Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА ДОДАТКА «MCHAT» ЗІ ЗАСТОСУВАННЯМ ТОКЕНІВ ДЛЯ АУТЕНТИФІКАЦІЇ З ВИКОРИСТАННЯМ REACT NATIVE, NODE.JS ТА MONGODB

DEVELOPMENT OF THE "MCHAT" APPLICATION WITH TOKEN-BASED AUTHENTICATION USING REACT NATIVE, NODE.JS AND MONGODB

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-41
Хільковець Руслан Володимирович

Керівник:
Вознюк Анастасія Вадимівна

Кваліфікаційну роботу
допущено до захисту
«10» 06 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

_____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *бакалавр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«20» 12 2024р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Хільковця Руслана Володимировича
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *«Розробка додатка "MChat" зі застосуванням токенів для аутентифікації з використанням React Native, Node.js та MongoDB»*
Керівник роботи: *асистент Вознюк А. В.*
затверджені наказом закладу вищої освіти від *«19» грудня 2024 р.* № *474/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра *«10» червня 2025 р.*
3. Вихідні дані до роботи: *VS Code, Git, React Native, Node.js, Express.js, socket.io методичні вказівки до виконання кваліфікаційної роботи бакалавра*
4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): *аналіз сучасного стану real-time месенджерів; функціональні й нефункціональні вимоги; опис спроектованої архітектуру системи; обґрунтування технічного стеку; опис реалізації клієнтської та серверної частин; структура бази даних; опис тестування та налагодження месенджеру; опис заходів для забезпечення захисту інформаційно-комп'ютерної системи*
5. Перелік графічного матеріалу: *10 рисунків, 2 додатки*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Вознюк А. В.*		
Специфікація вимог до розробленої системи	*Вознюк А. В.*		
Розробка об'єкта проектування	*Вознюк А. В.*		
Нормоконтроль	*Вознюк А. В.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту	2,36 %		
Академічна доброчесність	*Вознюк А. В.*		

7. Дата видачі завдання *«20» грудня 2024 р.*

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 04.02.2025 р.	Виконано
2	Аналіз проблеми розробки та впровадження об`єкту проектування	до 01.03.2025 р.	Виконано
3	Обгрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 15.03.2025 р.	Виконано
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 29.03.2025 р.	Виконано
5	Практична реалізація об'єкта проектування та розробка бази даних	до 26.04.2025 р.	Виконано
6	Тестування та налагодження об'єкта проектування	до 03.05.2025 р.	Виконано
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 10.06.2025 р.	Виконано

Здобувач вищої освіти Руслан ХІЛЬКОВЕЦЬ

Керівник кваліфікаційної роботи Анастасія ВОЗНЮК

АНОТАЦІЯ

Хільковець Р. В. Розробка додатка «MChat» зі застосуванням токенів для аутентифікації з використанням React Native, node.js та MongoDB. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

У першому розділі здійснено аналіз сучасних real-time месенджерів та їхніх функціональних особливостей , на основі чого сформульовано завдання на кваліфікаційну роботу. У другому розділі визначено функціональні та нефункціональні вимоги до розроблюваної системи, а також обґрунтовано вибір засобів, методів та алгоритмів розробки, зокрема React Native, Node.js та MongoDB. У третьому розділі детально описано практичну реалізацію мобільного додатка «MChat» , включаючи розробку серверної та клієнтської частин, бази даних, впровадження механізмів захисту та тестування системи. У висновках узагальнено результати роботи та підведено підсумки щодо досягнення поставлених цілей.

Ключові слова: мобільний додаток, MChat, React Native, Node.js, MongoDB, Socket.IO, JWT, аутентифікація, шифрування даних, тестування, UML-діаграми.

ABSTRACT

Khilkovets R. Development of the "MChat" Application with Token-based Authentication Using React Native, node.js and MongoDB. Manuscript. Qualification work of the bachelor's program "Software engineering" in the specialty "Software engineering". Lutsk National Technical University. Lutsk, 2025.

The bachelor's qualification thesis consists of an introduction, three chapters, conclusions, a list of references, and appendices.

In the first chapter, modern real-time messengers and their functional features are analyzed, based on which the objectives for the qualification work are formulated. The second chapter defines the functional and non-functional requirements for the developed system, and also justifies the choice of development tools, methods, and algorithms, including React Native, Node.js, and MongoDB. The third chapter details the practical implementation of the "MChat" mobile application, including the development of the server and client sides, the database, the implementation of security mechanisms, and system testing. The conclusions summarize the results of the work and draw conclusions regarding the achievement of the set goals.

Keywords: mobile application, MChat, React Native, Node.js, MongoDB, Socket.IO, JWT, authentication, data encryption, testing, UML diagrams.

ЗМІСТ

ВСТУП

Актуальність теми. Сучасне інформаційне суспільство характеризується стрімким зростанням обсягу цифрової комунікації, що, у свою чергу, підвищує вимоги до безпеки передачі даних, захисту конфіденційної інформації та автентифікації користувачів у мережевих системах. Однією з ключових задач сьогодення є створення ефективних і водночас захищених інструментів для обміну повідомленнями, особливо у випадках, коли інформація не повинна виходити за межі внутрішньої інфраструктури. У зв’язку з цим виникає потреба в розробці власних, ізольованих рішень, що не залежать від сторонніх серверів чи хмарних сервісів.

Метою кваліфікаційної роботи бакалавра є розробка мобільного чат-додатка «MChat» зі застосуванням токенів для аутентифікації, реалізованого на основі сучасних технологій React Native, Node.js та MongoDB.

Об’єктом кваліфікаційної роботи бакалавра є мобільний додаток «MChat», призначений для організації реального часу обміну повідомленнями між користувачами.

Предметом кваліфікаційної роботи бакалавра є практична реалізація механізмів аутентифікації через токени (JWT), інтерфейсних компонентів у React Native, серверної логіки на Node.js та схеми зберігання даних у MongoDB для забезпечення безпечного й надійного обміну повідомленнями.

Для досягнення поставленої мети були сформульовані такі завдання:

– проаналізувати сучасний стан real-time месенджерів;

– визначити функціональні і нефункціональні вимоги;

– спроектувати архітектуру системи;

– обґрунтувати вибір технічного стеку;

– реалізувати клієнтську та серверну частину;

– розробити базу даних;

– провести тестування та налагодження;

– забезпечити захист інформаційно-комп’ютерної системи.

Таким чином, робота спрямована на вирішення актуальної задачі – створення безпечної платформи для комунікації, яка задовольняє потреби користувачів у приватності, контрольованому середовищі обміну даними та технологічній незалежності.

РОЗДІЛ 1
АНАЛІЗ РОБОТИ СУЧАСНИХ REAL-TIME МЕСЕНДЖЕРІВ І ПОСТАНОВКА ЗАВДАНЬ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

В наш час існує безліч різних месенджерів, проте загалом всі користуються лише декількома. Абсолютна більшість опитаних користується Telegram. Майже 65 % надають перевагу Viber, 37 % – WhatsUp (рис. 1.1) [1].

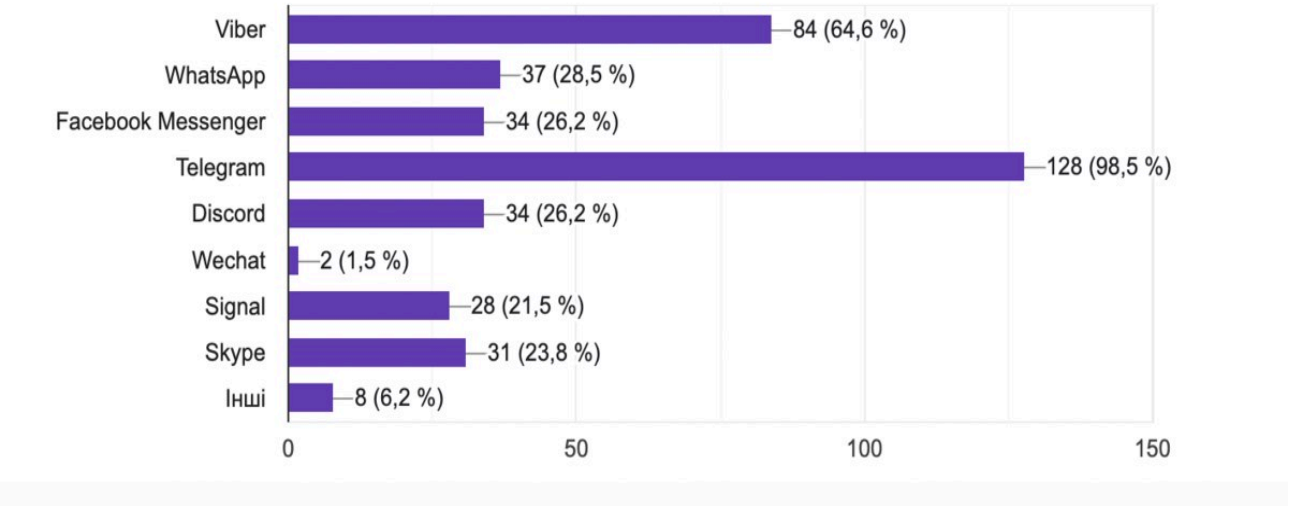

Рисунок 1.1 – Найпопулярніші месенджери в Україні [1]

У сучасному цифровому світі, де обсяг переданої інформації зростає щосекунди, питання конфіденційності, безпеки та надійності засобів комунікації виходить на перший план. З початком повномасштабної агресії проти України зросла потреба у незалежному, захищеному та прозорому месенджері, який би забезпечував повний контроль над передачею даних – як на рівні громадян, так і державних структур. Звичайні користувачі не можуть знати як працює месенджер «під капотом», тому обирають собі для користування той, який є зручнішим, а не безпечнішим, через що і трапляється так багато зливів особистих даних. Серед месенджерів найбільш захищеними опитані вважають Signal та Telegram (рис. 1.2).

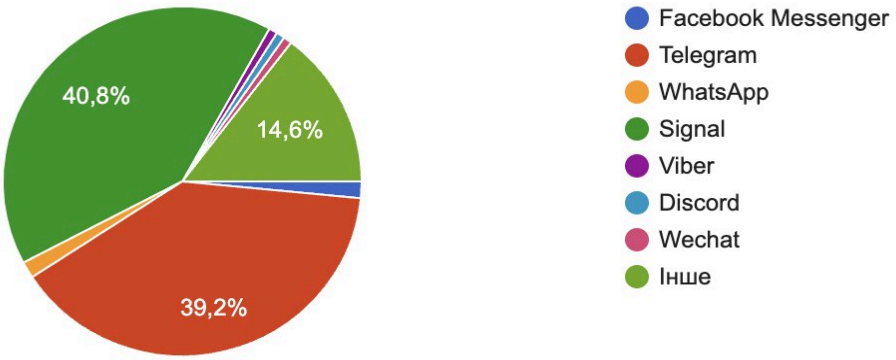

Рисунок 1.2 – Найбільш безпечні месенджери на думку користувачів [1]

Необхідно розібратися, що являють собою ці платформи.

Telegram – це універсальний месенджер, який підтримується на різних платформах. Він пропонує функцію наскрізного шифрування для чатів (відомих як «секретні чати»), а також підтримує відеодзвінки, VoIP, передачу файлів та інші корисні можливості [2].

Signal – це крос-платформний зашифрований месенджер, розроблений Signal Foundation. Він дає змогу обмінюватися приватним повідомленнями, файлами, аудіо, зображеннями та відео з підвищеним рівнем безпеки завдяки наскрізному шифруванню [3].

Facebook Messenger – це сервіс для обміну миттєвими повідомленнями, розроблений компанією Facebook. Він тісно інтегрований з основним додатком Facebook і працює на основі відкритого протоколу MQTT [4].

WhatsApp – це провідний приватний месенджер, спеціально розроблений для смартфонів, що надає користувачам широкий спектр можливостей для ефективної комунікації. Завдяки цій платформі, ви можете безперешкодно обмінюватися текстовими повідомленнями, надсилати зображення, ділитися відеофайлами та аудіозаписами, що робить спілкування більш насиченим та інтерактивним. Він став невід’ємною частиною повсякденного життя для мільйонів користувачів по всьому світу, пропонуючи зручний та надійний спосіб залишатися на зв’язку з друзями, родиною та колегами [5].

Viber – це застосунок для здійснення голосових дзвінків (VoIP) та обміну повідомленнями. Він прив’язується до вашого мобільного номера, але не використовує мобільну мережу для своєї роботи. Натомість, для дзвінків та обміну повідомленнями програмі необхідне інтернет-з’єднання. Є одним із найпопулярніших месенджерів [6].

Discord – це комплексна та багатофункціональна платформа, яка виходить за рамки звичайного обміну повідомленнями. Вона інтегрує в собі передові можливості миттєвої комунікації, потужні функції VoIP (голосового зв’язку через інтернет) та ефективні інструменти для цифрового розповсюдження контенту [7].

WeChat – це багатофункціональна мобільна платформа, розроблена компанією Tencent, яка надає користувачам можливість обмінюватися текстовими та голосовими повідомленнями. Крім того, для зручності використання на інших пристроях, WeChat також пропонує версії для персональних комп’ютерів та веб-переглядачів [8].

Попри велику кількість існуючих платформ , жодна з них не є повністю незалежною від впливу зовнішніх держав, приватних компаній або комерційних інтересів. Навіть при тому що в усіх них присутнє потужне шифрування, яке на даний час неможливо зламати, все одно вони можуть мати доступ до даних користувачів, а також в теорії надавати їх третім особам без попередньої згоди власників цих даних. Більшість з месенджерів є закритими пропрієтарними рішеннями, які працюють на серверах, розміщених за межами України. У разі кібератак, санкцій, блокувань чи зовнішнього тиску – доступ до даних, історії повідомлень або функціоналу може бути повністю втрачено.

У результаті аналізу наявних рішень та їхніх обмежень з’явилася ідея розробки національного месенджера, який функціонуватиме в межах закритої української інфраструктури та забезпечуватиме максимально можливий рівень безпеки при обміні критично важливою інформацією. Внутрішньою мережею, його зробить можливість реєстрації лише при наявності токена доступу, який надасть той, хто уповноважений це робити.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Метою кваліфікаційної роботи бакалавра є розробка мобільного чат-додатка «MChat» зі застосуванням токенів для аутентифікації, реалізованого на основі сучасних технологій React Native, Node.js та MongoDB.

Для досягнення поставленої мети були сформульовані такі завдання:

– проаналізувати сучасний стан real-time месенджерів;

– визначити функціональні і нефункціональні вимоги;

– спроектувати архітектуру системи;

– обґрунтувати вибір технічного стеку;

– реалізувати клієнтську та серверну частину;

– розробити базу даних;

– провести тестування та налагодження;

– забезпечити захист інформаційно-комп’ютерної системи.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розробленого програмного забезпечення та проектування програмного забезпечення

На сучасному етапі розвитку цифрових технологій обмін даними у реальному часі відіграє ключову роль як у побутовій комунікації, так і в державному управлінні, науковій діяльності та бізнесі. Враховуючи актуальні потреби у забезпеченні надійної та безпечної передачі інформації, особливо в умовах військового стану чи інших надзвичайних ситуацій, стає необхідним створення власного захищеного месенджера з повним контролем над передачею даних.

Визначальними критеріями для розробки такого програмного забезпечення є:

– використання сучасних механізмів шифрування даних;

– розробка авторизації користувачів за своїми кастомними токенами;

– використання системи аутентифікації на основі токенів (JWT) для захисту сесій та забезпечення цілісності користувацьких даних;

– можливість розширення кількості користувачів без втрати продуктивності системи;

– мінімальна затримка при передачі повідомлень;

– розміщення даних виключно на внутрішніх серверах під повним контролем розробників;

– уся система повинна бути доступною для користувачів, що володіють українською мовою;

– забезпечення доступу до системи через мобільні пристрої із будь-якими технічними характеристиками.

Для формування функціональних вимог, хорошим підходом є використання UML-діаграм, щоб наочно показати вимоги із використанням

акторів, таких як «Адміністратор» та звичайний «Користувач». Така схема зображена на рисунку 2.1.

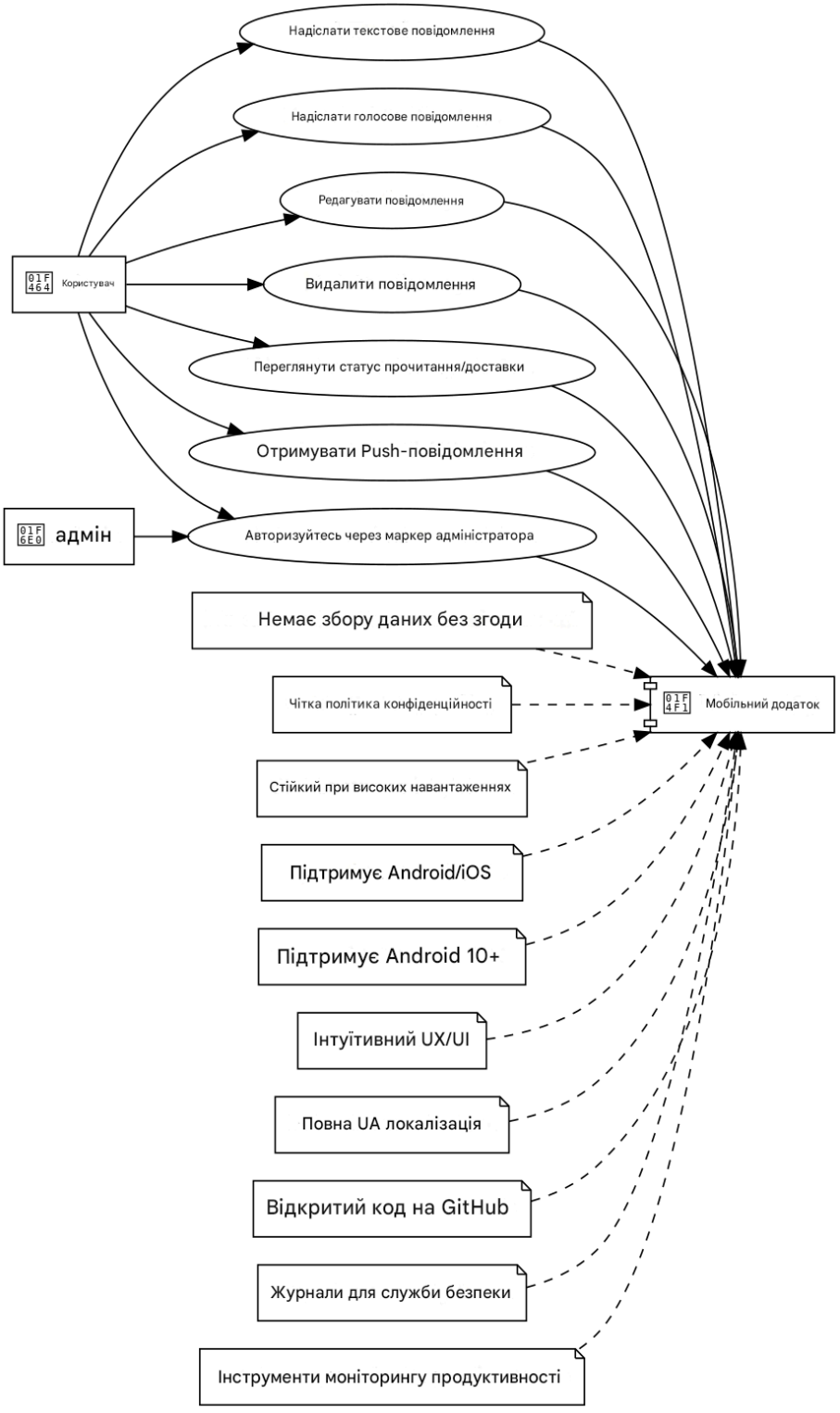

Рисунок 2.1 – UML-діаграма функціональних вимог

Функціональні вимоги до об’єкту розробки:

– надіслати текстове повідомлення;

– надіслати голосове повідомлення;

– редагувати повідомлення;

– видалити повідомлення;

– отримувати Push-повідомлення;

– переглянути статус прочитання/доставки;

– авторизуватися через маркер адміністратора;

– відсутність збору даних без згоди;

– чітка політика конфіденційності;

– стійкість при високих навантаженнях;

– підтримка Android/iOS;

– підтримка Android 10+;

– інтуїтивний UX/UI;

– повна UA локалізація;

– відкритий код на GitHub.

Нефункціональні вимоги:

– повна відмова від збору особистих даних без згоди;

– відкрита політика конфіденційності з прозорими умовами обробки даних;

– стабільна робота при високому навантаженні;

– платформи (Android, iOS);

– підтримка стабільної роботи для пристроїв із версією Android 10 і вище;

– зручний та інтуїтивно зрозумілий UX/UI;

– повна локалізація українською мовою;

– відкритий клієнтський і серверний код на GitHub;

– реєстрація критичних подій для службовців безпеки;

– інструменти реального моніторингу навантаження, активності та помилок.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

У сучасному світі розробки мобільних застосунків існує безліч технологій, які дозволяють створювати як нативні, так і кросплатформенні продукти. Традиційно для створення застосунків під Android використовуються Java або Kotlin, а для iOS – Swift або Objective-C (рис. 2.2) [9]. Ці мови й відповідні середовища розробки надають повний доступ до всіх нативних можливостей платформи, що дозволяє досягти максимальної продуктивності та стабільності, проте вимагає окремої розробки для кожної операційної системи.

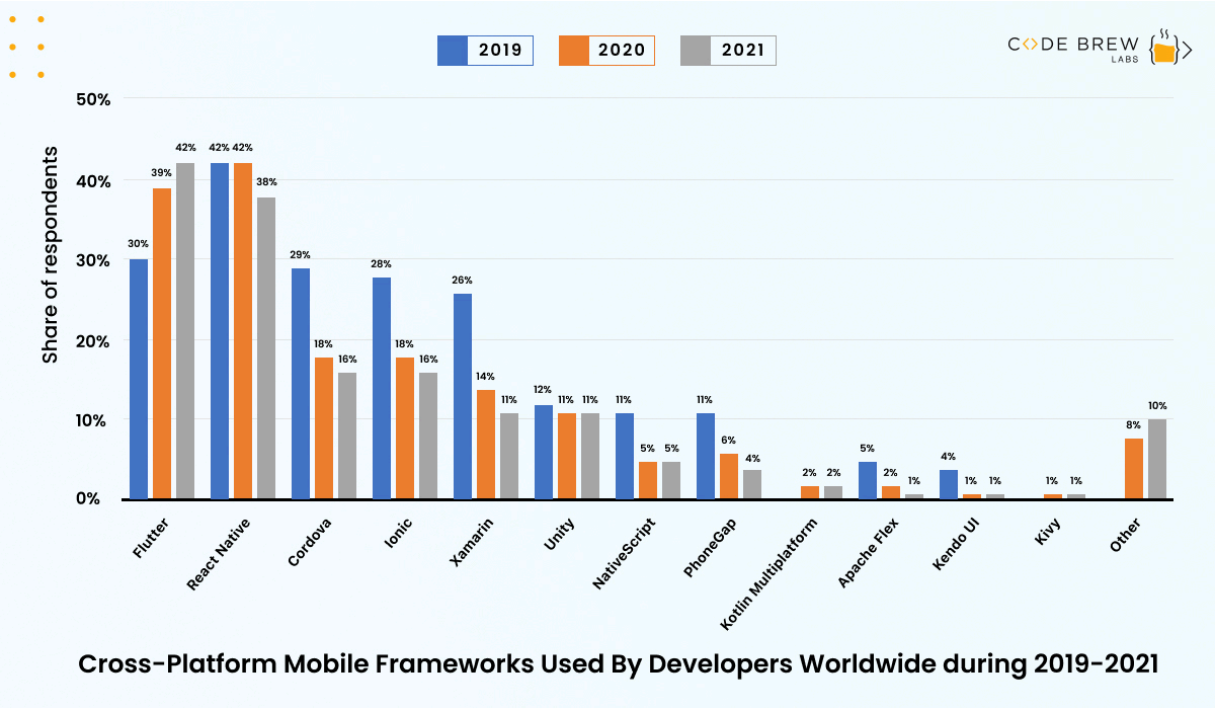

Рисунок 2.2 – Найпопулярніші фреймворки для мобільної розробки [9]

Окрім нативного підходу, з кожним роком усе більшого поширення набувають фреймворки для кросплатформної розробки. Однією з найпопулярніших технологій у цій галузі є Flutter – фреймворк від Google, який базується на мові Dart і дозволяє створювати візуально привабливі та

продуктивні застосунки. Flutter особливо цінується за свою гнучкість у роботі з UI та підтримку великої кількості віджетів.

Також широко використовується Xamarin – інструмент від Microsoft, що дозволяє писати код на C# і компілювати його в нативний код для Android та iOS. Ця технологія добре інтегрується з екосистемою .NET і Visual Studio, але має певні обмеження у гнучкості дизайну та продуктивності.

Іншим прикладом є Ionic – фреймворк, орієнтований на створення гібридних застосунків з використанням вебтехнологій: HTML, CSS і JavaScript. Такі застосунки працюють у WebView, що може впливати на швидкодію, проте забезпечує швидкий старт і просту розробку для невеликих проєктів.

На фоні цих рішень React Native займає особливе місце як фреймворк, що поєднує простоту JavaScript із можливістю створення нативних елементів інтерфейсу. Він дозволяє розробляти повноцінні мобільні застосунки для Android та iOS з єдиною кодовою базою, забезпечуючи при цьому доступ до нативних API і гнучкість розширення функціоналу. Саме завдяки такому балансу між зручністю, продуктивністю та спільнотою підтримки React Native став одним із найпоширеніших інструментів для створення сучасних мобільних застосунків [10]. Також спрощує роботу те, що сама розробка нагадує звичайну, та знайому багатьом верстку веб-сайтів, яка є дуже простою, тому маючи достатньо знань для того щоб займатися веб-розробкою, можна легко почати створювати мобільні застосунки з використанням React Native.

Як середовище розробки було обрано Visual Studio Code (VS Code) – один із найпопулярніших та найбільш гнучких редакторів коду серед сучасних розробників. Основними перевагами цього середовища є зручний та інтуїтивно зрозумілий інтерфейс, потужна система розширень, а також висока продуктивність навіть на комп’ютерах із середніми характеристиками. VS Code забезпечує повну підтримку екосистеми JavaScript/TypeScript та чудово інтегрується з інструментами, які активно використовуються під час створення мобільних застосунків на React Native.

Для контролю над типізацією було також використано TypeScript, який надає можливість писати код на JavaScript, проте із використанням статичної типізації. Це дозволяє уникати помилок із несумісністю типів даних та робить проект більш структурованим.

Було обрано Git як основну систему контролю версій для розробки, оскільки це найпопулярніша та найнадійніша платформа. Окрім того, Git забезпечує гнучку роботу з гілками, що дозволяє ефективно керувати паралельною розробкою та впроваджувати нові функціональні можливості без ризику пошкодження основної версії проекту.

Для зберігання та спільної роботи над кодом використовується GitHub – один із найпопулярніших хостингів репозиторіїв. Це дає можливість не лише зручно обмінюватися змінами з іншими розробниками, але й вести відкриту розробку, отримувати відгуки від спільноти, а також інтегрувати додаткові сервіси, такі як CI/CD, автоматичне тестування та аналіз коду. Завдяки такому підходу забезпечується прозорість розробки, контроль за всіма змінами в кодовій базі та можливість швидкого відновлення попередніх версій у разі потреби.

Серверна частина була розроблена за допомогою Node.js – середовища виконання JavaScript на стороні сервера. Вибір впав саме на цю технологію тому що це все ще той самий JavaScript, з яким я працював уже декілька років. Це дозволило швидко розібратися із роботою серверної частини, оскільки вже була необхідна для цього база знань. Як основний фреймворк для побудови серверної логіки було обрано Express.js, що дозволяє швидко та ефективно створювати REST API. Однією з головних переваг Express.js є його простота у створенні маршрутизації запитів, що значно спрощує розробку. Крім того, Express надає високу гнучкість у налаштуванні обробників запитів та middleware, що дозволяє кастомізувати поведінку сервера під конкретні вимоги проекту. Завдяки великій кількості готових модулів, доступних для розширення функціональності, Express стає ще більш потужним інструментом для розробки вебдодатків.

Для обміну повідомленнями в реальному часі була використана бібліотека Socket.IO, яка забезпечує постійний двосторонній канал зв’язку між сервером і клієнтом. Це дозволяє реалізувати ефективне спілкування в режимі реального часу, наприклад, для чатів або сповіщень. Як альтернативу, можна було використати багато вже готових рішень, таких як наприклад: Firebase, Rocket.Chat, Sendbird та багато інших. Мій вибір впав на socket.io, оскільки це дає максимальний контроль над усім, де всі рішення треба зробити з нуля, що дає впевненість у тому що розробники мають повне розуміння усіх процесів онлайн листування.

Основною частиною захищеності розробленої системи є те, що це повністю корпоративна внутрішня мережа, для доступу до якої необхідно мати спеціальний токен, який необхідно вказати при реєстрації, і який потім прив’язується до користувача, для того щоб можна було легко відслідкувати те, як він опинився в системі та адміна, який надав йому доступ. Така методика забезпечує максимальний рівень безпеки та дає відчуття захищеності користувачам, що і є основною метою цього програмного забезпечення.

Для реалізації механізму аутентифікації в системі застосовувалося використання JWT (JSON Web Tokens). Такий підхід дозволяє безпечно передавати дані між клієнтом і сервером, зберігаючи токени на клієнтській стороні для автентифікації запитів. Крім того, завдяки використанню JWT зменшується навантаження на сервер, оскільки перевірка автентифікації відбувається без необхідності зберігати сесії [11].

JSON Web Token (JWT) – це відкритий стандарт, детально описаний у RFC 7519, що являє собою елегантне рішення для безпечної та ефективної передачі інформації між різними сторонами. Його основна функція полягає у пакуванні даних у компактний, самодостатній JSON-об’єкт, який згодом можна легко пересилати через мережу.

JSON Web Token (JWT) може бути підписаний двома основними способами для забезпечення його безпеки. Останній метод є особливо надійним, оскільки підпис, створений за допомогою приватного ключа,

однозначно підтверджує, що токен був підписаний саме тією стороною, яка володіє цим закритим ключем, забезпечуючи високий рівень довіри до джерела інформації.

Ключовим аспектом безпеки JSON Web Token (JWT) є механізм його підпису, який може бути реалізований двома способами. Перший варіант – це симетричне шифрування за допомогою секретного ключа та алгоритму HMAC. У цьому випадку одна й та сама «таємниця» використовується як для створення підпису відправником, так і для його перевірки одержувачем. Другий, більш надійний метод, базується на асиметричному шифруванні [11].

Для гарантування високого рівня безпеки користувацьких даних, зокрема при зберіганні паролів, у розробленій системі було реалізовано механізм шифрування за допомогою бібліотеки bcrypt. Ця бібліотека є однією з найбільш надійних та широко визнаних у галузі кібербезпеки для хешування паролів. На відміну від простого шифрування, bcrypt застосовує криптографічні хеш-функції з додаванням «солі» (random salt), що значно ускладнює будь-які спроби злому паролів за допомогою атак типу brute-force або rainbow table.

Кожен пароль, який вводить користувач під час реєстрації або зміни облікових даних, проходить процес хешування з унікальною сіллю перед збереженням у базі даних. Таким чином, навіть у випадку несанкціонованого доступу до бази, зловмисник отримує лише набір хешів, які практично неможливо розшифрувати у зворотному напрямку без оригінального пароля.

Крім того, під час авторизації користувача система не зберігає і не порівнює відкриті паролі, а здійснює перевірку шляхом порівняння вхідного пароля (який також хешується) з уже збереженим хешем у базі. Це забезпечує ще один рівень захисту й відповідає сучасним стандартам безпеки у веброзробці.

Застосування bcrypt дозволило створити надійний захист конфіденційних даних користувачів і зробило систему більш стійкою до потенційних атак. Bcrypt – адаптивна криптографічна функція формування ключа, що використовується для безпечного зберігання паролів. Розробники: Нільс Провос

і David Mazières. Функція заснована на шифрі Blowfish, вперше представлена на USENIX у 1999 році. Для захисту від атак за допомогою райдужних таблиць bcrypt використовує сіль (salt); крім того, її можна сповільнити, щоб ускладнити атаки перебором [12].

Також, щоб мати можливість десь зберігати голосові повідомлення, було використано можливості Firebase Storage.

В системі присутнє логування за допомогою бібліотеки logger. Після кожної відносно важливої інтеракції користувача із додатком, відбувається запис логу на сервер, для того, щоб в майбутньому легше відтворювати можливі баги та мати можливість моніторити активність користувачів.

Щоб мати можливість зручно керувати запитами на сторону сервера було використано бібліотеку axios.js. Axios – це бібліотека JavaScript для виконання HTTP-запитів із Node.js, XMLHttpRequest або браузера. Як сучасна бібліотека, вона заснована на Promise API, Axios має деякі переваги, такі як захист від атак із підробкою міжсайтових запитів (CSFR) [13].

Для зручної роботи із запитами на API зі сторони клієнта, використовується бібліотека @tanstack/react-query. TanStack Query (раніше відомий як React Query) часто описують як відсутню бібліотеку для отримання даних для вебзастосунків, але, якщо говорити більш технічно, вона значно спрощує отримання, кешування, синхронізацію та оновлення стану сервера у ваших вебзастосунках [14].

Для тестування React Native-додатків широко використовується бібліотека @testing-library/react-native, яка дозволяє імітувати поведінку користувача – наприклад, натискання кнопок, введення тексту у форму, прокручування списків тощо.

Також в процесі створення месенджера, було створено графічний інтерфейс для адмін-панелі, щоб адміністратор системи легко міг створювати нові токени, видаляти старі, моніторити активність користувачів та дивитися логи.

Для пришвидшення розробки було використано бібліотеку React.js, розробка з використанням якої дуже нагадує розробку на React Native, що є хорошим плюсом з точки зору економії часу. React (раніше відомий як React.js або ReactJS) – це відкрита JavaScript-бібліотека, призначена для створення користувацьких інтерфейсів. Її основне завдання – вирішувати проблеми, пов’язані з частковим оновленням вмісту вебсторінки, що є типовим викликом при розробці односторінкових застосунків [15].

Було також використано готові UI-компоненти із бібліотеки Material-UI. Це дозволило думати над бізнес-логікою веб-додатку, а не зациклюватися на розробці візуальних компонентів та їхньої логіки.

РОЗДІЛ 3
РОЗРОБКА ДОДАТКА «MCHAT» ІЗ ЗАСТОСУВАННЯМ ТОКЕНІВ ДЛЯ АУТЕНТИФІКАЦІЇ

3.1 Практична реалізація об’єкта проектування

В першу чергу створюється серверна сторона застосунку, оскільки згідно встановлених правил розробки, основна частина логіки має бути розташована на стороні бекенду. Мною було обрано фреймворк express.js що працює на node.js. Із усіх фреймворків із якими я мав справу для написання серверної частини різноманітних систем, він є найпростішим. Можна легко створити простий сервер написавши всього декілька рядків коду. Розглянемо на практиці як це можна зробити.

Перш за все, необхідно переконатися чи встановлено у нас npm та node.js. Якщо ні, треба встановити, перейшовши на офіційні сайти даних технологій, завантаживши та виконавши процес інсталяції. Після цього ми маємо знову переконатися чи все було успішно встановлено. Це можна зробити дуже просто, просто зайшовши в будь-який термінал, який надає поточна ОС та написати декілька потрібних команд для перевірки версії технології (рис. 3.1). Якщо все так як на рисунку, значить інсталяція пройшла успішно і можна починати розробляти програмне забезпечення, тобто переходити на наступний етап.

Рисунок 3.1 – Перевірка версій встановлених технологій

Також треба мати попередньо встановлений редактор коду в якому буде зручно працювати, в моєму випадку це Visual Studio Code. Щоб почати роботу над проектом, спочатку необхідно створити папку та відкрити її в обраному

IDE. Тепер можна ініціалізувати проект командою npm init. Ця команда створить папку node_modules, де будуть базові модулі для роботи в середовищі node.js. Також можна помітити новостворені файли package.json та package-lock, ці файли містять інформацію про сам проект, встановлені залежності та скрипти, які будуть використані для запуску різноманітних кастомізованих завдань.

Для роботи з express.js треба його встановити, це робиться дуже просто, достатньо написати в консолі: npm install express, і фреймворк автоматично буде додано до проекту.

Дуже хорошою практикою є налаштування архітектури проекту на самому початку розробки, щоб потім над цим довго не сидіти та мати краще розуміння про те, що в проекті де знаходиться.

На рисунку 3.2 можна побачити стандартну структуру папок та файлів простого express.js проекта.

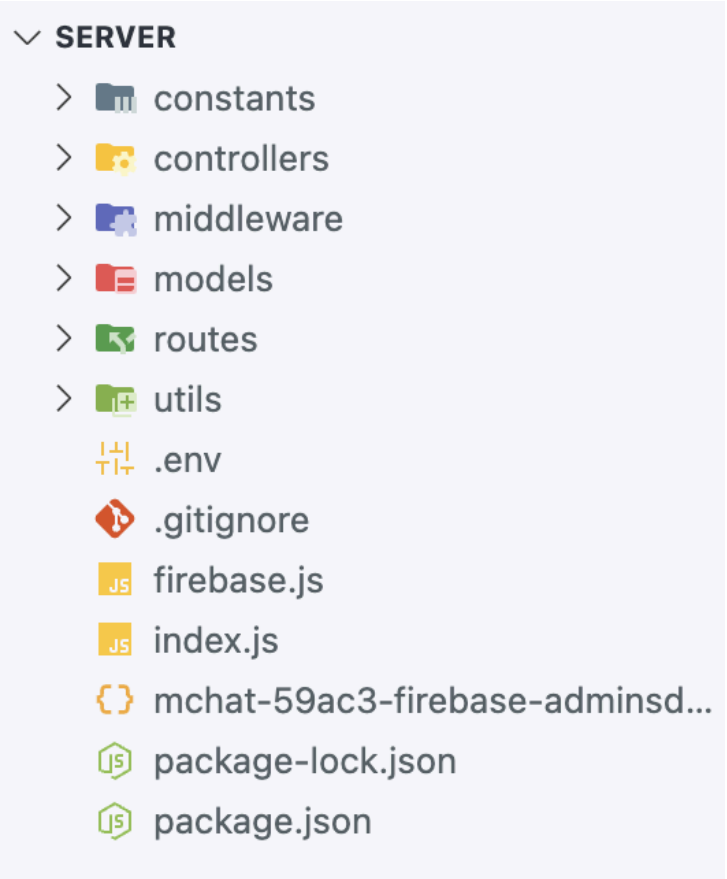

Рисунок 3.2 – Архітектура серверної частини додатку

В папці constants знаходяться глобальні константні значення які будуть використані у всьому проекті. Винесення певних даних, які десь дублюються, в окремий файл або просто експортну змінну є хорошою практикою, оскільки вона надає можливість змінювати значення у всіх місцях одночасно, де вона використовується, також це сприяє зменшенню дублювання значень.

Папка controllers містить контроллери, які по факту є методами, які приймають дані від клієнта, та надсилають йому відповідь.

В middleware міститься проміжний код, що буде прив’язуватися до кожного запиту. В нашому випадку це буде робитися для організації роботи JWT аутентифікації.

Папка models містить моделі для записів бази даних, тобто детальний опис кожного поля певної сутності.

В папці routes, розміщують роути, на які потім клієнт буде надсилати свої запити для отримання або модифікації даних з бази даних.

У папку utils зазвичай виносять функції, які часто повторюються в процесі розробки, аналогічно глобальним константам. Все це робиться для того щоб проект був легко адаптованим до майбутніх змін, для мінімізації виникнення багів та розбіжностями в логіці.

Файли із розширенням .env містять дуже чутливу інформацію. Це можуть бути секретні токени, ключі та адреси серверів.

Головним файлом проекту є файл index.js в якому міститься запуск сервера, самі сокети для роботи із real-time повідомленнями, підключення middleware та роутів.

На початку роботи необхідно створити та запустити сервер (ліст. 3.1).

Лістинг 3.1 – Створення та запуск простого сервера на express.js

```
const express = require("express");
const app = express();

app.use(express.json());
app.use(cors());
```

```
const expressServer = app.listen(port, (req, res) => {
 console.log(`server: ${port}`)
});
```

кінець лістингу 3.1

Після цього можна перейти до створення моделей та контроллерів. Зазвичай починають із авторизації та в нашому випадку із токенів, які потрібні для реєстрації. Для цього першим ділом необхідно створити модель користувача (ліст. 3.2).

Лістинг 3.2 – Модель схеми користувача

```
const moongoose = require('mongoose');
const { v4: uuidv4 } = require('uuid');
const userSchema = new moongoose.Schema(
{
   name: { type: String, required: true, minLength: 3, maxLength: 30 },
    email: { type: String, required: true, unique: true, lowercase: true
},
   password: { type: String, required: true, minLength: 8 },
   registerToken: { type: String, required: false, unique: true },
   phone: { type: String, required: false, unique: true },
   tag: { type: String, required: false, unique: true },
    playerId: { type: String, required: false, unique: true, default:
uuidv4 },
 },
 {
   timestamps: true,
 },
);
const userModel = moongoose.model('User', userSchema);
module.exports = userModel;
```

кінець лістингу 3.2

Зараз можна перейти до створення моделі токена. Код цієї моделі зображений у лістингу 3.3.

Лістинг 3.3 – Модель схеми токена

```
const mongoose = require('mongoose');
const tokenSchema = new mongoose.Schema(
```

```
 {
   token: { type: String, required: true, unique: true },
   isUsed: { type: Boolean },
   adminId: { type: mongoose.Schema.Types.ObjectId, ref: 'Admin',
required: true },
 },
 {
   timestamps: true,
 },
);
module.exports = mongoose.model('Token', tokenSchema);
```

кінець лістингу 3.3

Створивши необхідні для авторизації моделі, необхідно також створити утиліту, яка буде допомагати зі створенням JWT для забезпечення безпеки додатку (ліст. 3.4). В цьому лістингу можна побачити що в процесі створення токена використовується пакет jsonwebtoken, який надає зручний інтерфейс для легкої роботи із JWT. Також для того щоб токен не можна було підробити, цей пакет використовує унікальний ключ, який передається при його створенні. В мене він зберігається в змінних середовища, що гарантує безпеку та унеможливлює його взлом або викрадення.

Лістинг 3.4 – Утиліта для створення JWT

```
const jwt = require("jsonwebtoken");
const createToken = (_id) => {
     const jwtkey = process.env.JWT_SECRET_KEY;
     return jwt.sign({ _id }, jwtkey, { expiresIn: "30d" });
};
```

кінець лістингу 3.4

Зараз найкращим рішенням буде перейти до розробки контроллера для створення нового токена. Весь код контроллера знаходиться в додатку А.

Схема створення нового токена є доволі простою. Головною частиною є його генерація за допомогою пакета crypto. Так як токен може створити лише адміністратор системи, то для цього необхідно передавати у запит змінну adminId, яка має містити у собі валідний та наявний у базі даних запис із

такими унікальним ідентифікатором. Для створення нового токена авторизації, було використано генерацію рядка, який складається із 30 абсолютно рандомних символів та потім транслюється в hex-формат. Також кожен токен прив’язується до адміна, який його створив, і у разі успішного створення, це все логується.

Тепер маючи необхідну кодову базу, можна створити контроллер для реєстрації нового користувача. Код цього контроллера знаходиться в додатку Б.

Сам алгоритм доволі простий і не має якихось особливостей окрім використання кастомного токена авторизації. Ось список дій, які виконуються при реєстрації:

1) витягуються дані з тіла запиту;

2) відбувається пошук користувача в базі даних користувачів через адресу електронної пошти;

3) якщо користувач з такою поштою вже зареєстрований, система викине помилку про це і процес реєстрації буде зупинено;

4) далі проводиться валідація всіх даних, перевіряється правильність переданої інформації та перевірка паролю на те чи є він достатньо безпечним та складним, щоб унеможливити його злом;

5) після цього перевіряється чи існує такий токен авторизації, який є ідентичним отриманому від користувача та якщо такий наявний в базі даних, і він при цьому ще є невикористаним, створюється нова модель користувача із попередньо переданими даними;

6) далі відбувається процес шифрування пароля через пакет bcrypt, який використовує певну кількість кругів «соління», в моєму випадку їх 10, це підвищує безпеку системи в рази;

7) потім відбувається збереження нового користувача в базу даних та віддача результату виконання функції користувачу.

Після того як ми створили контроллер, необхідно створити також і роут, за яким можна буде надіслати запит на сервер для певних маніпуляцій із даними. Це робиться доволі просто. Все що необхідно, це у папці routes створити файл

роута, наприклад для маніпуляцій із обліковим записом користувача (ліст. 3.5), та додати туди потрібний код.

Лістинг 3.5 – Код роута для маніпуляцій із обліковим записом

```
const express = require('express');
const {
 registerUser,
 loginUser,
 findUser,
 getUser,
 findUsersByNameOrTag,
 checkPassword,
 updateUser,
} = require('../controllers/userController');

const router = express.Router();

router.post('/register', registerUser);
router.post('/login', loginUser);
router.get('/find/:userId', findUser);
router.get('/', getUser);
router.get('/find', findUsersByNameOrTag);
router.post('/check', checkPassword);
router.patch('/:userId', updateUser);
module.exports = router;
```

кінець лістингу 3.5

Маючи вже готовий роут, його треба підключити в головний файл – index.js (ліст. 3.6). Провівши такі маніпуляції із кодом, ми маємо готовий маленький сервер, який уже можемо запустити командою node index.js. Тепер можемо тестувати нашу систему.

Лістинг 3.6 – Підключення роута в файл index.js

```
const userRoute = require("./routes/userRoute");
app.use("/api/users", userRoute);
```

кінець лістингу 3.6

Маючи готові запити, необхідно якось зробити функціонал real-time спілкування, для того щоб користувач одразу після зміни даних на сервері,

отримував про це повідомлення. Це дуже корисно у нашому випадку, коли ми хочемо створити real-time мессенджер.

Із цим нам допоможе пакет socket.io, що є обгорткою над протоколом передачі даних Websockets. Для того щоб ним користуватися, спочатку необхідно його встановити командою npm install socket.io. Тепер можемо користуватися всіма можливостями цієї бібліотеки.

Все що необхідно, це створити екземпляр класу, який цей пакет надає, передати туди змінну в якій зберігається наш express server і при необхідності задати певні правила передавши в нього конфігурацію (ліст. 3.7).

Лістинг 3.7 – Створення socket.io сервера

```
const io = new Server(expressServer, {

 cors: {
   origin: "*",
   methods: ["GET", "POST", "PUT", "DELETE", "PATCH"],
 },

});
```

кінець лістингу 3.7

Тепер необхідно запустити сервер, що дасть можливість працювати із сокетами. Це робиться дуже просто, і показано на лістингу 3.8. Тут можна побачити що викликається метод on в уже створеного попередньо екземпляра класу Server. В сам метод передається колбек функція, що приймає параметром змінну socket, поле id якої дозволить ідентифікувати активного користувача та виконувати із ним певні дії.

Лістинг 3.8 – Запуск сервера socket.io та робота із сокетами

```
io.on("connection", (socket) => {

 socket.on("sendMessage", (message) => {
   const user = onlineUsers.get(message.recipientId);

   if (user) {
     io.to(user.socketId).emit("getMessage", message);
```

```
      io.to(user.socketId).emit("getNotification", {
        senderId: message.senderId,
        isRead: false,
        date: new Date(),
      });
    }
  });

  socket.on("typingStart", ({ chatId, senderId, recipientId }) => {
    const recipient = onlineUsers.get(recipientId);

    if (recipient) {
      io.to(recipient.socketId).emit("typingStart", { chatId, senderId });
    }
  });
});
```

кінець лістингу 3.8

Таким чином, маючи достатньо хороші знання мови JavaScript та користуючись вищенаведеним прикладом, можна легко створювати різноманітний функціонал для власних потреб.

Тепер можна переходити до розробки клієнтської частини додатку.

Так як використовується фреймворк React Native, спочатку необхідно його встановити. В попередніх прикладах, де розповідалося про створення серверної частини, було використано пакетний менеджер NPM. Використаємо його і для клієнтської частини.

Для встановлення React Native запустимо команду в терміналі: npm install -g react-native-cli. Тепер ми маємо встановлений фреймворк, це означає що можна створювати новий проект. Це робиться легко, через ще одну команду: npx react-native init MChat, де MChat це назва нашого застосунка.

Після успішного створення проекту, також хорошою практикою є створення архітектури, щоб у процесі розробки менше відволікатися, та наперед мати гарно структуровану інфраструктуру.

На рисунку 3.3 зображена структура проекту клієнтської частини, що відображає стандартний підхід до організації коду в сучасному застосунку. Вона включає всі необхідні директорії для розробки, тестування та підтримки додатку на різних платформах.

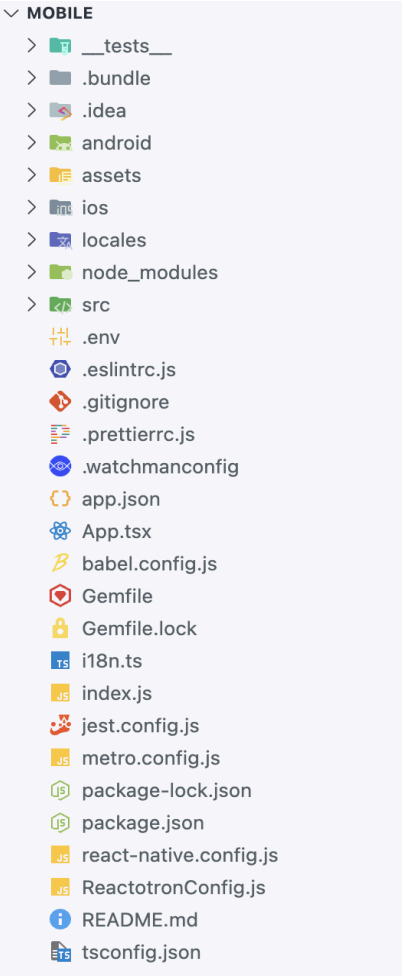

Рисунок 3.3 – Структура проекту клієнтської частини

На вищезгаданому рисунку можна побачити дуже багато файлів та папок, і кожна ця одиниця має важливу роль у роботі застосунка.

Папка __tests__ має юніт-тести для того, щоб при створенні нового функціоналу, або оновленні старого, мінімізувати кількість багів та невідповідностей в бізнес-логіці або попередити виникнення серйозних конфліктів серед компонентів.

В папці android, знаходиться нативна частина андроїд версії додатку. Там присутній нативний код платформи та всі необхідні залежності для коректної роботи на Android-платформі.

Директорія assets має в собі всі фото, іконки та шрифти, які використовуються в роботі додатку. Це дуже зручно, оскільки всі вони знаходяться в одній папці, що спрощує роботу із інтерактивними елементами.

Так само як папка android, папка ios містить в собі нативні модулі для підтримки роботи застосунку на платформі ios, що є дуже корисним.

В locales знаходяться файли інтернаціоналізації, що додає функіонал зміни мови, що є зручним тоді, коли наш додаток підтримує різні мови, це робить його більш міжнародним.

Обов'язковою частиною кожного проекту, в якому використовується мова програмування JavaScript, є папка node_modules, де знаходяться всі встановлені залежності, через використання який ми маємо можливість створювати функціонал на вищезгаданій мові.

Папка src містить в собі файли, які будуть створені в процесі розробки, і які є основою нашого застосунка. Це можуть бути різні файли: компоненти, константи, утиліти, конфігурації, мультимедіа, файли типізації та багато інших.

Для роботи з платформою Android, потрібно спочатку завантажити та встановити Android Studio. Розробка для ios можлива лише з використанням операційної системи MacOS через вбудований IDE XCode.

Маючи необхідну структуру проекту, можна починати розробку, для запуску збірки, треба запустити в терміналі команду npm start, після чого хорошою практикою є відкриття іншого терміналу в тій же папці, що й попереднього, і запуск там потрібної команди. Для запуску на платформі android, використовують npx react-native run-android, для ios команда майже ідентична: npx react-native run-ios.

Хорошою практикою на початку проекту, є налаштування навігації для різних екранів, з цим допоможе бібліотека @react-navigation/native. В попередньо створеній папці navigation, створимо компонент Navigation.tsx, який буде містити у собі налаштовану навігацію для різних роутів. Код файл компонента для простої навігації зображений на лістингу 3.9.

Лістинг 3.9 – Код файлу Navigation.tsx

```
const Navigation = () => {
 return (
   <NavigationContainer>
```

```
      <Stack.Navigator initialRouteName="Start">
        {privateRoutes.map((route: IRoute) => (
          <Stack.Screen {...route} key={route.name} />
        ))}
      </Stack.Navigator>
    </NavigationContainer>
  );
};
export default Navigation
```

кінець лістингу 3.9

Тепер необхідно налаштувати механізм стору, для зберігання глобальних даних, у моєму випадку це буде Context API, яке надає сама платформа React Native. Його було використано через популярність, простоту використання та можливість використання без додавання в проект нових залежностей. Створимо простий контекст для маніпуляцій із чатами (ліст. 3.10).

Лістинг 3.10 – Контекст для маніпуляцій з чатами

```
export const ChatContext = createContext();

export const ChatProvider = ({children}) => {
  return (
    <ChatContext.Provider
      value={{ }}>
      {children}
    </ChatContext.Provider>
  );
};
```

кінець лістингу 3.10

Навіть для того хто ніколи не працював із контекстом, легко можна розібратися у вищенаведеному коді. Все що потрібно, це створити змінну, яка буде містити в собі виклик функції createContext. Після чого з її допомогою треба створити обгортку яка буде батьківським компонентом для всіх інших компонентів, де будуть використовуватися змінні, що знаходяться в новоствореному контексті.

На основі бібліотеки @tanstack/react-query, було створено кастомний хук useAuthMutation, що дозволяє при кожному запиті, перевіряти чи є користувач

авторизований в системі. За принципом роботи, він є схожим на middleware, оскільки при кожному запиті перевіряє HTTP-код відповіді від сервера, і якщо запит був створений користувачем, який не пройшов аутентифікацію, виконується вихід із системи. Це зроблено для того щоб зловмисники не могли отримати доступ до будь-яких, навіть кешованих даних. Це не просто хороша, а навіть необхідна практика для кожної системи. Код присутній в лістингу 3.11.

Лістинг 3.11 – Кастомний хук useAuthMutation

```
import {useContext} from 'react';
import {
 useMutation,
 UseMutationOptions,
 UseMutationResult,
} from '@tanstack/react-query';
import {AuthContext} from '@/context/Auth/AuthContext';
interface ErrorWithStatus {
 status?: number;
 message?: string;
}
interface MutationOptions<TData, TVariables, TError>
 extends UseMutationOptions<TData, TError, TVariables> {}

export const useAuthMutation = <
 TData,
 TVariables,
 TError extends ErrorWithStatus,
>(
 options: MutationOptions<TData, TVariables, TError>,
): UseMutationResult<TData, TError, TVariables> & {isLoading: boolean} =>
{
 const {logout} = useContext(AuthContext);

 const mutation = useMutation<TData, TError, TVariables>(options);

 if (mutation?.error?.status === 401 || mutation?.error?.status === 403)
{
   logout();
 }

 return {
   ...mutation,
   isLoading: mutation.isPending,
 };
};
```

кінець лістингу 3.11

Для того щоб кожен запит до серверу автоматично містив у собі необхідний accessToken і не призводив до повернення кодів помилки, які вказують на відсутність авторизації (наприклад, HTTP 401 Unauthorized), необхідно реалізувати механізм додавання токена до заголовків HTTP-запитів. У випадку використання бібліотеки axios, яка є одним із найпоширеніших рішень для роботи з HTTP у JavaScript-застосунках, це реалізується за допомогою interceptors (перехоплювачів).

Така централізована обробка також дозволяє легко реалізувати повторну авторизацію при закінченні терміну дії токена або обробку помилок (наприклад, оновлення токена через refresh token). Фрагмент коду, в якому до кожного запиту автоматично додається accessToken, наведено на лістингу 3.12.

Лістинг 3.12 – Використання react-hook-form для полів реєстрації

```
const client = axios.create({
 baseURL: `${SERVER_URL}/api`,
});

client.interceptors.request.use(
 async config => {
   const accessToken = await SInfo.getItem('accessToken', {
     sharedPreferencesName: 'prefs',
     keychainService: 'keychainService',
   });
   config.headers.Authorization = `Bearer ${accessToken}`;
   return config;
 },
 error => {
   console.log('Error request', error);
   return Promise.reject(error);
 },
);
```

кінець лістингу 3.12

Після цього можна перейти до створення форми реєстрації. Для зручного управління формами, доцільно використати бібліотеку react-hook-form, яка спрощує роботу над валідацією та управлінням даних, та збирає всі дані з полів

у одному місці. Для її використання, необхідно встановити вищеописаний пакет, та почати ним користуватися. Приклад можна побачити у лістингу 3.13.

Лістинг 3.13 – Використання react-hook-form для полів реєстрації.

```
const {
   reset,
   control,
   handleSubmit,
   formState: {errors},
 } = useForm({
   resolver: yupResolver(registerSchema),
   defaultValues: {
     email: '',
     password: '',
     name: '',
     phone: '',
     token: '',
   },
 });
```

кінець лістингу 3.13

Після створення змінних, можемо їх використати у самій розмітці. Це дуже просто, необхідно імпортувати компонент-обгортку Controller та обгорнути ним компонент поля вводу, так можна використовувати можливості цієї бібліотеки. Це є хорошою практикою в розробці програмного забезпечення з використанням React Native. Використання показано в лістингу 3.14.

Лістинг 3.14 – Використання react-hook-form для полів реєстрації.

```
<Controller
      control={control}
name="token"
render={({field: {onChange, value}}) => (
<Input
      placeholder={t('input.AccessToken')}
value={value}
onChangeText={onChange}
error={errors?.token?.message || formErrors?.token}
/>
)}
/>
```

кінець лістингу 3.14

Всі поля виглядають приблизно однаково і є зображеними на рисунку 3.4.

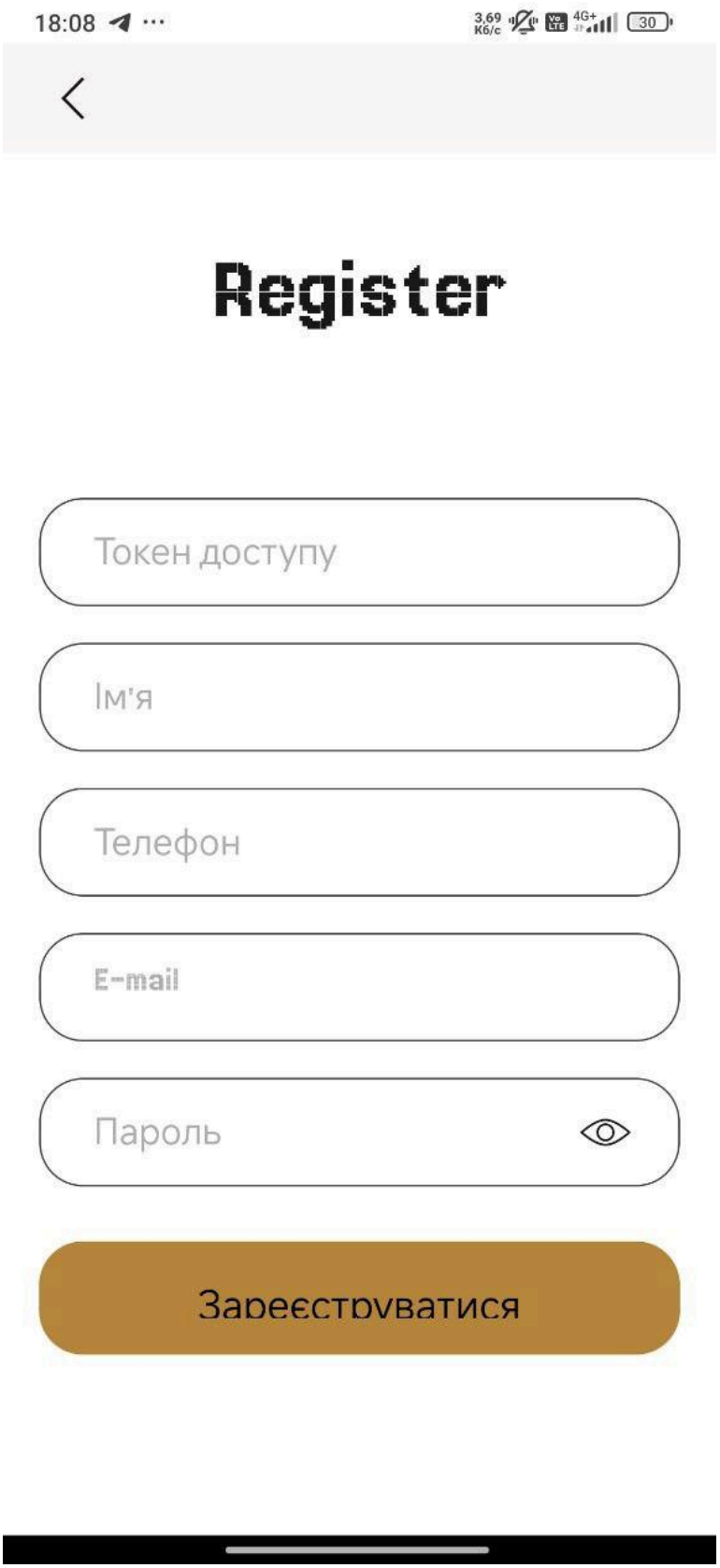

Рисунок 3.4 – Візуальний інтерфейс для полів реєстрації

Ввівши всі необхідні дані в поля форми реєстрації, необхідно їх надіслати на сервер, з цим допоможе вищезгаданий хук useAuthMutation. Приклад його використання можна побачити на лістингу 3.15.

Лістинг 3.15 – Надсилання даних на сервер

```
const {mutate: onRegister, isLoading} = useAuthMutation({
   mutationFn: Api?.auth?.register,
   onError: onRegisterError,
   onSuccess: onRegisterSuccess,
 });
 const onSubmit = (data: any) => {
   onRegister({...data, registerToken: data.token});
 };
```

кінець лістингу 3.15

Для роботи із real-time даними, необхідно встановити необхідну для цього бібліотеку. У моєму випадку це socket.io, яка надає все необхідне для використання цього протоколу, якщо готовий сервер, де він також використовується. Для прикладу можна взяти механізм прочитання повідомлення та повідомлення сервера про це. Цей процес зображено на лістингу 3.16.

Лістинг 3.16 – Фрагмент коду, де використовується socket.io

```
useEffect(() => {

   if(socket === null) return;
   socket.on('messageRead', ({messageId}) => {
     setMessages(prevMessages =>
       prevMessages.map(message =>
             message._id === messageId ? {...message, isRead: true} :
message,
       ),
     );
   });

   return () => {
     socket.disconnect();
   };
 }, [user]);
```

кінець лістингу 3.16

Лістинг 3.13 демонструє фрагмент коду, в якому реалізовано обробку подій у реальному часі за допомогою бібліотеки socket.io у React Native-застосунку. Використовується хук useEffect, що відповідає за виконання певної логіки після моніторингу змін у переданих залежностях. У цьому випадку як залежність вказано змінну user, тому колбек-функція всередині useEffect виконується кожного разу, коли user змінюється.

У тілі ефекту відбувається перевірка на наявність з’єднання через сокет: якщо об'єкт socket не ініціалізований (тобто дорівнює null), функція завершується достроково. Якщо ж з’єднання є, застосунок підписується на подію messageRead, яка надсилається сервером у момент, коли певне повідомлення в чаті позначається як прочитане. Коли ця подія надходить, компонент оновлює стан, використовуючи setMessages – попередній список повідомлень перебирається методом map, і якщо message._id збігається з messageId, що надійшов із сервера, відповідне повідомлення оновлюється: поле isRead змінюється на true.

Таким чином забезпечується реактивна поведінка інтерфейсу: прочитане повідомлення одразу позначається відповідним статусом без необхідності перезавантаження або ручного оновлення сторінки. У фіналі useEffect повертає функцію очистки, в якій здійснюється від’єднання сокету під час демонтажу компонента або при зміні користувача, що дозволяє уникнути зайвих підписок і витоків пам’яті. Цей механізм – простий, але ефективний приклад того, як можна організувати реалтайм-оновлення стану в React Native, забезпечуючи швидкий і зручний обмін інформацією між клієнтом і сервером.

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

Тестування дозволяє мінімізувати ризики появи помилок у фінальній версії застосунку, а також забезпечує його надійну й передбачувану поведінку під час використання реальними користувачами. Процес тестування базується на кількох загальноприйнятих принципах. По-перше, тестування показує

наявність дефектів, але не їх відсутність. Іншими словами, навіть якщо система проходить усі тести, це не гарантує повну відсутність помилок. По-друге, чим раніше виявлено помилку, тим меншими є витрати на її усунення. Тому тестування повинно починатися ще на ранніх етапах розробки. Крім того, існує принцип скупчення дефектів, згідно з яким більшість помилок концентрується в невеликій кількості модулів. Також важливо усвідомлювати, що неможливо протестувати абсолютно всі варіанти поведінки програми, тому необхідно визначити ключові випадки для перевірки. Крім того, після внесення змін у код необхідно повторно запускати тести, щоб перевірити, чи не порушилися інші частини програми (це називається регресійним тестуванням).

У сучасній розробці застосунків з використанням бібліотек React і React Native велика увага приділяється автоматизованому тестуванню. Зокрема, одним із найпоширеніших інструментів для цього є фреймворк Jest, який дозволяє створювати юніт-тести, інтеграційні тести та snapshot-тести. Його перевагами є швидкість виконання, підтримка моків (імітацій об'єктів або функцій), простий синтаксис, а також активна підтримка спільноти..

Види тестування, що застосовуються в процесі розробки, можна класифікувати за рівнем деталізації. Модульне (або юніт-тестування) перевіряє окремі функції чи компоненти незалежно від інших частин системи. Це дозволяє швидко локалізувати помилки у вузьких місцях. Інтеграційне тестування перевіряє взаємодію між модулями – наприклад, як компонент працює із зовнішнім API чи зі станом додатку. Регресійне тестування виконується після змін у коді та перевіряє, чи не зламалися інші частини системи. Snapshot-тести особливо популярні у React-екосистемі – вони дозволяють зберігати «знімки» виводу компонентів і при кожному новому запуску перевіряти, чи не відбулося небажаних змін у структурі. Автоматизоване тестування має численні переваги. Воно дозволяє запускати тести регулярно, наприклад, при кожному збереженні змін у коді або при запуску CI/CD-процесу. Це зменшує людський фактор, пришвидшує перевірку програми і дає впевненість у тому, що система працює стабільно. У React та

React Native автоматизовані тести допомагають уникнути багів, які можуть проявитися лише на окремих пристроях або при специфічній поведінці користувача.

Окрему увагу в контексті мобільної розробки варто звернути на те, що тестування інтерфейсів користувача є критичним, адже взаємодія з додатком відбувається саме через UI. Тестування має враховувати різні сценарії: натискання кнопок, поява помилок, завантаження даних, реакція на недоступність інтернету тощо. Це дозволяє гарантувати, що застосунок буде працювати передбачувано на різних пристроях з різною роздільною здатністю, операційною системою та версією. Таким чином, тестування є невід'ємною складовою розробки якісного програмного забезпечення. Його правильне впровадження дозволяє не лише вчасно виявити критичні помилки, але й значно підвищити загальну стабільність, масштабованість та зручність користування інформаційною системою. У розробці фронтенд-додатків, зокрема на React Native, тестування дає змогу забезпечити передбачувану поведінку компонентів, зменшити витрати часу на відлагодження і створити надійний, перевірений програмний продукт, готовий до використання кінцевим споживачем.

Фреймворк React Native вже має встановлений окремий фреймворк для тестування – Jest. Він і буде використовуватися в процесі тестування. За його допомогою можна тестувати також і правильність рендерингу компонентів, якщо при цьому використовувати ще одну популярну бібліотеку – @testing-library/react-native. Вона дозволяє дуже легко писати різноманітні тести для виконання яких, необхідний рендеринг потрібного компонента. Маючи мінімальну кількість знань, можна написати тести для якогось метода, які перевірять правильність його логіки. Для прикладу можна взяти функцію, яка генерує псевдовипадковий колір для аватарки користувача (ліст. 3.17).

Лістинг 3.17 – Unit-тест для функції генерації кольору аватара користувача

```
describe('getAvatarColor', () => {
 it('returns null if id is falsy', () => {
   expect(getAvatarColor('')).toBeNull();
```

```
    expect(getAvatarColor(null as unknown as string)).toBeNull();
    expect(getAvatarColor(undefined as unknown as string)).toBeNull();
  });
  it('returns a color from predefined set', () => {
    const id = 'user-id-12345678901234567890';
    const result = getAvatarColor(id);
        const colors = ['#FFA726', '#AB47BC', '#26C6DA', '#66BB6A',
'#FF7043'];
    expect(colors).toContain(result);
  });
  it('returns consistent color for the same id', () => {
    const id = 'user-id-12345678901234567890';
    const color1 = getAvatarColor(id);
    const color2 = getAvatarColor(id);
    expect(color1).toBe(color2);
  });

  it('handles short id strings without error', () => {
    const id = 'short-id';
    const result = getAvatarColor(id);
    expect(result).toBeUndefined();
  });
});
```

кінець лістингу 3.17

3.3 Розробка бази даних

MongoDB була обрана через її гнучкість і швидкість роботи з даними, що особливо важливо для динамічних застосунків із часто змінюваними структурами. На рисунку 3.5 можна побачити, як організовані таблиці та їх взаємозв’язки, що забезпечує ефективне управління даними та масштабованість системи. Вона дозволяє працювати з даними у форматі документів (BSON), що є природним способом зберігання JSON-подібних структур, які тісно пов’язані з логікою роботи сучасних вебзастосунків.

Було створено 6 таблиць в базі даних, для зберігання інформації. Кожна з цих таблиць призначена для зберігання специфічної інформації, що дозволяє чітко структурувати всі необхідні дані. Крім того, були ретельно налаштовані взаємозв'язки між цими таблицями, що забезпечує логічну інтеграцію даних, уникнення дублювання та підтримання послідовності інформації.

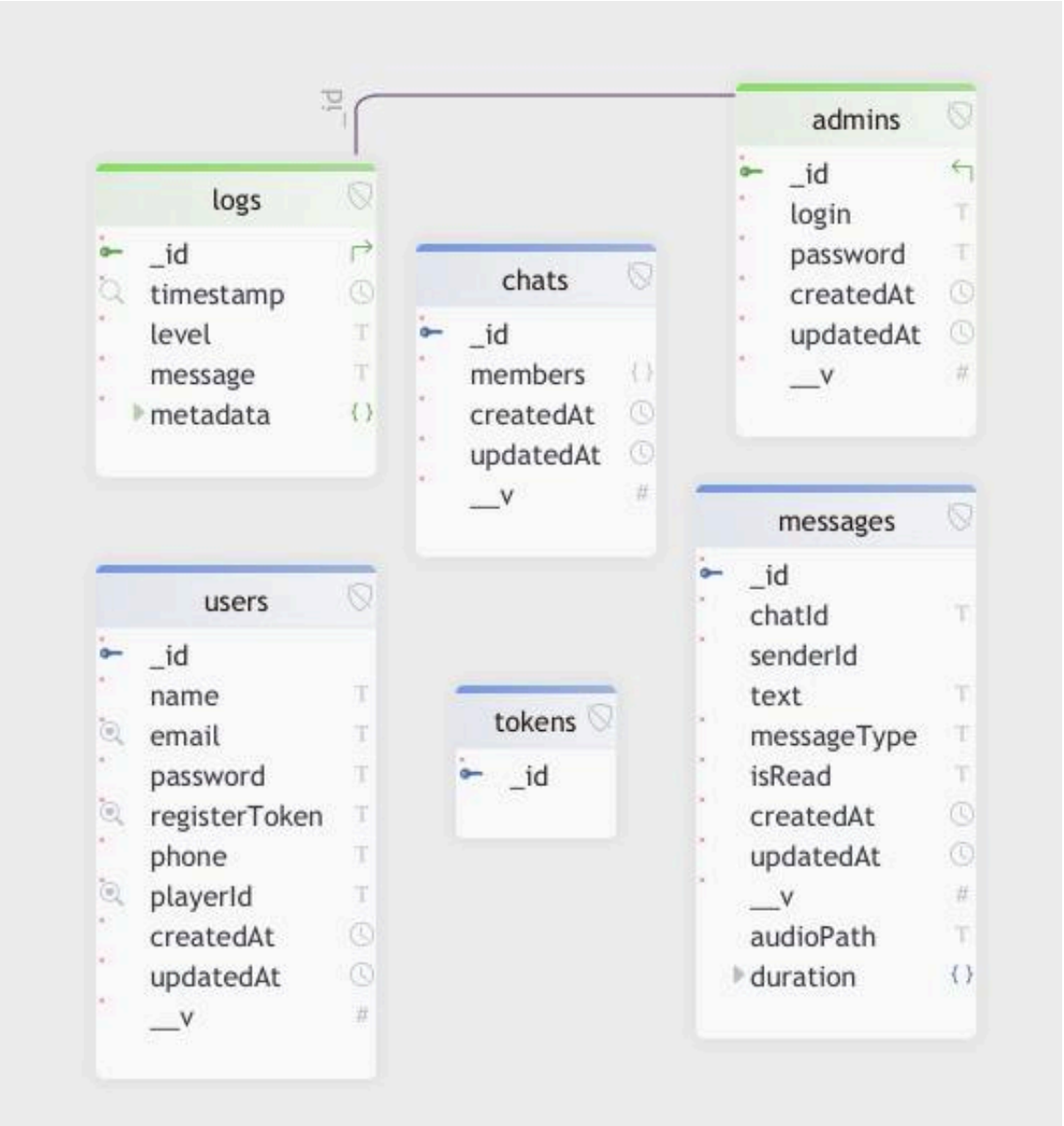

Рисунок 3.5 – Схема таблиць бази даних

На відміну від класичних реляційних баз даних (таких як MySQL або PostgreSQL), MongoDB не потребує попереднього визначення схеми таблиць (у MongoDB вони називаються колекціями). Це означає, що документи в одній колекції можуть мати різну структуру, кількість полів та вкладення. Така гнучкість дозволяє швидко розпочати розробку без необхідності створювати складні SQL-структури чи виконувати міграції при зміні моделі. Проте для зручності та стандартизації структури даних у цьому проєкті використовувався клієнт Mongoose – бібліотека для Node.js, яка забезпечує можливість оголосити схеми та моделі поверх MongoDB.

Mongoose дозволяє чітко описати структуру документів (типи полів, обов’язковість, унікальність тощо), а також надає зручні методи для взаємодії з колекціями, включаючи CRUD-операції, валідацію, middleware та зв’язки між моделями. Завдяки використанню Mongoose структура бази даних була логічно спроєктована, і хоча MongoDB як така не вимагає фіксованої схеми, схемність реалізується на рівні застосунку через оголошення моделей. Це дозволяє зберігати контроль над даними, уникати помилок і полегшувати масштабування проєкту. У межах цього проєкту було реалізовано 6 основних моделей (умовно – «таблиць»), які відповідають ключовим сутностям системи.

3.4 Захист інформаційно-комп’ютерної системи

Основний захист розробленої системи реалізований на стороні сервера шляхом використання JWT (JSON Web Token) для аутентифікації користувачів. Після успішного входу користувача, сервер генерує токен, який містить зашифровану інформацію про користувача та обмежений термін дії. Цей токен передається клієнту, який надалі додає його до кожного запиту до захищених ресурсів через заголовок авторизації. Сервер при кожному запиті перевіряє дійсність токена, його підпис, термін дії та, за потреби, роль користувача. Якщо токен прострочено або підроблено, доступ до ресурсів блокується. Такий підхід дозволяє забезпечити надійну, масштабовану та безпечну модель авторизації без необхідності зберегти сесію на сервері.

Шифрування паролів організовано за допомогою бібліотеки bcrypt. Це дозволяє зберігати паролі в зашифрованому вигляді, не зберігаючи їх у відкритому тексті в базі даних. Bcrypt – це один із найбільш безпечних алгоритмів для хешування паролів, який використовує соління та багаторазове хешування, щоб ускладнити злом пароля навіть при використанні потужних атак. Кожен пароль перед хешуванням проходить через процес соління. Це додає випадкові дані до пароля перед його хешуванням, що запобігає використанню попередньо обчислених таблиць хешів (rainbow tables).

Бібліотека bcrypt автоматично створює унікальний хеш для кожного пароля, що ускладнює атаки на базу даних. Після того, як пароль хешується і зберігається в базі даних, кожен наступний раз, коли користувач вводить свій пароль, система порівнює введений пароль з хешем у базі даних.

Важливою частиною авторизації, є використання спеціально згенерованого власного токена. Сам токен генерується через попередньо створений веб-додаток адмін-панель. Як це відбувається:

1) адмін заходить в адмін-панель;

2) переходить на вкладку де можна згенерувати потрібний токен авторизації;

3) натискає на кнопку «Згенерувати новий токен»;

4) після натискання на кнопку, надсилається запит на сервер, де генерується токен.

Він генерується за наступним алгоритмом, який зображено на рисунку 3.6.

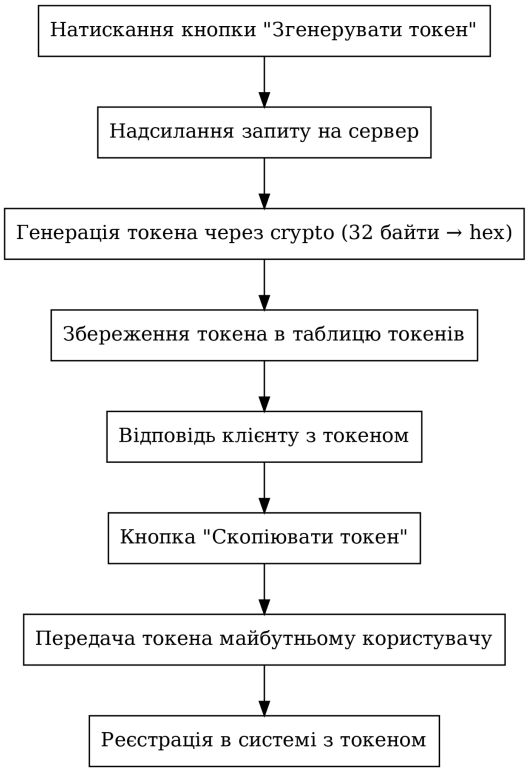

Рисунок 3.6 – Блок-схема алгоритму реєстрації в системі

Перше що може прийти в голову після ознайомлення із даним алгоритмом, це те що сам процес реєстрації є надто складним та комплексним, проте такий підхід гарантує набагато вищий рівень безпеки системи, оскільки в ній будуть присутні лише користувачі, існування яких було санкціоновано певним адміністратором.

Також було додано фаєрвол для захисту сервера від надмірної кількості запитів за певний проміжок часу. Це було зроблено для того щоб зловмисники не могли кинути гігантську кількість запитів за короткий проміжок часу, чим зламати стабільну роботу системи занадто її перенавантаживши. Для реалізації цього функціоналу, я додав до проекту пакет express-rate-limit, та використав його метод rateLimit (ліст. 3.18). Все що треба, це додати цей код в index.js. Тепер система є захищеною від надмірного навантаження.

Лістинг 3.18 – Захист від DDOS-атак на сервер

```
const limiter = rateLimit({
 windowMs: 1 * 60 * 1000, // 1 minute
 max: 100,
 message: "Too many requests from this IP, please try again later.",
});
```

кінець лістингу 3.18

3.4 Розробка інсталяційного пакета

У межах реалізації мобільного застосунку було підготовлено інсталяційний пакет для Android-платформи, що дозволяє розповсюджувати застосунок у вигляді APK-файлу (Android Package). Розробка інсталяційного пакета здійснювалась з використанням стандартних інструментів Android-білду, які вбудовані в екосистему React Native, зокрема – Gradle.

Після компіляції застосунку за допомогою команди npx react-native run-android або ./gradlew assembleRelease генерується релізна версія застосунку – apk файл. Цей файл можна інсталювати безпосередньо на Android-пристрій або завантажити у Google Play (після підписування ключем).

Під час встановлення APK автоматично створюється ярлик застосунку на головному екрані Android, що містить: назву застосунку; значок (іконка), вказаний у android/app/src/main/res/mipmap-*; доступ до головного екрану (наприклад, MainActivity). Ярлик створюється Android-системою автоматично згідно з метаданими, вказаними у AndroidManifest.xml. Конфігурація і підписування Для підготовки релізного APK необхідно створити ключ підпису (keystore), який зберігається в безпечному місці та додається у файл gradle.properties. Після проходження процесу авторизації, токен зберігається локально та автоматично додається до заголовків усіх запитів через axios.interceptors. Завдяки цьому користувач не потребує повторного входу при кожному запуску застосунку. Режими збірки «Розробка інсталяційного пакета» передбачає підтримку двох основних режимів:

– debug mode – для внутрішнього тестування, з доступом до React Native Debugger;

– release mode – оптимізований, з мінімальним розміром, відключеним дебагом, підписаний ключем.

Таким чином, створення інсталяційного пакета для Android дозволяє користувачу встановити застосунок як звичайну програму, без необхідності додаткових ручних налаштувань. Після інсталяції застосунок створює необхідні каталоги, зберігає конфігураційні дані, автоматично додає ярлик на головний екран і забезпечує зручну взаємодію з користувачем відповідно до стандартів Android-системи.

ВИСНОВКИ

У даній кваліфікаційній роботі було успішно розроблено прототип системи захищеного обміну повідомленнями у режимі реального часу із використанням персональних токенів автентифікації. Поставлені завдання виконано у повному обсязі: створено серверну та клієнтську частини застосунку, реалізовано обробку повідомлень через WebSocket-з'єднання, забезпечено безпечну автентифікацію користувачів за допомогою JWT токенів та впроваджено базові механізми шифрування даних.

У ході роботи над проєктом було успішно виконано всі поставлені завдання, що дозволило створити повноцінний програмний продукт.

Проведено глибокий аналіз ринку сучасних real-time месенджерів. У результаті були виділені їхні ключові архітектурні рішення та ефективні механізми безпеки, що стало науково-технічною основою для розробки власного додатка.

Сформульовано вичерпний перелік вимог до мобільного чат-додатка. Визначено функціональні (обмін текстовими й голосовими повідомленнями, статуси доставки) та нефункціональні (продуктивність, відмовостійкість, безпека) аспекти. Затверджено вимоги до UI/UX, локалізації українською мовою та підтримки платформ Android/iOS.

Спроєктовано стійку архітектуру системи за моделлю клієнт-сервер. Було створено детальну проектну документацію, що включає UML-діаграми (прецедентів, класів, компонентів) та ER-модель документної бази даних з відповідними Mongoose-схемами.

Обґрунтовано та підібрано оптимальний технологічний стек. Для кросплатформенної розробки було обрано React Native, для серверної логіки – Node.js та Express. Для зберігання даних використано гнучку СУБД MongoDB з Mongoose, а для комунікації в реальному часі – бібліотеку Socket.IO. Автентифікацію реалізовано на основі JWT.

Реалізовано клієнтську та серверну частини додатка. На клієнті розроблено React Native-компоненти, налаштовано навігацію та управління станом. На сервері створено REST API та WebSocket-сервер, інтегровано механізми автентифікації та обміну повідомленнями в реальному часі.

Розроблено та налаштовано базу даних у MongoDB. Було описано Mongoose-схеми для ключових сутностей (користувач, токен, повідомлення, чат), оптимізовано запити за допомогою індексів та забезпечено масштабовану структуру даних.

Проведено комплексне тестування та налагодження системи. Виконано юніт-тестування серверної логіки (Jest), компонентів React Native, а також інтеграційні перевірки REST API та WebSocket-з’єднань. Регресійне та кросплатформове тестування підтвердили стабільність і надійність додатка.

Забезпечено високий рівень безпеки інформаційної системи. Впроваджено шифрування паролів (bcrypt), валідацію JWT, захист від поширених веб-вразливостей (SQL/NoSQL-ін’єкції, XSS, CSRF). Уся комунікація відбувається через захищений протокол HTTPS, налаштовано систему логування та моніторингу для своєчасного виявлення загроз.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OhNxuB. «Я ненавиджу соцмережі. Вони вбили в мені людину». Що айтішників дратує в Facebook, LinkedIn, Instagram, Telegram: дослідження. URL: https://dev.ua/news/sotsmerezhi-1655060130 (дата звернення: 24.02.2025).

2. Учасники проектів Вікімедіа. Telegram – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Telegram (дата звернення: 26.02.2025).

3. Учасники проектів Вікімедіа. Signal – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Signal (дата звернення: 26.02.2025).

4. Учасники проектів Вікімедіа. Facebook Messenger – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Facebook_Messenger (дата звернення: 26.02.2025).

5. Учасники проектів Вікімедіа. WhatsApp – Вікіпедія. URL: https://uk.wikipedia.org/wiki/WhatsApp (дата звернення: 26.02.2025).

6. Учасники проектів Вікімедіа. Viber – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Viber (дата звернення: 27.02.2025).

7. Учасники проектів Вікімедіа. Discord – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Discord (дата звернення: 27.02.2025).

8. Учасники проектів Вікімедіа. WeChat – Вікіпедія. URL: https://uk.wikipedia.org/wiki/WeChat (дата звернення: 27.02.2025).

9. Top 8 Mobile App Development Frameworks in 2023 – Code Brew Labs. Code Brew Labs. URL: https://www.code-brew.com/top-8-mobile-app-development-frameworks-in-2023 (date of access: 28.02.2025).

10. Learning React Native. O'Reilly Online Learning. URL: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html (date of access: 25.03.2025).

11. Introduction to JSON Web Tokens. URL: https://jwt.io/introduction (date of access: 26.03.2025).

12. Учасники проектів Вікімедіа. Bcrypt – Вікіпедія. URL: https://uk.wikipedia.org/wiki/Bcrypt (дата звернення: 22.01.2025).

13. Використання Axios проти Fetch API у JavaScript – HTML+. HTML+. URL: https://html-plus.in.ua/using-axios-vs-fetch-in-javascript/ (дата звернення: 06.04.2025).

14. TanStack Query React Docs. TanStack. High Quality Open-Source Software for Web Developers. URL: https://tanstack.com/query/latest/docs/framework/react/overview (date of access: 08.04.2025).

15. Учасники проектів Вікімедіа. React – Вікіпедія. URL: https://uk.wikipedia.org/wiki/React (дата звернення: 12.04.2025).

ДОДАТКИ

Додаток А

Контроллер для токена авторизації

```
const crypto = require('crypto');
const logger = require('../utils/logger');
const tokenModel = require('../models/tokenModel');
const adminModel = require('../models/adminModel');
const { default: mongoose } = require('mongoose');

const createToken = async (req, res) => {
 const { adminId } = req.params;

 if (!mongoose.Types.ObjectId.isValid(adminId)) {
      return  res.status(400).json({  message:  'Invalid  Admin  ID
format' });
 }

 if (!adminId) {
      return  res.status(400).json({  message:  'Admin  ID  must  be
provided' });
 }

 try {
   const adminExists = await adminModel.findById(adminId);
   if (!adminExists) {
     return res.status(404).json({ message: 'Admin not found' });
   }

   const newToken = crypto.randomBytes(32).toString('hex');

     const  tokenDoc  =  new  tokenModel({  token:  newToken,  isUsed:
false, adminId });
   await tokenDoc.save();

    logger.info(`Admin  with  id  =  ${adminId}  created  a  new  token
${newToken}`);
```

```
    const tokens = (await tokenModel.find({ adminId })) || [];

    res.status(200).json(tokens);
  } catch (err) {
    console.error(err);
    res.status(500).json({ error: 'Internal Server Error' });
  }
};
```

Додаток Б

Контроллер для реєстрації

```
const registerUser = async (req, res) => {
 try {
    const { name, registerToken, email, password, phone, tag } =
req.body;

   let user = await userModel.findOne({ email });

   if (user) {
     return res
       .status(400)
       .json({ message: "User with given email already exists" });
   }

   if (!name || !email || !password || !registerToken)
       return res.status(400).json({ message: "All fields must be
provided" });

   if (!validator.isEmail(email)) {
     return res.status(400).json({ message: "Invalid email" });
   }

   if (!validator.isStrongPassword(password))
       return res.status(400).json({ message: "Password must be a
strong" });

   const tokenDoc = await tokenModel.findOne({
     token: registerToken,
     isUsed: false,
   });

   if (!tokenDoc)
     return res
       .status(400)
          .json({ message: "Invalid or already used registration
token" });

    user = new userModel({ name, email, password, registerToken,
phone, tag });

   const salt = await bcrypt.genSalt(10);
   user.password = await bcrypt.hash(password, salt);

   await user.save();

   tokenDoc.isUsed = true;
   await tokenDoc.save();

   const token = createToken(user._id);

   logger.info(
```

```
            `User with email ${email} registered. Used token
${registerToken}`
    );

          res.status(200).json({   _id:   user._id,   name,   email,
registerToken, token });
  } catch (err) {
    console.error(err);
    res.status(500).json({ message: "Server Error", error: err });
  }
};
```