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 кросплатформного багатокористувацького маркованого чату

Засвідчую, що в цій кваліфікаційній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студент гр. ІПЗ-21-2
_____ Д.О. Рябець

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-21-2 Рябцю Данилу Олександровичу

1. Тема: Розробка кросплатформного багатокористувацького маркованого чату затверджено наказом по КНУ № 200с від «10» квітня 2025 р.

2. Термін подання студентом закінченої роботи: «15» червня 2025р.

3. Вихідні дані по роботі: Розроблювана система має забезпечувати стабільну роботу навіть при великій кількості одночасних взаємодій з даними

4. Зміст пояснювальної записки (перелік питань, що їх треба розробити): У пояснювальній записці розглядається створення кросплатформного багатокористувацького застосунку у вигляді чату з функціями нотаток і тегів. Описано вибір технологій, структуру даних, логіку синхронізації, роботу інтерфейсу та механізми захисту й відновлення даних.

5. Перелік ілюстративного матеріалу: Приклад структури бази даних, знімки екрану інтерфейсу системи, схема взаємодії модулів сервісу.

Календарний план:

№	Найменування етапів кваліфікаційної роботи	Термін виконання етапів роботи
1	Розгляд літературних джерел та пошук інтернет-ресурсів з заданої тематики	25.12.24 – 01.01.25
2	Аналіз існуючих методів вирішення проблеми	02.01.25 – 13.01.25
3	Формулювання актуальності роботи і постановка завдань	14.01.25 – 30.01.25
4	Оформлення матеріалів першого розділу роботи	31.01.25 – 11.02.25
5	Створення функціональної системи та алгоритму додатку	12.02.25 – 28.02.25
6	Оформлення матеріалів другого розділу роботи	01.04.25 – 21.03.25
7	Розробка баз даних, інтерфейсу програмного забезпечення, програмних модулів	22.03.25 – 12.04.25
8	Оформлення додатків	13.04.25 – 18.04.25
9	Тестування розробленої програми	19.04.25 – 27.04.25
10	Оформлення пояснювальної записки	28.04.25 – 30.05.25

Дата видачі завдання: «15» лютого 2025 р.

Студент _____ Д. О. Рябець

Керівник роботи _____ І.А. Котов

РЕФЕРАТ

СИСТЕМА ЧАТІВ, НОТАТКИ, КРОСПЛАТФОРМНІСТЬ, СИНХРОНІЗАЦІЯ, АВТОРИЗАЦИЯ GOOGLE, GOOGLE ДИСК API, ТЕГУВАННЯ

Пояснювальна записка: 74 с., 35 рис., 2 дод., 15 джерел.

Метою кваліфікаційної роботи є створення системи чатів для збереження нотатків з підтримкою класифікації, групування та ефективного пошуку записів.

Проведений аналіз різних програмних застосунків показав які рішення інтерфейсу користувача будуть найбільш ефективними для зручного користування та швидкого аналізу записів. Для синхронізації на різних пристроях та завдяки швидкій авторизації було прийнято рішення використовувати Google Drive API, що дозволяє зберігати та обробляти дані у форматі JSON без потреби в окремому серверному сховищі.

Для впорядкування інформації впроваджено систему тегів, що дає змогу знаходити необхідні записи за ключовими словами. Створено інтерфейс користувача з мінімалістичним та адаптивним дизайном. Розроблену систему протестовано на різних пристроях з операційними системами Window, Linux та Android.

Проблематику орієнтування у системах збереження нотаток та необхідність у новому підході до складової інтерфейсу у програмному забезпеченні було розглянуто на Всеукраїнській науково-практичній WEB конференції CISN 2025.

ABSTRACT

CHAT SYSTEM, NOTES, CROSS-PLATFORM, SYNCHRONIZATION, GOOGLE AUTHORIZATION, GOOGLE DRIVE API, TAGGING

Explanatory Note: 74 pages, 35 figures, 2 appendices, 15 sources.

The goal of this qualification work is to create a chat system for storing notes with support for classification, grouping, and efficient search of entries.

The analysis of various software applications showed which user interface solutions would be the most effective for convenient use and quick analysis of records. For synchronization on different devices and due to quick authorization, it was decided to use the Google Drive API, which allows storing and processing data in JSON format without the need for a separate separate server storage.

To organize information, a tag system was implemented, enabling users to find the necessary entries using keywords. A user interface with a minimalist and adaptive design was created. The developed system was tested on different devices running Windows, Linux, and Android operating systems.

The issue of orientation in note-taking systems and the need for a new approach to the interface component in software were presented at the All-Ukrainian Scientific and Practical WEB Conference CISN 2025.

ЗМІСТ

ВСТУП

Об’єм інформації у сучасному світі, що ми отримуємо кожного дня, став непомірно великим. Через те, що мережа Інтернет дуже стрімко розвивається, користувачі отримують велику кількість даних з різних джерел. Частина цих даних може знадобитися через невизначену кількість часу і її бажано зберегти. Такі умови вимагають дуже великих затрат пам’яті, як людської так і цифрової. Ці вимоги роблять утримання інформації в пам’яті людини неможливим, а їх систематизування у звичайних застосунках нотатків стає все більш складним.

Списки і лінійні записи вже не можуть дати повноцінного комфорту запису, редагування та, що ще більш важливо, орієнтування у тій кількості інформації, що надходить до нас з різних джерел. Тому виникає необхідність у сучасних і більш інтуїтивно зрозумілих методах структурування записів. Цей метод має бути знайомим користувачам, мати можливість додавати теги та групувати нотатки. Найбільш перспективним методом, що буде задовольняти усі вимоги є створення інтерфейсу у вигляді системи чатів, що буде мати діалогову структуру, яка буде розділяти нотатки по даті, що вже надасть візуально поділений вигляд записам в одній групі, а чати дозволять групувати кожний запис як повідомлення.

Крім того, для додаткової систематизації є можливість додати систему тегів на кожен запис, щоб класифікувати їх за різними типами або цінністю Теги дозволять не тільки фільтрувати повідомлення, але й швидко знаходити потрібну інформацію серед великої кількості записів різного ступеня важливості. Зі свого боку користувач отримає зрозумілий та ефективний інструмент для зберігання інформації, у якому буде легко як і записувати так і шукати все необхідне завдяки пошуку та фільтрації.

1 ЗАВДАННЯ РОЗРОБКИ КРОСПЛАТФОРМНОГО БАГАТОКОРИСТУВАЦЬКОГО ЧАТУ З СИСТЕМОЮ ТЕГІВ

1.1. Аналіз принципів функціонування розподілених систем багатокористувацьких чатів

Усі знайомі нам багатокористувацькі чати такі як Discord, Telegram, WhatsApp, загалом мають дуже спільний як функціонал так і технічну реалізацію. І також усі вони базуються на перевірених часом архітектурних підходах, серед яких більше всього має поширення клієнт-серверна модель. Ця архітектура дозволяє поділити усі частини сервісу на різні частини основними з яких і є клієнт як користувацький інтерфейс, а також логіку всього сервісу як серверну частино. Це допомагає спростити масштабування, модифікацію та підвищує стабільність продукту. Сучасні сервіси різних типів, таких як браузери, поштові системи, соціальні мережі, месенджери та хмарні сховища засновані саме на клієнт-серверній архітектурі, бо вона вже стала стандартом в розробці багатокористувацьких сервісів.

Пов’язана така поширеність з тим, що автономна робота у більшій кількості випадків просто не можлива, або якнайменше недоцільна, тому що обмін повідомленнями, синхронізація, збереження історії чатів, забезпечення безпеки — всі ці задачі вирішуються більш ефективно через клієнт-серверну модель. В такій архітектурі сервер виступає у ролі централізованого вузла, що зберігає дані, розподіляє ресурси, реалізує автентифікацію користувачів, контролює доступ, оптимізує трафік та забезпечує стабільність.

Це особливо актуально для сервісів чату, де важливо не тільки зберегти історію повідомлень, але й забезпечити швидку передачу даних, сповіщення, синхронізацію, безпеку даних, підтримку різних технологій передачі та реалізацію інтеграцій з іншими сервісами. А також клієнт-сервер дозволяє ефективно реалізовувати функціонал сервісу навіть при великих навантаженнях і великій кількості підключень.

Як зазначено у документації MDN Web Docs [1], сутність клієнт-серверної взаємодії полягає в тому, що користувач через клієнтський інтерфейс ініціює з’єднання з сервером, відправляючи йому запити певного типу й очікує стандартизовану відповідь. Така взаємодія будучи спершу односторонньою дозволяє організовувати обмін даними між великою кількістю клієнтських зв’язків також дозволяючи використання звернення до інших серверних сервісів. У свою чергу сервер не тільки обробляє кожний запит, але й впроваджує стандартизацію відповідей для того, щоб клієнтська сторона могла інтерпретувати дані коректно не зважаючи на різницю реалізованих клієнтських застосунків.

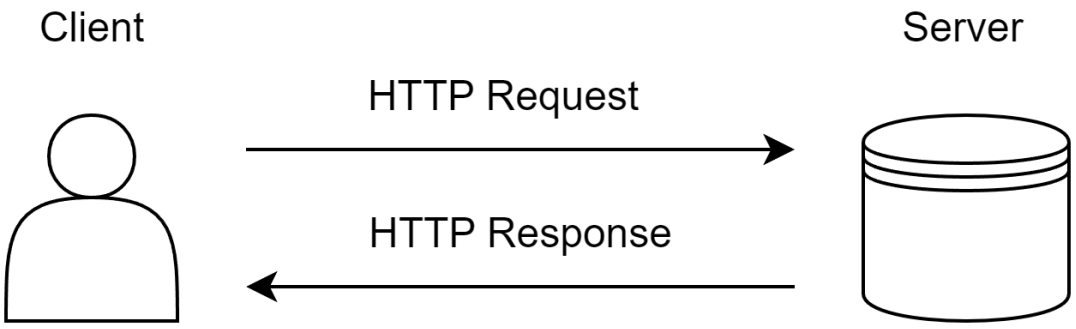


Рисунок 1.1 – Схема взаємодії клієнт-серверної моделі

Спираючись на документацію також можна стверджувати, що така модель забезпечує передбачуваність у взаємодії, тому що принцип незмінний і сам по собі є простим та передбачуваним. Клієнт відсилає запити використовуючи протоколи HTTP або HTTPS, використовує стандартизовані методі, як наприклад GET для отримання інформації, POST для відправки нових даних, PUT для оновлення або DELETE для видалення (див. рис. 1.1). Це далеко не всі методи але і їх достатньо для коректної взаємодії клієнт-серверу. Відповідаючи на запит, сервер пересилає дані у вигляді структурно оформлених об’єктів які можна інтерпретувати у інтерфейсі користувача, а

також статус відповіді, що може вказати про успішність виконанні запиту, що також можна відобразити у інтерфейсі користувача. Завдяки цьому користувач легко може отримувати інформацію, будь то очікуваний результат роботи або сповіщення про помилку у функціонуванні програми або навіть діях користувача при цьому функціонально не оброблюючи кожен тип відповіді окремо.

Важливо зазначити, що запити, хоч і мають відповідати певним стандартам заповнення, але вони не обмежені у видах інформації яку передають. Це можуть бути як текстові параметри якогось об’єкта, так і складні структури, які можуть включати дані для входу, токени для автентифікації, дані з заповнених форм, користувацькі налаштування, прикріплені об’єкти або іншу важливу для обробки інформацію, що буде використана для детальної або просто коректної відповіді сервера. В залежності від логіки та реалізації функціоналу сервісу, сервер може виконувати велику кількість різних обчислень, запитів до інших сервісів, виконувати перевірку будь то авторизації, або просто валідності даних, редагувати базу даних або запускати алгоритми обробки реагуючи тим самим на запит клієнтської частини.

Така система буквально є фундаментом для більшості сучасних веб-сервісів, де кожна з сторін виконує чіткі функції роблячи взаємодію передбачуваною, ефективною та зручною. Така модульність також не дозволяє клієнту мати доступ до внутрішньої реалізації серверної частини, але може використовувати її функціонал у тому обсязі, що передбачено розробником.

Клієнт-серверна модель є лише фундаментом для реалізації, але не основним функціоналом. Якщо розбирати саме систему багатокористувацьких чатів, сервер надає широкий спектр різного функціоналу. Авторизація і перевірка користувача, це першочергова технологія, що для користувача здається лише першим кроком, але насправді має місце серед усіх інших

технічних рішень, кожен раз надсилаючи серверу дані для виконання функцій від імені користувача. Сервер займається тим, що перевіряє дані користувача як при авторизації так і після неї, у той час коли клієнт зберігає дані для входу або токени доступу, щоб користувачу не довелось кожен перезапуск входити у систему.

Для отримання та відправки повідомлень клієнт надсилає запити на оновлення, наприклад при завантаженні сторінки, або запит на відправку, редагування чи видалення повідомлення, у такому випадку серверна частина повинна забезпечити такі функції як читання з бази даних, відправляючи їх у стандартизованому вигляді до клієнта оновлюючи вміст інтерфейсу, та оновлення бази даних для збереження та синхронізації історії чатів. При цьому база даних може містити не лише сам вміст повідомлень, а купу іншої інформації, таку як час відправки, прикріплені файли, маркування про зміну оригінального вмісту повідомлення, та іншу інформацію, яку спочатку клієнт надає через інтерфейс, а сервер зберігає та повертає у зворотному напрямку.

Особливу увагу завжди треба приділяти безпеці даних, саме для цього вже є реалізація авторизації, але цього може виявитись не достатньо. Оскільки все, що було зроблено людьми має вразливості, не можна розраховувати, що дані з сервера не попадуть до третьої сторони. Для забезпечення безпеки використовують різні методи шифрування при яких сервер не може отримати вміст повідомлень, але може передати зашифровані дані які шифруються та розшифровуються вже на стороні клієнта.

Важливо відмітити, що у сучасних месенджерах ключовою є реалізація оновлення в реальному часі. Використовуються такі технології як наприклад WebSocket, що дозволяє клієнтам отримувати оновлення від серверу без постійних оновлень сторінки. WebSocket – дозволяє встановити двосторонній зв’язок між клієнтом та сервером, що дає змогу відправляти та отримувати відповіді без постійних запитів [1]. Це дозволяє користувачу бачити зміни

майже у реальному часі і завжди бути на зв’язку якщо мова йде про багатокористувацьку систему чатів.

1.2. Аналіз програмних технологій розробки і забезпечення багатокористувацьких чатів

Сфера веб-застосунків стрімко розвивається і технології змінюють одна одну з небаченою раніше швидкістю. Якщо раніше написання кросплатформного програмного забезпечення означало велику кількість роботи та часу, то вже станом на сьогодні це є як стандартом розробки, так і легким у реалізації рішенням, для поширення програмного застосунку. Все це стало можливим через використання технологій розробки веб-сторінок для написання клієнтської частини як встановлюваного програмного забезпечення. Для кінцевого користувача зміни не помітні, але, як для розробників, це великий крок до легкого написання інтерфейсів технологіями HTML.

Основною проблемою у реалізації кросплатформних додатків є те, що для кожної операційної системи, а іноді і окремих пристроїв треба писати окремий код для повноцінного функціонування. Вони добре працюють і мають глибоку інтеграцію в систему, але таке розділення робить дуже складним як розробку самих додатків так і їх подальше оновлення. Деякі функції, що передбачені на одному пристрої можуть з’явитися через велику кількість часу і оновлень на іншому.

Прогресивний веб-додаток – це програмний встановлюваний веб-застосунок, що э модифікацією веб-сайту, що може працювати на будь яких пристроях. Посилаючись на документацію WEB.DEV, PWA не має відрізнятись від звичайних додатків на пристрої. Застосунок буде мати свій значок, буде з’являтись у пошуку додатків операційної системи (див. рис. 1.2), відкриватись не у браузері, а в повноцінному вікні як інші додатки (див. рис.

1.3), може обробляти сповіщення, посилання та інтегруватись в операційну систему і за потреби працювати без доступу до мережі [2].

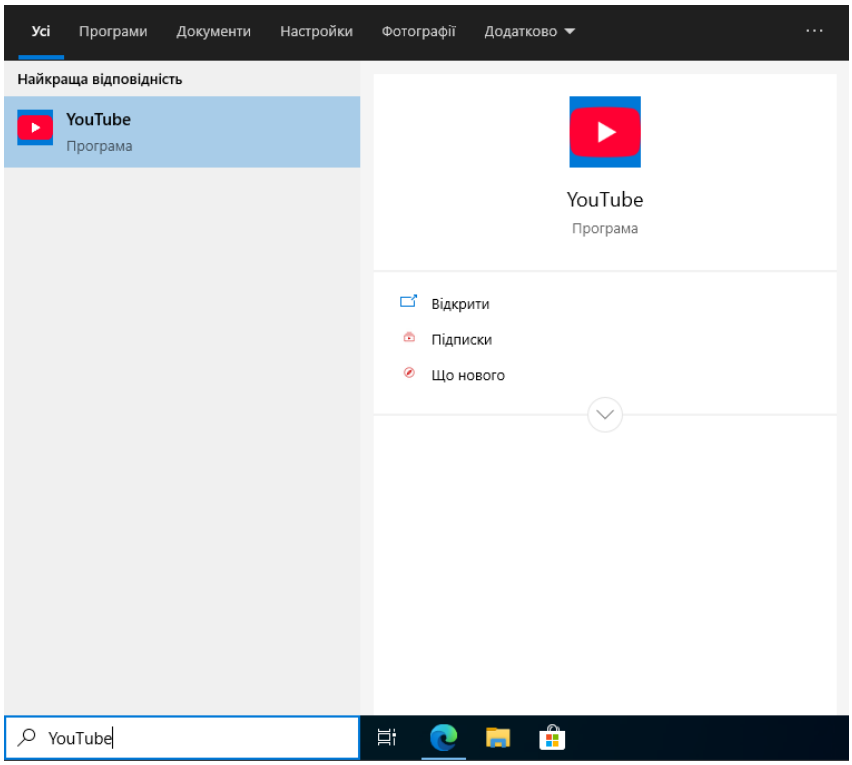


Рисунок 1.2 – Знімок екрану меню пошуку Windows 10

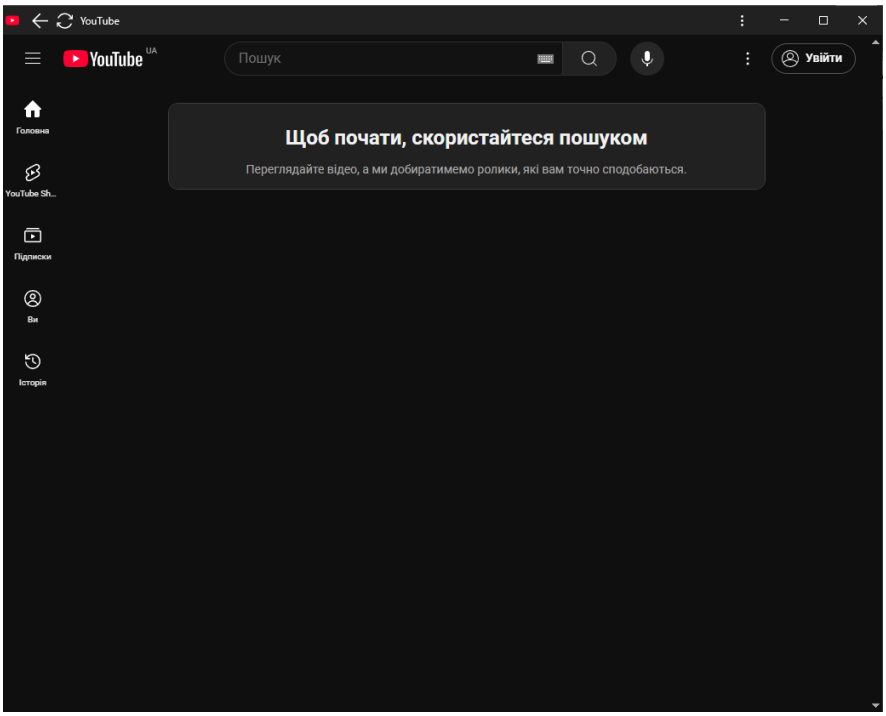


Рисунок 1.3 – Знімок екрану вікна встановлюваного додатку YouTube

Переваги такого підходу у тому, що функціональна частина додатку реалізується не для кожного пристрою окремо, а для всіх одночасно. Та частина, що повинна автономно працювати, реалізується однаковим кодом для всіх пристроїв і операційних систем, що мають підтримку PWA. Тоді коли серверна частина також функціонує без будь-яких затримок так само як на веб-сторінці. Така технологія отримує переваги як веб-сторінки, так і десктопної реалізації сервісу.

Серед усіх переваг PWA також треба відмітити легке встановлення. Реалізація додатку має такий самий набір технологій, що й розробка веб-сайту. Тобто програмний код застосунку можна писати на звичайному JavaScript, що підтримується у кожному браузері як і CSS та HTML. Оскільки розробка багатокористувацьких чатів потребує мов програмування, що дозволяють створювати динамічні системи з підтримкою обміну повідомленнями, при цьому не встановлюючи без потреби зайве програмне забезпечення, основною мовою обирається саме JavaScript. Оскільки JavaScript вже вбудований в усіх браузерах.

За серверну частину зазвичай відповідає Node.js, що дозволяє використовувати JavaScript на бекенді. Node.js – це саме середа для виконання JavaScript коду. Він працює через запуск лише ядра браузеру для виконання коду не використовуючи при цьому повноцінний браузер [3]. Основна перевага такого рішення, на відміну від звичайного JavaScript, є те, що Node.js дозволяє асинхронно виконувати код, що дозволяє будь-якій кількості клієнтів одночасно користуватись сервісом не блокуючи виконання.

Також використання Node.js має перевагу у тому, що мова написання не відрізняється від тої, що використовується на стороні клієнту. Розробник може писати весь код використовуючи лише JavaScript не вивчаючи нові мови програмування для написання серверної частини. Це спрощує розробку якщо самих розробників не велика кількість або навіть один, а також стає простіше

аналізувати різні складові сервісу будучи розробником як клієнтської так і серверної частини.

Додаткової зручності також додає менеджер пакетів npm. Вже станом на 2022 другий рік він має більше двох мільйонів пакетів, що робить його як дуже великим так і тим, що вміщує у себе модулі та бібліотеки для будь-якої задачі[3]. Цей менеджер пакетів зручний тим, що автоматично встановлює залежності, оновлення і звісно встановлює пакети до проекту, майже не вимагаючи від користувача зайвих дій.

Як приклад Vue.js має свою поширеність серед розробників і також підтримує PWA. Він надає саме повну підтримку розробки прогресивних веб-застосунків, і дозволяє розробникам розробляти додатки, що працюють на різних пристроях. У цьому фреймворку є офіційний плагін, що легко інтегрувати у проект [4]. Він додає автоматично усі компоненти PWA такі як маніфести, налаштування та сервіс-воркер, що є основним компонентом для PWA.

Головна перевага Vue.js у цій взаємодії з PWA полягає у тому, що розробник не повинен самостійно налагоджувати механізми функціонування прогресивного веб-додатку. Усі налаштування виконуються автоматично або через файл конфігурації, що дозволяє як уникнути ручного налагодження, так і взяти участь у конфігуруванні в залежності від потреб розробника. Також налаштування маніфесту в Vue.js нічим не обмежується і так само гнучко дозволяє налаштувати додаток починаючи від його значка закінчуючи темами оформлення та іншими параметрами, що необхідні для корректного відображення.

Після налаштування такого додатку, його можна встановити через будь-який браузер з підтримкою PWA. Серед таких браузерів також є Google Chrome, що дозволяє встановити будь-який додаток через відповідне меню. Судячи з інформації представленої у довідці Google Chrome, веб-сторінка може навіть не бути прогресивною. Достатньо мати веб-сайт зі своїм

посиланням, але тоді розробник не буде мати можливості впровадити встановлення додатку у інтерфейс свого додатку та налаштувати окремо елементи встановлюваного програмного комплексу [5]. А також треба відмітити, що автономна робота не передбачена у веб-сайтів, що не розроблялись як прогресивний веб-додаток. Тобто Google Chrome хоч і дозволяє встановити навіть не PWA додатки, але різниця у тому, що з прогресивним веб-додатком розробник може як налаштувати усе необхідне для його роботи за межами браузера так і зробити його автономним.

Оскільки PWA працює здебільшого на браузерній основі у вигляді звичайних додатків як Google Chrome, зрозуміло, що і збереження даних користувача реалізовується таким самим чином. У звичайних браузерах наявне локальне сховище, яким користуються розробники для збереження як короткочасних даних користувача, так і довгострокових, що також треба записувати. Виходячи з цього, стає зрозуміло, що розробка нічим не ускладнюється, а навпаки дозволяє розширити вже знайомі розробникам технології за допомогою бібліотек та фреймворків, що спрощують запис даних до локального сховища веб-сторінки, але вже на базі прогресивного веб-додатку.

Основним для збереження даних середовищем для PWA є IndexedDB. Він підтримується у будь-якому браузері, тому ніколи не виникне проблем із підтримкою прогресивного веб-застосунку. Основною перевагою IndexedDB є те, що місткість цього сховища є достатньо великою, щоб записувати велику кількість різних необхідних даних користувача для коректної роботи веб-додатку. Також підтримується асинхронна робота та автономний режим [1]. Але серед таких значних переваг є і недоліки, і це взаємодія з цим API. Він виявляється досить громіздким, і з ним стає важко працювати.

Це не є проблемою якщо використовувати бібліотеки для обгортки і взаємодії з API сховища. Dixie.js – це обгортка для IndexedDB, яка вирішує наявні проблеми у цьому API, а також надає на заміну комфортний

мінімалістичний API для взаємодії з базою даних. Серед виправлень основними є виправлення стосовно обробки помилок у IndexedDB. Оригінальний API має дуже не очевидну і важку схему обробки. У Dixie.js це було виправлено використовуючи звичні підходи сучасної розробки, з якими стикалася більшість розробників. Крім того запити в IndexedDB також реалізовані не гнучко через що дуже складно писати довгі або складні запити. А синтаксис нативного API дуже багаторівневий, у той час як Dixie.js більше схожий на взаємодію з масивами у звичайних мовах програмування [6].

1.3. Аналіз існуючих програмних комплексів багатокористувацьких чатів з системами тегів

Серед величезної кількості застосунків багатокористувацьких чатів, у яких є функціонал для класифікації чатів, на сьогоднішній день найбільшої популярності набрали такі платформи як Discord та Teleram. Підхід для систематизації у них має як і щось спільне, так і дуже багато відмінностей. На прикладі цих програм можна виявити основні закономірності та основні дизайнерські рішення у розробці чатів та їх систематизації.

Discord – це одна з найпоширеніших систем для обміну повідомленнями. Найбільшу популярність вона має через геймерів, і як стверджує статистка, що найменше 90% їх користувачів грають у відеоігри [7]. Платформа з самого початку позиціонувала себе як месенджер для геймерів, що також має свій вплив на інтерфейс. Геймерам більш за все необхідно було забезпечити не великий простір для вираження думок, а систему, що дозволить розмежувати за призначенням кожну тему для обговорення на велику кількість гілок. Ця потреба є наслідком розвитку індустрії відеоігор де кожна гра починає набирати свою аудиторію у той час як у цієї аудиторії також є велика кількість інших інтересів які можна обговорити з кимось. Розвиток платформи призвів до початку утворення навіть груп для обговорення не тільки ігор, а різних інтересів навіть не цифрових.

З іншого боку Telegram, що, на відміну від Discord, не є вузьконаправленим і не має чіткої аудиторії. Він є одним із самих популярних месенджерів, і якщо спиратись на інформацію з офіційного сайту, розробники приділяють велику увагу швидкості додатку та захисті даних [8]. Головною перевагою є те, що Telegram не відстає по кількості фукнціональних рішень, а оновлення інтерфейсу та введення різних візуальних рішень є звичними. На сьогоднішній день застосунок має усі технічні та візуальні ознаки і рішення, що є в інших месенджерах і навіть більше, що робить його гарним прикладом оформлення системи чатів.

Ключовим рішенням при розробці стала система каналів у серверах Discord. Під серверами тут мається на увазі функціонал розмежування спільнот на групи, що мають у собі різний набір чатів, які називаються каналами (див. рис. 1.4). Канали дозволяють структурувати чати за темами та функціями, що забезпечує розмежування контенту чатів на одному сервері(див. рис. 1.5). Така система дозволяє мати велику кількість серверів за набором інтересів при цьому маючи ще більшу кількість різних за призначенням каналів, при цьому ефективно орієнтуватись у цьому просторі.

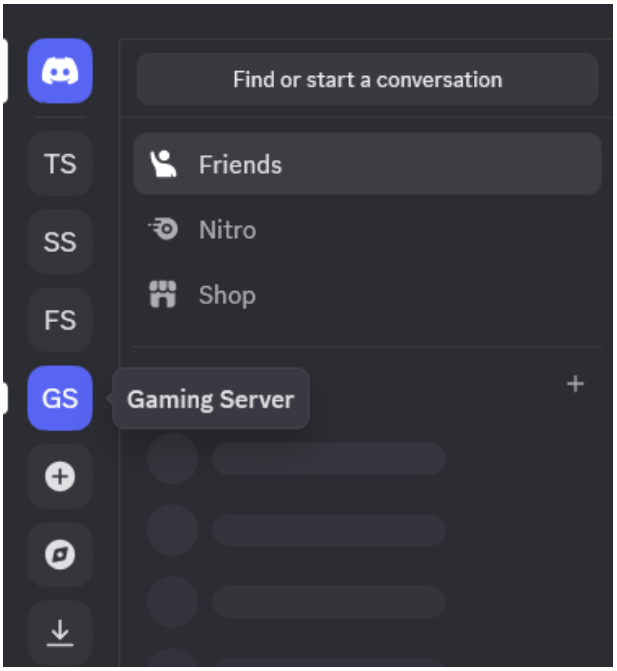


Рисунок 1.4 – Знімок екрану списку серверів програми Discord

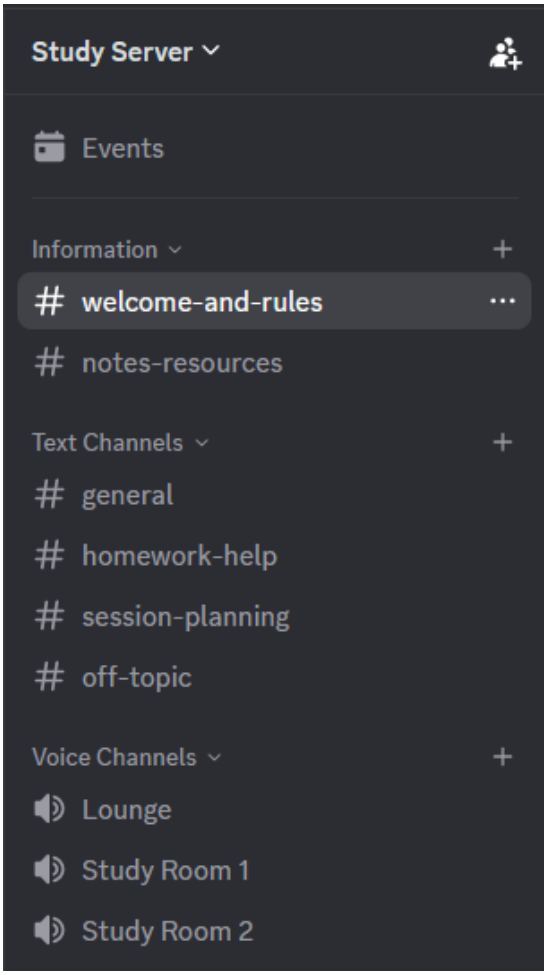


Рисунок 1.5 – Знімок екрану списку каналів програми Discord

Зі сторони Telegram, останні версії набули функції розділяти групи на гілки (Topics), тобто на різні чати. Канал у Telegram, як і сервер в Discord, це станом на зараз група чатів, до яких має доступ певна кількість людей (див. рис. 1.6). До оновлення, що додає гілки – сервер Discord та канал у Telegram не були так схожі, тому що канал мав лише один чат який не можна було переключити і вже більш був схожий на звичайний чат для великої кількості людей. Гілки дозволяють мати в одному каналі або групі вести обговорення на різні теми або зберігати важливу інформацію, що стосується предмету обговорення (див. рис. 1.7). Ця технологія дуже схожа на систему серверів та

каналів Discord. Станом на зараз, вони дійсно не сильно відрізняються, а єдина не візуальна різниця є лише у технічній реалізації збереження повідомлень.

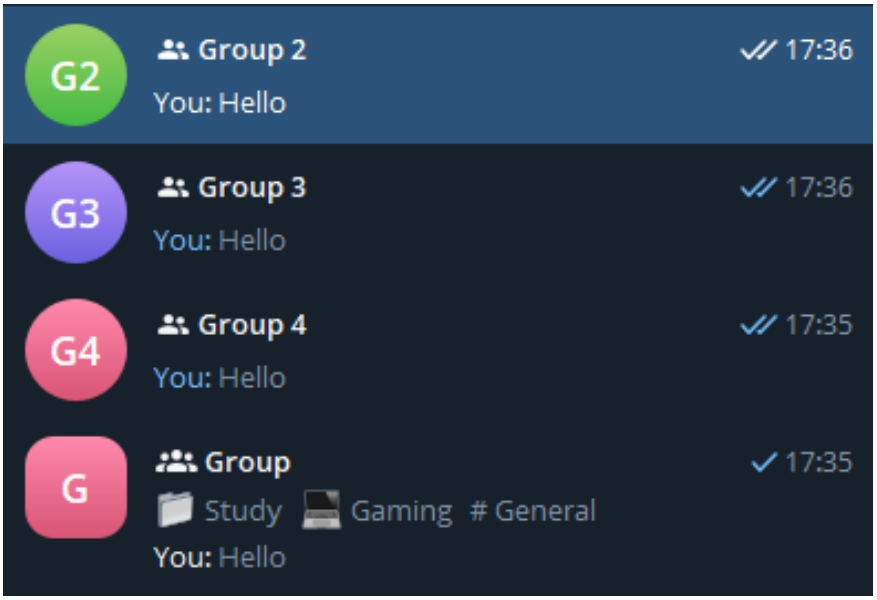


Рисунок 1.6 – Знімок екрану списку каналів програми Telegram

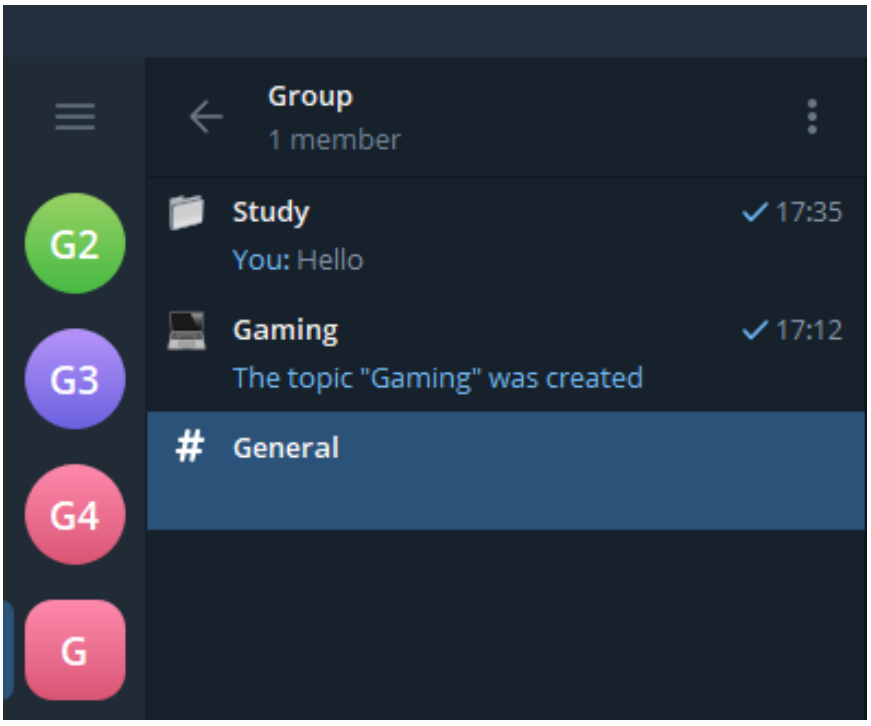


Рисунок 1.7 – Знімок екрану списку гілок програми Telegram

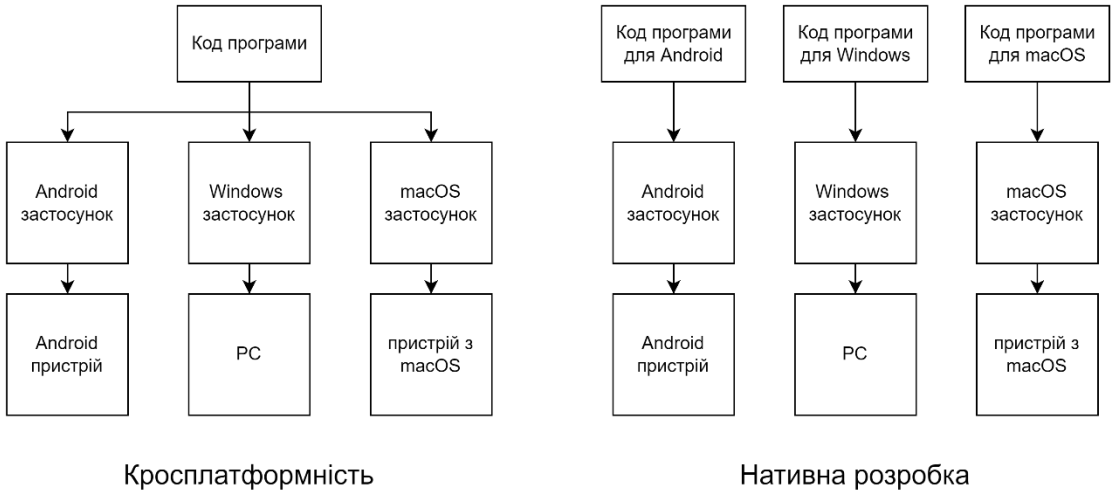
Такі системи, що розмежують і систематизують теми, дуже добре підходять для розробки системи збереження нотаток. Коли нотатки

систематизовані у вигляді чату з окремою темою, користувач легко може зорієнтуватись у великій кількості інформації.

1.4. Аналіз сучасних принципів розробки кросплатформних програмних комплексів

Для того щоб розібратись у тому, які саме є методи розробки кросплатформних програмних застосунків, спочатку треба сформулювати означення для самого терміну кросплатформності. Кросплатформність або багатоплатформність – це здатність програмного забезпечення реалізовуватись на більш ніж одній платформі. Тобто працювати на двох або більше операційних системах чи інших платформах здатних реалізовувати програми. Ця здібність досягалась протягом років різними підходами і методами, зі своїми перевагами і недоліками.

Першим підходом і хронологічно – є нативна кросплатформна розробка. Це метод розробки програмних застосунків з використанням мов програмування, що підтримуються окремими операційними системами і розмежовувати розробку на кожен пристрій. Основною проблемою є кількість роботи, бо на кожен пристрій треба написати свій код застосунку, а крім того мови можуть дуже сильно відрізнятись вже не говорячи про різницю у бібліотеках. Цей метод важко назвати кросплатформним через те, що розробка застосунку під різні пристрої відбувається окремо (див. рис. 1.8)



Кросплатформність Нативна розробка

Рисунок 1.8 – Схеми методів розробки кросплатформних додатків

Іншим вже описаним методом є використання прогресивних веб-застосунків. Це дозволяє уникнути таких проблем як сумісність, та дистрибуція програмного забезпечення, а реалізація для кожного пристрою практично не відрізняється та не відокремлюється від основного коду програмного застосунку, але має проблему з глибиною взаємодії з операційною системою.

Існує також підхід до розробки програм, що базується на використанні браузерного середовища або спеціалізованих фреймворків, таких як Electron або React Native. Ці технології дозволяють створювати додатки, що можуть працювати на різних платформах не вимагаючи написання коду під кожну систему.

Electron – це фреймворк, що дозволяє створювати десктопні застосунки за допомогою вже звичних веб-технологій як JavaScript, HTML та CSS. За відображення інтерфейсу відповідає двигун Chromium, а за виконання логіки застосунку – середа Node.js [9]. Якщо код у обраному фреймворку можна скомпілювати у звичайну HTML сторінку з кодом JavaScript, то Electron без проблем запустить її на любому пристрої.

З іншого боку React Native використовує компоненти фреймворка React. Тут важливо з самого початку будувати додаток з урахуванням особливостей React Native, впроваджуючи його елементи та стилі протягом усього проекту. Теоретично можливо використовувати React Native без React, але розробники не рекомендують цього робити, оскільки набагато простіше і ефективніше зосередитися на розробці додатку, а не на побудові архітектури з нуля [10].

API – це інтерфейс взаємодії між різними програмними компонентами. API дозволяє одному програмному комплексу запитувати у іншого певні дані, не маючи представлення про реалізовану логіку запитуваного (див. рис. 1.9).

У кросплатформній розробці API грає ключову роль, забезпечуючи доступ до зовнішніх сервісів, базам даних та функціям пристроїв.

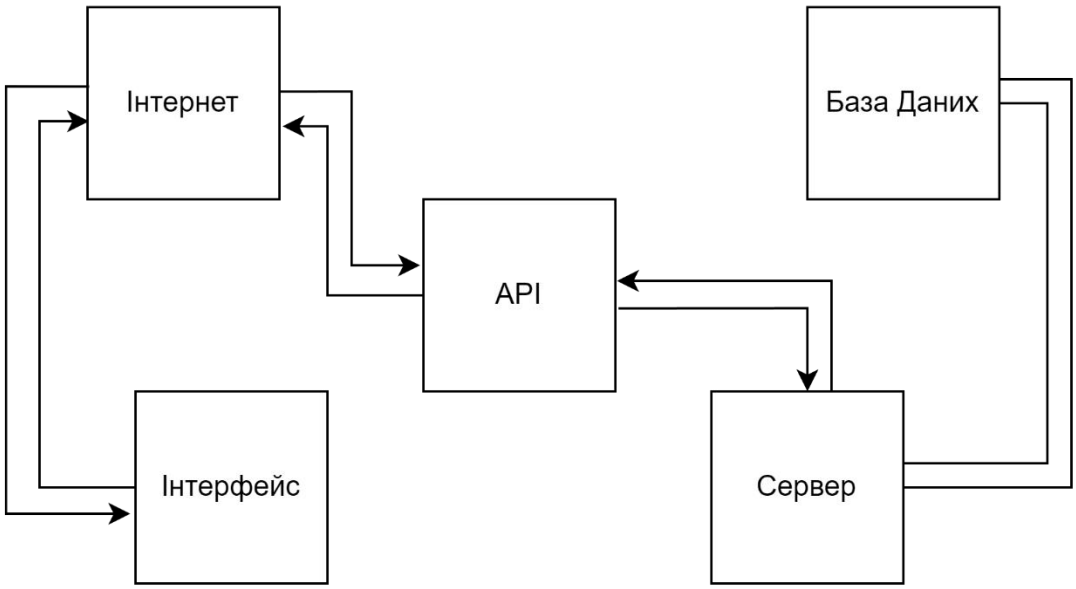


Рисунок 1.9 – Схема взаємодії API

Функціональність кросплатформних програмних застосунків більш за все залежить від їх здатності взаємодіяти з зовнішніми джерелами даних, та розширювати власні можливості через бібліотеки. Саме інтеграція API і підключення модулів дозволяє реалізовувати сучасні вимоги до користувацьких застосунків. У фреймворку Electron в поєднанні з Node.js розробнику надається доступ до великої кількості функцій, що дозволяє створювати додатки, що можуть взаємодіяти з локальними файлами, процесами, мережею та іншими елементами операційної системи. Доступ до зовнішніх сервісів реалізується за стандартними механізмами HTTP-запитів. Для цього можуть використовуватися як вбудовані модулі, так і популярні JavaScript бібліотеки, що забезпечує гнучкість розроблюваних рішень.

У свою чергу React Native надає зручні засоби для роботи з API при розробці застосунків. Використання фреймворку React дозволяє підключити набагато більше різних функціональних інструментів, що дозволяють надсилати запити, оброблювати відповіді і керувати мережевими помилками.

А встановлення та підключення різних модулів відбувається через стандартні менеджери пакетів, що спрощує доступ до готових рішень.

Не дивлячись на велику кількість переваг у кросплатформній розробці, вона не позбавлена певних труднощів, що необхідно враховувати при виборі архітектурного підходу.

Одним з найбільш важливих недоліків є значне споживання системних ресурсів, особливо якщо казати про Electron, оскільки кожен додаток на його основі працює на вбудованому браузері, і реалізовує веб-сторінку. Це має вплив як на розмір виконуємого файлу, так і на завантаженість оперативної пам’яті. Такі додатки менш ефективні з точки зору споживання ресурсів, а також можуть працювати повільніше ніж нативні аналоги.

Якщо казати про React Native, то основні складнощі виникають з необхідності працювати з нативним кодом, додаючи специфічний функціонал. Не дивлячись на обширну підтримку зі сторони бібліотек та модулів, деякі функції платформ потребують взаємодії з такими мовами як наприклад Java, що не дуже фігурує у розробці веб-додатків і може створити лишню ускладненість у розробці та повисити вимоги до розробників.

Також слід враховувати, що відображення інтерфейсу буде обмежене. Кросплатформні застосунки через різницю у рішеннях зовнішнього вигляду на різних пристроях не зможуть з точністю відповідати візуальним стандартам платформ для яких розроблюються. Це більш критично для мобільних платформ, де різниця між рішеннями у інтерфейсі дуже впливає на сприйняття користувачем. Інтерфейс може виглядати не звично і вибиватись з загального плану. Треба також згадати про складності з оновленням залежностей. При використанні сторонніх бібліотек часто можуть виникати конфлікти версій, відсутність підтримки або невідповідність новим вимогам платформ.

Таким чином, кросплатформні рішення являють собою хоч і дуже зручний інструмент для розробки широкого спектру різних програмних

застосунків, але потребують обережного підходу при проектуванні, та більш ретельної оцінки потенційних обмежень

1.5 Постановка завдання розробки кросплатформного багатокористувацького маркованого чату з системою тегів

У ході аналізу різних систем та методів розробки було прийнято рішення реалізовувати користувацький інтерфейс у вигляді SPA, або односторінкового додатку. Такий підхід дозволяє забезпечити безперервну взаємодію користувача з додатком без перезавантажень, що особливо важливо для системи чатів, де є вимога до відгуку на дії користувача та швидкість оновлення даних.

Як базова технологія реалізації даного типу інтерфейсу було обрано використання React, що є популярним сучасним рішенням для побудови інтерфейсів, що дозволяє організовувати структуру застосунку у вигляді компонентів. Підхід з компонентами є дуже зручним та більше схожий на взаємодію у звичайних мовах програмування. Також це дозволяє ефективно розмежувати відповідальність різних частин у рамках інтерфейсу. Перевага також є у тому, що побудова та розширення інтерфейсу, а також функціоналу, спрощується через компонентний підхід.

Важливим аргументом на користь React стала його адаптивність до різних платформ. Це є важливим фактором при розробці оскільки додаток призначений для кросплатформного використання і має гарно працювати на будь-яких пристроях. Як технологія веб-розробки, React дозволяє уникнути прив’язки до конкретної операційної системи чи браузера, а інтерфейс, що будується на його основі, однаково відображається на всіх веб-двигунах.

React дозволить реалізувати інтерфейс, що буде швидко реагувати на зміни станів, забезпечуючи плавну взаємодію без необхідності у великій кількості нагромаджень у коді, та постійних завантажень, або складних переходів між сторінками. Так як уся логіка додатка сконцентрована на

взаємодії з чатами, повідомленнями та тегами, було важливо, щоб елементи інтерфейсу моментально оновлювались згідно з діями користувача. Так React виявився найбільш зручною середою для створення динамічної структури.

Для створення кросплатформного застосунку було вирішено використовувати Electron, що дозволяє запускати веб застосунки у вигляді повноцінних десктопних програм. Головною причиною для вибору саме цієї технології була легка взаємодія з системними викликами, що підтримує наприклад системні діалоги. Застосунок також можна запускати у фоновому режимі, зберігати стан між сесіями, відкривати файли через системні меню та інтегрувати повідомлення у трей. Все це впливає на сприйняття застосунку як повноцінної десктопної програми.

Однією з головних причин вибору Electron стало бажання забезпечити стабільність, автономність та єдиний користувацький досвід, незалежно від платформи. Також важливим фактором є простота дистрибуції. Electron дозволяє створити окремий інсталяційний файл, який можна поширювати серед користувачів без попереднього встановлення залежностей. Забезпечення автономності дає змогу встановити програму та одразу почати працювати незалежно від операційної системи на якій реалізовується додаток, будь то Windows чи Linux.

Завдяки цій технології, користувач отримує інтерфейс, що буде виглядати на усіх пристроях однаково і поводитись як нативна програма, з можливістю запуску без браузера, становлення через інсталятор, створення ярликів, підтримка фонового режиму та взаємодія з файловою системою.

Таким чином, для забезпечення кросплатформності додатку, Electron став оптимальним вибором для реалізації проекту, оскільки дозволяє створити універсальний, простий у встановленні та використанні застосунок, який буде виглядати та працювати однаково добре на різних операційних системах. Це особливо важливо в умовах відсутності серверної архітектури та прагнення зробити додаток максимально доступним для користувача.

Проаналізувавши технічні особливості, архітектурні вимоги та специфіку реалізації багатокористувацького маркованого чату з системою тегів, можна виділити такі основні задачі, що мають бути реалізовані в рамках розробки програмного комплексу:

1. Створити структурну модель архітектури, що демонструє взаємозв’язок між клієнтським додатком, Google API та хмарним сховищем;
2. Розробити функціональну схему взаємодії користувача з застосунком
3. Спроектувати логіку зберігання даних у форматі JSON;
4. Побудувати алгоритми основної логіки взаємодії користувача з програмним комлексом;
5. Розробити модель адаптивного користувацького інтерфейсу програмного комплексу;
6. Реалізувати клієнтський інтерфейс у вигляді веб-додатку з використанням сучасних технологій;
7. Налаштувати механізм роботи з Google Drive API. Реалізувати синхронізацію, збереження та авторизацію;
8. Створити програмні модулі для автономної взаємодії з нотатками та списком чатів;
9. Додати функціонал керування тегами, серед яких прив’язка до повідомлень, редагування та фільтрація повідомлень;
10. Реалізувати підтримку нотаток як окремого модуля з керуванням записами у чатах.

Архітектура додатка має повністю обійтись без серверної сторони, що також значить часткову або повну автономність додатку. Це дозволяє користувачу зберігати всі дані локально і при можливості у власному хмарному середовищі без передачі їх на стороні сервери. Такий підхід підвищує рівень приватності, знижує залежність від інтернет-з’єднання та усуває необхідність у постійному обслуговуванні серверної частини.

2 РОЗРОБКА ФУНКЦІОНАЛЬНИХ МОДЕЛЕЙ І АЛГОРИТМІВ БАГАТОКОРИСТУВАЦЬКОГО ЧАТУ З СИСТЕМОЮ ТЕГІВ

2.1. Розробка структурної моделі маркованого чату з системою тегів

Сучасні веб-застосунки часто поєднують гнучкий інтерфейс, автономність і взаємодію з зовнішніми сервісами. Це вже скоріш стало стандартом для розробки. У випадку цього проекту розглядається односторінковий додаток на React, який реалізує модель чату зі збереженням повідомлень у вигляді нотаток, де кожне повідомлення може мати систему тегів.

Цікавою особливістю архітектури є відмова від особистого серверу. Уся логіка додатку буде реалізовуватися лише на стороні клієнту, що буде зменшувати витрати на підтримку. Такий підхід робить систему більш стійкою до різних обставин, і уникає проблем з підтримкою свого власного хостингу. А оскільки користувацькі дані не передаються на сторонні сервери, окрім Google Drive, безпека даних менше стає проблемою, а сам сервіс має велику ступінь довіри серед користувачів.

Програма працює автономно на стороні клієнту та як з серверною стороною взаємодіє лише з Google Drive API для синхронізації даних. Це дозволяє уникнути потреби у власному сервері, зберігаючи при цьому повну функціональність багатокористувацького застосунку і розробляти систему лише з урахуванням автономної роботи приділяючи увагу лише відправці даних на хмарне сховище.

Щоб побудувати схему взаємодії між частинами додатку та розібратись у структурі вихідного проекту можна побудувати діаграму розгортання (див. рис. 2.1). На схемі видно, що основними об’єктами додатку є автономна логіка та інтерфейс програми, що має з’єднання лише з API хмарного сховища.

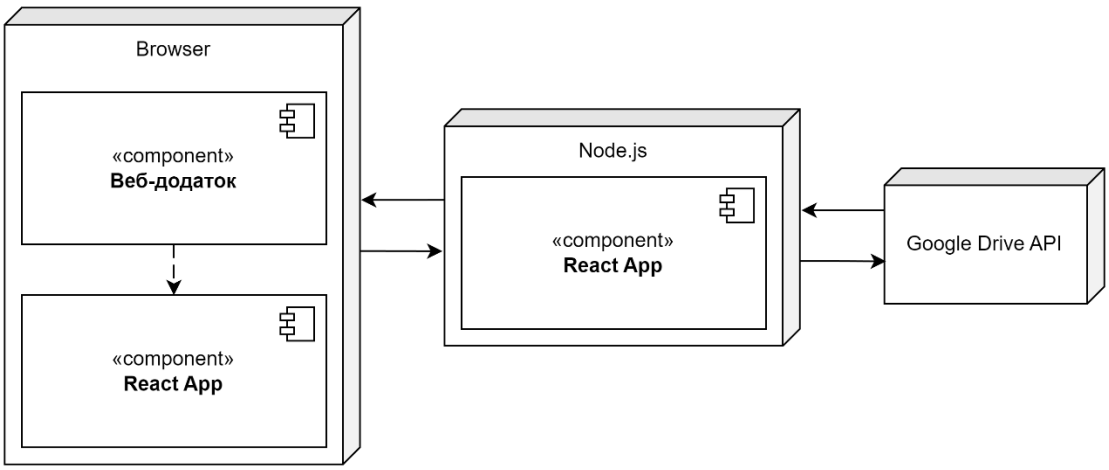


Рисунок 2.1 – Діаграма розгортання

Важливо також зазначити, що Google Drive API буде єдиним зовнішнім зв’язком у додатку, через який застосунок отримує та завантажує JSON-файл, що є базою даних з чатами та повідомленнями. Такий підхід і забезпечить автономність, оскільки користувач не залежить від стороннього серверу, гарантує приватність завдяки зберіганню данних на особистому хмарному середовищу. Все це можна побачити на діаграмі розгортання.

Це рішення також значно спрощує технічне обслуговування та масштабування проекту. Відсутність бекенд-сервера означає, що немає потреби витрачати ресурси на розгортання, хостинг чи підтримку інфраструктури, що може бути критично важливо для невеликих команд або індивідуальних розробників. У той же час, Google Drive виступає надійним інструментом збереження даних, що вже має високий рівень доступності, резервного копіювання та захисту від збоїв.

Граф діалогу дозволить побачити більш обширно цикл взаємодії саме користувача з програмою, при цьому відображаючи процеси програми пов’язані з діями на клієнті (див. рис. 2.2).

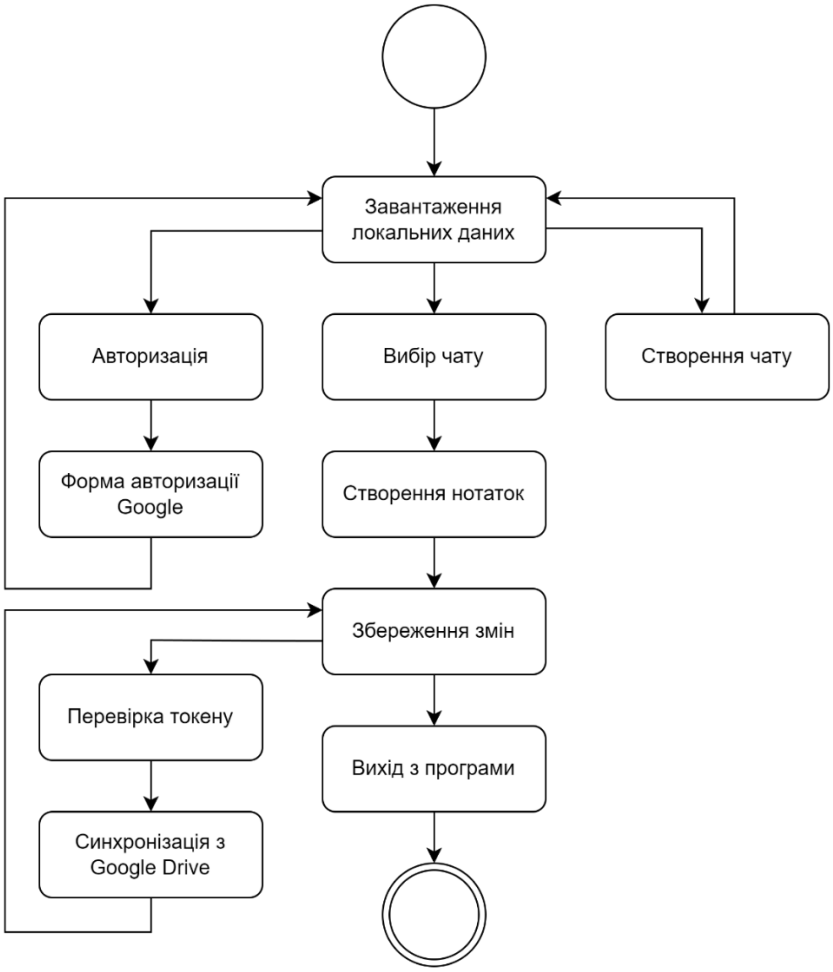


Рисунок 2.2 – Граф діалогу

На цій діаграмі показано, що після запуску програми ми завантажуємо локальні дані, як наприклад налаштування користувача та останню збережену синхронізовану базу даних. Після цього користувач може пройти авторизацію, обрати вже створений чат або створити новий. При виборі авторизації відображається форма для входу з Google API з підтвердженням на обробку даних з Google Drive. Після цього відбувається перевірка токена і при успішному вході відбувається синхронізація даних з Google Drive. У разі вибору чата або його створення, користувач може додавати нотатки. Дані зберігаються при будь-яких змінах даних. Програма завершується при виході з програми, що завершує сесію.

Також треба побудувати структурну схему, щоб зрозуміти які ролі виконує кожна частина програми і як вони взаємодіють між собою (див. рис 2.3). Ця схема коротко покаже архітектуру та структуру взаємодії компонентів системи.

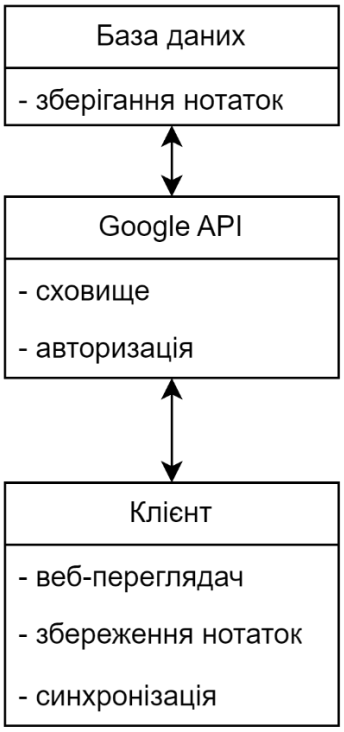


Рисунок 2.3 – Структурна схема

Схема відображає структурну організацію застосунку, побудовану навколо трьох ключових рівнів: клієнта, Google API та бази даних. На рівні клієнта знаходиться веб-додаток, який запускається у браузері користувача і забезпечує інтерфейс для взаємодії, локальне збереження нотаток та функцію синхронізації даних. Цей рівень безпосередньо підключений до Google API, який виконує роль проміжного шару для авторизації користувача та доступу до хмарного сховища. У свою чергу, Google API, пов’язаний з базою даних, що представлена у вигляді Google Drive, де фізично зберігається JSON-файл з чатами та нотатками.

2.2. Розробка функціональної моделі багатокористувацького доступу програмного комплексу маркованого чату з системою тегів

Для того, щоб побачити основну діяльність процесу роботи програми та реалізацію бізнес-логіки можна побудувати IDEF0 схему. Вона дозволить у будь-якому масштабі оцінити сутність зв’язків між процесами та те які дані використовуються в конкретному блоці.

Діаграма верхнього рівня (див. рис. 2.4) показує, що основним результатом роботи для користувача буде оновлення інтерфейсу. Це пояснюється тим, що додаток займається тільки збереженням даних, і для кінцевого користувача основним результатом є відображення нотаток у інтерфейсі, що відбувається через його оновлення.

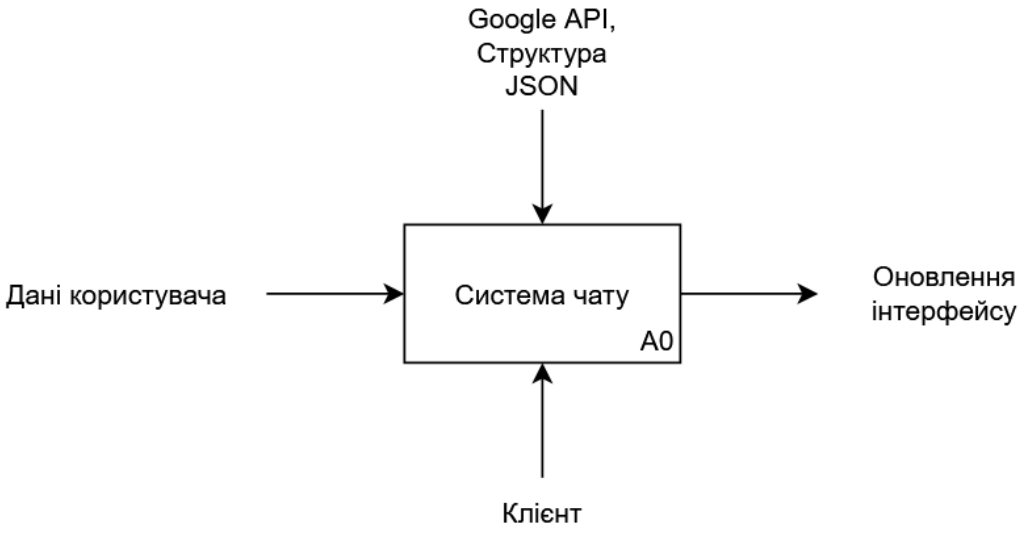


Рисунок 2.4 – Функціональна схема додатку

Далі розбираючи блок на наступному рівні декомпозиції можна виділити основні блоки функціонування програми (див. рис. 2.5). Серед них найбільш важливими є авторизація, завантаження даних та синхронізація з Google Drive.

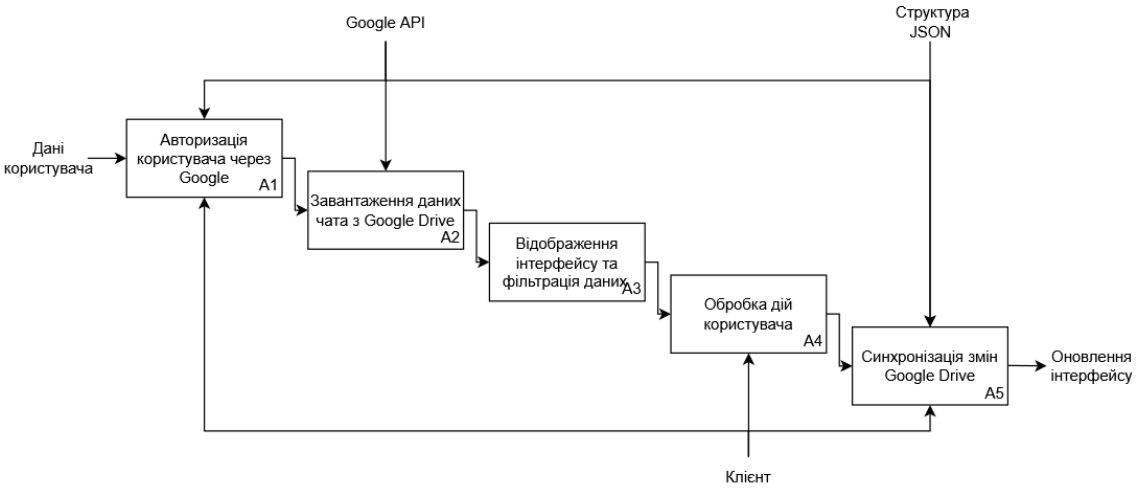


Рисунок 2.5 – Функціональна схема додатку, перший рівень декомпозиції

Блок авторизації має вертати у результаті своєї діяльності токен, для виконання дій від імені користувача на його акаунті. Це дозволяє користувачу певний час не підтверджувати дії, дозволяючи застосунку використовувати ресурси Google Drive, для зберігання його даних.

Оскільки функціональна схема дозволяє розбивати блоки на більшу кількість рівнів декомпозиції, можна розібрати також головні блоки, що мають ключову роль у функціоналі програмного комплексу. Розбиваючи блок авторизації отримуємо схему другого рівня декомпозиції (див. рис. 2.6).

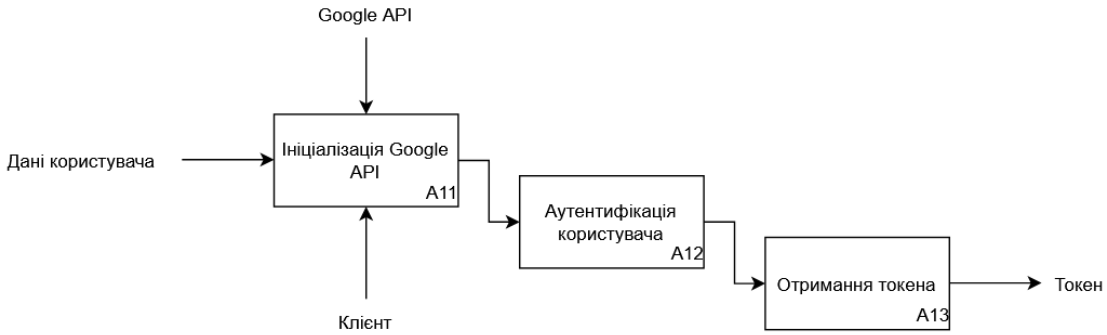


Рисунок 2.6 – Функціональна схема додатку, другий рівень декомпозиції блоку A1

Як видно блок займається тим, що відображає інтерфейс Google API для авторизації користувача задля того, щоб отримати та зберегти токен. А оскільки процеси авторизації лишаються на стороні Google, більш детально описати схему взаємодії просто не можливо через відсутність доступу до внутрішньої логіки API.

Далі розіб’ємо схему з блоку завантаження даних з Google Drive (див. рис 2.7). Схема буде дуже обмежена через те, що взаємодія відбувається з зовнішнім сервісом до якого немає прямого доступу, що не дає розібрати процеси більш детально.

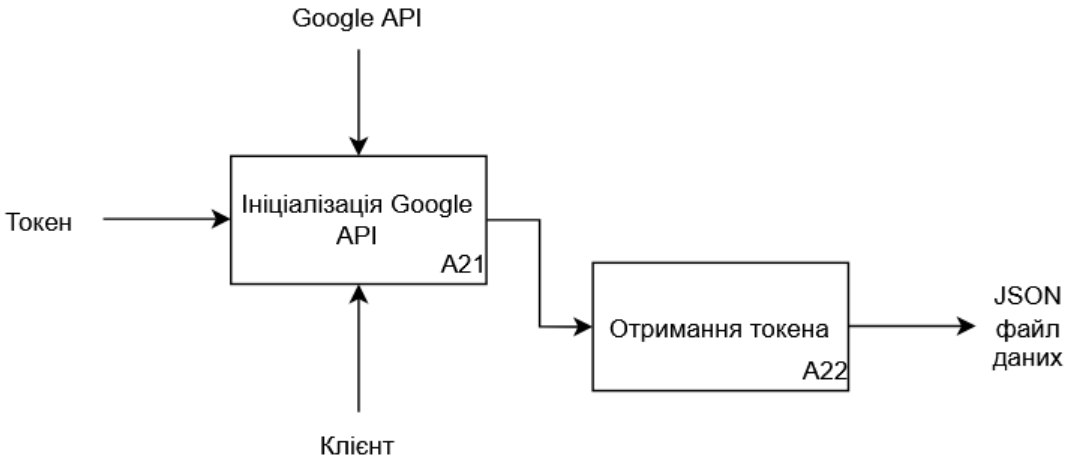


Рисунок 2.7 – Функціональна схема додатку, другий рівень декомпозиції блоку A2

Як видно завантаження файлу не є проблемою при наявності токена, а основний механізм роботи буде залежати від посилання на API, що одразу буде завантажувати необхідний файл без попередньої ініціалізації.

Останнім блоком для декомпозиції стане блок синхронізації змін, щоб побачити які процеси відбуваються вже на етапі завантаження даних на Google Drive (див. рис 2.8).

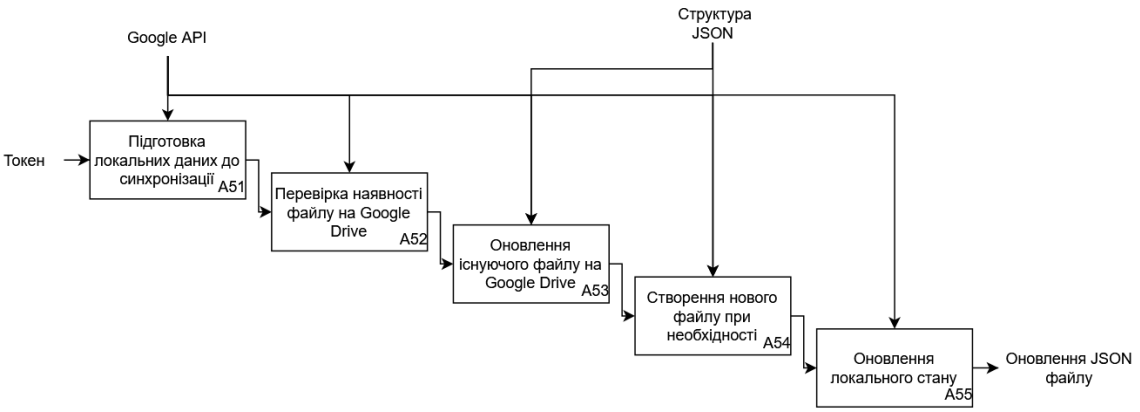


Рисунок 2.7 – Функціональна схема додатку, другий рівень декомпозиції блоку A5

Ця схема має дуже багато станів як для одного блоку взаємодії. Через те, що взаємодія з API відбувається за прописаними алгоритмами клієнтської сторони, такі етапи підготовки необхідні, щоб забезпечити стабільну роботу. Цей блок займається не тільки тим, щоб відправити файл, а також, забезпеченням коректного запису, усіма етапами перевірки та оновленням вмісту.

На другому рівні декомпозиції процес синхронізації починається з блоку підготовки локальних даних, де відбувається ініціалізація та підготовка локальних даних, включаючи отримання токену з локального сховища для роботи з Google API. Далі блок перевірки наявності файлу перевіряє, чи є файл на хмарному сховищі Google Drive. І якщо файл вже існує, що буде найпоширенішим результатом роботи програми, де з використанням структури запису бази даних у вигляді JSON-файлової структури, дані на Google Drive оновлюється. В випадку відсутності файлу активується блок створення нового файлу, що відправить на хмарне сховище наявну на пристрої базу даних у вигляді JSON структури. Обидва варіанти приведуть програму до блоку оновлення локального стану незалежно від того, чи співпадають дані на Google Drive з наявними локально чи ні.

2.3. Розробка структури бази даних програмного комплексу маркованого чату

Проектування структури бази даних є ключовим у створенні програмного комплексу маркованого чату. Головною метою даної бази даних є забезпечення збереження та оновлення чатів, повідомлень та системи маркування нотаток за допомогою тегів. Особливістю системи є її одноосібна модель взаємодії – усе керування даними здійснюється лише одним користувачем, що дозволяє значно спростити логіку обробки та збереження інформації. Такий підхід є доцільним для персонального використання, та є наслідком системи авторизації Google, але не окрім авторського розмежування не має жодних недоліків та дозволить створювати різні структури чатів для подальшого аналізу.

Базова модель передбачає наявність трьох основних сутностей: чат, повідомлення та тег. Кожен чат містить набір повідомлень, кожне повідомлення може бути помічене одним або кількома тегами. Таким чином між повідомленнями та тегами існує зв’язок «багато до багатьох», що реалізується через проміжну таблицю.

Сутність чату зберігає загальну інформацію про окремий чат. Кожен чат має унікальний ідентифікатор, що дозволяє розмежовувати чати між собою у базі даних не спираючись на їх не постійне ім’я. Поле назви служить для представлення самої назви у інтерфейсі користувача. Також реалізовано поле для картинки або іконки у текстовому варіанті.

Повідомлення відповідає за збереження окремих повідомлень у чаті. До основних полів належать: ідентифікатор, текст повідомлення, дата, тип, а також список тегів, з якими це повідомлення пов’язано. Ключовим елементом структури є зв’язок повідомлень із тегами. Кожне повідомлення може мати безліч тегів, що визначають його тематику. Такий підхід не лише структурує велику кількість інформації, але й дозволяє реалізовувати функції пошуку та фільтрації.

Сутність тегу дозволяє здійснювати категоризацію повідомлень. До її складу входять такі поля як: ідентифікатор, назва, колір та можливо буде розширено полем опису. Наявність тегів надає можливість здійснювати пошук, фільтрацію та швидку навігацію серед великої кількості повідомлень.

А для реалізації зв’язку багато-до-багатьох між повідомленнями та тегами використовується таблиця, яка містить пари ідентифікаторів повідомлення та тегу. Завдяки цій таблиці можна також ефективно виконувати вибірку повідомлень, пов’язаних з певним тегом, що особливо корисно для реалізації функцій фільтрації, пошуку та сортування. Наприклад, користувач може швидко відобразити усі повідомлення, що відмічені тегами «важливо» або будь-яким іншим тегом, без необхідності перебирати весь масив даних. Окрім того, структура добре масштабується, що дозволить також додавати інформацію для швидкого виведення, або статистики.

Логічні зв’язки між сутностями зображено на ER-діаграмі (див. рис. 2.8), де показано, як один чат може містити велику кількість повідомлень і як кожне повідомлення може мати зв’язок з багатьма тегами.

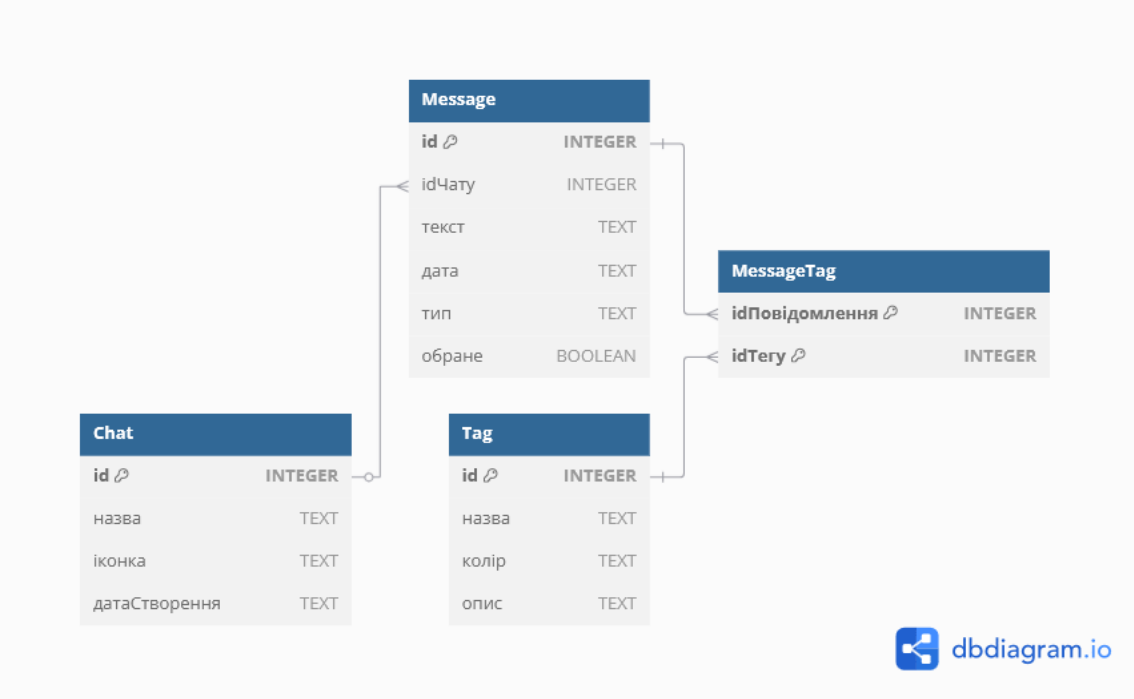


Рисунок 2.8 – ER-діаграма бази даних

Для кращого розуміння і наочного прикладу нижче наведено структуру JSON файлу яка імітує дані для одного чату:

```
{
  “chat”: {
    “id”: “chat001”,
    “назва": “Робочі нотатки”,
    “іконка": “WorkIcon”,
    “датаСтворення”: “2025-04-01T10:00:00Z”,
    “повідомлення”: [
      {
        “id”: “msg001”,
        “текст”: “Підготувати звіт до п’ятниці”,
        “дата”: “2025-05-20T14:35:00Z”,
        “тип”: “текст”,
        “обране”: true,
        “теги”: [“завдання”, “терміново”]
      },
      {
        “id”: “msg002”,
        “текст”: “Ідеї для нового проєкту”,
        “дата”: “2025-05-20T15:10:00Z”,
        “тип”: “текст”,
        “обране”: false,
        “теги”: [“мозковий штурм”]
      }
    ]
  },
  “теги”: [
    { “id”: “tag001”, “назва”: “завдання”, “колір”:
“#FFAA00” },
    { “id”: “tag002”, “назва”: “терміново”, “колір”:
“#FF0000” },
    { “id”: “tag003”, “назва”: “мозковий штурм”, “колір”:
“#00AAFF” }
  ]
}
```

Цей приклад демонструє, як можна одночасно зберігати загальні метадані про чат, пов’язані з ним повідомлення та відповідні теги, забезпечуючи структуроване зберігання й гнучке маркування і все це у текстовому форматі, що підтримується завдяки стандарту JSON-файлової структури.

Незважаючи на початкову простоту запропонована структура має гнучкість до подальшого розширення. Наприклад, можна додати підтримку вкладень, валідацію тегів, створення підкатегорій, реалізувати історію змін, або додати авторизацію для користувачів якщо додаток буде переноситись з Google Drive API на централізований сервер. Тобто навіть якщо відмовитись від вихідних вимог проекту і встановити основну взаємодію не з Google API, а з власним сервером і своєю авторизацією – перенесення буде проходити без ускладнень та критичних змін, а JSON структуру можна буде змінити на аналоги сервісів збереження даних.

При проектуванні структури таблиць важливим кроком стало визначення відповідних типів даних для кожного поля. Наприклад, для ідентифікаторів використано тип INTEGER, що дозволяє ефективно індексувати записи та забезпечує унікальність. Текстові поля, як назва, текст, опис та назва іконки реалізовані через тип TEXT, оскільки вони можуть містити у собі довільні за довжиною символи, виключно з юнікод-символами.

Поле дата в таблицях теж визначається полем формату TEXT, що забезпечує сумісність у системах синхронізації, зокрема Google Drive API. Такий формат легко сортується та підтримується у більшості баз даних і мов програмування без необхідності додаткового перетворення. А також поле дати буде використовуватись лише для розмежування нотаток хронологічно.

Не дивлячись на те, що структура досить проста, але проектована структура дозволить з легкістю масштабувати функціонал. Прикладів розширення безліч і частина з них вже була перерахована. Структура передбачає навіть розширення системи чатів на систему категорій з чатами, щоб групувати вже самі списки нотаток на різні типи.

Проектована структура також є універсальною тому підтримується переведення і подальша інтеграція з іншими сервісами для зберігання як наприклад SQLite або іншими системами управління базами даних чи навіть хмарними базами даних.

2.4. Розробка блок-схем алгоритмів програмних модулів комплексу маркованого чату

Блок-схеми алгоритмів представляють собою логічну структуру роботи ключових блоків або модулів програми. Вони дозволяють побачити взаємодію між подіями, станами та операціями більш детально. Особливістю даного застосунку є можливість працювати в автономному режимі з локальною базою даних, а також синхронізуватися з Google Drive за умови, що користувач авторизований у систему. Синхронізація повинна хоча б частково дозволяти користувачу редагувати записи з різних пристроїв автономно, при цьому мінімально шкодити актуальності.

Спочатку потрібно побудувати схему, яка продемонструє логіку ініціалізації програми (див. рис. 2.9).

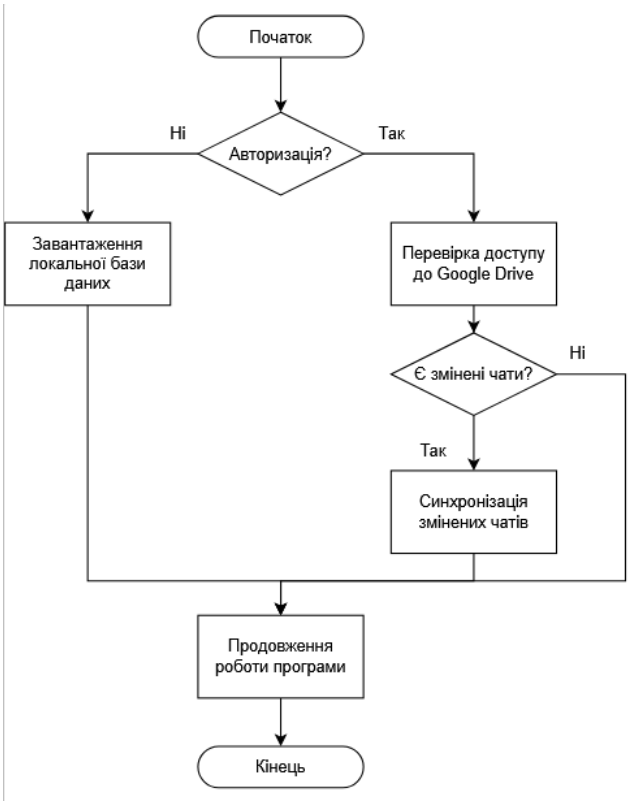


Рисунок 2.9 – Блок-схема алгоритму ініціалізації програми

На цій схемі відображено основні блоки запуску застосунку. Якщо користувач авторизованиій, система виконує синхронізацію у процесі роботи.

Щоб зрозуміти, як саме відбувається створення нового чату та внесення змін до нотаток, необхідно звернутись до відповідної блок-схеми (див. рис. 2.10). Вона відображає послідовність дій користувача та програмної логіки при створенні, редагуванні і маркуванні змінених чатів для подальшої синхронізації

Рисунок 2.10 – Блок-схема алгоритму створення чату або нотатки

Діаграма ілюструє створення нового чату. На ній демонструється як користувач редагує нотатки та оновлює її вміст і після цього встановлюється прапорець про модифікацію, для синхронізації цього чату.

Для відображення внутрішньої логіки локального збереження інформації у базу даних застосунку доцільно розглянути окрему блок-схему (див. рис. 2.11). Вона описує принцип дії автоматичного збереження змін, тобто відстеження змін, перевірка актуальності даних і запис до локального сховища.

Рисунок 2.11 – Блок-схема алгоритму збереження даних у базу даних

Схема показує, як застосунок реагує на зміну вмісту нотаток де спочатку перевіряється, чи справді були зміни, і лише тоді відбувається оновлення локальної бази даних. Процес завершується встановленням прапорця про зміну на «істину».

Процес синхронізації вимагає чіткого розуміння логіки, щоб уникнути конфліктів і дублювання. Відповідна блок-схема (див. рис. 2.12) дозволяє візуально ознайомитись з механізмом виявлення змінених чатів та оновлення тих чатів, що потребують синхронізації.

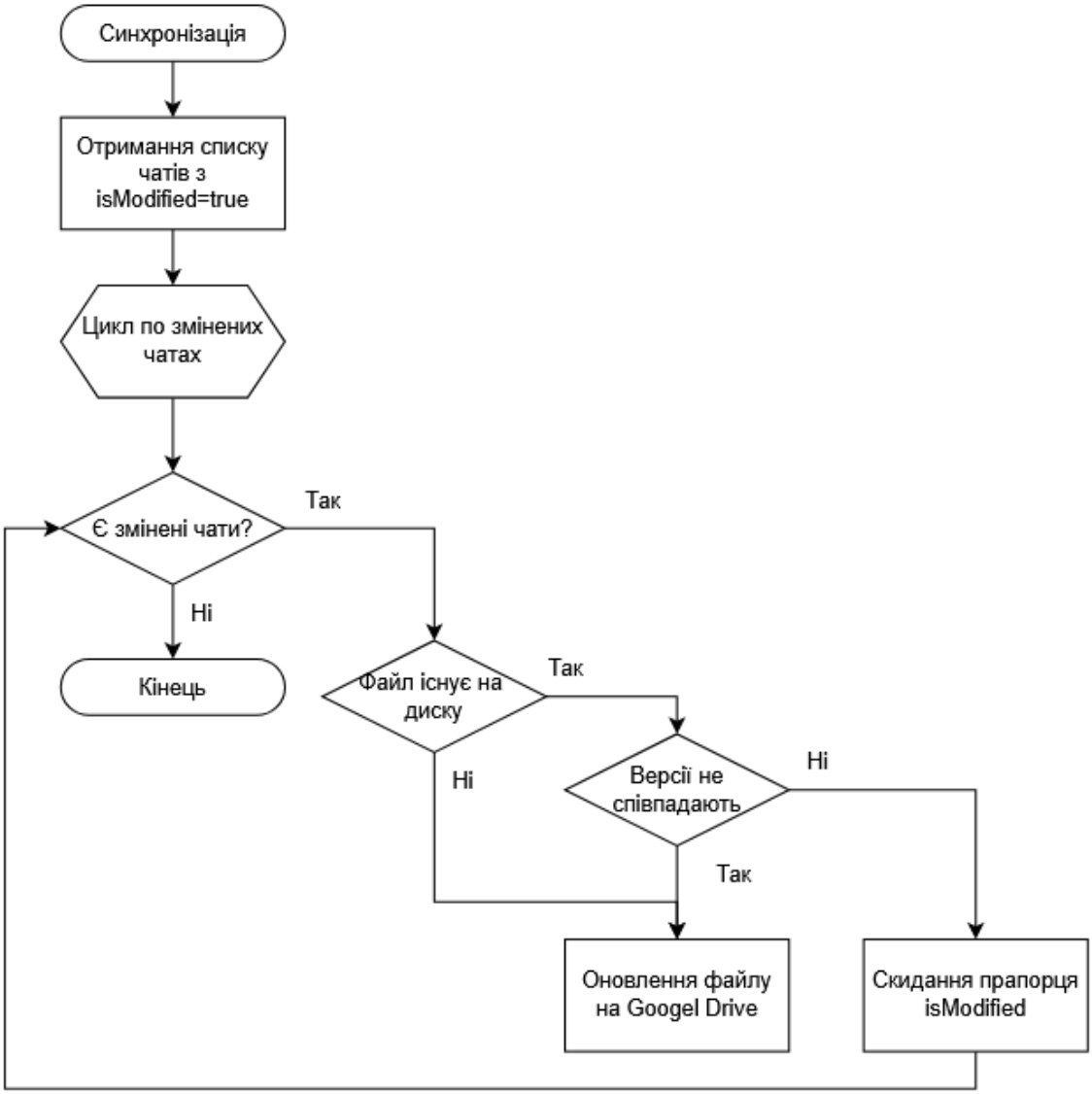


Рисунок 2.12 – Блок-схема алгоритму синхронізації з Google Drive

Ця блок-схема демонструє процес синхронізації де спочатку збираються всі чати з прапорцем, що говорить про попереднє редагування у цьому чаті. Потім кожен перевіряється на наявність у хмарному сховищі та актуальності.

Якщо файл відрізняється або відсутній – він оновлюється редагованою версією з локального сховища.

Наступним кроком було розбити авторизацію на окрему блок-схему (див. рис. 2.13). Вона опише процедуру взаємодії з Google OAuth та збереженні токену в локальному середовищі.

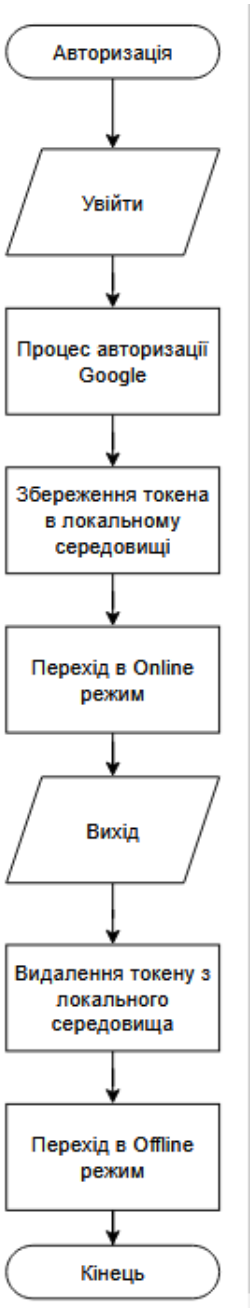


Рисунок 2.13 – Блок-схема алгоритму авторизації

Схема відображає процес входу через Google OAuth, авторизацію та отримання токену з його подальшим збереженням. Показано приблизний сценарій виходу де токен видаляється, а система повертається у автономний режим.

2.5. Розробка моделі візуального інтерфейсу маркованого чату

У цьому пункті розроблюються схеми, що приблизно описують візуальну частину розроблюваного додатку. Макет показує як може виглядати користувацький інтерфейс у мобільній та десктопній версії. Створено дві моделі для представлення схематичного та стилізованого макету розроблюваного додатку.

На рисунку 2.14 показано схему широкоформатної, тобто десктопної, версії додатку. На схемі містяться основні блоки, що повинні бути реалізовані у програмі. Вона показує приблизне розташування об’єктів відносно один одного.

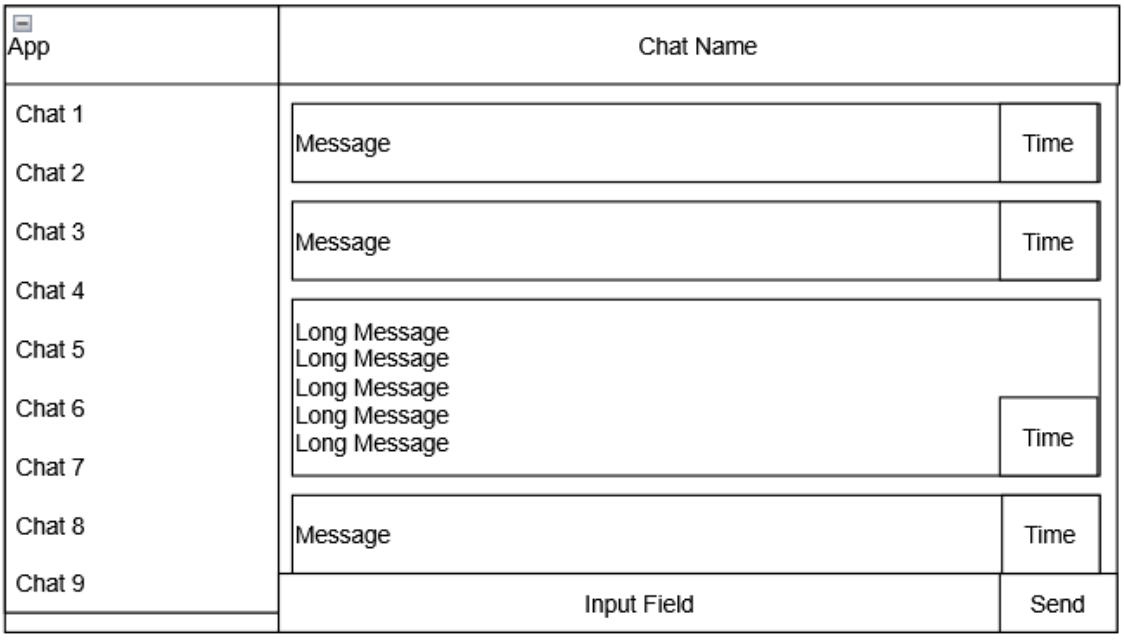


Рисунок 2.14 – Схематичний макет розроблюваного додатку. Широкоформатна версія

На рисунку 2.15 показано схему, що описує розташування об’єктів у мобільній версії додатку. Показано саме систему перемикання чатів, що відображає їх список.

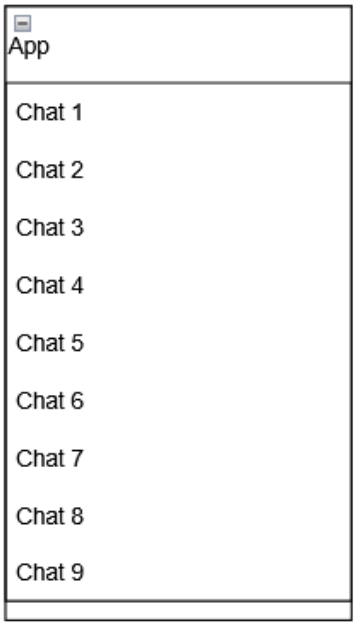


Рисунок 2.15 – Схематичний макет розроблюваного додатку. Мобільна версія списку чатів

На рисунку 2.16 також показано мобільну версію додатку вже у інтерфейсі чату де показано повідомлення та поле для введення.

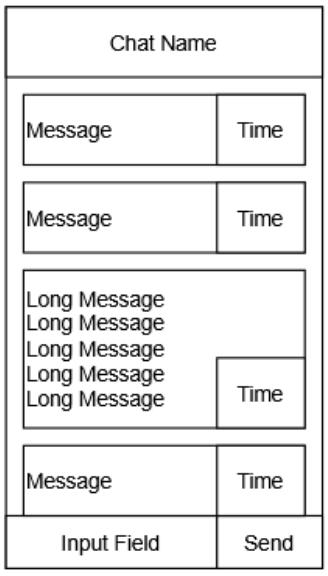


Рисунок 2.16 – Схематичний макет розроблюваного додатку. Мобільна версія чату

На рисунку 2.17 представлена приблизна стилізована схема користувацького інтерфейсу для широкоформатної версії.

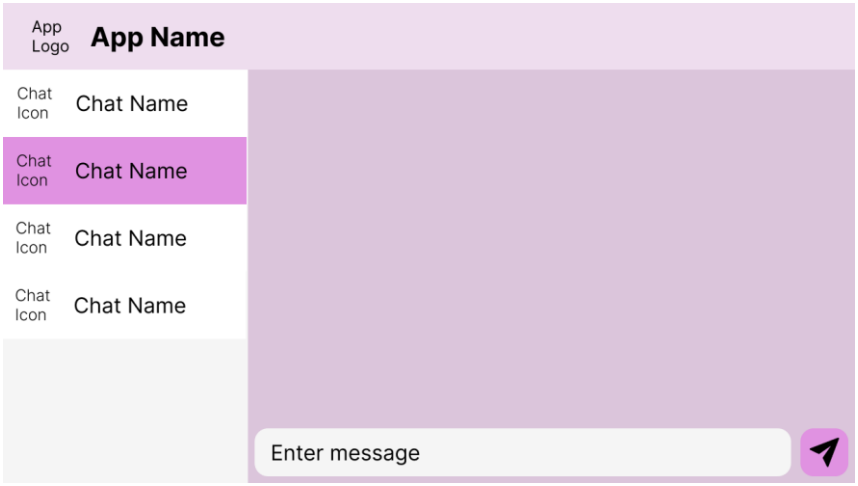


Рисунок 2.17 – Стилістичний макет розроблюваного додатку. Мобільна версія чату

Рисунок 2.18 ілюструє стилізований макет списку чатів у мобільній версії додатку. У ньому показано як інтерфейс адаптується до вузького екрану.

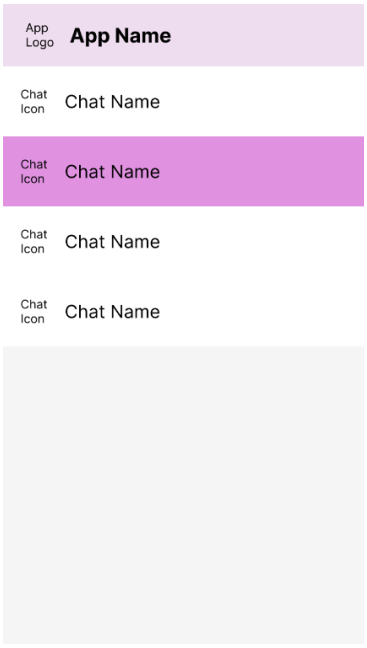


Рисунок 2.18 – Стилістичний макет розроблюваного додатку. Мобільна версія чату

Рисунок 2.19 показує стилістично оформлений макет списку нотаток у чаті на мобільній версії додатку.

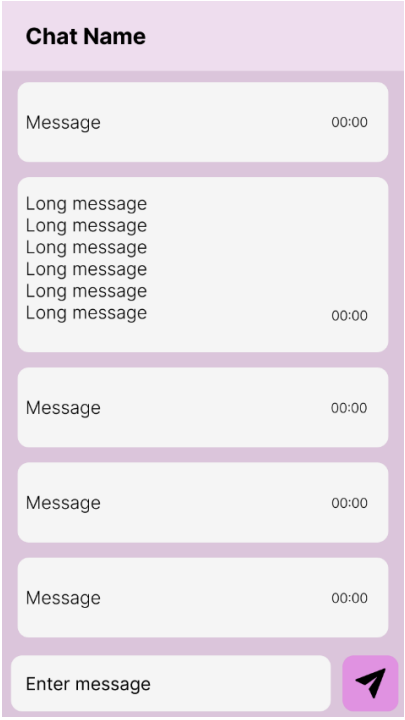


Рисунок 2.19 – Стилістичний макет розроблюваного додатку. Мобільна версія списку чатів

На рисунку 2.20 зображено стилістично оформлений макет з прикладом модального вікна.

Рисунок 2.20 – Стилістичний макет розроблюваного додатку. Модальне вікно

Ці схеми дозволять розробити інтерфейс включаючи як його розмітку та структуру, так і кольорову схему, що проілюстрована у стилістичних макетах.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ КРОСПЛАТФОРМНОГО БАГАТОКОРИСТУВАЦЬКОГО МАРКОВАНОГО ЧАТУ

3.1. Розробка візуального інтерфейсу комплексу маркованого чату

Першим кроком при розробці додатку є розмітка сторінки на основні блоки, що й будуть складати структуру інтерфейсу. Інтерфейс буде мати 3 основних блоки. Перший блок – це головна верхня панель додатку на якій буде відображатись інформація для орієнтування, як наприклад назва чату. Далі блок бокової панелі, за стандартом знаходиться у лівій стороні додатку і відображає список чатів. Останнім блоком буде частина вільного місця, що лишилась після заповнення першими двома блоками і буде відповідати вікну обраного чату відображаючи нотатки.

Якщо розбирати ієрархію блоків у html коді, виглядатиме вона приблизно таким чином:

```
<div class="left-panel">
  <div class="top-panel-component">
  </div>
  <div class="panel-content">
    <div class="chat-name-panel">
    </div>
  </div>
  <div class="primary-button">
  </div>
</div>

<div class="right-panel">
  <div class="top-panel-component">
  </div>
  <div class="panel-content">
    <div class="messages-field">
    </div>
    <div class="chat-input-field">
    </div>
  </div>
</div>
```

На рисунку 3.1 показано, як виглядає готове розташування блоків інтерфейсу. Для кращої видимості було виставлено тимчасові кольори блоків.

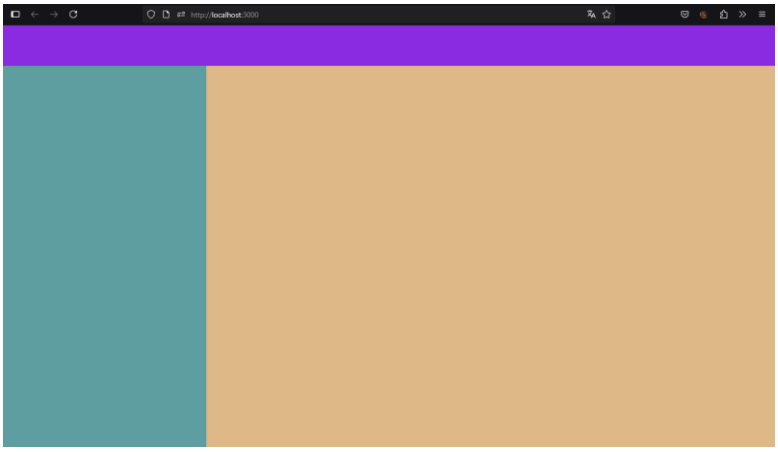


Рисунок 3.1 – Скріншот додатку з початковим розбиттям інтерфейсу на блоки

Далі після розміщення блоків на свої місця і перевірки на коректність відображення цих блоків можна встановити для них стандартну схему кольорів, що була передбачена у стилістичному макеті у другому розділі.

Оскільки передбачається також зміна кольорової схеми, було також розроблено файл з основною палітрою. Їх запис дозволить використовувати для одних і тих же елементів різні кольори при різних схемах. Після встановлення вже підготовленої схеми кольорів маємо результат на рисунку 3.2.

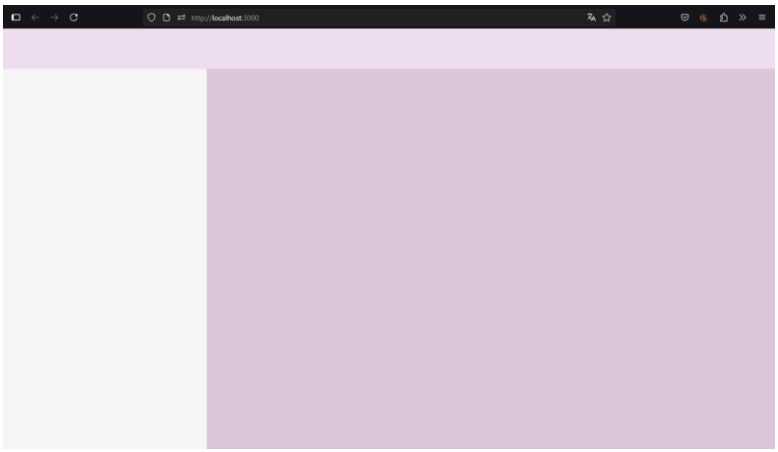


Рисунок 3.2 – Скріншот додатку з розбиттям інтерфейсу на блоки та кольоровою схемою

Для схеми кольорів було обрано невелику модель з 3х кольорів для блоків, основного кольору тексту, основний тон, фон модального вікна, а також кольори для підсвічування помилки та попередження. Кольори зберігаються у файлі JavaScript як модуль, що дозволяє через код змінювати тему використовуючи модуль ThemeManager.

Схема світлої теми зберігається у такому вигляді:

```
light: {
    "--primary-color": "#e092e1",
    "--main-panel-color": "#eeddee",
    "--background-color": "#dbc5db",
    "--side-background-color": "#f5f5f5",
    "--widget-color": "#f5f5f599",
    "--warning-color": "#f39c12",
    "--error-color": "#e74c3c",
    "--foreground-color": "#333",
},
```

Надалі це дозволить розширити список великою кількістю різних схем кольорів додатку, а використання модулю і збереження їх у коді дозволить легко змінювати кольорову схему через змінні.

Також треба додати основні компоненти, які будуть використовуватись у додатку. Це елементи кнопок, поле для введення, поле відображення нотатки та чат. Ці елементи вже розроблені як об’єкти React, що дозволить використовувати їх багаторазово з різними даними. Ці елементи будуть мати кращий вигляд якщо додавати іконки. За допомогою бібліотеки Lucide можна додати велику кількість svg іконок, що впроваджуються у проект як звичайні об’єкти React [11]. На рисунку 3.3 зображено як ці елементи тимчасово розміщені через код на сторінці додатку.

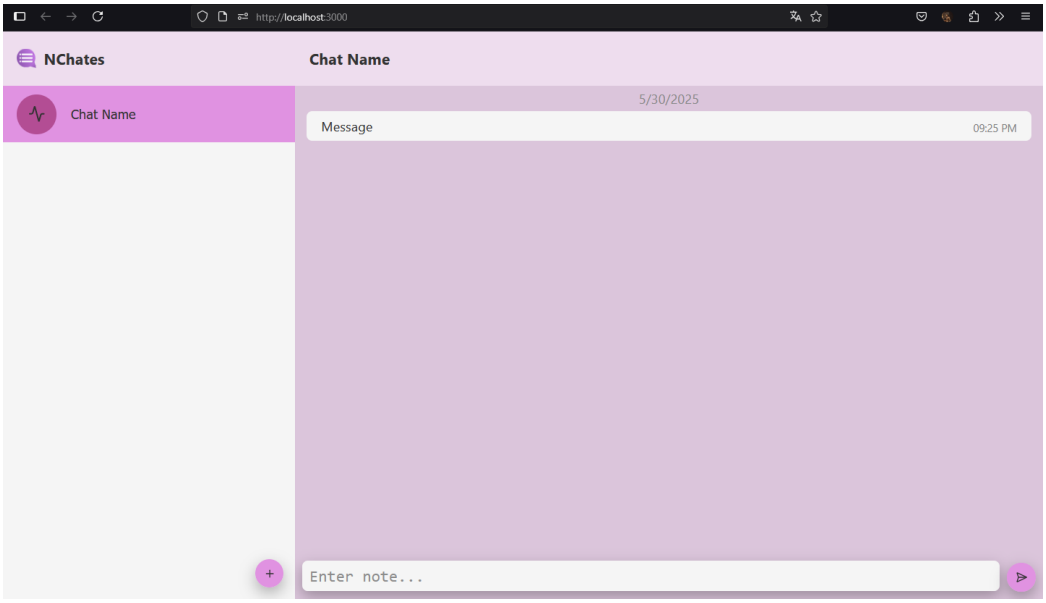


Рисунок 3.3 – Скріншот додатку з елементами інтерфейсу

Також об’єктами можна зробити й інші елементи інтерфейсу як наприклад модальне вікно. Як об’єкт модальне вікно може не мати нічого всередині себе і бути лише обгорткою для інших об’єктів або набору об’єктів. Реалізація модального вікна включає у себе лише розробку контейнеру в якому перед цим додано приймання частини html коду у відведений елемент контенту. На рисунку 3.4 показано модальне вікно без вмісту.

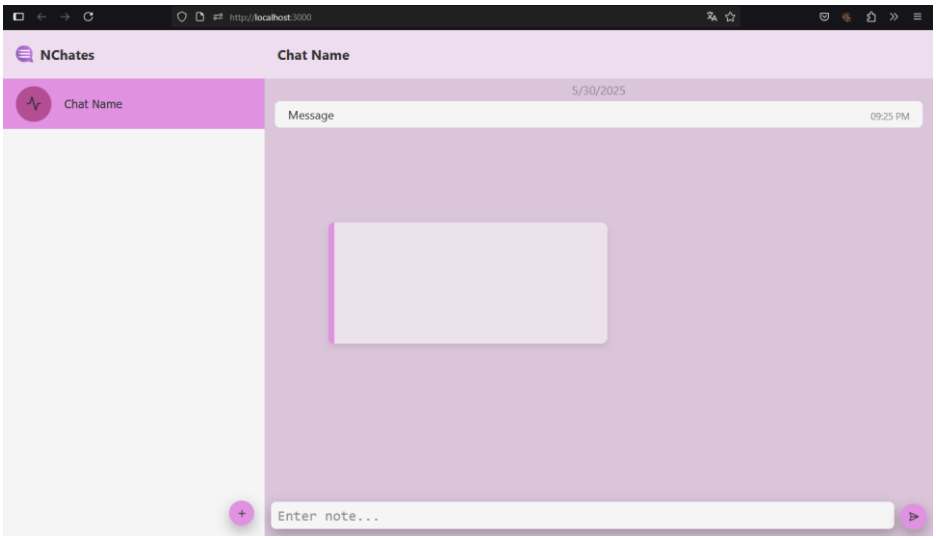


Рисунок 3.4 – Скріншот прикладу модального вікна без вмісту

Використовуючи вже готові об’єкти серед яких і кнопки і вже готове модальне вікно, у якому попередньо налаштовано додавання контенту, можна додати до інтерфейсу меню створення нового чату (див. рис. 3.5). Наперед у меню також продумано вибір іконки та кольору чату.

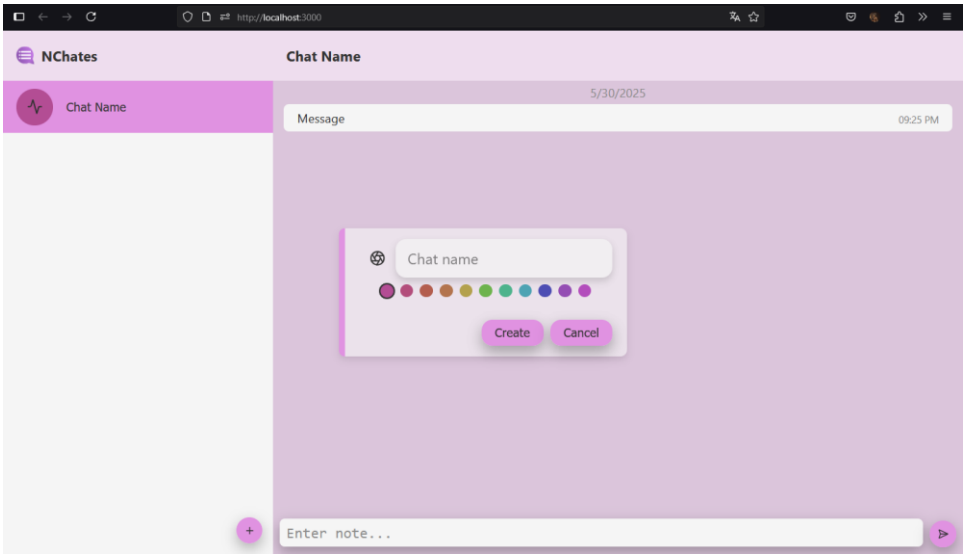


Рисунок 3.5 – Скріншот меню створення нового чату

Ці модулі та панелі вже є прототипом інтерфейсу, що надалі можна розширювати функціоналом та реалізовувати необхідні для функціонування додатку методи.

3.2. Розробка бази даних програмного комплексу

Наступним кроком треба впровадити систему керування базою даних. Спочатку треба виділити для цього клас, що буде зберігати базу даних у локальному сховищі, що і буде відповідати за автономну роботу додатку.

Спочатку як вказано в документації у Dexie.js встановлюємо модуль ази даних через пакетний менеджер, наприклад npm. Після цього одразу можна розроблювати систему таблиць[12]. Використовуючи бібліотеку Dexie клас через конструктор слідуючи схемі json яку було попередньо розроблено.

```
class NotesDatabase extends Dexie {
  constructor() {
    super('NotesDB');
    this.version(1).stores({
```

```
      chats: '++id, name, icon, color, tags, edited'
    });
    this.chats = this.table('chats');
  }
```

Далі треба зазначити наперед методи синхронізації з Google Drive API, що на цьому етапі можна залишити без реалізації:

```
async syncToDrive() {
  return;
}

async syncFromDrive() {
  return;
}
```

Далі реалізується створення чату у базі даних. Метод приймає як і у інтерфейсі лише назву чату, колір та іконку у текстовому форматі. Кожен наступний метод буде використовувати функції збереження та завантаження з Google Drive для автоматичної синхронізації даних. Це може бути як тимчасовим рішенням так і постійним в залежності від ефективності.

```
async createChat(name, icon, color) {
  await this.syncFromDrive();
  const chat = {
    name,
    icon,
    color,
    creationDate: new Date().toISOString(),
    messages: [],
    tags: [],
    edited: true
  };
  const chatId = await this.chats.add(chat);
  await this.syncToDrive();
  return chatId;
}
```

Після реалізації створення чатів також треба реалізувати отримання інформації про наявні у локальному сховищі чати.

```
async getChats() {
  await this.syncFromDrive();
  return this.chats.toArray();
}
```

Далі створюємо функцію видалення:

```
async deleteChat(id) {
  await this.syncFromDrive();
  await this.chats.delete(id);
  await this.syncToDrive();
}
```

Наступним кроком, щоб завершити повністю реалізацію списку чатів у базі даних, необхідно створити функцію для відстеження оновлення чату, що буде відмічати чи відбувалось редагування чату у автономному режимі.

```
async updateChat(id, updates) {
  await this.syncFromDrive();
  const chat = await this.chats.get(id);
  if (!chat) return;
  const updatedChat = { ...chat, ...updates, edited: true };
  await this.chats.put(updatedChat);
  await this.syncToDrive();
}
```

Це був останній крок для запису змін у чатах і настав час реалізовувати додавання, редагування та видалення нотатків у чатах. Спочатку реалізуємо додавання нової нотатки до бази даних.

```
async addMessage(chatId, text, tags = [], type = "text") {
  await this.syncFromDrive();
  const chat = await this.chats.get(chatId);
  if (!chat) return;
  const newMessage = {
    id: `msg${Date.now()}`,
    content: text,
    timestamp: new Date().toISOString(),
    type: type,
    tags: tags
  };
  chat.messages = chat.messages || [];
  chat.messages.push(newMessage);
  chat.edited = true;
  await this.chats.put(chat);
  await this.syncToDrive();
  return newMessage.id;
}
```

Як і з системою чатів треба додати функцію для відображення усіх нотатків, що записані у чаті у локальному сховищі.

```
async getMessages(chatId) {
  await this.syncFromDrive();
  const chat = await this.chats.get(chatId);
  return chat?.messages || [];
}
```

Також функція для редагування вмісту запису, що також буде відмічати, що цей чат має зміни створені в автономному режимі.

```
async updateMessage(chatId, messageId, updates) {
  await this.syncFromDrive();
  const chat = await this.chats.get(chatId);
  if (!chat) return;
  const index = chat.messages?.findIndex(msg => msg.id ===
messageId);
  if (index === -1 || index === undefined) return;
  chat.messages[index] = { ...chat.messages[index],
...updates };
  chat.edited = true;
  await this.chats.put(chat);
  await this.syncToDrive();
}
```

А також реалізація видалення запису:

```
async deleteMessage(chatId, messageId) {
  await this.syncFromDrive();
  const chat = await this.chats.get(chatId);
  if (!chat) return;
  chat.messages = chat.messages?.filter(msg => msg.id !==
messageId);
  chat.edited = true;
  await this.chats.put(chat);
  await this.syncToDrive();
}
```

3.3. Розробка основних програмних модулів програмного комплексу маркованого чату

Для повноцінного функціонування програми необхідно забезпечити синхронізацію з гугл диском. Для цього спочатку треба розробити модуль авторизації. Спираючись на документацію, щодо авторизації акаунту Google

Identity про використання OAuth 2.0, треба спочатку авторизувати сервіс у Google Console, для того, щоб налаштувати права на доступ до даних користувача зі свого додатку, а після цього написати код, що буде отримувати для користувача токен доступу до його акаунту [13]. Для початку реалізуємо клас кнопки, що потім помістимо у модальне вікно авторизації, яка буде відкривати меню авторизації OAuth використовуючи бібліотеку react-oauth.

```
const GoogleLoginButton = () => {
  const login = useGoogleLogin({
    onSuccess: async (tokenResponse) => {
      console.log("Log in success:", tokenResponse);
      localStorage.setItem('google_token',
tokenResponse.access_token);

      try {
        const token = tokenResponse.access_token;

      } catch (error) {
        console.error('Error while fetching user info:',
error);
      }
      window.location.reload();
    },
    onError: () => {
      console.error("Authorization error");
    },
    scope: 'https://www.googleapis.com/auth/drive.file
https://www.googleapis.com/auth/drive',

  });
```

Для того, щоб кнопка відображалась також у повернення треба вказати html частину коду. Використання особистого ідентифікатору або класу для цієї кнопки не є ефективним, бо кнопка авторизації буде лише в одному місці і це модальне вікно, що буде реалізовано пізніше.

```
  return (
    <button
      onClick={login}
      style={{
        display: "flex",
        alignItems: "center",
        gap: "10px",
        padding: "10px 20px",
        border: "none",
```

```
          borderRadius: "50px",
          backgroundColor: "#fff",
          color: "#000",
          fontSize: "clamp(14px, 1vw + 0.5vh, 50px)",
          fontWeight: "bold",
          cursor: "pointer",
          outline: "none",
        }}
      >
        <img

src="https://fonts.gstatic.com/s/i/productlogos/googleg/v6/24px.svg"
          alt="Google Icon"
          style={{
            width: "clamp(20px, 1vw + 0.5vh, 50px)",
            height: "clamp(20px, 1vw + 0.5vh, 50px)",
          }}
        />
        Login with Google
      </button>
    );
```

Як і сказано вище, використання цієї кнопки передбачено у блоці модального вікна авторизації аккаунту, у якому окрім виведення кнопки також є перевірка авторизації, яка ховає модальне вікно якщо токен записаний у локальне сховище, що і говорить про авторизованість користувача в системі. Нижче наведено код реалізації модального вікна:

```
const GoogleAuth = () => {
  const [isAuthenticated, setIsAuthenticated] =
useState(false);

  useEffect(() => {
    const checkToken = async () => {
      const token = localStorage.getItem('google_token');
      if (await isTokenValid(token)) {
        setIsAuthenticated(true);
      } else {
        setIsAuthenticated(false);
      }
    };
    checkToken();
  }, []);

  return (
    <>
      {!isAuthenticated ? (
```

```
          <Widget>
            <p style={{ marginBottom: "15px" }}>
              To use the service, you need to log in through
your Google account.
            </p>
            <GoogleOAuthProvider clientId={clientId}>
              <GoogleLoginButton />
            </GoogleOAuthProvider>
          </Widget>
        ) : null}
      </>
    );
}
```

Далі реалізуємо клас виклики якого будуть спрямовані на спрощення взаємодії з Google Drive. Цей модуль буде мати основні функції, що будуть використовуватись для синхронізації файлу бази даних з пристрою до хмарного сховища. Спираючись на документацію для розробників Google Workspace, маючи токен після авторизації, можна відправляти запити на сервер Google Drive, що буде виконувати задані дії щодо файлів хмарного сховища, що були у запиті [14]. Тобто використання може здійснюватися навіть без додаткових бібліотек.

Реалізація починається з функції, що повертає ідентифікатор файлу на хмарному сховищу, щоб не шукати його кожен раз за ім’ям самостійно. Функція буде спрощувати лише знаходження файлу.

```
async function fetchChatFileId(token) {
  try {
    const query = encodeURIComponent(`name =
"${FILE_NAME}"`);
    const url =
`https://www.googleapis.com/drive/v3/files?q=${query}&fields
=files(id,name,modifiedTime)`;

    const response = await fetch(url, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      },
      cache: 'no-store',
    });

    if (!response.ok) {
```

```
      const errorText = await response.text();
      throw new Error(`Failed to fetch file list:
${response.status} ${response.statusText} - ${errorText}`);
    }

    const data = await response.json();

    if (data.files && data.files.length > 0) {
      const latestFile = data.files.reduce((latest, current)
=> {
        return new Date(current.modifiedTime) > new
Date(latest.modifiedTime) ? current : latest;
      });
      return latestFile.id;
    }

    return null;
  } catch (error) {
    console.error('Error fetching chat file ID:', error);
    throw error;
  }
}
```

Далі реалізація функції завантаження чатів з хмарного сховища, що використовує у запиті ідентифікатор файлу з чатами, який отримується у методі, що описаний вище.

```
async function downloadChats(token) {
  try {
    const fileId = await fetchChatFileId(token);
    if (!fileId) {
      console.log('No chat file found in Google Drive');
      return [];
    }

    const url =
`https://www.googleapis.com/drive/v3/files/${fileId}?alt=med
ia&_=${Date.now()}`;

    const response = await fetch(url, {
      headers: {
        'Authorization': `Bearer ${token}`,
      },
      cache: 'no-store',
    });

    if (!response.ok) {
      const errorText = await response.text();
```

```
      throw new Error(`Failed to download chats:
${response.status} ${response.statusText} - ${errorText}`);
    }

    const text = await response.text();

    if (!text.trim()) {
      console.log('Chat file is empty');
      return [];
    }

    try {
      const chats = JSON.parse(text);
      return Array.isArray(chats) ? chats : [];
    } catch (parseError) {
      console.error('Failed to parse chat data:',
parseError);
      console.error('Raw data:', text.substring(0, 200)); //
Первые 200 символов для отладки
      return [];
    }

  } catch (error) {
    console.error('Download error:', error);
    return [];
  }
}
```

А далі логічним продовженням є розробка методу, що буде навпаки завантажувати з локального сховища на хмарне:

```
async function uploadChats(token, chats) {
  try {
    if (!Array.isArray(chats)) throw new Error('Chats must
be an array');

    const fileId = await fetchChatFileId(token);
    const boundary = `boundary_${Date.now()}`;
    const delimiter = `--${boundary}`;

    const multipartBody = [
      delimiter,
      'Content-Type: application/json; charset=UTF-8',
      '',
      JSON.stringify({ name: FILE_NAME, mimeType: MIME_TYPE
}),
      delimiter,
      `Content-Type: ${MIME_TYPE}`,
      '',
      JSON.stringify(chats, null, 2),
```

```
      `${delimiter}--`
    ].join('\r\n');

    const response = await fetch(

`https://www.googleapis.com/upload/drive/v3/files${fileId ?
`/${fileId}` : ''}?uploadType=multipart`,
      {
        method: fileId ? 'PATCH' : 'POST',
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': `multipart/related;
boundary=${boundary}`,
        },
        body: multipartBody,
        cache: 'no-store',
      }
    );

    if (!response.ok) {
      const errorText = await response.text();
      throw new Error(`Failed to upload chats:
${response.status} ${response.statusText}\n${errorText}`);
    }

    const result = await response.json();
    console.log(`Chats successfully ${fileId ? 'updated' :
'created'} in Google Drive`);
    return result;

  } catch (error) {
    console.error('Upload error:', error);
    throw error;
  }
}
```

Це основний функціонал файлів авторизації і синхронізації у хмарному сховищі, але база даних так і залишилась без їх виклику, тому треба доповнити систему бази даних реалізацією синхронізації з хмарного сховища та навпаки.

Як і до цього для реалізації синхронізації необхідно розробити метод для завантаження з диску та на диск. Нижче наведено реалізацію завантаження на диск:

```
async syncToDrive() {
  if (!this.canSync()) {
    console.log('Sync to Drive skipped: no token or
offline');
```

```
      return;
    }

    try {
      const token = this.getAccessToken();

      const localChats = await this.chats.toArray();

      const chatsToSync = localChats.map(chat => ({
        ...chat,
        edited: false
      }));

      await uploadChats(token, chatsToSync);

      await this.chats.toCollection().modify({ edited: false
});

      console.log('Sync to Drive completed');
    } catch (error) {
      console.error('Sync to Drive failed:', error);
    }
}
```

Тепер треба реалізувати функцію завантаження з диску:

```
async forceSyncFromDrive() {
  const token = this.getAccessToken();
  if (!token || !this.isOnline) {
    throw new Error('Cannot sync: no access token or
offline');
  }

  try {
    const remoteChats = await downloadChats(token);
    if (remoteChats && remoteChats.length > 0) {
      await this.chats.clear();
      await this.chats.bulkAdd(remoteChats.map(chat => ({
...chat, edited: false })));
    }
    console.log('Force sync from Drive completed');
  } catch (error) {
    console.error('Force sync from Drive failed:', error);
    throw error;
  }
}
```

3.4. Розробка програмних модулів функціонування тематичних тегів

При розробці системи тегів було проаналізовано статтю з детальним посібником, що ілюструє способи представлення тегів та основні принципи при розробці зовнішнього вигляду тегів. Спираючись на статтю Tags UI Design, основними принципами є лаконічність тегів, тема тегів повинна виділятись на тлі теми програми, структура тегу повинна виділяти лише ті об’єкти у тегу, що мають інформаційну цінність [15]. Але назва тегу залежить від користувача, тому лаконічність тегу не реалізується.

Реалізація підтримки тегів починається з їх запису при введені та відправці нотатки. Для цього необхідно додати поле вводу у інтерфейс:

```
<div className="tags-input-row">
  <input
    className="tags-input"
    value={tags}
    onChange={(e) => setTags(e.target.value)}
    placeholder="Tags separated by commas"
    spellCheck={false}
  />
</div>
```

Додаємо при відправці теги, що у функції відправлення парсяться видаляючи коми:

```
const handleSend = async () => {
    if (!content.trim() || !chatId || sending) return;

    setSending(true);

    const parsedTags = tags
      .split(",")
      .map((tag) => tag.trim())
      .filter((tag) => tag.length > 0);

    if (editingMessage) {
      await db.updateMessage(chatId, editingMessage.id, {
        content: content,
        tags: parsedTags,
      });
      onEditComplete();
    } else {
      await db.addMessage(chatId, content, parsedTags);
      onMessageSent();
    }
```

```
    setContent("");
    setTags("");
    setSending(false);
  };
```

А також хук на редагування нотатки, щоб заповнити теги у поле, які вже є на записі:

```
useEffect(() => {
  if (editingMessage) {
    setContent(editingMessage.content);
    setTags((editingMessage.tags || []).join(", "));
  } else {
    setContent("");
    setTags("");
  }
}, [editingMessage]);
```

Тепер треба зробити відображення тегів на самих нотатках, щоб бачити без фільтрації до якої категорії відноситься запис:

```
{message.tags?.length > 0 && (
  <div className="message-tags">
    {message.tags.map((tag, index) => (
      <span key={index} className="message-tag">{tag}</span>
    ))}
  </div>
)}
```

На цьому реалізація системи тегів зі сторони відображення у повідомленнях реалізовані. Залишилось додати на головну сторінку панель для фільтрування нотаток. Для цього реалізовується клас панелі:

```
const TagsFilterPanel = ({ tags, selectedTag, onSelectTag })
=> {
  const allTags = Array.from(new Set(tags));

  return (
    <div className="tags-filter-panel">
      <span
        className={`filter-tag ${selectedTag === null ?
"active" : ""}`}
        onClick={() => onSelectTag(null)}
      >
        All
      </span>
      {allTags.map((tag, index) => (
```

```
        <span
          key={index}
          className={`filter-tag ${selectedTag === tag ?
"active" : ""}`}
          onClick={() => onSelectTag(tag)}
        >
          {tag}
        </span>
      ))}
    </div>
  );
};
```

І додаємо цю панель до головної сторінки у бокову правую панель

```
<div className="right-panel">
  <TopPanelComponent>
    <div style={{ display: 'flex', alignItems: 'center',
gap: '10px' }}>
      {isMobile && (
        <button
          onClick={() => setLeftPanelVisible(true)}
          style={{
            background: 'none',
            border: 'none',
            color: 'var(--foreground-color)',
            cursor: 'pointer',
            fontSize: '1.5em',
          }}
        >
          <ArrowLeft size={20} />
        </button>
      )}
      <h3 style={{ margin: 0 }}>{activeChat ?
`${activeChat.name}` : ""}</h3>
    </div>
  </TopPanelComponent>
  <TagsFilterPanel
    tags={allTagsInChat}
    selectedTag={selectedTag}
    onSelectTag={setSelectedTag}
  />

  <div className="panel-content">
    <MessagesField
      messages={messages}
      selectedTag={selectedTag}
      onDelete={async (id) => {
        await db.deleteMessage(activeChat.id, id);
        forceReload();
      }}
```

```
        onEdit={handleEditStart}
      />

      <ChatInputField
        chatId={activeChat?.id}
        onMessageSent={forceReload}
        editingMessage={editingMessage}
        onEditComplete={handleEditComplete}
        onEditCancel={handleEditCancel}
      />
    </div>
  </div>
```

3.5. Розробка програмних модулів функціонування нотаток

Основними програмними модулями для функціонування додатку – є більша частина реалізації інтерфейсу та система керування базою даних. Поєднання цих систем і дає як результат повноцінно функціонуючу програму, що можна використовувати автономно.

Реалізація інтерфейсу поділена на велику кількість блоків, основні з яких виділити буде важко. Інтерфейс включає у себе обгортку основних елементів HTML, для більш модульного підходу до розробки, та реалізація власних класів, що будуть слугувати обгорткою для основних елементів та відповідати вже за повноцінні блоки.

Відображення повідомлень є ключовим у роботі. Для реалізації потрібно спочатку підготувати контейнер, який буде відповідати за відображення. Він також буде розділяти, сортувати та фільтрувати відображені нотатки:

```
const MessagesField = ({ messages = [], onDelete, onEdit,
selectedTag }) => {
  const renderMessagesWithDateSeparators = () => {
    const elements = [];
    let lastDate = null;

    const sorted = [...messages]
      .filter(msg => !selectedTag || (msg.tags ||
[]).includes(selectedTag))
      .sort((a, b) => new Date(a.timestamp) - new
Date(b.timestamp));

    sorted.forEach((msg) => {
      const msgDate = new
Date(msg.timestamp).toLocaleDateString();
```

```
      const msgTime = new
Date(msg.timestamp).toLocaleTimeString([], {
        hour: "2-digit",
        minute: "2-digit",
      });

      if (msgDate !== lastDate) {
        lastDate = msgDate;
        elements.push(
          <div key={"separator-" + msgDate} className="date-
separator">
            {msgDate}
          </div>
        );
      }

      elements.push(
        <MessageItem
          key={msg.id}
          message={msg}
          time={msgTime}
          onDelete={() => onDelete?.(msg.id)}
          onEdit={() => onEdit?.(msg)}
        />
      );
    });

    return elements;
  };

  return (
    <div className="messages-field" style={{ flexGrow: "3",
overflow: "auto" }}>
      {renderMessagesWithDateSeparators()}
    </div>
  );
};
```

А також підготуємо для нього елемент, що й буде відображати запис. Для цього створимо окремий елемент, що використовує функціонал описаний вище у реалізації системи тегів.

```
const MessageItem = ({ message, onDelete, onEdit, time }) =>
{
  const [showMenu, setShowMenu] = useState(false);
  const menuRef = useRef(null);

  const handleContextMenu = (e) => {
    e.preventDefault();
```

```
    setShowMenu(true);
  };

  const handleCloseMenu = () => setShowMenu(false);

  useEffect(() => {
    const handleClickOutside = (e) => {
      if (menuRef.current &&
!menuRef.current.contains(e.target)) {
        setShowMenu(false);
      }
    };

    if (showMenu) {
      document.addEventListener("mousedown",
handleClickOutside);
    } else {
      document.removeEventListener("mousedown",
handleClickOutside);
    }

    return () => {
      document.removeEventListener("mousedown",
handleClickOutside);
    };
  }, [showMenu]);

  return (
    <div
      className="message-item"
      onContextMenu={handleContextMenu}
      onClick={handleCloseMenu}
    >
      <div className="message-main">
        <p className="message-text">{message.content}</p>

        <div className="message-right">
          {!showMenu && <span className="message-
time">{time}</span>}

          {showMenu && (
            <div className="message-actions" ref={menuRef}>
              <PrimaryButton type="transparent" onClick={()
=> onEdit(message)}>
                <Pencil />
              </PrimaryButton>
              <PrimaryButton type="transparent" onClick={()
=> onDelete(message.id)}>
                <Trash2 />
              </PrimaryButton>
            </div>
```

```
          )}
        </div>
      </div>

      {message.tags?.length > 0 && (
        <div className="message-tags">
          {message.tags.map((tag, index) => (
            <span key={index} className="message-
tag">{tag}</span>
          ))}
        </div>
      )}
    </div>
  );

};};
```

ВИСНОВКИ

Протягом виконання кваліфікаційної роботи було розроблено систему для створення, редагування та видалення нотаток у вигляді чатової структури, що підтримує систему тегів. Впровадження системи чатів повинно було дозволити інтуїтивно швидко та ефективно орієнтуватись у групах записаних нотатків, а система тегів фільтрувати та знаходити нотатки за їх категоріями.

Кожен розділ роботи визначав кінцевий результат, так як підготовка була розподілена на декілька етапів. У першому розділі було проаналізовано подібні системи, що стали відправною точкою як для знаходження ефективних рішень у інтерфейсі користувача, так і до впровадження нових. А також було проаналізовано і обрано підходящі технології розробки серед яких основною було обрано React.

Другий розділ допоміг розробити схеми взаємодії, щоб виділити головні випадки які необхідно обробити та реалізувати у додатку. Також схеми інтерфейсу, що розділили усі можливі етапи користування застосунком на розмітку сторінки у вигляді різних панелей, модальних вікон, кнопок та інших елементів користувацього інтерфейсу.

У третьому розділі було поетапно розроблено додаток, а кожен етап розробки описує окремий блок, що відповідав за інтерфейс, базу даних, модулі функціонування тегів та модулі синхронізації, авторизації у системі Google Drive.

Результатом розробки стала система, що ефективно розділяє нотатки, групує їх за чатами та дозволяє ефективно орієнтуватися у будь-якій кількості записів через впровадження системи тегів яка дозволила фільтрувати нотатки за різними критеріями. Розроблювана система отримала велику кількість можливостей для налаштування, як зі сторони користувацького інтерфейсу так і говорячи про систему чатів та тегів, що отримали зміну іконок, кольорів та власних назв.

ПЕРЕЛІК ПОСИЛАНЬ

1. MDN Web Docs – Client-server overview [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/Learn_web_development/Extensions/Server-side/First_steps/Client-Server_overview;

2. Web.Dev – OS Integration [Електронний ресурс] – Режим доступу до ресурсу: https://web.dev/learn/pwa/os-integration;

3. Node.js – Introduction to Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/en/learn/getting-started/introduction-to-nodejs;

4. Vue CLI – Guide [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/introduction.html;

5. Google Help – Use web apps [Електронний ресурс] – Режим доступу до ресурсу: https://support.google.com/chrome/answer/9658361;

6. Dexie.js – Samples [Електронний ресурс] – Режим доступу до ресурсу: https://dexie.org/docs/Samples;

7. Discord – About Discord [Електронний ресурс] – Режим доступу до ресурсу: https://discord.com/company;

8. Telegram – Telegram FAQ [Електронний ресурс] – Режим доступу до ресурсу: https://telegram.org/faq;

9. Electron – Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://www.electronjs.org/docs/latest;

10. React Native – Get Started with React Native [Електронний ресурс] – Режим доступу до ресурсу: https://reactnative.dev/docs/environment-setup;

11 Lucide – What is Lucide [Електронний ресурс] – Режим доступу до ресурсу: https://lucide.dev/guide/;

12. Dexie.js – Get started with Dexie in React [Електронний ресурс] – Режим доступу до ресурсу: https://dexie.org/docs/Tutorial/React;

13. Google Identity – Using OAuth 2.0 to Access Google APIs [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/identity/protocols/oauth2;

14. Google Workspace – Google Drive API overview [Електронний ресурс] – Режим доступу до ресурсу: https://developers.google.com/drive /api/guides/about-sdk;

15. Pixso – [Full Guide] Tags UI Design [Електронний ресурс] – Режим доступу до ресурсу: https://pixso.net/tips/tags-ui.

Додаток А

А.1 – Посібник користувача.

А.1.1 – Інформація про застосунок

Розроблюваний додаток, що розглядається – є багатокористувацьким кросплатформним додатком, що орієнтований на збереження нотаток. Особливістю системи є інтеграція з Google Drive API, що дозволяє зберігати та синхронізувати дані на різних пристроях. Це дозволяє виключити необхідність у власному сервері. А робота сервісу виконана у рамках SPA доступного через браузер.

А.1.2 – Авторизація

При запуску програми користувач одразу може пройти авторизацію через Google API. Користувачу необхідно натиснути на кнопку входу до системи Google. Потім надати доступ до системи Google Drive і підтвердити вхід. Після цього додаток автоматично зберігає зміни у додатку.

А.1.3 – Створення чату

Для того щоб створити новий чат, користувачу необхідно натиснути на кнопку у лівій панелі з іконкою «+». Після цього відкриється модальне вікно, де користувач може вписати назву чату, обрати його іконку та обрати колір. Після натискання на кнопку «Створити» – модальне вікно зникне і з’явиться новий чат.

А.1.4 – Робота з нотатками

Натиснувши на чат, користувач обирає його і може продивлятись записані нотатки у правій панелі сайту.

Для створення нотатки, користувач повинен написати в полі для введення свій запис та натиснути на кнопку з іконкою паперового літака.

Додатково вище поля для вводу нотатки є поле для введення тегів які можна також заповнити за необхідністю розділюючи їх через кому.

Натиснувши на нотатку правою кнопкою миші у десктопній версії, або утримуючи на нотатці у мобільній версії – користувач може відкрити меню для редагування та видалення нотатки. Після натискання на кнопку видалення, обране повідомлення видаляється з чату. Якщо натиснути на кнопку редагування, користувач через поле вводу для відправки може відредагувати запис та відправивши його підтвердити редагування.

А.1.5 – Синхронізація

Синхронізація на пристроях відбувається автоматично після авторизації користувача і не потребує його прямого втручання.

Додаток А.2 – Лістинг коду

А.2.1 – Код файлу MainPage.jsx, головна сторінка

```
import { Plus, ArrowLeft } from "lucide-react";
import React, { useState, useEffect, useRef, useCallback }
from "react";

import ChatCreatorWidget from
"../../components/fields/ChatCreatorWidget";
import TopPanelComponent from
"../../components/fields/TopPanelComponent";
import ChatInputField from
"../../components/utility/ChatInputField";
import ChatNamePanel from
"../../components/utility/ChatNamePanel";
import MessagesField from
"../../components/utility/MessagesField";
import TagsFilterPanel from
"../../components/fields/TagsFilterPanel";
import PrimaryButton from
"../../components/utility/PrimaryButton";
import { db } from "../../backend/Database";
import GoogleAuth from '../../backend/GoogleAuthorization';
import MainMenuButton from
"../../components/utility/MainMenuButton";

import "./MainPage.css";

const MainPage = () => {
  const [selectedTag, setSelectedTag] = useState(null);
```

```
  const [messages, setMessages] = useState([]);
  const [isMobile, setIsMobile] = useState(false);
  const [leftPanelVisible, setLeftPanelVisible] =
useState(true);
  const [activeChat, setActiveChat] = useState(null);
  const [chatList, setChatList] = useState([]);
  const [isCreating, setIsCreating] = useState(false);
  const [reloadFlag, setReloadFlag] = useState(false);
  const [editingMessage, setEditingMessage] =
useState(null);
  const [editingChat, setEditingChat] = useState(null);

  const handleDeleteChat = async (chatId) => {
    if (!window.confirm(`Remove chat?`)) return;
    try {
      await db.deleteChat(chatId);
      if (activeChat?.id === chatId) {
        setActiveChat(null);
      }
      await loadChats();
    } catch (err) {
      console.error("Chat delete error:", err);
    }
  };

  const handleEditChat = (chat) => {
    setEditingChat(chat);
    setIsCreating(true);
  };

  const handleCreateOrUpdateChat = async (name, icon, color)
=> {
    try {
      if (editingChat) {
        // Редактирование существующего чата
        await db.updateChat(editingChat.id, { name, icon,
color });
        await loadChats();
        setIsCreating(false);
        setEditingChat(null);
      } else {
        // Создание нового чата
        await db.createChat(name, icon, color);
        await loadChats();
        setIsCreating(false);
      }
    } catch (err) {
      console.error("Error while creating/updating chat:",
err);
    }
  };
```

```
  const handleCancelCreateOrEdit = () => {
    setIsCreating(false);
    setEditingChat(null);
  };

  const forceReload = () => setReloadFlag(!reloadFlag);
  const allTagsInChat = messages.flatMap(msg => msg.tags ||
[]);

  const touchStartX = useRef(null);

  const handleTouchStart = useCallback((event) => {
    touchStartX.current = event.touches[0].clientX;
  }, []);

  const handleTouchEnd = useCallback((event) => {
    if (touchStartX.current === null) return;
    const touchEndX = event.changedTouches[0].clientX;
    const deltaX = touchEndX - touchStartX.current;
    const swipeThreshold = 50;
    if (Math.abs(deltaX) > swipeThreshold) {
      setLeftPanelVisible(deltaX > 0);
    }
    touchStartX.current = null;
  }, []);

  const handleChatSelect = (chat) => {
    setActiveChat(chat);
    setEditingMessage(null);
    if (isMobile) {
      setLeftPanelVisible(false);
    }
  };

  const loadChats = async () => {
    const chats = await db.chats.toArray();
    setChatList(chats);
    if (chats.length > 0 && !activeChat) {
      setActiveChat(chats[0]);
    }
  };

  const loadMessages = async () => {
    if (!activeChat) return;
    const msgs = await db.getMessages(activeChat.id);
    setMessages(msgs);
  };

  const handleEditStart = (message) => {
    setEditingMessage(message);
```

```
  };

  const handleEditComplete = () => {
    setEditingMessage(null);
    forceReload();
  };

  const handleEditCancel = () => {
    setEditingMessage(null);
  };

  useEffect(() => {
    document.addEventListener("touchstart",
handleTouchStart);
    document.addEventListener("touchend", handleTouchEnd);

    const checkScreenSize = () => {
      const aspectRatio = window.innerWidth /
window.innerHeight;
      setIsMobile(aspectRatio < 1);
    };

    checkScreenSize();
    window.addEventListener("resize", checkScreenSize);

    loadChats();
    loadMessages();

    return () => {
      document.removeEventListener("touchstart",
handleTouchStart);
      document.removeEventListener("touchend",
handleTouchEnd);
      window.removeEventListener("resize", checkScreenSize);
    };
  }, [activeChat, reloadFlag, handleTouchStart,
handleTouchEnd]);

  return (
    <div className="chat-page">
      <div
        className={`left-panel ${isMobile ?
(leftPanelVisible ? 'mobile-full' : 'hidden') : ''}`}
      >
        <div
          style={{
            position: "absolute",
            display: "flex",
            right: "2vh",
            bottom: "2vh",
          }}
```

```
        >
          <PrimaryButton
            type={"icon"}
            onClick={() => setIsCreating(true)}
          >
            <Plus />
          </PrimaryButton>
        </div>

        <TopPanelComponent>
          <div style={{ display: "flex", flexDirection:
"row", alignItems: "center"}}>
            <MainMenuButton /><h3>NChates</h3>
          </div>
        </TopPanelComponent>

        <div className="panel-content">
          <ChatNamePanel
            chatList={chatList}
            activeChat={activeChat}
            setActiveChat={handleChatSelect}
            onDelete={handleDeleteChat}
            onEdit={handleEditChat}
          />
        </div>
      </div>

      <div className="right-panel">
        <TopPanelComponent>
          <div style={{ display: 'flex', alignItems:
'center', gap: '10px' }}>
            {isMobile && (
              <button
                onClick={() => setLeftPanelVisible(true)}
                style={{
                  background: 'none',
                  border: 'none',
                  color: 'var(--foreground-color)',
                  cursor: 'pointer',
                  fontSize: '1.5em',
                }}
              >
                <ArrowLeft size={20} />
              </button>
            )}
            <h3 style={{ margin: 0 }}>{activeChat ?
`${activeChat.name}` : ""}</h3>
          </div>
        </TopPanelComponent>
        <TagsFilterPanel
          tags={allTagsInChat}
```

```
            selectedTag={selectedTag}
            onSelectTag={setSelectedTag}
          />

          <div className="panel-content">
            <MessagesField
              messages={messages}
              selectedTag={selectedTag}
              onDelete={async (id) => {
                await db.deleteMessage(activeChat.id, id);
                forceReload();
              }}
              onEdit={handleEditStart}
            />

            <ChatInputField
              chatId={activeChat?.id}
              onMessageSent={forceReload}
              editingMessage={editingMessage}
              onEditComplete={handleEditComplete}
              onEditCancel={handleEditCancel}
            />
          </div>
        </div>

        {isCreating && (
          <ChatCreatorWidget
            onSubmit={handleCreateOrUpdateChat}
            onCancel={handleCancelCreateOrEdit}
            editingChat={editingChat}
          />
        )}
        <GoogleAuth></GoogleAuth>
      </div>
    );
};

export default MainPage;
```