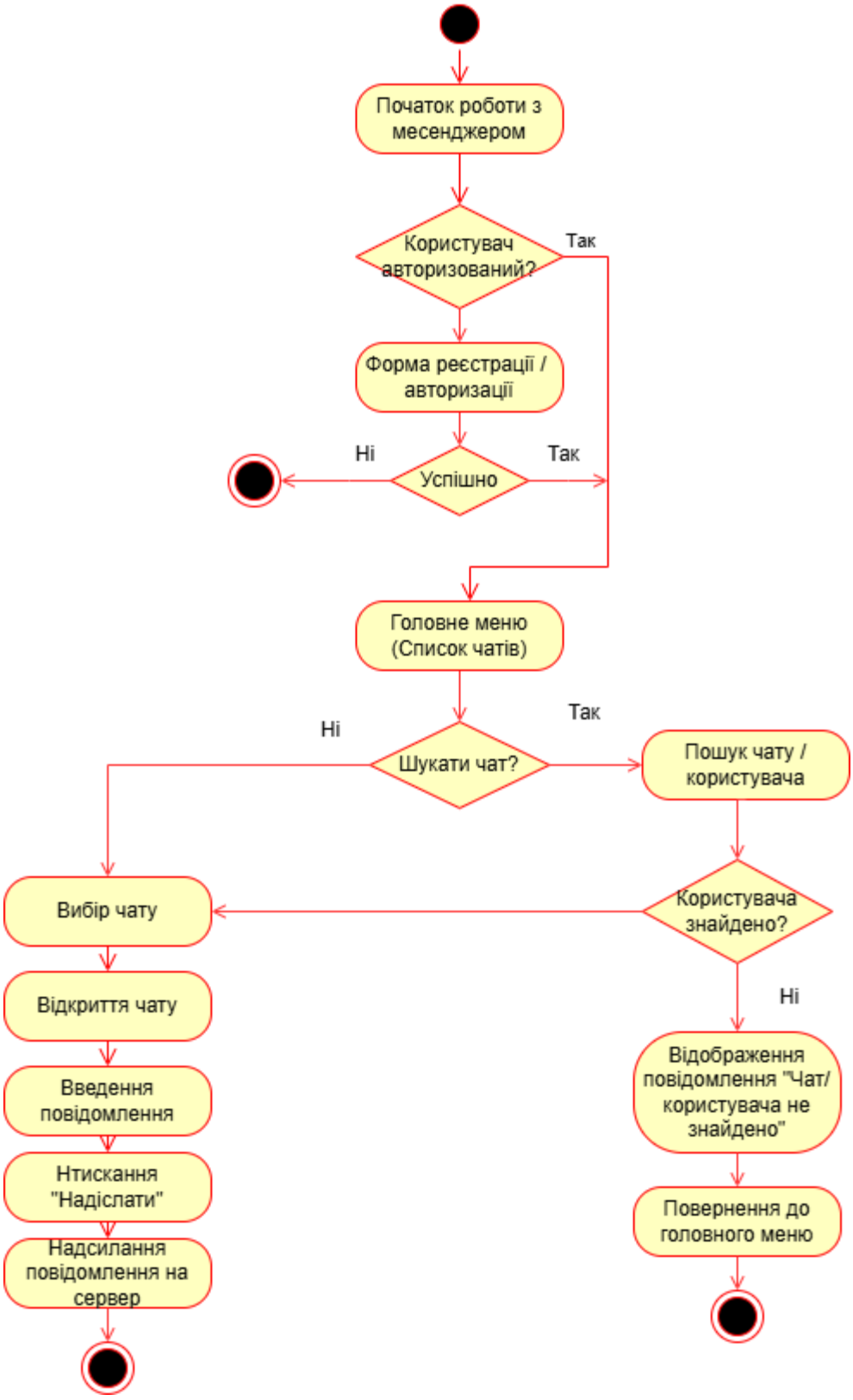

Рисунок 2.4 – Функціональна схема програмного комплексу

Запропонована функціональна модель дозволяє чітко структурувати основні бізнес-процеси системи, визначити місця інтеграції модулів автоматизованої модерації, а також забезпечує необхідний рівень гнучкості для впровадження

механізмів виявлення спаму та токсичних повідомлень, що є критично важливим для сучасних месенджерів. Вона слугує основою для побудови архітектури програмного комплексу, реалізації контролю якості контенту, а також створює підґрунтя для подальшого масштабування й розширення функціональних можливостей системи.

2.4 Розробка схеми бази знань програмної системи месенджера

У традиційних системах автоматизованої модерації з використанням штучного інтелекту база знань формується як комплексний внутрішній ресурс. Вона зазвичай включає лінгвістичні словники (набір ключових слів, токенів, мовних шаблонів), набір правил (алгоритми класифікації, фільтрації контенту), кейси (архів анотованих прикладів токсичних повідомлень, дискусій) і статистичні моделі, навчені на історичних даних для прогнозування порушень. Така система потребує постійного оновлення та навчання, що пов’язано зі значними витратами часу та обчислювальних ресурсів.

У запропонованій архітектурі програмної системи месенджера основна увага приділяється організації та зберіганню результатів аналізу кожного повідомлення. Кожен текст, що надходить від користувача, після проходження через Perspective API, зберігається у базі даних разом із відповідними мітками. Таким чином, база знань у цій системі виконує роль реєстру стану кожного повідомлення та його атрибутів без необхідності підтримки окремих словників або правил.

Всі дані структуруються у вигляді окремих записів у Firebase, де для кожного повідомлення додатково фіксується результат автоматичної AI-модерації. Основними полями таких записів є: ідентифікатор відправника, текст повідомлення, мітка часу, а також статус повідомлення, який базується на оцінках від Perspective API. Відповідно до отриманих результатів аналізу, повідомлення може бути або відразу відображене у чаті, або автоматично замасковане для інших учасників розмови з можливістю подальшого перегляду за бажанням користувача.

На рисунку 2.5 представлено блок-схему процесу обробки повідомлення, яка ілюструє основні етапи перевірки, збереження й подальшої роботи з контентом. Після того, як користувач надсилає повідомлення, система відправляє його на перевірку

через Perspective API. Далі отримані результати використовуються для додавання відповідних міток у базу даних Firebase. Наступним кроком є розсилання повідомлення всім користувачам чату. В залежності від того, чи має повідомлення мітку “Blocked”, воно або одразу відображається у чаті, або відображається з маскуванням вмісту (рис. 2.5).

Рисунок 2.5 – Схема обробки повідомлення

Структуру зв’язків між основними сутностями бази даних представлено на ER-діаграмі (рисунок 2.6). Система складається з трьох основних сутностей: chats (чати), users (користувачі) та messages (повідомлення). Кожен чат містить список користувачів, кожен користувач може брати участь у багатьох чатах, а також надсилати багато повідомлень. Повідомлення містить інформацію про відправника, час надсилання, статус та текст. Таке моделювання забезпечує просте масштабування

системи, а також зручний доступ до історії переписки, даних для аналітики та модерації.

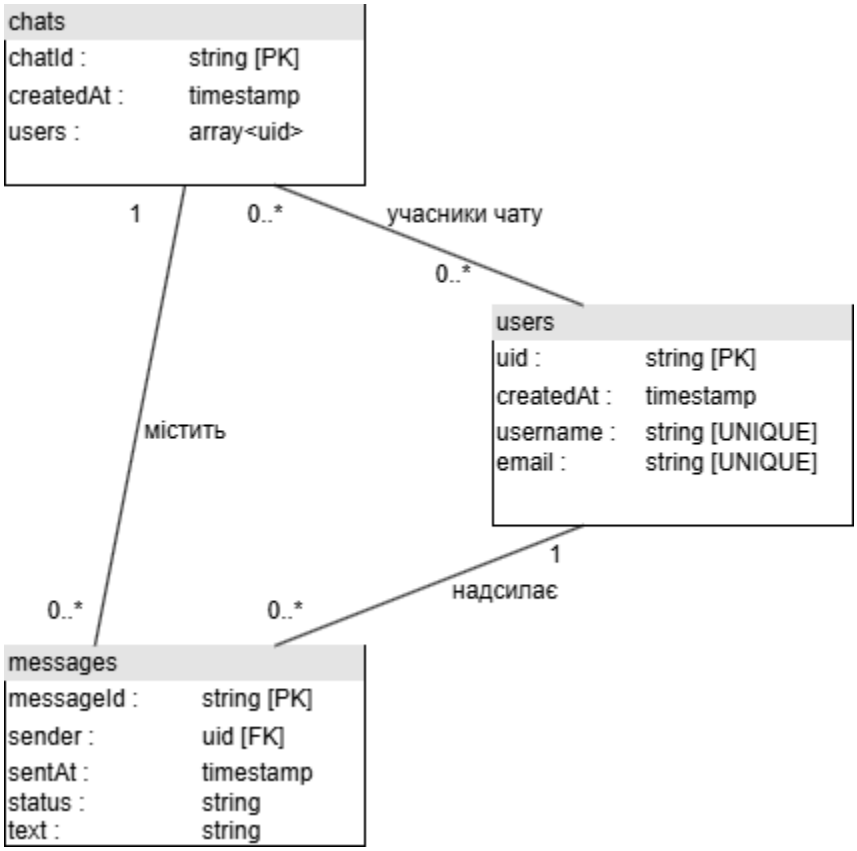

Рисунок 2.6 – ER-діаграма зв’язків між основними сутностями бази даних

Завдяки такій організації база знань є максимально гнучкою: у разі зміни вимог або розширення функціональності система може легко адаптуватися до нових типів атрибутів чи додаткових полів, які повертає Perspective API. Водночас зберігається можливість оперативного пошуку, аналітики й подальшого удосконалення політики модерації.

2.5 Розробка блок-схем алгоритмів програмних модулів месенджера

Для успішної реалізації клієнт-серверного месенджера, критично важливо чітко формалізувати логіку взаємодії основних програмних компонентів системи. Блок-схеми дозволяють наочно представити алгоритми роботи ключових модулів – таких як обробка та надсилання повідомлень, взаємодія з сервером штучного інтелекту, збереження даних та управління користувачами. Особливу увагу в даному розділі

приділено побудові алгоритмів, які забезпечують автоматичне виявлення та обробку спаму й токсичних повідомлень, інтеграції із зовнішнім AI-сервісом, а також реалізації механізмів маркування потенційно небезпечного контенту. Деталізація цих процесів у вигляді блок-схем дозволяє уникнути неоднозначностей у розробці, підвищити прозорість архітектури програмного забезпечення і закладає основу для подальшого тестування й масштабування системи.

Для початку користування додатком користувачу необхідно пройти процедуру реєстрації відповідно до алгоритму, наведеного на рисунку 2.7.

Рисунок 2.7 – Алгоритм реєстрації користувача

Після успішної реєстрації та авторизації користувача в нього з’являється можливість відкриття чату з іншим користувачем. Логіка цього процесу наведена на рисунку 2.8. При натисканні на іконку співрозмовника система здійснює перевірку в базі даних: якщо чат між цими користувачами вже існує, відбувається завантаження його ідентифікатора та історії повідомлень. У випадку, якщо спільний чат відсутній, створюється новий запис і формується його ідентифікатор. Далі система намагається отримати історію чату: якщо повідомлення вже є – вони відображаються користувачу, якщо історія відсутня – виводиться повідомлення “Поки немає повідомлень”.

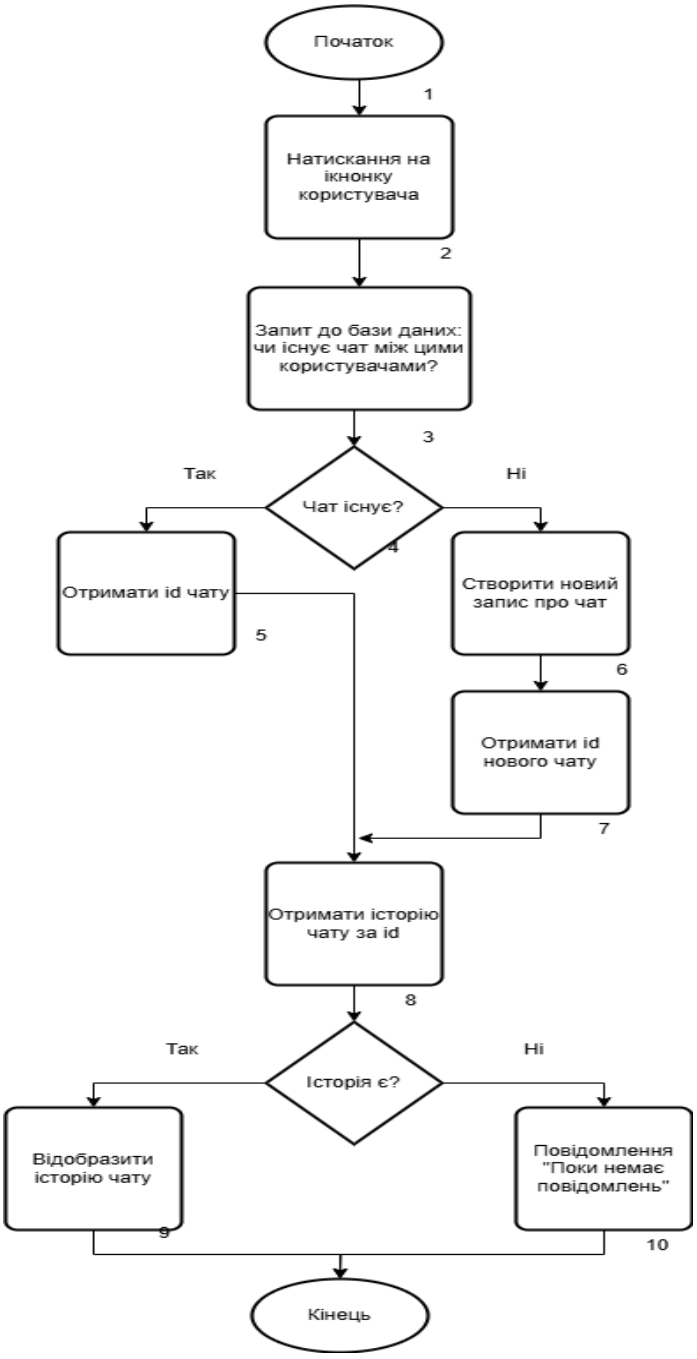

Рисунок 2.8 – Алгоритм відкриття чату

Після відкриття чату користувач можу надіслати повідомлення. З логікою цього процесу можна ознайомитись переглянувши рисунку 2.9.

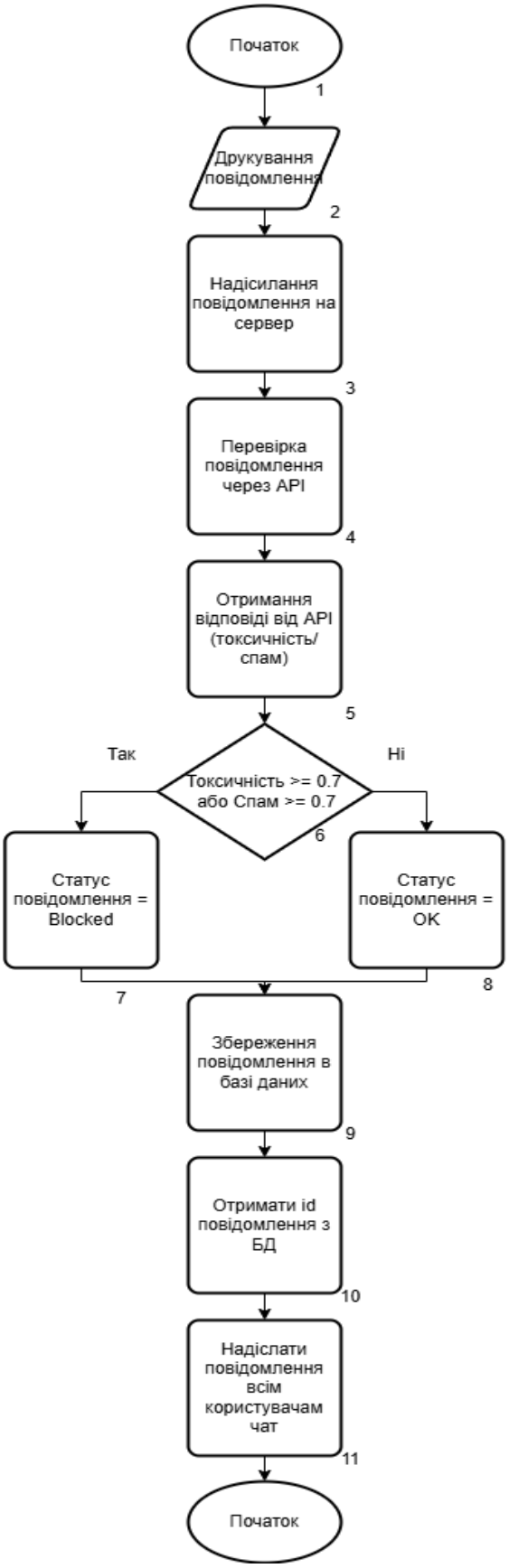

Рисунок 2.9 – Алгоритм надсилання повідомлень

Представлені алгоритми є одними з основних для роботи додатку. Їх формалізація забезпечує надійність, прозорість та ефективність основних бізнес-процесів месенджера, а також закладає фундамент для подальшого розвитку й масштабування системи.

2.6 Розробка моделі візуального інтерфейсу месенджера

Графічний інтерфейс додатку складається з мінімально необхідної кількості вікон та елементів, що робить його інтуїтивно зрозумілим та зручним для користувача. Початковим вікном застосунку є вікно авторизації, яке призначене для входу користувача до системи. Дане вікно містить декілька основних компонентів:

- Кнопка перемикання теми, що дозволяє користувачеві змінювати вигляд інтерфейсу з темної на світлу тему або навпаки, забезпечуючи додатковий комфорт при роботі;
- Поле для введення email – тут користувач має ввести свою електронну адресу, зареєстровану у системі;
- Поле для введення паролю, яке використовується для введення захищеного паролю користувача;
- Кнопка “Увійти” – призначена для підтвердження введених даних та спроби авторизації у додатку;
- Текст з гіперпосиланням на сторінку реєстрації, який дає змогу новим користувачам швидко перейти до створення облікового запису у разі відсутності облікового запису.

Усі елементи розташовані у центральній частині екрану, що дозволяє максимально сфокусувати увагу користувача на виконанні основної дії – вході до системи. Ознайомитись із зовнішнім виглядом даного вікна можна на рисунку 2.10.

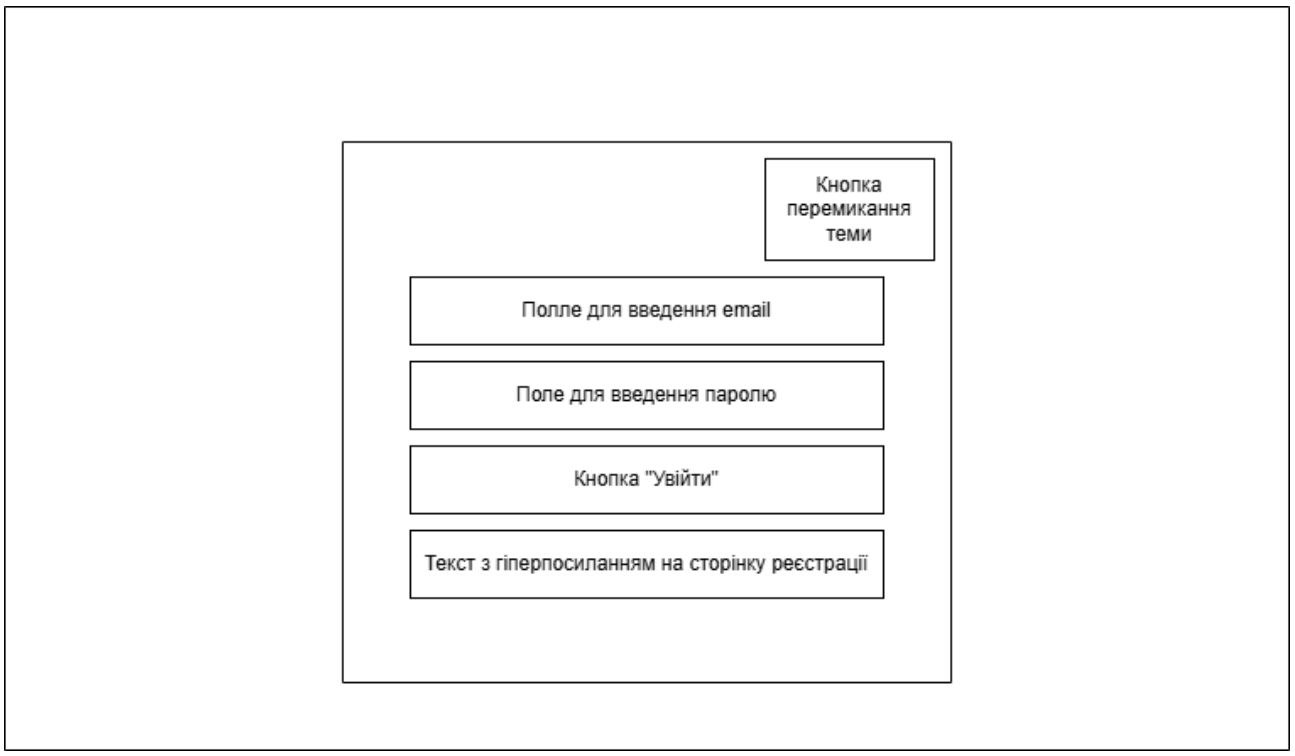

Рисунок 2.10 – Сторінка авторизації

Сторінка реєстрації передбачає створення нового облікового запису користувача і майже не відрізняється від сторінки авторизації. Дане вікно містить усі необхідні для цієї процедури елементи:

- Кнопка перемикання теми – дозволяє обирати між темною і світлою темою інтерфейсу;
- Поле для введення username – тут користувач задає бажане ім’я користувача (нікнейм), яке буде відображатися у чатах;
- Поле для введення email, що призначене для введення електронної пошти, яка буде використовуватись для входу та відновлення паролю;
- Поле для введення паролю – поле для створення паролю для захисту облікового запису;
- Кнопка “Зареєструватись” – підтверджує введені дані і надсилає запит на створення нового облікового запису;
- Текст з гіперпосиланням на сторінку авторизації – якщо користувач уже має обліковий запис, він може швидко повернутися на екран входу до системи.

Всі елементи логічно згруповані, що сприяє простоті та зрозумілості процесу реєстрації. Зовнішній вигляд вікна реєстрації наведено на рисунку 2.11.

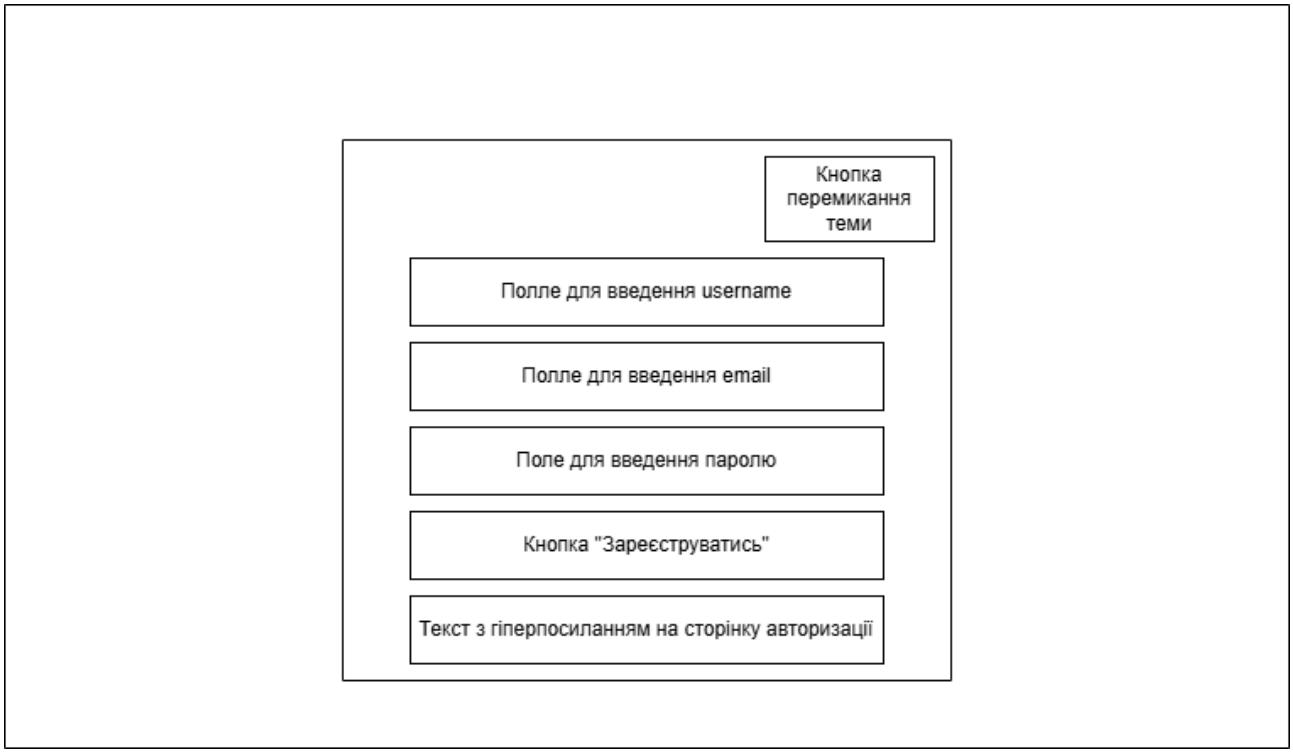

Рисунок 2.11 – Сторінка реєстрації

Після авторизації або реєстрації користувач потрапляє до головного вікна месенджера, яке містить такі основні блоки ліва бокова панель і центральна область.

Розглянемо компоненти лівої бокової панелі:

- Кнопка перемикання теми для зміни оформлення інтерфейсу;
- Кнопка виходу для завершення роботи з додатком та виходу з облікового запису;
- Список чатів або результати пошуку – тут відображаються всі активні чати користувача, а також результати пошуку за введеним запитом;
- Строка пошуку користувача – дозволяє швидко знаходити потрібних співрозмовників або створювати нові чати.

В свою чергу центральна область чату складається з:

- Верхньої панелі, яка містить username користувача, з яким наразі ведеться переписка, для зручності ідентифікації активного діалогу;

- Списку повідомлень, у якому відображаються всі вхідні та вихідні повідомлення у вибраному чаті. Всі повідомлення візуально відділені одне від одного для зручності сприйняття інформації;
- Поле для введення повідомлення, розташоване у нижній частині, через яке користувач може вводити та надсилати текстові повідомлення у чат.

Такий підхід до організації інтерфейсу дозволяє користувачеві зручно перемикатися між чатами, швидко знаходити потрібних співрозмовників, вести діалоги та налаштовувати вигляд додатку під свої вподобання. Відповідний макет інтерфейсу головного вікна месенджера зображено на рисунку 2.12.

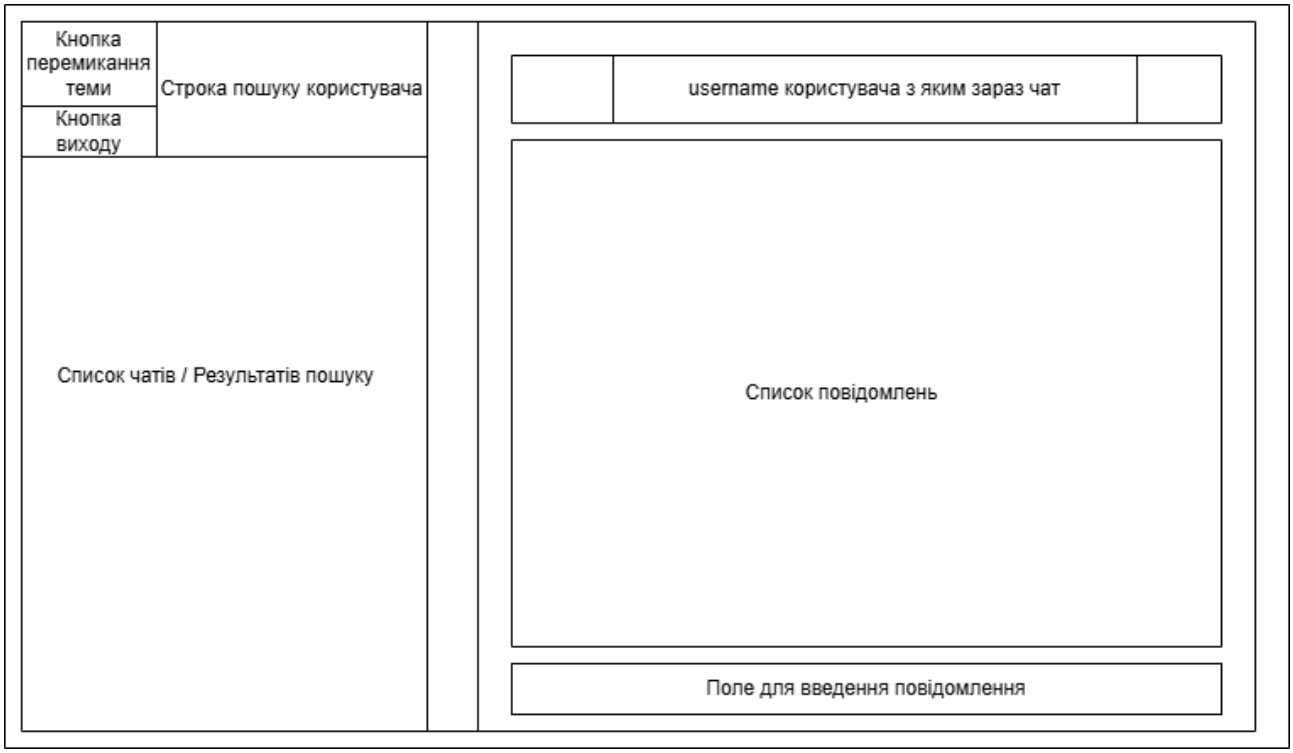

Рисунок 2.12 – Головна сторінка

У ході розробки ескізів графічного інтерфейсу додатку месенджера було дотримано принципів мінімалізму, зручності та інтуїтивності для кінцевого користувача. Запропоновані макети покривають усі основні сценарії роботи з додатком – авторизацію, реєстрацію та безпосереднє користування чатами.

Основною перевагою обраного підходу стало просте та зрозуміле компонування елементів, яке не перевантажує користувача зайвою інформацією. Чіткий поділ вікон на функціональні зони (авторизація, реєстрація, головний екран месенджера) дає змогу легко орієнтуватись у додатку навіть недосвідченим

користувачам. Кожен екран містить лише ті елементи, які необхідні для виконання відповідної дії, що значно спрощує навігацію та покращує загальний досвід взаємодії. Додаткова увага була приділена питанням безпеки: відсутність зайвих функцій у вікні авторизації та реєстрації відповідає сучасним вимогам захисту даних. Реалізація перемикання між світлою та темною темою дозволяє адаптувати інтерфейс до індивідуальних уподобань користувача, підвищуючи комфорт роботи з додатком у різних умовах освітлення.

Таким чином, створені ескізи слугують якісною основою для подальшої реалізації сучасного, функціонального та безпечного інтерфейсу месенджера, що відповідає як технічним, так і ергономічним вимогам.

3. РОЗРОБКА ПРОГРАМНИХ МОДУЛІВ МЕСЕНДЖЕРА З ІНТЕГРАЦІЄЮ AI

3.1 Розробка візуального інтерфейсу месенджера

Візуальний інтерфейс користувача є важливим компонентом програмного забезпечення, зокрема у месенджерах, де зручність та швидкість взаємодії безпосередньо впливають на якість користувацького досвіду. Для реалізації клієнтської частини застосунку було обрано фреймворк Angular, який забезпечує гнучку побудову інтерфейсу та підтримується компанією Google. Це рішення дозволяє створити логічну структуру елементів керування та адаптувати інтерфейс під потреби цільової аудиторії. Архітектура та функціонал Angular відповідають всім потребам для створення месенджера:

- Компонентний підхід

  Кожен елемент інтерфейсу (авторизація, список чатів, вікно повідомлення) реалізовано як ізольований компонент з власним HTML, CSS та логікою. Це забезпечує модульність і структурованість. Модульність відповідає за легке повторне використання компонентів, наприклад, аватар користувача у різних розділах. Структурованість уявляє собою чітку організацію коду для великих проєктів;

- TypeScript

  Сувора типізація зменшує кількість помилок на етапі розробки та спрощує підтримку;

- Двостороннє зв’язування даних

  Миттєва синхронізація інтерфейсу з даними, наприклад, оновлення списку чатів при новому повідомленні без перезавантаження;

- Інтегровані інструменти

  Серед інтегрованих інструментів варто виокремити роутинг, що забезпечує ефективну навігацію між розділами (чати/профіль/налаштування); реактивні форми, що відповідають за валідацію даних у реальному часі (реєстрація, налаштування); Angular

CLI, що являє собою автоматизацію створення компонентів, сервісів і розгортання. Angular також чудово підходить для командної розробки завдяки чіткій структурі проєкту та підтримці тестування. Таким чином, Angular став логічним вибором для реалізації клієнтської частини месенджера, дозволяючи досягти високого рівня продуктивності, гнучкості та підтримуваності інтерфейсу.

Відповідно до технічних вимог та макетів було створено такі ключові сторінки:

- Сторінка реєстрації;
- Сторінка авторизації;
- Головна сторінка.

Кожна з вищезазначених сторінок виконана в єдиному візуальному стилі, що ґрунтується на принципах мінімалізму та інтуїтивної взаємодії. Дизайн інтерфейсу побудовано таким чином, щоб користувач міг швидко зорієнтуватися навіть без попереднього досвіду використання подібних застосунків. Основна увага приділялася логічній структурі, простоті навігації та адаптивності до різних розмірів вікна застосунку.

Далі детально розглянемо кожну зі сторінок, її функціональне наповнення, ключові компоненти та особливості реалізації у середовищі Angular.

Сторінка авторизації

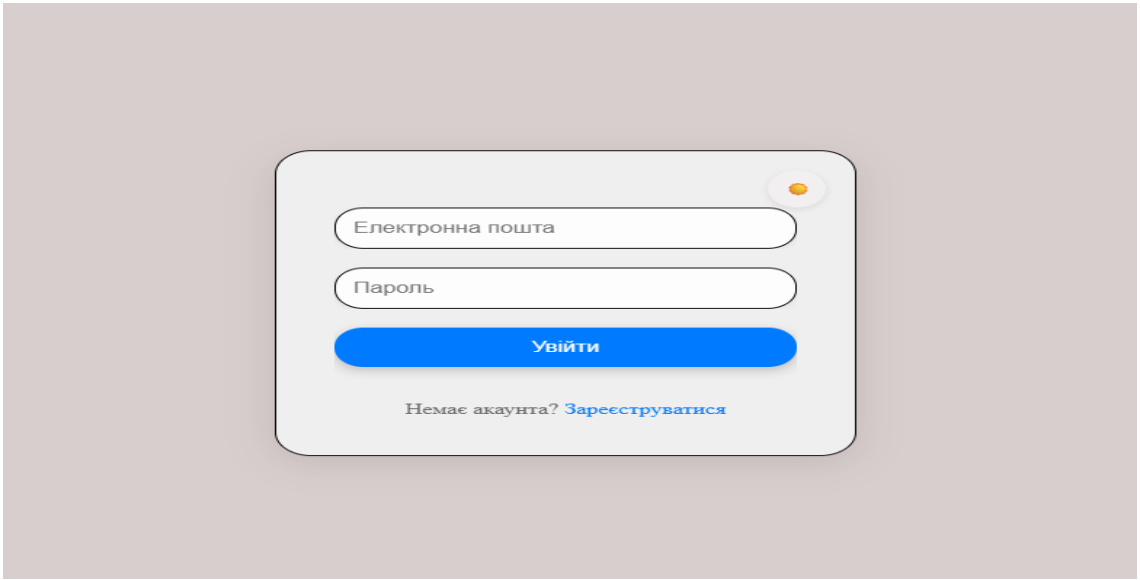

Рисунок 3.1 – Скріншот сторінки авторизації

Сторінка авторизації є першим екраном, з яким взаємодіє користувач після запуску застосунку. Вона призначена для входу до системи на основі облікових даних, збережених під час реєстрації. На рисунку 3.1 представлено зовнішній вигляд цієї сторінки.

```typescript
import { Component, inject } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { ThemeToggleComponent } from '../../common-ui/theme-toggle/theme-toggle.component';
import { AuthService } from '../../auth/auth.service';


@Component({
  selector: 'app-login-page',
  imports: [
    ReactiveFormsModule,
    ThemeToggleComponent
  ],
  templateUrl: './login-page.component.html',
  styleUrl: './login-page.component.css'
})

export class LoginPageComponent {

  router = inject(Router)

  goToRegister() {
    this.router.navigate(['/registration']);
  }

  authService = inject(AuthService)

  authForm = new FormGroup ({
    email: new FormControl('', {
      validators: [Validators.required, Validators.email],
      nonNullable: true
    }),
    password: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    })
  })

  onSubmit() {
    if (!this.authForm.valid) return;

    const { email, password } = this.authForm.value as {
      email: string;
      password: string;
    };

    this.authService.login(email, password).subscribe({
      next: () => this.router.navigate(['/']),
      error: err => console.error('Login error', err)
    });
  }
}
```

Рисунок 3.2 – TypeScript код компонента LoginPageComponent

Інтерфейс сторінки авторизації реалізовано з використанням компонентного підходу Angular. Вся логіка зосереджена у компоненті LoginPageComponent, який

відповідає за ініціалізацію форми, валідацію введених даних, обробку подій входу та навігацію на сторінку реєстрації. Для ознайомлення з реалізацією цього компонента див. рисунок 3.2.

HTML-шаблон форми авторизації реалізовано наступним чином (див. рис. 3.3).

```
<div class="box">
  <app-theme-toggle class="theme-toggle-btn"></app-theme-toggle>

  <form (ngSubmit)="onSubmit()" class="auth-form" [formGroup]="authForm">
    <div class="form-group">
      <input type="email" class="form-input" formControlName="email" placeholder="Електронна пошта" required>
    </div>
    <div class="form-group">
      <input type="password" class="form-input" formControlName="password" placeholder="Пароль" required>
    </div>
    <button type="submit" class="submit-btn">Увійти</button>
    <div class="toggle-link">
      Немає акаунта? <a (click)="goToRegister()">Зареєструватися</a>
    </div>
  </form>
</div>
```

Рисунок 3.3 – HTML реалізація сторінки авторизації

Усі стилі компонента зосереджено у відповідному CSS-файлі, де використано змінні для забезпечення темної/світлої теми, закруглення елементів, прозорий фон з ефектом backdrop-filter, а також плавні переходи для інтерактивності.

Таким чином, дана сторінка демонструє зразкове використання Angular-інструментів: реактивні форми**,** вбудована валідація**,** роутинг**,** ін’єкція залежностей - усе це робить реалізацію надійною, розширюваною та зручною для підтримки.

Сторінка реєстрації

Коротко розглянемо сторінку реєстрації**,** яка за структурою та стилістикою є дуже подібною до раніше описаної сторінки авторизації. Основне її призначення – створення нового облікового запису користувача через заповнення відповідної форми. На рисунку 3.4 зображено зовнішній вигляд сторінки реєстрації.

З точки зору реалізації, сторінка побудована за тією ж логікою, що й сторінка авторизації. Angular-компонент RegistrationPageComponent відповідає за реактивну форму з трьома полями (login, email, password), локальну валідацію, а також виклик методу register (...) сервісу AuthService. Оскільки структура HTML, логіка валідації та стилі дуже схожі на сторінку авторизації, достатньо ознайомитись з фрагментом TypeScript-коду (див. рис. 3.5). HTML-шаблон містить аналогічні елементи: інпут-поля, кнопку надсилання та навігаційне посилання, а також кастомні компоненти.

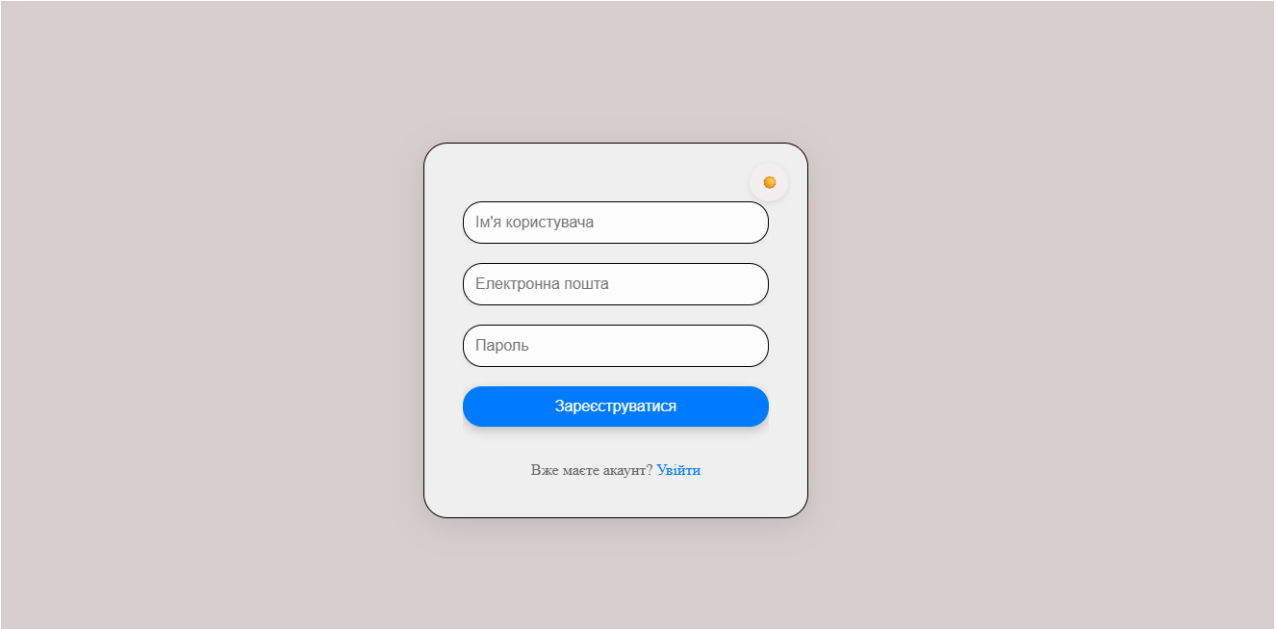

Рисунок 3.4 – Скріншот сторінки реєстрації

```typescript
import { Component, inject } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { ThemeToggleComponent } from '../../common-ui/theme-toggle/theme-toggle.component';
import { AuthService } from '../../auth/auth.service';

@Component({
  selector: 'app-registration-page',
  imports: [
    ReactiveFormsModule,
    ThemeToggleComponent
  ],
  templateUrl: './registration-page.component.html',
  styleUrl: './registration-page.component.css'
})
export class RegistrationPageComponent {
  private router = inject(Router);
  private authService = inject(AuthService);

  goToLogin() {
    this.router.navigate(['/login']);
  }

  authForm = new FormGroup({
    login: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    }),
    email: new FormControl('', {
      validators: [Validators.required, Validators.email],
      nonNullable: true
    }),
    password: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    })
  });

  onSubmit() {
    if (!this.authForm.valid) return;

    const { login, email, password } = this.authForm.value as {login: string; email: string; password: string;};

    this.authService.register(email, password, login).subscribe({
      next: () => {
        this.router.navigate(['/login']);
      },
      error: err => {
        console.error('Registration error:', err);
      }
    });
  }
}
```

Рисунок 3.5 – TypeScript код компонента RegistrationPageComponent

Це дозволяє зробити висновок про загальну уніфікацію логіки аутентифікації користувача, що є перевагою Angular-підходу – повторне використання компонентів і сервісів дозволяє зменшити дублювання коду, спростити тестування і підтримку.

Головна сторінка

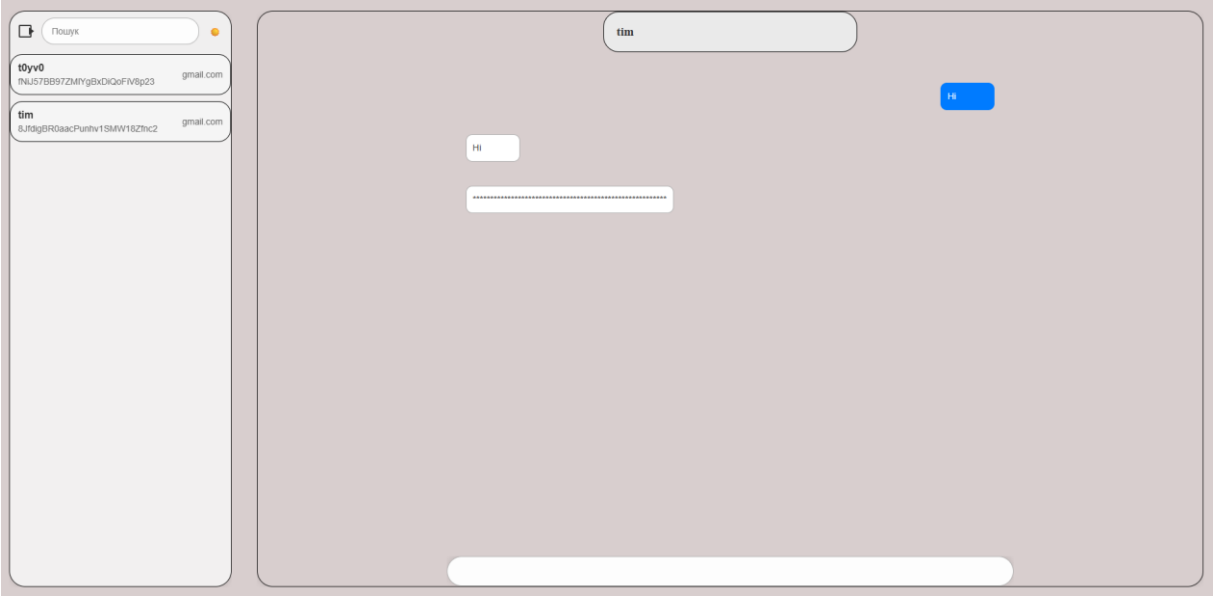

Рисунок 3.6 – Скріншот головної сторінки

Головна сторінка є центральним елементом усього інтерфейсу месенджера, адже саме вона забезпечує доступ до функціоналу пошуку користувачів, перегляду чатів, перегляду повідомлень і надсилання нових. Вона формується з двох основних частин – бокової панелі (sidebar) та вікна активного чату (chat window), які реалізовані як окремі Angular-компоненти. На рисунку 3.6 наведено зовнішній вигляд цієї сторінки.

Структура реалізована у вигляді контейнерного компонента (див. рис. 3.7).

```
1  <div class="main-wrraper">
2      <app-sidebar></app-sidebar>
3      <app-chat-window *ngIf="chatService.isChatOpen()" />
4  </div>
```

Рисунок 3.7 – HTML-код головної сторінки

Angular-компонент HomepagePageComponent відповідає за інтеграцію бокової панелі (app-sidebar) та вікна чату (app-chat-window) за допомогою сервісу ChatService. При відсутності активного чату вікно чату не відображається завдяки умовному рендерингу через ngIf.

Компонент app-sidebar реалізує логіку пошуку користувачів та відображення знайдених чатів. Нижче наведено реалізацію HTML коду цього компонента (див. рис. 3.8). Цей код є прикладом використання Angular-директив *ngFor та *ngIf, реактивних форм і динамічного рендерингу компонентів на основі введених даних.

```
<div class="side-bar">
  <div class="search-header">
    <app-log-out-button class="logout-button"></app-log-out-button>
    <form class="search-form" [formGroup]="searchChatForm">
      <input class="search_chat" formControlName="userLogin" placeholder="Пошук">
    </form>
    <app-theme-toggle class="theme-toggle"></app-theme-toggle>
  </div>
  <ul class="chat-list">
    <app-chat-card *ngFor="let user of searchUserService.searchedUsers()" [user]="user" ></app-chat-card>
    <li *ngIf="
      searchUserService.searchedUsers().length === 0 &&
      searchChatForm.get('userLogin')?.value
    ">
    Такого користувача не знайдено
    </li>
  </ul>
</div>
```

Рисунок 3.8 – HTML-код шаблону app-sidebar

Компонент app-chat-window відповідає за відображення активного діалогу, історію повідомлень, і можливість надсилати нові повідомлення в реальному часі. Детальніше з реалізацією його HTML-коду можна ознайомитись нижче (див. рис. 3.9).

```
<div class="current-chat-window">
    <div class="chat-info">
      <h3 class="chat-info-username" *ngIf="selectedUser() as user">{{ user.username }}</h3>
    </div>

    <div class="messages-container" id="messagesContainer">
      <app-message-container *ngFor="let message of messages"  [messageID]="message.id" [sender]="message.sender
      [text]="message.text" [sentAt]="message.sentAt" [status]="message.status"></app-message-container>
    </div>
    <form class = "message-form" [formGroup] = "inputMessageForm" (ngSubmit)="onSubmit()">
      <input class = "input-message" formControlName = "inputMessage">
    </form>
</div>
```

Рисунок 3.9 – HTML-код шаблону app-chat-window

Таким чином, головна сторінка об'єднує кілька самостійних Angular-компонентів, кожен із яких відповідає за окрему частину функціональності: бокову панель, чати та форму введення повідомлень.

Загалом архітектура застосунку побудована відповідно до принципів розділення та гнучкість підтримки. відповідальностей, що забезпечує

масштабованість, зручність тестування Завдяки цьому користувач отримує швидкий доступ до всіх необхідних функцій месенджера в єдиному інтерфейсі.

3.2 Розробка бази даних і бази знань месенджера

Функціонування месенджера з інтеграцією систем штучного інтелекту передбачає необхідність надійного зберігання структурованих даних, а також реалізації безпечного механізму обміну повідомленнями. У межах цієї системи база даних виконує роль централізованого сховища користувацької інформації, структури чатів та повідомлень, а також міток, що відображають результат автоматичної модерації контенту.

Для реалізації бази даних використовується хмарний сервіс Firebase Firestore, який забезпечує:

- Гнучку схему зберігання документів у форматі JSON;
- Високу масштабованість;
- Вбудовані засоби авторизації та контроль доступу;
- Підтримку роботи в режимі реального часу.

Firestore налічує всі основні колекції які були описані в розділі 2.4 такі як: users (облікові записи користувачів), chats (документи, що описують чати), messages (підколекція всередині кожного чату, яка містить усі повідомлення).

Кожне повідомлення після надсилання перевіряється сервером через Perspective API, але самі оцінки API не зберігаються. Натомість, результат аналізу інтерпретується й класифікується сервером як:

- “Ok” – безпечне повідомлення, яке одразу відображається в чаті;
- “Blocked” – потенційно небезпечне повідомлення (токсичне, спам), яке автоматично маскується на клієнті.

Це дозволяє зберігати в базі лише необхідну метаінформацію, мінімізуючи розмір записів і захищаючи користувачів від розкриття проміжних даних AI.

Для забезпечення приватності та захисту від несанкціонованого доступу застосовано спеціалізовані правила доступу Firestore (див. рис. 3.10) зокрема:

- Користувач може створювати власний профіль (/users/{userId}) лише за умови, що request.auth.uid == userId;
- Доступ до чату (/chats/{chatId}) дозволено лише учасникам, які вказані у полі users;
- Повідомлення (/messages/{messageId}) можна створювати, читати, редагувати тільки в межах чатів, учасником яких є автентифікований користувач.

Ці правила гарантують, що жоден сторонній користувач не зможе прочитати або змінити повідомлення у чужому чаті, та те що лише автентифіковані учасники мають повний контроль над своїми чатами.

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read: if true;
      allow create: if request.auth.uid == userId;
      allow update: if request.auth.uid == userId;
    }

    match /chats/{chatId} {
      allow create: if request.auth != null
                    && request.resource.data.users is list
                    && request.auth.uid in request.resource.data.users;

      allow read, update, delete: if request.auth != null
                                  && resource.data.users is list
                                  && request.auth.uid in resource.data.users;

      match /messages/{messageId} {
        allow create: if request.auth != null
                      && exists(/databases/$(database)/documents/chats/$(chatId))
                      && request.auth.uid in get(/databases/$(database)/documents/chats/$(chatId)).data.users;

        allow read: if request.auth != null
                    && exists(/databases/$(database)/documents/chats/$(chatId))
                    && request.auth.uid in get(/databases/$(database)/documents/chats/$(chatId)).data.users;

        allow update, delete: if request.auth != null
                              && exists(/databases/$(database)/documents/chats/$(chatId))
                              && request.auth.uid in get(/databases/$(database)/documents/chats/$(chatId)).data.users;
      }
    }
  }
}
```

Рисунок 3.10 – Спеціалізовані правила доступу Firestore

У розробленій системі база знань не є окремим фізичним компонентом або сховищем даних. Вона реалізується у вигляді логічного шару, який функціонує на рівні серверного програмного модуля, що взаємодіє з Perspective API.

База знань включає в себе:

- Набір критеріїв оцінювання текстових повідомлень, зокрема токсичність, образливість, спам тощо;

- Порогові значення, наприклад: якщо TOXICITY > 0.7, то повідомлення вважається небезпечним;
- Логіку ухвалення рішень, яка інтерпретує результати аналізу й присвоює статус "Ok" або "Blocked";
- Вбудовані правила реакції: повідомлення зі статусом "Blocked" не видаляється, а лише маскується на клієнті, зберігаючись у базі даних із відповідною позначкою.

Така форма реалізації бази знань дозволяє:

- оперативно оновлювати політику фільтрації без зміни структури БД;
- інкапсулювати знання у вигляді кодової логіки, що легко модифікується й перевіряється;
- забезпечити адаптивність: у майбутньому система може переходити на інші AI-сервіси або моделі з новими правилами без зміни архітектури.

Розроблена структура бази даних та реалізація логічної бази знань забезпечують надійне зберігання повідомлень і гнучку модерацію контенту на основі результатів аналізу AI-сервісу Perspective API. Замість зберігання повної відповіді моделі, система фіксує лише підсумковий статус повідомлення, що дозволяє зменшити обсяг даних і зберегти конфіденційність. Обрана архітектура дає змогу масштабувати систему без зміни структури даних – зокрема, легко інтегрувати інші AI-сервіси, додавати нові критерії фільтрації або змінювати порогові значення. Повідомлення можуть бути оперативно класифіковані, відфільтровані або повторно перевірені. Такий підхід створює основу для подальшого розвитку системи автоматичної модерації без втрати сумісності з існуючими даними.

3.3 Розробка програмних модулів месенджера

Розробка клієнтської та серверної частин месенджера здійснювалась у середовищі Visual Studio Code з використанням мов програмування TypeScript та Node.js. Вибір цих технологій був зумовлений низкою факторів.

TypeScript забезпечує високу продуктивність, статичну типізацію, а також є основною мовою для розробки в рамках фреймворку Angular, що значно полегшує

створення масштабованих компонентів користувацького інтерфейсу. Зі свого боку, Node.js є однією з найшвидших платформ для створення серверних додатків завдяки неблокуючій архітектурі та широкій екосистемі. Його простота в опануванні та активна спільнота зробили його оптимальним вибором для реалізації логіки взаємодії з базою даних, з'єднанням у режимі реального часу через Socket.IO, а також інтеграції зі сторонніми AI-сервісами.

З урахуванням функціоналу системи, необхідним кроком під час розробки стало визначення ключових інтерфейсів (моделей), які описують структуру даних, що циркулюють між клієнтською частиною, сервером і базою даних. Ці інтерфейси формують основу для типізації, спрощують обробку запитів і відповідають логічним сутностям предметної області – чатам, повідомленням і користувачам.

Інтерфейс Message

```
1  export interface Message {
2      id: string | null;
3      sender: string;
4      text: string;
5      sentAt: any;
6      chatId : string;
7      status: string;
8  }
```

Рисунок 3.11 – Код інтерфейсу повідомлення

Інтерфейс повідомлення включає (див. рис. 3.11) унікальний ідентифікатор (id), автора повідомлення (sender), текст повідомлення (text), час надсилання (sentAt), ідентифікатор чату (chatId) та статус повідомлення (status). Останнє поле зберігає результат модерації – "Ok" або "Blocked".

Інтерфейс UserSearchResponse

```
1  export interface UserSearchResponse {
2      uid: string;
3      username: string;
4      email: string;
5  }
```

Рисунок 3.12 – Код інтерфейсу пошуку

Використовується для формування відповіді при пошуку користувачів у систем (див. рис. 3.12) і містить основну ідентифікаційну інформацію: унікальний UID, ім’я користувача та адресу електронної пошти.

Ці інтерфейси забезпечують сильну типізацію даних у клієнтській частині додатку, підвищуючи стабільність, читабельність і передбачуваність коду в умовах масштабування системи.

Для реалізації обміну повідомленнями в режимі реального часу в клієнтську частину системи інтегровано окремий сервіс – SocketService, який інкапсулює логіку підключення до Socket.IO-сервера та обробки основних подій (див. рис. 3.13).

```
import { Injectable } from '@angular/core';
import { io, Socket } from 'socket.io-client';

@Injectable({
  providedIn: 'root'
})
export class SocketService {
    private socket: Socket;
    constructor() {
        this.socket = io('http://diplom-production-da57.up.railway.app');
    }

    joinChat(chatId: string) {
        this.socket.emit('join-chat', chatId);
    }

    leaveChat(chatId: string) {
        this.socket.emit('leave-chat', chatId);
    }

    sendMessage(data: any) {
        this.socket.emit('send-message', data);
    }

    onMessage(callback: (data: any) => void) {
        this.socket.on('receive-message', callback);
    }

    disconnect() {
        this.socket.disconnect();
    }
}
```

Рисунок 3.13 – Код зв’язку клієнта з сервером

Основні функції модуля:

- joinChat – підключення до кімнати чату на сервері;
- leaveChat – вихід із чату;
- sendMessage – надсилання повідомлення через сокет-з’єднання;
- onMessage – підписка на вхідні повідомлення (receive-message);
- disconnect – ручне завершення з’єднання.

Модуль реалізований як інжектабельний Angular-сервіс (@Injectable), що дозволяє використовувати його в будь-якому компоненті через dependency injection. Це забезпечує централізоване керування з’єднанням та мінімізує дублювання коду.

Для реалізації автентифікації користувачів у системі використано Firebase Authentication, а логіку обробки подій авторизації, реєстрації та виходу з облікового запису інкапсульовано в окремому Angular-сервісі AuthService.

```
register(email: string, password: string, username: string): Observable<void> {
  const firestore = this.firestore;
  const auth = this.firebaseAuth;

  return from(this.checkUsernameAvailable(username)).pipe(
    switchMap(isAvailable => {
      if (!isAvailable) {
        throw new Error('Логін уже зайнятий');
      }
      return from(
        createUserWithEmailAndPassword(auth, email, password)
          .then(userCredential => {
            const uid = userCredential.user.uid;
            const userDocRef = doc(firestore, 'users', uid);
            return setDoc(userDocRef, {
              uid,
              email: email.toLowerCase(),
              username: username.toLowerCase(),
              createdAt: new Date()
            });
          })
      );
    })
  );
}
```

Рисунок 3.14 – Код методу реєстрації користувача

```
checkUsernameAvailable(username: string): Promise<boolean> {
    const usersRef = collection(this.firestore, 'users');
    const q = query(usersRef, where('login', '==', username));
    return getDocs(q).then(snapshot => snapshot.empty);
}
```

Рисунок 3.15 – Код методу перевірки зайнятості логіна

Основні функції:

- login – здійснює вхід користувача в систему за допомогою Firebase;
- register – створює обліковий запис у Firebase Auth і паралельно зберігає додаткові дані в колекції users (див. рис. 3.14);
- checkUsernameAvailable – перевірка зайнятості логіна через запит у Firestore (див. рис. 3.15);
- logout – вихід користувача з системи з очищенням сесійних даних.

Особливості реалізації полягають в тому, що поточний авторизований користувач доступний через user$ – реактивний стрім, що дозволяє легко підписуватись на зміну стану автентифікації. Всі основні дії реалізовані через RxJS-оператори (from, switchMap), що забезпечує зручну обробку асинхронних подій та гнучку інтеграцію з іншими компонентами Angular. Валідація логіна (username) відбувається на стороні клієнта до створення облікового запису, що зменшує ризик конфліктів при реєстрації.

Одним з важливих функціональних модулів клієнтської частини є сервіс пошуку користувачів – UserSearchService (див. рис. 3.16). Він забезпечує доступ до колекції користувачів у Firebase Firestore та реалізує логіку текстового пошуку за логінами для ініціації нових чатів.

```typescript
import { Injectable, signal } from '@angular/core';
import { collection, getDocs, query, where, Firestore } from '@angular/fire/firestore';
import { UserSearchResponse } from './search.interface';

@Injectable({
  providedIn: 'root',
})
export class UserSearchService {
  searchedUsers = signal<UserSearchResponse[]>([]);

  constructor(private firestore: Firestore) {}

  async searchUser(parameters: Record<string, any>) {
    const searchTerm: string = (parameters['userLogin'] || '').toLowerCase().trim();

    if (!searchTerm) {
      this.searchedUsers.set([]);
      return;
    }

    const usersRef = collection(this.firestore, 'users');
    const q = query(
      usersRef,
      where('username', '>=', searchTerm),
      where('username', '<=', searchTerm + '\uf8ff')
    );

    const snapshot = await getDocs(q);
    const results = snapshot.docs.map(
      (doc) => doc.data() as UserSearchResponse
    );
    this.searchedUsers.set(results);
  }
}
```

Рисунок 3.16 – Код сервісу пошуку користувачів

Основні особливості сервісу:

- Пошук здійснюється за полем username у колекції users на Firestore;
- Для пошуку застосовується діапазонний фільтр (>= searchTerm, <= searchTerm + '\uf8ff'), що дозволяє імітувати пошук за префіксом;
- Результати пошуку зберігаються у реактивному сигналі searchedUsers, що інтегрується з Angular Signal API і автоматично оновлює інтерфейс при зміні даних;
- При пустому рядку пошуку масив результатів скидається до порожнього;
- Сервіс приймає об'єкт parameters, з якого виділяє userLogin – пошукову строку. Цей підхід дозволяє легко масштабувати метод для додаткових фільтрів у майбутньому.

Таким чином, UserSearchService виконує роль окремого шару логіки, відповідального за запити до бази даних і формування результатів пошуку, які потім використовуються при створенні нових діалогів між користувачами.

Серверна частина месенджера реалізована з використанням платформи Node.js, бібліотеки Express для створення HTTP-сервера та Socket.IO для забезпечення двостороннього зв’язку між клієнтами в режимі реального часу (див. рис. 3.17).

```
1   const express = require('express');
2   const http = require('http');
3   const { Server } = require('socket.io');
4   const cors = require('cors');
5   const axios = require('axios');
6   const admin = require('firebase-admin');
7   const serviceAccount = require('./serviceAccountKey.json');
8   admin.initializeApp({
9     credential: admin.credential.cert(serviceAccount)
10  });
```

Рисунок 3.17 – Код ініціалізації серверного оточення

```
27  io.on('connection', (socket) => {
28    console.log('User connected:', socket.id);
29
30    socket.on('join-chat', (chatId) => {
31      socket.join(chatId);
32      console.log(`Socket ${socket.id} joined chat ${chatId}`);
33    });
34
35  >   socket.on('send-message', async (data) => {···
75    });
76
77    socket.on('leave-chat', (chatId) => {
78      socket.leave(chatId);
79    });
80
81    socket.on('disconnect', () => {
82      console.log('User disconnected:', socket.id);
83    });
84  });
```

Рисунок 3.18 – Код основних подій сервера

Після запуску сервер очікує підключення клієнтів через Socket.IO та обробляє основні події (див. рис. 3.18):

- join-chat – підключення користувача до вказаного чату;
- leave-chat – вихід із чату;
- send-message – обробка надісланого повідомлення;

- receive-message – надсилання повідомлення всім учасникам чату;
- disconnect – від’єднання клієнта.

Усі повідомлення зберігаються у Firebase Firestore – в колекції chats/{chatId}/messages. Дані включають текст повідомлення, автора, мітку часу та статус модерації.

Після надсилання повідомлення сервер виконує запит до Perspective API, який аналізує текст на наявність токсичності або спаму. Залежно від результатів, повідомлення позначається як “Ok” або “Blocked”. Детальний опис інтеграції з Perspective API, логіки класифікації та роботи з AI-модерацією буде подано в наступному розділі.

У разі успішної перевірки повідомлення розсилається всім учасникам чату. Якщо ж виникає помилка при запиті до API, система надсилає клієнту повідомлення зі статусом “error”.

Такий підхід дозволяє досягти миттєвого реагування на дії користувачів, забезпечити централізовану модерацію повідомлень і надійне збереження всієї історії листування.

Архітектура додатку побудована на сучасному технологічному стеку: Angular + TypeScript для клієнта та Node.js + Socket.IO для сервера. Такий підхід забезпечив високу швидкодію, модульність і простоту масштабування.

Всі ключові функціональні блоки винесено в окремі сервіси – автентифікація, обмін повідомленнями, пошук користувачів. Інтерфейси чітко визначають структуру даних, що передаються між клієнтом, сервером та базою, що покращує контроль за типами і зменшує кількість помилок на етапі розробки.

Серверна частина реалізує логіку обробки повідомлень у реальному часі з використанням бібліотеки Socket.IO, а також виконує перевірку текстів через Perspective API. Отримані результати використовуються для автоматичної модерації контенту без втручання користувача. Взаємодія з Firebase Firestore забезпечує зберігання даних і керування правами доступу.

Загалом, реалізована структура дозволяє ефективно розподіляти обов’язки між модулями, підтримувати чисту архітектуру коду та забезпечити надійне функціонування системи в реальному часі.

3.4 Розробка підсистеми штучного інтелекту для виявлення спаму та токсичних повідомлень у чатах

У сучасних комунікаційних системах особливу роль відіграє автоматична модерація контенту. З метою забезпечення безпечного середовища спілкування у межах розробленого месенджера було реалізовано підсистему штучного інтелекту, що дозволяє виявляти спам і токсичні повідомлення.

Для реалізації цієї підсистеми було обрано готовий хмарний сервіс – Perspective API, розроблений дослідницьким підрозділом Google Jigsaw. Цей сервіс надає REST API для аналізу текстових повідомлень на наявність різних негативних атрибутів, таких як токсичність, агресія, спам, образливість тощо.

Принцип роботи Perspective API полягає в тому, що він отримує на вхід текстове повідомлення та повертає ймовірнісну оцінку кожного запитаного атрибута. Значення коливаються в межах від 0 до 1, де вища оцінка означає більшу ймовірність наявності відповідної характеристики в тексті.

Серед доступних атрибутів є такі як:

- TOXICITY – загальна токсичність повідомлення;
- INSULT – наявність образ;
- SPAM – ознаки спаму;
- THREAT – погрози;
- PROFANITY – ненормативна лексика.

У межах даної роботи використовуються лише атрибути TOXICITY та SPAM.

Метрика TOXICITY частково охоплює інші негативні аспекти – зокрема, INSULT, THREAT та PROFANITY, хоча й без чіткої деталізації кожної з категорій. У разі потреби, підключення додаткових атрибутів не потребує змін у структурі системи й може бути реалізоване за кілька хвилин шляхом розширення JSON-запиту до API.

Формат запиту до API складається з того, що сервіс очікує HTTP POST-запит у форматі JSON, де вказується текст повідомлення та список атрибутів, які потрібно проаналізувати (див. рис. 3.19).

Рисунок 3.19 – Приклад HTTP POST-запиту у форматі JSON

Як відповідь API повертає відповідь у вигляді об’єкта з оцінками ймовірностей в форматі JSON (див. рис. 3.20).

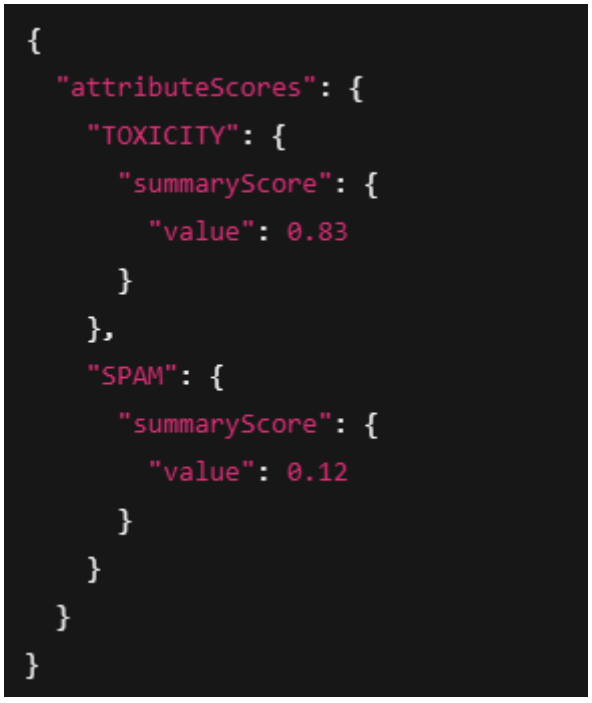

Рисунок 3.20 – Приклад формату відповіді від API

На основі отриманих чисел система виконує просту логіку класифікації: якщо хоча б один із показників перевищує заданий поріг, повідомлення вважається небажаним. Цей підхід забезпечує автоматичне прийняття рішення без залучення людини-модератора, зменшуючи ризики появи небажаного контенту в чатах.

3.5 Розробка програмних засобів інтеграції месенджера і підсистеми AI

Однією з ключових функцій розробленого месенджера є автоматична модерація повідомлень на основі аналізу їх вмісту за допомогою штучного інтелекту. Для реалізації цієї функціональності було здійснено інтеграцію серверної частини системи з хмарним сервісом Perspective API, який було описано у попередньому розділі.

Процес інтеграції виконується безпосередньо на рівні Node.js-сервера, який отримує повідомлення клієнта, надсилає його текст на аналіз до Perspective API, інтерпретує результат, а потім ухвалює рішення про відображення або блокування повідомлення. Ця логіка реалізована в рамках обробника події send-message модуля Socket.IO.

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const cors = require('cors');
const axios = require('axios');
const admin = require('firebase-admin');
const serviceAccount = require('./serviceAccountKey.json');
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});

const db = admin.firestore();
const app = express();
app.use(cors());

const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"]
  }
});
```

Рисунок 3.21 – Код ініціалізації серверної частини з підключенням до Perspective API

Коли клієнт надсилає нове повідомлення до сервера через подію send-message, спочатку виконується запит до Perspective API із текстом цього повідомлення. У

відповідь сервер отримує оцінки рівня токсичності (TOXICITY) та ймовірності спаму (SPAM). Далі реалізується логіка класифікації, що визначає, чи є повідомлення допустимим для відображення (див. рис. 3.22).

```
37    const response = await axios.post(
38      `https://commentanalyzer.googleapis.com/v1alpha1/comments:analyze?key=${PERSPECTIVE_KEY}`,
39      {
40        comment: { text: data.text },
41        languages: ['en'],
42        requestedAttributes: { TOXICITY: {}, SPAM: {} }
43      }
44    );
45
46    const toxicity = response.data.attributeScores.TOXICITY
47      ? response.data.attributeScores.TOXICITY.summaryScore.value
48      : 0;
49    const spam = response.data.attributeScores.SPAM
50      ? response.data.attributeScores.SPAM.summaryScore.value
51      : 0;
52
53    data.status = (toxicity > 0.7 || spam > 0.7) ? 'blocked' : 'ok';
```

Рисунок 3.22 – Код основної логіки інтеграції з Perspective API

Ці значення порівнюються з встановленими порогами (> 0.7), після чого статус повідомлення (Ok або Blocked) зберігається в базі даних Firebase Firestore разом із текстом і автором повідомлення. Усі учасники відповідного чату отримують повідомлення через подію receive-message, а на стороні клієнта реалізовано маскування повідомлень зі статусом blocked (див. рис. 3.23).

```
55    const saved = await db.collection('chats')
56    .doc(data.chatId)
57    .collection('messages')
58    .add({
59      sender: data.sender,
60      text: data.text,
61      sentAt: admin.firestore.FieldValue.serverTimestamp(),
62      status: data.status
63    });
64
65  data.id = saved.id;
66  io.to(data.chatId).emit('receive-message', data);
```

Рисунок 3.23 – Код збереження повідомлення у Firestore та розсилка по сокетах

У разі помилки під час запиту до Perspective API (наприклад, при втраті з’єднання або перевищенні ліміту), сервер повертає повідомлення зі статусом error, інформуючи користувача про те, що модерація тимчасово недоступна (див. рис. 3.24).

```
69      } catch (e) {
70        data.status = 'error';
71        data.error = 'Moderation unavailable';
72        socket.to(data.chatId).emit('receive-message', data);
73        console.error(e?.response?.data || e);
74      }
```

Рисунок 3.23 – Код обробки помилок при запиті до Perspective API

Переваги реалізованої інтеграції:

- Легкість масштабування – атрибути, які аналізуються, легко змінюються або доповнюються;
- Незалежність клієнта – вся модерація виконується на сервері, що захищає модель від зловживань;
- Прозора логіка – класифікація побудована на чітко заданих порогах, які можна гнучко налаштовувати;
- Гнучкість зберігання – зберігається лише результат аналізу, без проміжних значень.

У межах розділу було реалізовано повноцінну інтеграцію клієнт-серверної архітектури з підсистемою штучного інтелекту. Отримані оцінки від Perspective API дозволяють у реальному часі класифікувати повідомлення та застосовувати політику модерації. Такий підхід значно підвищує безпеку спілкування та мінімізує потребу в ручному нагляді за контентом у чатах.

У рамках реалізації було створено зручний та інтуїтивно зрозумілий інтерфейс користувача, який забезпечує доступ до основного функціоналу месенджера. На наведених нижче скріншотах продемонстровано ключові елементи клієнтської частини застосунку, зокрема вікно авторизації, список чатів, вікно обміну повідомленнями, а також приклади спрацювання системи модерації. Окремо представлено приклад взаємодії із підсистемою штучного інтелекту, яка аналізує повідомлення та реагує на виявлення спаму або токсичних висловлювань. Це підтверджує ефективність інтеграції AI-компонента в реальних сценаріях використання та демонструє готовність продукту до подальшого практичного впровадження.

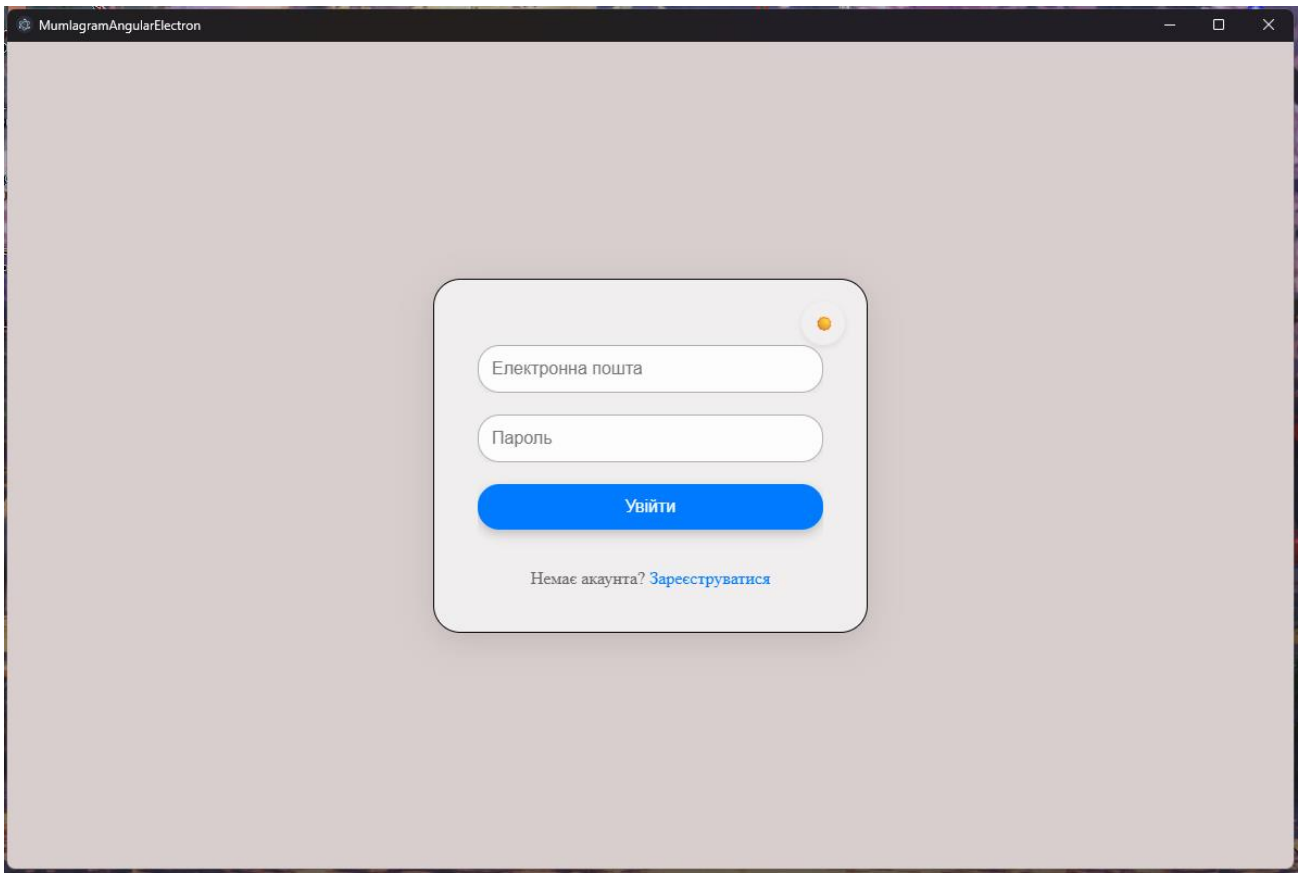

Рисунок 3.24 – Скріншот вікна авторизації

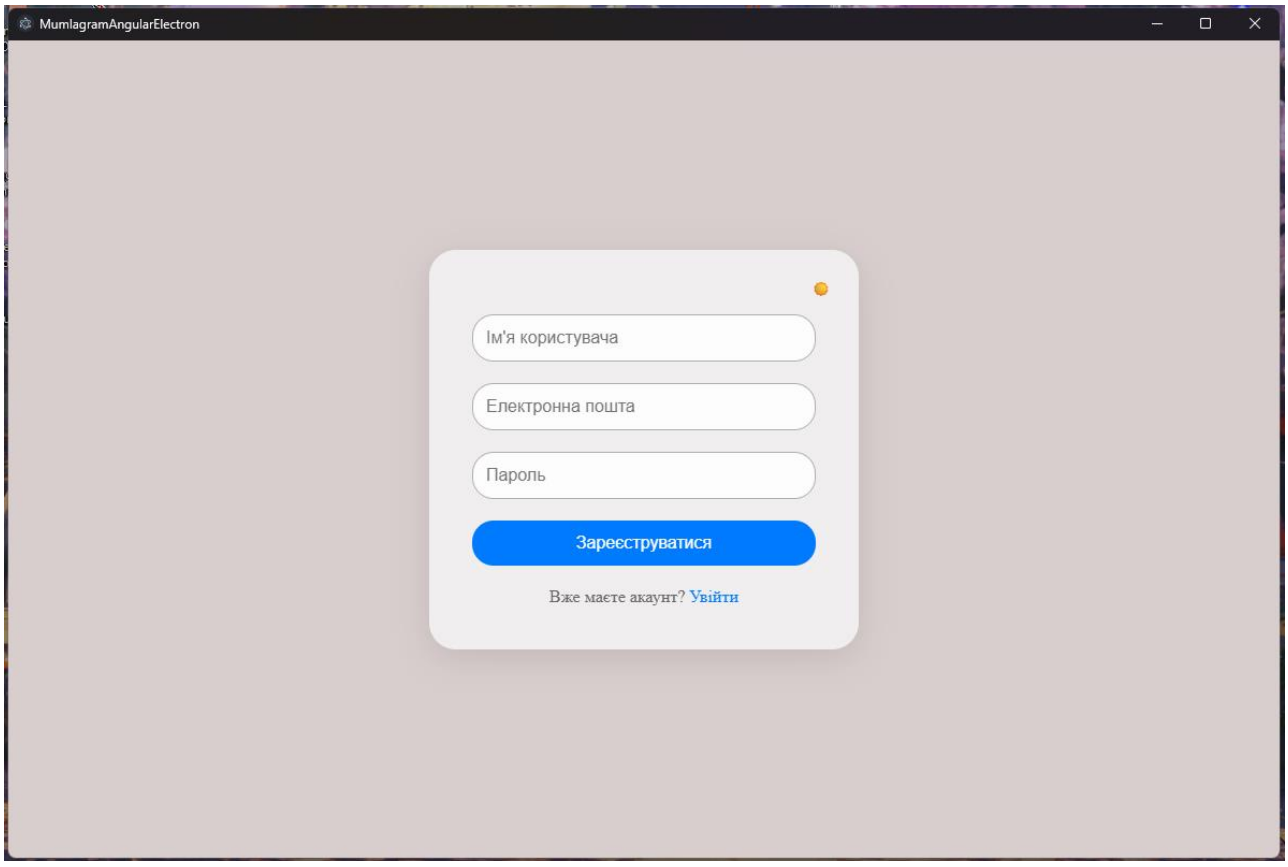

Рисунок 3.25 – Скріншот вікна реєстрації

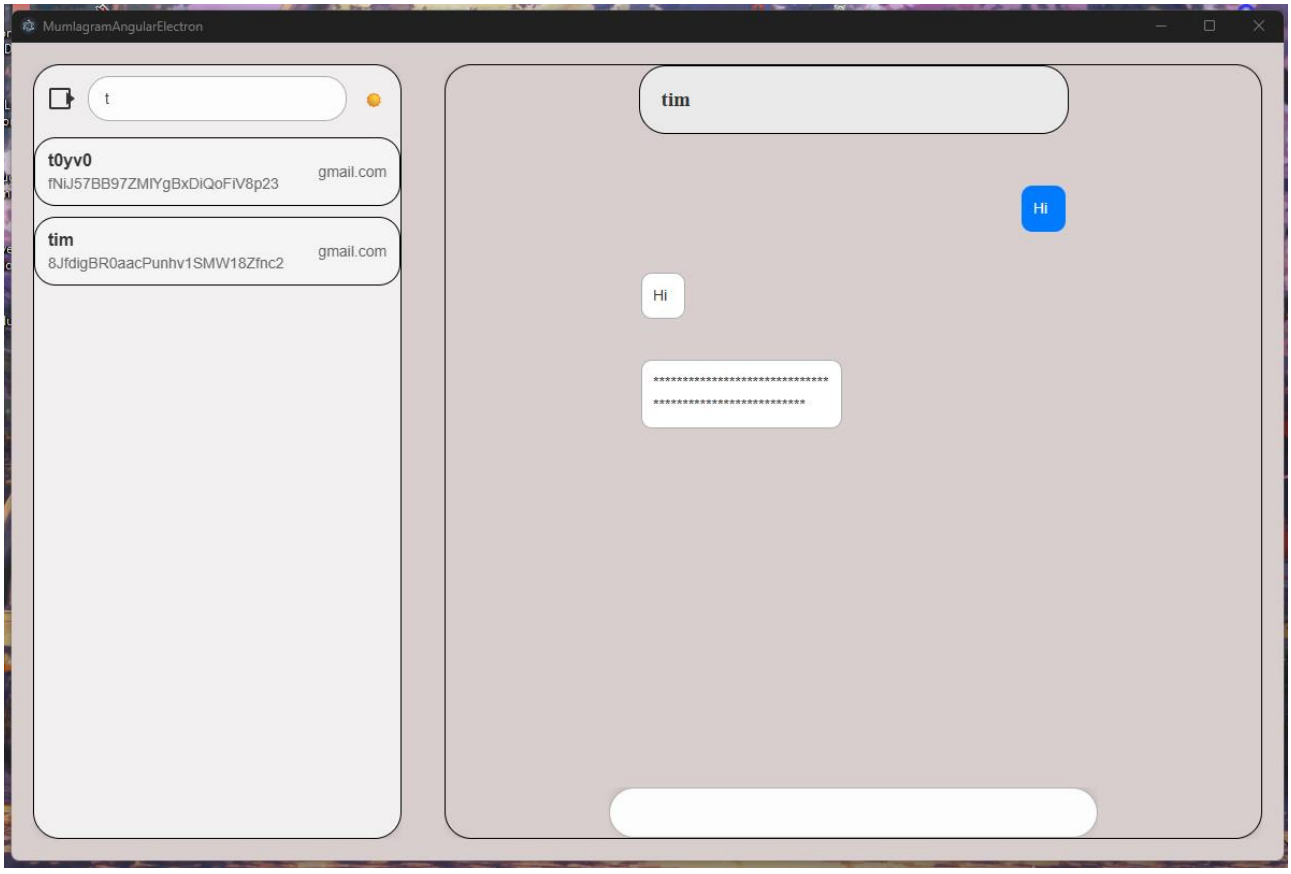

Рисунок 3.26 – Скріншот головного вікна месенджера

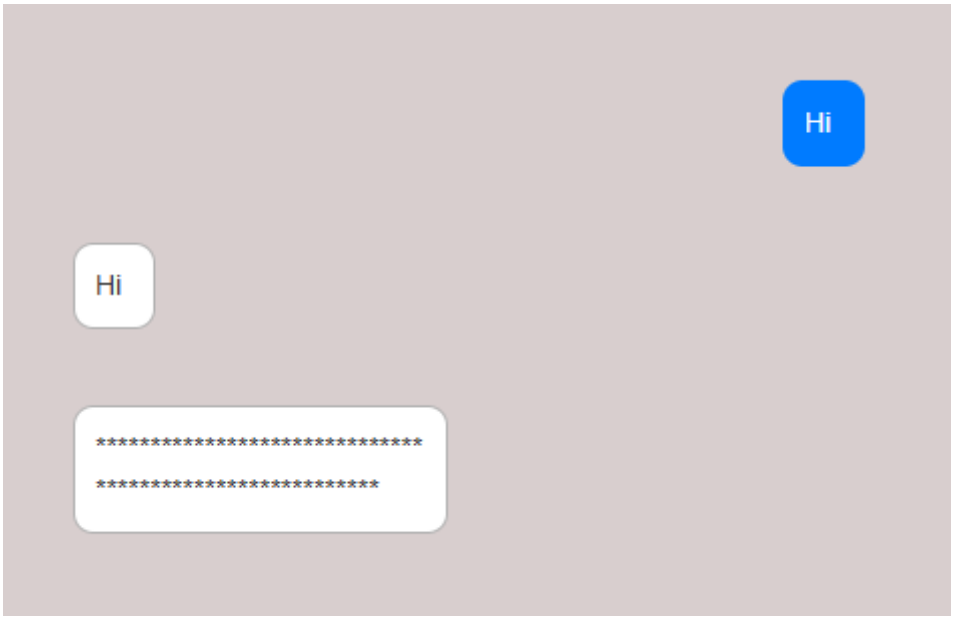

Рисунок 3.27 – Приклад замаскованого токсичного чи спам повідомлення

Висновки

Метою даної роботи було створення програмного засобу для обміну повідомленнями, інтегрованого з підсистемою штучного інтелекту, яка забезпечує автоматичне виявлення та фільтрацію токсичних і спам-повідомлень у режимі реального часу.

На початковому етапі роботи було проведено детальний аналіз існуючих аналогів, зокрема популярних месенджерів і рішень зі штучною модерацією контенту. Було визначено їхні функціональні можливості, архітектурні підходи та методи боротьби з токсичністю в чатах.

На основі аналізу сформульовано вимоги до майбутньої системи, серед яких ключовими стали: реалізація обміну повідомленнями в режимі реального часу, автоматична класифікація вмісту на основі AI-сервісів та простота масштабування архітектури.

Після цього було здійснено комплексне проєктування архітектури майбутньої системи. Основну увагу зосереджено на побудові структурної моделі месенджера з урахуванням підсистеми штучного інтелекту для фільтрації повідомлень. Було створено концептуальні схеми, які відображають взаємодію між клієнтською, серверною частинами та зовнішнім AI-сервісом. Окремо опрацьовано логіку руху даних усередині системи, що дозволило формалізувати ключові сценарії обміну інформацією між користувачами, зберіганням у базі даних та аналізом повідомлень на токсичність.

Крім того, було розроблено макети інтерфейсу користувача, що дало змогу ще на етапі проєктування врахувати зручність взаємодії з системою та визначити основні функціональні компоненти. В результаті проєктування було закладено логічну основу для реалізації програмного коду, забезпечено цілісність архітектури та визначено технічні рішення, які були реалізовані в наступних розділах.

Під час розробки програмного забезпечення було використано сучасні фреймворки та платформи, що забезпечили швидку, масштабовану й безпечну реалізацію функціональності. Клієнтська частина месенджера створювалася з використанням Angular та мови TypeScript, що дозволило реалізувати компонентну

архітектуру, реактивну логіку та зручний інтерфейс. Для з’єднання в режимі реального часу було застосовано бібліотеку Socket.IO, яка забезпечує двосторонню комунікацію між клієнтами й сервером через WebSocket.

Серверна частина реалізована на Node.js із використанням Express для обробки HTTP-запитів та взаємодії з базою даних Firestore. У процесі розробки програмних модулів було чітко виокремлено сервіси, відповідальні за автентифікацію, обробку повідомлень, модерацію та пошук користувачів. Логіка модерації базується на інтеграції з AI-сервісом Perspective API, який аналізує текст повідомлень перед збереженням у базі. Повідомлення, що визначаються як потенційно небезпечні, автоматично маркуються й маскуються на клієнті.

Завдяки правильній організації структури коду, використанню інжектабельних сервісів та дотриманню принципів реактивного програмування, вдалося досягти високої надійності, зручності підтримки та можливості подальшого розширення системи. Усі реалізовані компоненти повністю інтегруються між собою, утворюючи єдину узгоджену систему з підтримкою автоматичної модерації в реальному часі.

Після завершення основних етапів розробки було отримано функціональний програмний продукт, який повністю реалізує заявлену архітектуру та логіку роботи. Система забезпечує обмін повідомленнями в режимі реального часу, автоматичну модерацію контенту за допомогою зовнішнього AI-сервісу, автентифікацію користувачів і пошук співрозмовників. Усі компоненти клієнтської та серверної частин успішно взаємодіють між собою, а структура бази даних відповідає вимогам безпеки та масштабованості.

Отриманий програмний комплекс є завершеним з погляду реалізації основного функціоналу та готовим до проведення функціонального й користувацького тестування з метою виявлення можливих недоліків і підготовки до розгортання в реальному середовищі.

Список джерел

1. Connell, Adam. Most Popular Messaging Apps in 2023. Adam Connell, https://adamconnell.me/popular-messaging-apps/

2. Kemp, Simon. Digital 2024: Global Overview Report. DataReportal, https://datareportal.com/social-media-users

3. Telegram. MTProto Protocol Documentation. Telegram Core, https://core.telegram.org/mtproto

4. Meta. Messenger Platform Documentation. Facebook for Developers, https://developers.facebook.com/docs/messenger-platform

5. Signal. Signal Protocol Documentation. Signal Foundation, https://signal.org/docs/

6. Discord. Developer Documentation: Introduction. Discord Developer Portal, https://discord.com/developers/docs/intro

7. Erlang Solutions. 20 Years of Open Source Erlang: Interview with Anton Lavrik from WhatsApp, https://www.erlang-solutions.com/blog/20-years-of-open-source-erlang-openerlang-interview-with-anton-lavrik-from-whatsapp/

8. Telegram. Antispam API. Telegram Core, https://core.telegram.org/api/antispam

9. Meta. Meta’s Approach to Safer Private Messaging. Meta Newsroom, https://about.fb.com/news/2021/12/metas-approach-to-safer-private-messaging/

10. Newman, Lily Hay. “Facebook Messenger's New Safety Alerts Are Just the Beginning.” Wired, https://www.wired.com/story/facebook-messenger-safety-alerts-encryption/

11. Signal. Keeping Spam off Signal. Signal Blog, https://signal.org/blog/keeping-spam-off-signal/.

12. LeapXpert. How WhatsApp’s New Security Features Make Blocking Spam Much Easier, https://www.leapxpert.com/how-whatsapps-new-security-features-make-blocking-spam-much-easier/

13. Discord. AutoMod FAQ. Discord Support, https://support.discord.com/hc/en-us/articles/4421269296535-AutoMod-FAQ

14. Discord. New Anti-Spam, Raid, and AutoMod Safety Update. Discord Blog, https://discord.com/blog/new-anti-spam-raid-automod-safety-update

15. Norton. WhatsApp Scams to Avoid. Norton LifeLock, https://us.norton.com/blog/online-scams/whatsapp-scams

16. The Scottish Sun. “Parents Warned over WhatsApp Cash Scam.” The Scottish Sun, https://www.thescottishsun.co.uk/tech/14749904/whatsapp-cash-scam-warning-parents/

17. Uzi, Shira. “Vet Chris Brown Used in Catfishing Scam.” Daily Mail, https://www.dailymail.co.uk/news/article-14701805/Bondi-vet-Chris-Brown-catfish-ai-scam.html

18. Cyber Insider. Viber Messenger Abused for Delivering Pegasus Spyware, https://cyberinsider.com/viber-messenger-abused-for-delivering-pegasus-spyware-on-targets/

19. The Media Line. Telegram’s Hate Speech Increased 433% Since October 7, https://themedialine.org/top-stories/telegrams-hate-speech-increased-433-since-october-7-2023-new-privacy-policies-useless-experts-say/

20. National Institutes of Health. Cyberbullying Linked to Suicidal Thoughts in Adolescents, https://www.nih.gov/news-events/nih-research-matters/cyberbullying-linked-suicidal-thoughts-attempts-young-adolescents

21. Dev.ua. Telegram Pushes Erotic Channels – Research, https://dev.ua/news/telehram-pushyt-stromni-kanaly-doslidzhennia-1734442804

Додатки

Додаток А – Код Програми

Серверна частина

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const cors = require('cors');
const axios = require('axios');
const admin = require('firebase-admin');
const serviceAccount = require('./serviceAccountKey.json');
admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});

const db = admin.firestore();
const app = express();
app.use(cors());

const server = http.createServer(app);
const io = new Server(server, {
  cors: {
    origin: "*",
    methods: ["GET", "POST"]
  }
});

const PORT = 3000;
const PERSPECTIVE_KEY = "";

io.on('connection', (socket) => {
  console.log('User connected:', socket.id);
```

```
  socket.on('join-chat', (chatId) => {
    socket.join(chatId);
    console.log(`Socket ${socket.id} joined chat ${chatId}`);
  });

  socket.on('send-message', async (data) => {
    try {
      const response = await axios.post(
`https://commentanalyzer.googleapis.com/v1alpha1/comments:analyze?key=
${PERSPECTIVE_KEY}`,
        {
          comment: { text: data.text },
          languages: ['en'],
          requestedAttributes: { TOXICITY: {}, SPAM: {} }
        }
      );

      const toxicity = response.data.attributeScores.TOXICITY
        ? response.data.attributeScores.TOXICITY.summaryScore.value
        : 0;
      const spam = response.data.attributeScores.SPAM
        ? response.data.attributeScores.SPAM.summaryScore.value
        : 0;

      data.status = (toxicity > 0.7 || spam > 0.7) ? 'blocked' : 'ok';

      const saved = await db.collection('chats')
      .doc(data.chatId)
      .collection('messages')
      .add({
        sender: data.sender,
        text: data.text,
```

```
        sentAt: admin.firestore.FieldValue.serverTimestamp(),
        status: data.status
      });

    data.id = saved.id;
    io.to(data.chatId).emit('receive-message', data);

      io.to(data.chatId).emit('receive-message', data);
    } catch (e) {
      data.status = 'error';
      data.error = 'Moderation unavailable';
      socket.to(data.chatId).emit('receive-message', data);
      console.error(e?.response?.data || e);
    }
  });

  socket.on('leave-chat', (chatId) => {
    socket.leave(chatId);
  });

  socket.on('disconnect', () => {
    console.log('User disconnected:', socket.id);
  });
});

server.listen(PORT, () => {
  console.log(`Socket.IO server running on
http://localhost:${PORT}/`);
});
```

Код app.routs.ts

```
import { Routes } from '@angular/router';
import { LoginPageComponent }        from './pages/login-page/login-
page.component';
```

```
import { HomepagePageComponent }     from './pages/homepage-
page/homepage-page.component';
import { RegistrationPageComponent } from './pages/registration-
page/registration-page.component';
import { canActivateAuth, canActivateNoAuth }           from
'./auth/access.guard';


export const routes: Routes = [
  {
    path: '',
    component: HomepagePageComponent,
    canActivate: [ canActivateAuth ]
  },
  {
    path: 'login',
    component: LoginPageComponent,
    canActivate: [ canActivateNoAuth ]
  },
  {
    path: 'registration',
    component: RegistrationPageComponent,
    canActivate: [ canActivateNoAuth ]
  },
  {
    path: '**',
    redirectTo: ''
  }
];
```

Код access.guard.ts

```
import { inject } from '@angular/core';
import { Router, UrlTree } from '@angular/router';
import { AuthService } from './auth.service';
import { map, take } from 'rxjs/operators';
```

```
import { Observable } from 'rxjs';

export const canActivateAuth = (): Observable<boolean|UrlTree> => {
  const auth = inject(AuthService);
  const router = inject(Router);

  return auth.user$.pipe(
    take(1),
    map(user => {
      if (user) {
        return true;
      }
      return router.createUrlTree(['/login']);
    })
  );
};

export const canActivateNoAuth = (): Observable<boolean | UrlTree> =>
{
  const auth = inject(AuthService);
  const router = inject(Router);

  return auth.user$.pipe(
    take(1),
    map(user => {
      if (user) {
        return router.createUrlTree(['/']);
      }
      return true;
    })
  );
};
```

Код сервісу auth.service.ts

```
import { Injectable } from '@angular/core';
import {
  Auth,
  createUserWithEmailAndPassword,
  signInWithEmailAndPassword,
  signOut,
  user,
  User
} from '@angular/fire/auth';
import { Firestore } from '@angular/fire/firestore';
import { collection, doc, getDocs, query, setDoc, where } from
'firebase/firestore';
import { from, Observable, switchMap } from 'rxjs';


@Injectable({ providedIn: 'root' })
export class AuthService {
  user$: Observable<User | null>;


  constructor(private firebaseAuth: Auth, private firestore:
Firestore) {
    this.user$ = user(this.firebaseAuth);
  }


  login(email: string, password: string): Observable<void> {
    return from(signInWithEmailAndPassword(this.firebaseAuth, email,
password).then(() => {}));
  }


  checkUsernameAvailable(username: string): Promise<boolean> {
    const usersRef = collection(this.firestore, 'users');
    const q = query(usersRef, where('login', '==', username));
    return getDocs(q).then(snapshot => snapshot.empty);
```

```
  }

  register(email: string, password: string, username: string):
Observable<void> {
    const firestore = this.firestore;
    const auth = this.firebaseAuth;

    return from(this.checkUsernameAvailable(username)).pipe(
      switchMap(isAvailable => {
        if (!isAvailable) {
          throw new Error('Логін уже зайнятий');
        }
        return from(
          createUserWithEmailAndPassword(auth, email, password)
            .then(userCredential => {
              const uid = userCredential.user.uid;
              const userDocRef = doc(firestore, 'users', uid);
              return setDoc(userDocRef, {
                uid,
                email: email.toLowerCase(),
                username: username.toLowerCase(),
                createdAt: new Date()
              });
            })
        );
      })
    );
  }

  logout(): Observable<void> {
    return from(signOut(this.firebaseAuth).then(() =>
sessionStorage.clear()));
  }
```

```
}
```

Код сервісу chat.service.ts

```
import { Injectable, signal } from '@angular/core';
import { UserSearchResponse } from './search.interface';
import { Firestore, collection, query, where, getDocs, addDoc, orderBy
} from '@angular/fire/firestore';
import { Auth } from '@angular/fire/auth';
import { SocketService } from './socket.service';
import { Message } from './message.interface';

@Injectable({ providedIn: 'root' })
export class ChatService {
  isChatOpen = signal(false);
  selectedUser = signal<UserSearchResponse | null>(null);
  public chatId: string | null = null;
  messages = signal<Message[]>([]);

  constructor(
    private firestore: Firestore,
    private auth: Auth,
    private socketService: SocketService
  ) {}

  setChatId(id: string) { this.chatId = id; }

  async getOrCreateChatId(currentUserId: string, selectedUserId:
string): Promise<string> {
    const userPair = [currentUserId, selectedUserId].sort();
    const chatsRef = collection(this.firestore, 'chats');
    const q = query(chatsRef, where('users', '==', userPair));
    const querySnapshot = await getDocs(q);

    if (!querySnapshot.empty) {
```

```
      return querySnapshot.docs[0].id;
    } else {
      const docRef = await addDoc(chatsRef, { users: userPair,
createdAt: new Date() });
      return docRef.id;
    }
  }

  async loadMessagesForChat(chatId: string) {
    const messagesRef = collection(this.firestore,
`chats/${chatId}/messages`);
    const q = query(messagesRef, orderBy('sentAt'));
    const snapshot = await getDocs(q);
    this.messages.set(snapshot.docs.map(doc => ({
      id: doc.id,
      ...(doc.data() as Omit<Message, 'id'>)
    })));
  }

  openChatWith(user: UserSearchResponse) {
    this.closeChat();
    this.selectedUser.set(user);
    this.isChatOpen.set(true);
    const currentUid = this.auth.currentUser?.uid;
    if (currentUid) {
      this.getOrCreateChatId(currentUid, user.uid).then(chatId => {
        this.setChatId(chatId);
        this.socketService.joinChat(chatId);
        this.loadMessagesForChat(chatId);
      });
    }
  }
```

```
  closeChat() {
    this.isChatOpen.set(false);
    this.selectedUser.set(null);
    if (this.chatId)
      this.socketService.leaveChat(this.chatId);
    this.chatId = null;
    this.messages.set([]);
  }
}
```

Код сервісу socket.service.ts

```
import { Injectable } from '@angular/core';
import { io, Socket } from 'socket.io-client';

@Injectable({
  providedIn: 'root'
})
export class SocketService {
    private socket: Socket;
    constructor() {
        this.socket = io('');
    }

    joinChat(chatId: string) {
        this.socket.emit('join-chat', chatId);
    }

    leaveChat(chatId: string) {
        this.socket.emit('leave-chat', chatId);
    }

    sendMessage(data: any) {
        this.socket.emit('send-message', data);
```

```
    }

    onMessage(callback: (data: any) => void) {
        this.socket.on('receive-message', callback);
    }

    disconnect() {
        this.socket.disconnect();
    }
}
```

Код сервісу user-search.service.ts

```
import { Injectable, signal } from '@angular/core';
import { collection, getDocs, query, where, Firestore } from
'@angular/fire/firestore';
import { UserSearchResponse } from './search.interface';

@Injectable({
  providedIn: 'root',
})
export class UserSearchService {
  searchedUsers = signal<UserSearchResponse[]>([]);

  constructor(private firestore: Firestore) {}

  async searchUser(parameters: Record<string, any>) {
    const searchTerm: string = (parameters['userLogin'] ||
'').toLowerCase().trim();

    if (!searchTerm) {
      this.searchedUsers.set([]);
      return;
    }
```

```
    const usersRef = collection(this.firestore, 'users');
    const q = query(
      usersRef,
      where('username', '>=', searchTerm),
      where('username', '<=', searchTerm + '\uf8ff')
    );

    const snapshot = await getDocs(q);
    const results = snapshot.docs.map(
      (doc) => doc.data() as UserSearchResponse
    );
    this.searchedUsers.set(results);
  }
}
```

Код інтерфейсу Messages

```
export interface Message {
  id: string | null;
  sender: string;
  text: string;
  sentAt: any;
  chatId : string;
  status: string;
}
```

Код інтерфейсу UserSearchResponse

```
export interface UserSearchResponse {
  uid: string;
  username: string;
  email: string;
}
```

Код стилів компонента app-registration-page

```
:host {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  padding: 20px;
}

.box {
  position: relative;
  background: var(--chat-bg);
  backdrop-filter: blur(10px);
  width: 100%;
  max-width: 400px;
  padding: 60px 40px 40px 40px;
  border-radius: 25px;
  box-shadow: 0 8px 32px var(--shadow-color);
  text-align: center;
}

app-theme-toggle.theme-toggle-btn {
  position: absolute;
  top: 20px;
  right: 20px;
  display: flex;
  align-items: center;
  justify-content: center;
  height: 32px;
  width: 32px;
}
```

```
/* Форма */
.form-group {
  margin-bottom: 20px;
}

.form-input {
  width: 100%;
  padding: 12px;
  border: 1px solid var(--border-color);
  border-radius: 20px;
  background: var(--input-bg);
  color: var(--text-color);
  font-size: 16px;
  transition: all 0.3s ease;
}

.form-input:focus {
  border-color: var(--accent-color);
  outline: none;
}

.submit-btn {
  width: 100%;
  padding: 12px;
  border: none;
  border-radius: 20px;
  background: var(--accent-color);
  color: white;
  font-size: 16px;
  cursor: pointer;
  transition: opacity 0.3s;
```

```
}

.submit-btn:hover {
  opacity: 0.9;
}

.toggle-link {
  margin-top: 20px;
  color: var(--secondary-text);
}

.toggle-link a {
  color: var(--accent-color);
  text-decoration: none;
  cursor: pointer;
}
```

Код HTML розмітки компонента app-registration-page

```
<div class="box">
  <app-theme-toggle class="theme-toggle-btn"></app-theme-toggle>

  <form (ngSubmit)="onSubmit()" class="auth-form"
[formGroup]="authForm">
    <div class="form-group">
      <input type="text" class="form-input" formControlName="login"
placeholder="Ім'я користувача" required>
    </div>
    <div class="form-group">
      <input type="email" class="form-input" formControlName="email"
placeholder="Електронна пошта" required>
    </div>
    <div class="form-group">
      <input type="password" class="form-input"
formControlName="password" placeholder="Пароль" required>
```

```
    </div>
    <button type="submit" class="submit-btn">Зареєструватися</button>
    <div class="toggle-link">
      Вже маєте акаунт? <a (click)="goToLogin()">Увійти</a>
    </div>
  </form>
</div>
```

Код TypeScript компонента app-registration-page

```
import { Component, inject } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { ThemeToggleComponent } from '../../common-ui/theme-toggle/theme-toggle.component';
import { AuthService } from '../../auth/auth.service';

@Component({
  selector: 'app-registration-page',
  imports: [
    ReactiveFormsModule,
    ThemeToggleComponent
  ],
  templateUrl: './registration-page.component.html',
  styleUrl: './registration-page.component.css'
})
export class RegistrationPageComponent {
  private router = inject(Router);
  private authService = inject(AuthService);

  goToLogin() {
    this.router.navigate(['/login']);
  }
```

```
  authForm = new FormGroup({
    login: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    }),
    email: new FormControl('', {
      validators: [Validators.required, Validators.email],
      nonNullable: true
    }),
    password: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    })
  });

  onSubmit() {
    if (!this.authForm.valid) return;

    const { login, email, password } = this.authForm.value as {login:
string; email: string; password: string;};

    this.authService.register(email, password, login).subscribe({
      next: () => {
        this.router.navigate(['/login']);
      },
      error: err => {
        console.error('Registration error:', err);
      }
    });
  }
}
```

Код стилів компонента app-login-page

```
:host {
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  padding: 20px;
}

.box {
  position: relative;
  background: var(--chat-bg);
  backdrop-filter: blur(10px);
  width: 100%;
  max-width: 400px;
  padding: 60px 40px 40px 40px;
  border-radius: 25px;
  box-shadow: 0 8px 32px var(--shadow-color);
  text-align: center;
}

app-theme-toggle.theme-toggle-btn {
  position: absolute;
  top: 20px;
  right: 20px;
  display: flex;
  align-items: center;
  justify-content: center;
  height: 32px;
  width: 32px;
}
```

```
.form-group {
  margin-bottom: 20px;
}

.form-input {
  width: 100%;
  padding: 12px;
  border: 1px solid var(--border-color);
  border-radius: 20px;
  background: var(--input-bg);
  color: var(--text-color);
  font-size: 16px;
  transition: all 0.3s ease;
}

.form-input:focus {
  border-color: var(--accent-color);
  outline: none;
}

.submit-btn {
  width: 100%;
  padding: 12px;
  border: none;
  border-radius: 20px;
  background: var(--accent-color);
  color: white;
  font-size: 16px;
  cursor: pointer;
  transition: opacity 0.3s;
}
```

```
.submit-btn:hover {
  opacity: 0.9;
}

.toggle-link {
  margin-top: 20px;
  color: var(--secondary-text);
}

.toggle-link a {
  color: var(--accent-color);
  text-decoration: none;
  cursor: pointer;
}
```

Код HTML розмітки компонента app-login-page

```
<div class="box">
  <app-theme-toggle class="theme-toggle-btn"></app-theme-toggle>

  <form (ngSubmit)="onSubmit()" class="auth-form"
[formGroup]="authForm">
    <div class="form-group">
      <input type="email" class="form-input" formControlName="email"
placeholder="Електронна пошта" required>
    </div>
    <div class="form-group">
      <input type="password" class="form-input"
formControlName="password" placeholder="Пароль" required>
    </div>
    <button type="submit" class="submit-btn">Увійти</button>
    <div class="toggle-link">
```

```
      Немає акаунта? <a (click)="goToRegister()">Зареєструватися</a>
    </div>
  </form>
</div>
```

Код TypeScript компонента app-login-page

```
import { Component, inject } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule, Validators } from '@angular/forms';
import { Router } from '@angular/router';
import { ThemeToggleComponent } from '../../common-ui/theme-toggle/theme-toggle.component';
import { AuthService } from '../../auth/auth.service';



@Component({
  selector: 'app-login-page',
  imports: [
    ReactiveFormsModule,
    ThemeToggleComponent
  ],
  templateUrl: './login-page.component.html',
  styleUrl: './login-page.component.css'
})

export class LoginPageComponent {

  router = inject(Router)

  goToRegister() {
    this.router.navigate(['/registration']);
  }
```

```
  authService = inject(AuthService)

  authForm = new FormGroup ({
    email: new FormControl('', {
      validators: [Validators.required, Validators.email],
      nonNullable: true
    }),
    password: new FormControl('', {
      validators: Validators.required,
      nonNullable: true
    })
  })

  onSubmit() {
    if (!this.authForm.valid) return;

    const { email, password } = this.authForm.value as {
      email: string;
      password: string;
    };

    this.authService.login(email, password).subscribe({
      next: () => this.router.navigate(['/']),
      error: err => console.error('Login error', err)
    });
  }
}
```

Код стилів компонента app-home-page

```
.main-wrraper {
    display: flex;
    gap: 40px;
    margin: 20px;
```

```
    height: calc(100vh - 40px);
    min-width: 600px;
}
```

Код HTML розмітки компонента app-home-page

```
<div class="main-wrraper">
    <app-sidebar></app-sidebar>
    <app-chat-window *ngIf="chatService.isChatOpen()" />
</div>
```

Код TypeScript компонента app-home-page

```
import { Component, inject } from '@angular/core';
import { ChatService } from '../../services/chat.service';
import { SidebarComponent } from '../../common-
ui/sidebar/sidebar.component';
import { ChatWindowComponent } from '../../common-ui/chat-window/chat-
window.component';
import { CommonModule } from '@angular/common';

@Component({
  selector: 'app-homepage-page',
  standalone: true,
  imports: [CommonModule, SidebarComponent, ChatWindowComponent],
  templateUrl: './homepage-page.component.html',
  styleUrl: './homepage-page.component.css'
})
export class HomepagePageComponent {
  chatService = inject(ChatService);
}
```

Код стилів компонента app-theme-toggle

```
:host-context(.dark-theme) .sun-icon {
  display: none;
}

:host-context(.dark-theme) .moon-icon {
  display: block;
}

.theme-toggle-btn {
  background: none;
  border: none;
  cursor: pointer;
  padding: 8px;
  transition: opacity 0.3s;
  display: flex;
  align-items: center;
  justify-content: center;
  gap: 8px;
}

.theme-toggle-btn:hover {
  opacity: 0.8;
}

.moon-icon {
  display: none;
}
```

Код HTML розмітки компонента app-theme-toggle

```
<button class="theme-toggle-btn" (click)="toggleTheme()">
    <span class="sun-icon">🌞</span>
    <span class="moon-icon">🌙</span>
```

```
</button>
```

Код TypeScript компонента app-theme-toggle

```
import { Component, OnInit} from '@angular/core';

@Component({
  selector: 'app-theme-toggle',
  imports: [],
  templateUrl: './theme-toggle.component.html',
  styleUrl: './theme-toggle.component.css'
})
export class ThemeToggleComponent implements OnInit {
  ngOnInit() {
    if (localStorage.getItem('theme') === 'dark') {
      document.body.classList.add('dark-theme');
    }
  }

  toggleTheme() {
    document.body.classList.toggle('dark-theme');

    const isDark = document.body.classList.contains('dark-theme');
    localStorage.setItem('theme', isDark ? 'dark' : 'light');
  }
}
```

Код стилів компонента app-sidebar

```
:host {
  flex: 0 1 30%;
  min-width: 200px;
  max-width: 350px;
}

.side-bar {
```

```
  display: flex;
  flex-direction: column;
  height: 100%;
  flex: 0 1 30%;
  min-width: 200px;
  max-width: 350px;
  background-color: var(--chat-bg);
  border-radius: 25px;
  box-shadow: 0 2px 8px var(--shadow-color);
  backdrop-filter: blur(5px);
  overflow: hidden;
}

.search-header {
  display: flex;
  align-items: center;
  padding: 10px;
  gap: 8px;
}

.logout-button {
  width: 32px;
  height: 32px;
  color: var(--accent-color);
  transition: color 0.2s, transform 0.1s;
  cursor: pointer;
  flex-shrink: 0;
}

.search-form {
  flex: 1;
  display: flex;
```

```
}

.search_chat {
  flex: 1;
  padding: 12px 15px;
  border: 1px solid var(--border-color);
  border-radius: 20px;
  outline: none;
  font-size: 14px;
  background-color: var(--input-bg);
  color: var(--text-color);
  transition: border-color 0.3s, box-shadow 0.3s;
  margin: 0;
}

.search_chat:focus {
  border-color: var(--accent-color);
  box-shadow: 0 0 0 2px rgba(0, 123, 255, 0.25);
}

.chat-list {
  flex: 1;
  overflow-y: auto;
  min-height: 0;
  margin-top: 5px;
  background-color: rgba(255, 255, 255, 0.1);
  width: 100%;
  box-sizing: border-box;
}

app-theme-toggle,
.theme-toggle {
```

```
  display: flex;
  align-items: center;
  justify-content: center;
  height: 32px;
  width: 32px;
  flex-shrink: 0;
}
```

Код HTML розмітки компонента app-sidebar

```
<div class="side-bar">
  <div class="search-header">
    <app-log-out-button class="logout-button"></app-log-out-button>
    <form class="search-form" [formGroup]="searchChatForm">
      <input class="search_chat" formControlName="userLogin"
placeholder="Пошук">
    </form>
    <app-theme-toggle class="theme-toggle"></app-theme-toggle>
  </div>
  <ul class="chat-list">
    <app-chat-card *ngFor="let user of
searchUserService.searchedUsers()" [user]="user" ></app-chat-card>
    <li *ngIf="
      searchUserService.searchedUsers().length === 0 &&
      searchChatForm.get('userLogin')?.value
    ">
    Такого користувача не знайдено
    </li>
  </ul>
</div>
```

Код TypeScript компонента app-sidebar

```
import { Component, inject } from '@angular/core';
import { ChatCardComponent } from '../chat-card/chat-card.component';
import { FormBuilder, ReactiveFormsModule } from '@angular/forms';
import { debounceTime, switchMap } from 'rxjs';
```

```
import { UserSearchService } from '../../services/user-
search.service';

import { takeUntilDestroyed } from '@angular/core/rxjs-interop';

import { CommonModule } from '@angular/common';

import { LogOutButtonComponent } from '../log-out-button/log-out-
button.component';

import { ThemeToggleComponent } from '../theme-toggle/theme-
toggle.component';


@Component({
  selector: 'app-sidebar',
  imports: [
    CommonModule,
    LogOutButtonComponent,
    ThemeToggleComponent,
    ChatCardComponent,
    ReactiveFormsModule
  ],
  templateUrl: './sidebar.component.html',
  styleUrl: './sidebar.component.css'
})

export class SidebarComponent {
  fb = inject(FormBuilder)
  searchUserService = inject(UserSearchService)



  searchChatForm = this.fb.group({
    userLogin: ['']
  })


  constructor() {
    this.searchChatForm.valueChanges.pipe(
```

```
      debounceTime(300),
      switchMap(formValue => {
        return this.searchUserService.searchUser(formValue)
      }),
      takeUntilDestroyed()
    ).subscribe();
  }
}
```

Код стилів компонента app-message-container

```
:host {
    min-width: 10%;
    max-width: 45%;
    width: auto;
    margin: 10px;
    padding: 10px;
}

.message {
    font-family: Arial, sans-serif;
    font-size: 14px;
    line-height: 1.5;
    word-wrap: break-word;
    min-width: 10%;
    max-width: 45%;
    width: auto;
    margin: 10px;
    padding: 10px;
    border: 1px solid var(--border-color);
    border-radius: 10px;
    background-color: var(--message-bg);
    color: var(--text-color);
    transition: background-color 0.3s, border-color 0.3s;
```

```
}

.my-message {
    justify-self: end;
    background-color: var(--accent-color);
    color: white;
    border-color: transparent;
}

.received-message {
    justify-self: start;
}
```

Код HTML розмітки компонента app-message-container

```
<div class="message" [ngClass]="messageType()"
    (mouseenter)="onMouseEnter()"
    (mouseleave)="onMouseLeave()"
    (click)="onClick()"
    [title]="shouldMask ? 'Текст приховано. Натисни для постійного перегляду.' : ''"
    style="cursor: pointer;"
>
    <ng-container *ngIf="shouldMask; else showText">
      {{ maskedText }}
    </ng-container>
    <ng-template #showText>
      {{ text }}
    </ng-template>
</div>
```

Код TypeScript компонента app-message-container

```
import { Component, computed, inject, Input } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Auth } from '@angular/fire/auth';
```

```
@Component({
  selector: 'app-message-container',
  imports: [
    CommonModule
  ],
  templateUrl: './message-container.component.html',
  styleUrl: './message-container.component.css'
})
export class MessageContainerComponent {
  auth = inject(Auth);

  @Input() messageID: string | null = null;
  @Input() sender!: string;
  @Input() text!: string;
  @Input() sentAt!: string;
  @Input() status: string = 'ok';

  isRevealed = false;
  isHovered = false;

  messageType = computed(() => {
    const currentUid = this.auth.currentUser?.uid;
    if (!currentUid) return 'receive-message';
    return this.sender === currentUid ? 'my-message' : 'received-
message';
  });

  get maskedText(): string {
    return this.text.replace(/./g, '*');
  }

  get shouldMask(): boolean {
```

```
    const currentUid = this.auth.currentUser?.uid;
    return this.status === 'blocked' && this.sender !== currentUid &&
!this.isRevealed && !this.isHovered;
  }

  onClick() {
    if (this.status === 'blocked' && this.sender !==
this.auth.currentUser?.uid) {
      this.isRevealed = !this.isRevealed;
    }
  }

  onMouseEnter() {
    if (this.status === 'blocked' && this.sender !==
this.auth.currentUser?.uid && !this.isRevealed) {
      this.isHovered = true;
    }
  }

  onMouseLeave() {
    if (this.status === 'blocked' && this.sender !==
this.auth.currentUser?.uid && !this.isRevealed) {
      this.isHovered = false;
    }
  }
}
```

Код стилів компонента app-log-out-button

```
.logout-button {
  background: none;
  border: none;
  padding: 8px;
  cursor: pointer;
  position: relative;
  width: 32px;
```

```
  height: 32px;
}

.logout-button::before {
  content: "";
  position: absolute;
  top: 50%;
  left: 4px;
  width: 16px;
  height: 16px;
  border: 2px solid currentColor;
  border-radius: 2px;
  transform: translateY(-50%);
}

.logout-button::after {
  content: "";
  position: absolute;
  top: 50%;
  right: 4px;
  width: 0;
  height: 0;
  border-top: 6px solid transparent;
  border-bottom: 6px solid transparent;
  border-left: 8px solid currentColor;
  transform: translateY(-50%);
}

.logout-button {
  color: #333;
}
.logout-button:hover {
```

```
  color: #000;
}
```

Код HTML розмітки компонента app-log-out-button

```
<button type="button"class="logout-button"aria-
label="Logout"(click)="onLogout()"></button>
```

Код TypeScript компонента app-log-out-button

```
import { Component, inject } from '@angular/core';
import { AuthService } from '../../auth/auth.service';
import { Router } from '@angular/router';

@Component({
  selector: 'app-log-out-button',
  templateUrl: './log-out-button.component.html',
  styleUrls: ['./log-out-button.component.css']
})
export class LogOutButtonComponent {
  private authService = inject(AuthService);
  private router      = inject(Router);

  onLogout(): void {
    this.authService.logout().subscribe({
      next: () => this.router.navigate(['/login']),
      error: err => console.error('Logout error', err)
    });
  }
}
```

Код стилів компонента app-chat-window

```
:host {
    display: flex;
    flex: 1 1 600px;
    min-width: 400px;
}
```

```
.current-chat-window {
    flex: 1 1 600px;
    display: flex;
    flex-direction: column;
    background-color: var(--bg-color);
    border-radius: 25px;
    padding: 0;
    box-shadow: 0 2px 8px var(--shadow-color);
    min-width: 400px;
    overflow: hidden;
}

.chat-info {
    min-height: 60px;
    width: 60%;
    max-width: 400px;
    margin: 0 auto;
    background-color: var(--chat-info-bg);
    border-radius: 25px;
    display: flex;
    align-items: center;
    padding: 20px 60px 20px 20px;
    transition: background-color 0.3s;
    position: relative;
}

.messages-container {
    flex: 1;
    overflow-y: auto;
    width: 60%;
    margin: 0 auto;
```

```
    padding: 20px;
    /*background-color: var(--bg-color);*/
}

.message-form {
    margin: 0 auto;
    width: 60%;
}



.input-message {
    width: 100%;
    padding: 15px 20px;
    border: 1px solid var(--border-color);
    border-radius: 25px;
    outline: none;
    position: sticky;
    background: var(--input-bg);
    color: var(--text-color);
    box-shadow: 0 -2px 10px var(--shadow-color);
    transition: border-color 0.3s, box-shadow 0.3s;
}

.input-message:focus {
    border-color: var(--accent-color);
    box-shadow: 0 0 0 2px rgba(0, 123, 255, 0.25);
}
```

Код HTML розмітки компонента app-chat-window

```
<div class="current-chat-window">
    <div class="chat-info">
      <h3 class="chat-info-username" *ngIf="selectedUser() as user">{{
user.username }}</h3>
    </div>
```

```
    <div class="messages-container" id="messagesContainer">
      <app-message-container *ngFor="let message of messages"
[messageID]="message.id" [sender]="message.sender"
[text]="message.text" [sentAt]="message.sentAt"
[status]="message.status"></app-message-container>
    </div>
    <form class = "message-form" [formGroup] = "inputMessageForm"
(ngSubmit)="onSubmit()">
      <input class = "input-message" formControlName = "inputMessage">
    </form>
</div>
```

Код TypeScript компонента app-chat-window

```
import { Component, computed, effect, EffectRef, OnInit, OnDestroy }
from '@angular/core';
import { MessageContainerComponent } from '../message-
container/message-container.component';
import { ChatService } from '../../services/chat.service';
import { FormBuilder, ReactiveFormsModule, FormGroup } from
'@angular/forms';
import { CommonModule } from '@angular/common';
import { Message } from '../../services/message.interface';
import { Auth } from '@angular/fire/auth';
import { SocketService } from '../../services/socket.service';


@Component({
  selector: 'app-chat-window',
  standalone: true,
  imports: [
    MessageContainerComponent,
    ReactiveFormsModule,
    CommonModule
  ],
  templateUrl: './chat-window.component.html',
  styleUrl: './chat-window.component.css'
```

```
})
export class ChatWindowComponent implements OnInit, OnDestroy {
  currentUserId: string | null = null;
  inputMessageForm: FormGroup;
  messages: Message[] = [];
  loading = false;
  private messagesEffectRef: EffectRef | undefined;

  constructor(
    private chatService: ChatService,
    private auth: Auth,
    private fb: FormBuilder,
    private socketService: SocketService
  ) {
    this.currentUserId = this.auth.currentUser?.uid ?? null;
    this.inputMessageForm = this.fb.group({
      inputMessage: ['']
    });

    this.messagesEffectRef = effect(() => {
      this.messages = this.chatService.messages();
      this.loading = false;
    });
  }

  selectedUser = computed(() => this.chatService.selectedUser());

  ngOnInit() {
    this.socketService.onMessage((data: Message) => {
      if (data.chatId === this.chatService.chatId) {
        let idx = -1;
        if (data.id) {
```

```
        idx = this.messages.findIndex(m => m.id === data.id);
      }
      if (idx === -1) {
        idx = this.messages.findIndex(m =>
          m.text === data.text &&
          m.sender === data.sender &&
          m.sentAt === data.sentAt &&
          m.chatId === data.chatId
        );
      }
      if (idx !== -1) {
        this.messages[idx] = { ...this.messages[idx], ...data };
      } else {
        this.showMessage(
          data.id,
          data.sender,
          data.text,
          data.sentAt,
          data.chatId,
          data.status
        );
      }
    }
  });
}

ngOnDestroy() {
  if (this.messagesEffectRef) this.messagesEffectRef.destroy();
}

showMessage(
  messageID: string | null,
```

```
    senderID: string,
    messageText: string,
    sendingTime: string,
    chatId: string,
    status: string
  ) {
    this.messages.push({
      id: messageID,
      sender: senderID,
      text: messageText,
      sentAt: sendingTime,
      chatId: chatId,
      status: status
    });
  }

  onSubmit() {
    const message = this.inputMessageForm.value.inputMessage?.trim();
    if (!message || !this.chatService.chatId) return;

    const tempMessage: Message = {
      id: null,
      sender: this.currentUserId!,
      text: message,
      sentAt: new Date().toISOString(),
      chatId: this.chatService.chatId,
      status: ''
    };

    this.inputMessageForm.reset();
    if (tempMessage.chatId) {
      this.showMessage(
```

```
        tempMessage.id,
        tempMessage.sender,
        tempMessage.text,
        tempMessage.sentAt,
        tempMessage.chatId,
        tempMessage.status
      );
      this.socketService.sendMessage(tempMessage);
    }
  }
}
```

Код стилів компонента app-chat-card

```
.chat-card {
    font-family: Arial, sans-serif;
    display: flex;
    justify-content: space-between;
    align-items: center;
    padding: 12px;
    border-bottom: 1px solid var(--border-color);
    cursor: pointer;
    user-select: none;
    transition: border-color 0.3s;
}

.chat-card-info {
    width: 80%;
    display: flex;
    flex-direction: column;
    justify-content: space-between;
    gap: 4px;
}
```

```
.chat-name {
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
    max-width: 100%;
    font-weight: bold;
    color: var(--text-color);
}

.last-message {
    white-space: nowrap;
    overflow: hidden;
    text-overflow: ellipsis;
    max-width: 100%;
    font-size: 14px;
    color: var(--secondary-text);
    transition: color 0.3s;
}

.chat-meta {
    width: 20%;
    display: flex;
    flex-direction: column;
    align-items: flex-end;
    gap: 5px;
}

.message-time {
    font-size: 14px;
    color: var(--secondary-text);
    transition: color 0.3s;
}
```

```
.unread-count {
    display: none;
    font-size: 12px;
    border-radius: 15px;
    padding: 2px 6px;
    background-color: var(--unread-bg);
    color: var(--bg-color);
    font-weight: bold;
    min-width: 20px;
    text-align: center;
    transition: background-color 0.3s;
}
```

Код HTML розмітки компонента app-chat-card

```
<li class="chat-card" (click)="openChat()">
    <div class="chat-card-info">
        <div class="chat-name">{{ user?.username }}</div>
        <div class="last-message">{{ user?.uid || 'Немає повідомлень'
}}</div>
    </div>
    <div class="chat-meta">
        <time class="message-time">{{ user?.email || '' }}</time>
        <span class="unread-count">2</span>
    </div>
</li>
```

Код TypeScript компонента app-chat-card

```
import { Component, inject, Input } from '@angular/core';
import { UserSearchResponse } from '../../services/search.interface';
import { ChatService } from '../../services/chat.service';

@Component({
  selector: 'app-chat-card',
```

```
  imports: [],
  templateUrl: './chat-card.component.html',
  styleUrl: './chat-card.component.css'
})
export class ChatCardComponent {
  @Input() user?: UserSearchResponse
  private chatService = inject(ChatService)

  openChat() {
    if (!this.user) return;
    this.chatService.openChatWith(this.user);
  }

}
```

Додаток Б – Налаштування правил бази даних Firebase

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {
    match /users/{userId} {
      allow read: if true;
      allow create: if request.auth.uid == userId;
      allow update: if request.auth.uid == userId;
    }

    match /chats/{chatId} {
      allow create: if request.auth != null
                    && request.resource.data.users is list
                    && request.auth.uid in
request.resource.data.users;

      allow read, update, delete: if request.auth != null
                                  && resource.data.users is list
```

```
                                    && request.auth.uid in
resource.data.users;


      match /messages/{messageId} {
        allow create: if request.auth != null
                        &&
exists(/databases/$(database)/documents/chats/$(chatId))
                        && request.auth.uid in
get(/databases/$(database)/documents/chats/$(chatId)).data.users;


        allow read: if request.auth != null
                       &&
exists(/databases/$(database)/documents/chats/$(chatId))
                       && request.auth.uid in
get(/databases/$(database)/documents/chats/$(chatId)).data.users;


        allow update, delete: if request.auth != null
                                    &&
exists(/databases/$(database)/documents/chats/$(chatId))
                                    && request.auth.uid in
get(/databases/$(database)/documents/chats/$(chatId)).data.users;
      }
    }
  }
}
```