Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Інститут електроенергетики
(інститут)

Факультет інформаційних технологій
(факультет)

Кафедра Програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
кваліфікаційної роботи ступеня

бакалавра
(назва освітньо-кваліфікаційного рівня)

студента *Лохвицького Нікіти Сергійовича*
(ПІБ)

академічної групи *121-19-2*
(шифр)

напряму підготовки *121 Інженерія програмного забезпечення*
(код і назва напряму підготовки)

на тему: *Розробка програмного забезпечення месенджеру з використанням React Native*

Керівники	Прізвище, ініціали	Оцінка за шкалою		Підпис
		рейтинговою	інституційною	
кваліфікаційної роботи	*Проф. Мещеряков Л.І.*			
розділів:				
спеціальний	*Проф. Бердник М.Г.*			
економічний	*доц. Касьяненко Л.В.*			
Рецензент	*доц. Шедловський І.А*			
Нормоконтролер	*ст. викл. Мартиненко А.А.*			

Дніпро
2023

Міністерство освіти і науки України
НТУ «Дніпровська політехніка»

ЗАТВЕРДЖЕНО:
завідувач кафедри
програмного забезпечення комп'ютерних систем
(повна назва)

_____ М.О. Алексєєв
(підпис) (прізвище, ініціали)

« » _____ 20 23 року

ЗАВДАННЯ
на дипломний проект

бакалавра
(назва освітньо-кваліфікаційного рівня)

студенту *121-19-2* *Лохвицькому Нікіті Сергійовичу*
(група) (прізвище та ініціали)

Тема дипломного проекту *Розробка програмного забезпечення месенджеру з використанням React Native*

затверджена наказом ректора НТУ «ДП» від *16 . 05 .2023 р.* № *350 -с*

Розділ	Зміст виконання	Термін виконання
Спеціальний	***На основі матеріалів виробничої практики та інших науково-технічних джерел провести аналіз стану рішення проблеми та постановку задачі.*** ***Обґрунтувати вибір та здійснити реалізацію методів вирішення проблеми***	***13.05.2023 р.***
Економічний	***Провести розрахунок трудомісткості розробки програмного забезпечення, витрат на створення ПО й тривалості його розробки***	***27.05.2023 р.***

Завдання видав _____ *Проф. Мещеряков Л.І.*
(підпис) (посада, прізвище, ініціали)

Завдання прийняв до виконання _____ *Лохвицький Н.С.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *14 .01 .2023 р.*

Термін подання дипломного проекту до ДЕК ___

РЕФЕРАТ

Пояснювальна записка: 88 с., 16 рис., 3 дод., 23 джерел.

Об'єкт розробки: програмне забезпечення месенджеру з використанням React Native.

Мета кваліфікаційної роботи: розробка месенджеру з використанням React Native для забезпечення зручного та безпечного спілкування користувачів з інтуїтивно-зрозумілим та сучасним привабливим інтерфейсом.

У вступній частині кваліфікаційної роботи проведено аналіз та розглянуто сучасний стан проблеми, визначено конкретну мету дослідження та галузь її застосування, наведено обґрунтування актуальності обраної теми та деталізовано постановку задачі.

У першому розділі кваліфікаційної роботи було проведено детальний аналіз предметної галузі, розглянуто актуальність поставленої задачі та цілей розробки, сформульовано конкретну постановку завдання, а також визначено вимоги до програмної реалізації, використовуваних технологій та програмних засобів.

У другому розділі кваліфікаційної роботи було проаналізовано наявні рішення, обрані платформи для розробки, проведено проектування та розробка програми. Також була надана детальна інформація про роботу програми, включаючи опис алгоритму та структури її функціонування. В розділі було розглянуто процес виклику та завантаження програми, визначено вхідні та вихідні дані, а також надано характеристику параметрів технічних засобів.

У економічному розділі кваліфікаційної роботи була визначена трудомісткість розробленої інформаційної системи. Крім того, було проведено розрахунок вартості роботи зі створення програми та оцінено необхідний час для її реалізації.

Практичне значення полягає у розробці мобільного застосунку з приємним інтерфейсом, який дозволяє користувачам безпечно, швидко та ефективно листуватися.

Актуальність даного програмного продукту визначається значним попитом на здійснення дистанційних покупок у мережі інтернет, що значно спрощує процес отримання товару клієнтом та економить фінансові витрати на відкриття стаціонарного магазину.

Список ключових слів: СМАРТФОН, ЧАТ, ЛИСТУВАННЯ, REACT NATIVE, NODEJS, SQLITE.

ABSTRACT

Explanatory note: 88 pages, 16 figures, 3 appendices, 23 references.

Development object: messaging software using React Native.

Objective of the qualification work: to develop a messaging app using React Native to facilitate convenient and secure communication between users with an intuitive, modern, and attractive interface.

The introductory part of the qualification work includes an analysis of the current state of the problem, defines the specific research objective and its application area, justifies the relevance of the chosen topic, and details the task formulation.

Chapter 1 of the qualification work conducts a detailed analysis of the subject area, discusses the relevance of the task and development goals, formulates the specific task, and identifies requirements for software implementation, technologies used, and software tools.

Chapter 2 of the qualification work analyzes existing solutions, selects development platforms, conducts design and program development. It also provides detailed information about the program, including algorithm description and functional structure. The chapter discusses the process of program invocation and loading, defines input and output data, and provides characteristics of technical device parameters.

The economic section of the qualification work determines the labor intensity of the developed information system. Additionally, it calculates the cost of program development and estimates the required time for implementation.

The practical significance lies in the development of a mobile application with a pleasant interface that allows users to communicate safely, quickly, and efficiently.

The relevance of this software product is determined by the high demand for online shopping, which simplifies the process of obtaining goods for customers and saves financial costs of opening a physical store.

List of keywords: SMARTPHONE, CHAT, MESSAGING, REACT NATIVE, NODEJS, SQLITE.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ

UI - User Interface (інтерфейс користувача)

JSON - JavaScript Object Notation (простий формат обміну даними)

CRUD - Create, Read, Update, Delete (операції створення, читання, оновлення, видалення)

HTTP - Hypertext Transfer Protocol (протокол передачі гіпертексту)

DB - Database (база даних)

WebSocket - WebSocket Protocol (протокол двостороннього спілкування)

ВСТУП

У сучасному світі зростає популярність месенджерів як зручного та швидкого засобу комунікації. Завдяки мобільним пристроям із доступом до Інтернету, люди можуть спілкуватись з будь-якого місця та в будь-який час. Це сприяє покращенню особистої та професійної комунікації, забезпечує зв'язок між людьми незалежно від географічних відстаней і створює нові можливості для співпраці та обміну інформацією.

В рамках цієї дипломної роботи пропонується розробка програмного забезпечення месенджеру для мобільного пристрою з використанням React Native. React Native - це потужна технологія, яка дозволяє створювати кросплатформові мобільні додатки з використанням JavaScript та перевагами React, такими як ефективність, швидкість розробки та кросплатформеність.

Важливою складовою розробки месенджеру є серверна частина, яка забезпечує обробку, збереження та передачу повідомлень між користувачами. Для цього використовується технологія node.js, що дозволяє створювати швидкі та масштабовані мережеві додатки з використанням JavaScript. Ця комбінація React Native та node.js надає нам потужні інструменти для створення мобільного месенджеру, який буде забезпечувати зручну та безпечну комунікацію між користувачами.

Метою цієї дипломної роботи є розробка месенджеру, який відповідатиме сучасним вимогам до функціональності, безпеки та зручного інтерфейсу користувача. Основні завдання роботи включають аналіз вимог до месенджеру, проектування архітектури системи, реалізацію клієнтської та серверної частини додатку, а також проведення тестування для забезпечення якості продукту.

Дана дипломна робота має важливе значення, оскільки месенджери стають необхідною складовою сучасного суспільства і мають великий потенціал для покращення комунікації та співпраці між людьми. Розробка мобільного месенджеру з використанням React Native та node.js сприятиме подальшому розвитку цієї галузі і надасть користувачам зручний інструмент для спілкування та обміну інформацією.

У решті роботи буде проведений аналіз вимог, розглянуто процес розробки месенджеру, описано реалізацію функціональних компонентів, а також проведено тестування та оцінку якості розробленого продукту.

Таким чином, розробка мобільного месенджеру з використанням React Native та node.js є актуальною та перспективною задачею, яка вимагає високої кваліфікації та знань у сфері програмування та розробки мобільних додатків.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Загальні відомості з предметної області

Месенджери є важливою складовою комунікаційної інфраструктури у сучасному світі. Вони дозволяють користувачам обмінюватись повідомленнями, файлами та мультимедійним вмістом, надаючи широкий спектр можливостей для особистого та професійного спілкування.

Месенджери стали популярними завдяки своїй зручності та доступності. Завдяки ним, користувачі можуть спілкуватись в реальному часі, незалежно від географічних відстаней, з використанням текстових повідомлень, голосових або відеодзвінків. Вони також надають можливість створювати групові чати, обмінюватись фотографіями та відео, використовувати емодзі та стікери для виразності спілкування.

Месенджери широко використовуються як в особистих, так і в комерційних цілях. Вони дозволяють людям підтримувати зв'язок з друзями, родиною та колегами, а також спілкуватись у групах за спільними інтересами. У бізнесі месенджери використовуються для комунікації з клієнтами, проведення переговорів, обміну документами та організації робочих груп.

У даному дипломному проекті розробляється месенджер, який надає можливість дослідити та впровадити практичні аспекти розробки месенджер додатків з використанням популярних технологій.

Таким чином, дослідження та розробка месенджеру сприяють збагаченню знань у сфері розробки комунікаційних додатків та дозволяють освоїти навички створення мобільних додатків з використанням сучасних інструментів та технологій.

1.2. Призначення розробки та область застосування

Метою розробки даного проекту є створення месенджеру, який надає зручний та ефективний спосіб комунікації між користувачами. Основним завданням є реалізація функціональності текстових повідомлень.

Цей месенджер розробляється з використанням React Native для реалізації мобільного інтерфейсу та Node.js для побудови бекенду з використанням websocket і express. В якості бази даних використовується SQLite, яка забезпечує мінімалістичний та ефективний спосіб збереження даних.

Хоча цей месенджер не є інноваційним у сфері комунікаційних додатків, він надає можливість користувачам обмінюватись повідомленнями у зручний спосіб. Він може бути використаний як у особистих цілях для підтримки зв'язку з друзями та родиною, так і в комерційних цілях для спілкування з колегами та клієнтами.

1.3. Підстава для розробки

Підставами для розробки та виконання кваліфікаційної роботи є:

- освітня програма 121 «Інженерія програмного забезпечення»;
- навчальний план та графік навчального процесу;
- наказ ректора Національного технічного університету «Дніпровська політехніка» № 350-с від 16.05.2023 р;
- завдання на кваліфікаційну роботу на тему «Розробка програмного забезпечення месенджеру з використанням React Native».

1.4. Постановка завдання

Метою даного проекту є розробка месенджера, який надає зручну та надійну комунікаційну платформу для користувачів. Завдання передбачає створення React Native UI частини для мобільного інтерфейсу, Node.js, websocket та express

для бекенду, а також використання SQLite бази даних як мінімалістичного та ефективного рішення для збереження даних.

Основні характеристики завдання:

- розробка зручного та інтуїтивно зрозумілого інтерфейсу, який дозволяє користувачам легко комунікувати шляхом текстових повідомлень;
- розробка системи аутентифікації та безпеки, що забезпечує захист приватності користувачів та їх даних;
- реалізація можливості створення групових чатів для організації комунікації в більших спільнотах;
- забезпечення оптимальної продуктивності та швидкості роботи за допомогою ефективного використання ресурсів та оптимізації коду;

1.4.1. Можливості програмного забезпечення

Можливості програмного забезпечення включають до себе:

- реєстрація: можливість зареєструвати власний аккаунт, за допомогою якого можна зберігати та керувати особистою інформацією, такою як список чатів;
- налаштування: можливість змінювати реєстраційні дані користувача
- список чатів: список, що дозволяє зручно та ефективно керувати чатами, в яких перебуває користувач, і оновлювати інформацію про них у реальному часі;
- список повідомлень: список, що дозволяє зручно переглядати повідомлення від інших користувачів та оновлювати інформацію про повідомлення у реальному часі;
- Інформація про чат: вікно, де можна переглянути інформацію про чат та актуальну інформацію про учасників чату.

Ці функціональні особливості є ключовими для забезпечення зручності та ефективності роботи інтернет-магазину настільних ігор головоломок.

1.4.2. Опис інтерфейсу користувача

При першому запуску клієнт потрапить на екран аутентифікації. З екрана аутентифікації користувач може перейти на екран реєстрації. Після успішної реєстрації або аутентифікації, користувач буде перенаправлений на екран списку чатів.

В месенджері користувач має можливість виконувати наступні дії. Він може натиснути на кнопку, розташовану угорі ліворуч, щоб увійти в налаштування та змінити свої особисті дані. Також, натиснувши на кнопку, розташовану угорі праворуч, користувач може перейти до вікна створення нового чату.

Якщо у користувача вже є чати, він може натиснути на відповідний чат у списку, щоб перейти до списку повідомлень цього чату. У вікні з повідомленнями користувач може вводити нові повідомлення у полі внизу та надсилати їх. Він також може повернутися до списку чатів, натиснувши кнопку угорі ліворуч. Крім того, він може перейти до вікна з інформацією про чат, натиснувши кнопку угорі праворуч.

У вікні з інформацією про чат користувач може додати нового учасника до чату або вийти з чату. Якщо користувач бажає вийти зі свого облікового запису, він може натиснути кнопку угорі праворуч у налаштуваннях та здійснити вихід з аккаунту.

1.5. Вимоги до програми або програмного виробу

1.5.1. Вимоги до функціональних характеристик

Основними функціональними характеристиками месенджера є:

- реєстрація та аутентифікація користувачів: користувачі повинні мати можливість створити обліковий запис з унікальним ім'ям та паролем. Система повинна перевіряти правильність введених даних та

забезпечувати безпеку паролів. користувачі повинні мати можливість увійти в систему за допомогою свого облікового запису.

- організація комунікації: користувачі повинні мати змогу створювати особисті чати з іншими користувачами та обмінюватися текстовими повідомленнями. Повідомлення повинні відображатися у реальному часі.
- групові чати: користувачі повинні мати можливість створювати групові чати для спілкування з більшою кількістю учасників. Месенджер повинен підтримувати створення та управління каналами для організації комунікації в спільнотах або організаціях.
- безпека та приватність: система повинна забезпечувати захист конфіденційності повідомлень та персональних даних користувачів.
- налаштування профілю: користувачі повинні мати можливість налаштувати свій профіль, включаючи фотографію та статус.

Ці функціональні характеристики забезпечують зручну та ефективну комунікацію між користувачами, забезпечують безпеку та приватність даних, а також надають можливості для налаштування та персоналізації месенджера.

1.5.2. Вимоги до інформаційної безпеки

Кожен користувач мусить аутентифікуватись під власним логіном та паролем, щоб забезпечити безпеку своїх особистих даних. Система мусить безвідмовно обробляти аутентифікаційні дані та шифрувати їх у вигляді токену. Перед кожною операцією система мусить перевіряти токен на валідність.

1.5.3. Вимоги до складу та параметрів технічних засобів

Вимоги до складу та параметрів технічних засобів для месенджера на платформі React Native включають наступне:

- операційна система: месенджер повинен підтримувати використання на мобільних пристроях з операційною системою Android або iOS.

- пам'ять: мобільний пристрій користувача повинен мати достатньо оперативної пам'яті для плавної роботи месенджера.
- вільне місце на пристрої: користувачеві потрібно мати достатньо вільного місця на пристрої для збереження додатку та даних, пов'язаних з месенджером.

1.5.4. Вимоги до інформаційної та програмної сумісності

Вимоги до інформаційної та програмної сумісності для месенджера включають наступне:

- Інформаційна сумісність: месенджер повинен забезпечувати обмін інформацією між мобільними пристроями на платформах Android та iOS. Дані, такі як повідомлення, користувачі та налаштування, повинні зберігатися та передаватися між клієнтською та серверною частинами месенджера з використанням SQLite бази даних.
- Програмна сумісність: месенджер розроблено з використанням React Native, Node.js, websocket та express, що вимагає наявності відповідних середовищ виконання на серверній та клієнтській сторонах. Для оптимальної роботи месенджера на мобільних пристроях користувачів необхідно мати оновлену версію операційної системи Android або iOS.

РОЗДІЛ 2
ПРОЕКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

2.1. Функціональне призначення програми

Функціональне призначення мого месенджера на React Native полягає у створенні зручного та ефективного засобу комунікації для користувачів. Основні функції месенджера включають:

1. Аутентифікація:
   - Користувачі можуть зареєструватися та увійти в систему, використовуючи електронну пошту або інші облікові дані.
   - Після успішної аутентифікації користувачі отримують доступ до основного функціоналу месенджера.
2. Налаштування користувача:
   - Користувачі можуть переглянути свій профіль та редагувати інформацію, таку як ім'я, фотографія тощо.
   - Можливість змінити налаштування облікового запису, такі як пароль або електронна пошта.
3. Список чатів:
   - Після успішної аутентифікації користувачі потрапляють до списку чатів, де вони можуть бачити всі свої чати.
   - Чати сортуються за різними параметрами, такими як останнє повідомлення або активність учасників.
   - Список чатів оновлюється у реальному часі, коли користувач отримує нові повідомлення.
4. Вікно повідомлень:
   - Користувачі можуть відкривати чати зі списку та переглядати повідомлення, відсортовані за часом та відправником.
   - Повідомлення оновлюються у реальному часі, коли користувач отримує нові повідомлення.

5. Інформація про чат:
    - Користувачі можуть переглядати додаткову інформацію про чат, включаючи список учасників.
    - Вони можуть додавати нових учасників та виходити з чату за потреби.
6. Взаємодія з сервером:
    - Месенджер взаємодіє з сервером на основі технологій HTTP або WebSocket для передачі даних між клієнтом та сервером.
    - За допомогою HTTP-запитів користувач може виконувати дії, такі як аутентифікація, зміна налаштувань тощо.
    - WebSocket забезпечує постійне оновлення даних у реальному часі, таких як нові повідомлення чи зміни у списку чатів.
7. База даних:
    - Сервер месенджера взаємодіє з базою даних SQLite, де зберігаються інформація про користувачів, повідомлення та налаштування.

Ці функції дозволяють користувачам легко аутентифікуватися, взаємодіяти з чатами, отримувати та надсилати повідомлення у реальному часі та керувати своїм профілем та налаштуваннями. Використання реального часу та взаємодія з сервером на основі HTTP та WebSocket дозволяють забезпечити швидку та ефективну роботу месенджера.

2.2. Опис застосованих математичних методів

При розробці месенджера на платформі React Native, використовувалися лише прості математичних методів або використання більш складних методів було автоматичним за допомогою завантажених бібліоток.

2.3. Опис використаної архітектури та шаблонів проектування

2.3.1. Архітектура Unidirectional Data Flow

При створенні клієнтського React Native застосунку для месенджера була використана архітектура, що базується на шаблоні проектування "Односторонній потік даних" (англ. "Unidirectional Data Flow") або UDF.

Цей шаблон проектування організовує структуру застосунку таким чином, щоб дані йшли в одному напрямку, що сприяє простоті управління станом та прогнозованості поведінки. Односторонній потік даних зазвичай використовується разом зі становим управлінням (state management) для забезпечення консистентності та ефективного оновлення інтерфейсу користувача.

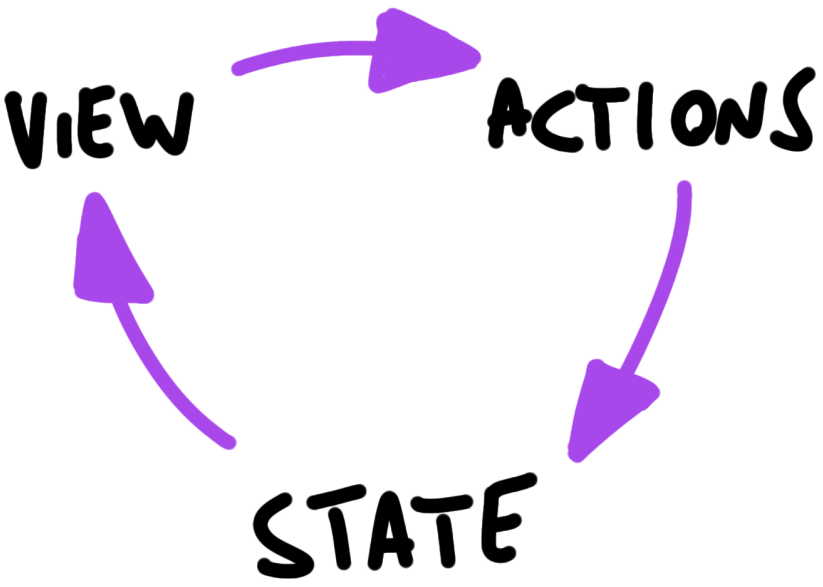

Рис. 2.1. Графічне представлення архітектури UDF

Основні складові архітектури "Одностороннього потоку даних" включають:

- компоненти (Components): компоненти вашого застосунку виконують роль відображення (View) та взаємодіють з користувачем. Вони приймають дані зі стану та відображають їх на екрані. Компоненти можуть відправляти події (actions) для оновлення стану;
- стан (State): стан є центральним елементом вашої архітектури. Це місце, де зберігаються дані вашого застосунку. Стан може бути представлений у вигляді об'єкта, який містить потрібні дані, такі як інформація про користувача, повідомлення чату тощо;
- дії (Actions): дії представляють події, які відбуваються в вашому застосунку, такі як натискання кнопок, введення тексту тощо. Вони ініціюють зміни стану та оновлення інтерфейсу користувача.

Ця архітектура допомагає розділити логіку вашого застосунку на чітко визначені частини та полегшує управління станом та оновлення інтерфейсу користувача. Вона також сприяє тестуванню та розширенню вашого застосунку в майбутньому.

Як і будь-яка архітектура, даний підхід має свої переваги та недоліки. Переваги Unidirectional Data Flow:

- простота розуміння: Unidirectional Data Flow (однобічний потік даних) має просту та прозору структуру, що полегшує розуміння та налагодження коду. Дані рухаються у визначеному напрямку, що сприяє прогнозовуваності та зменшує складність взаємодії між компонентами;
- зручна налагоджуваність: оскільки дані рухаються в одному напрямку, виявлення та виправлення помилок стає простішим. Ви можете відстежувати потік даних та точно знати, які компоненти залежать від змін даних;
- прогнозованість та стабільність: Unidirectional Data Flow дозволяє забезпечити прогнозованість стану додатку. Компоненти завжди знають, з якими даними працюють і як їх модифікувати. Це зменшує ймовірність неочікуваних станів і покращує стабільність програми;

- тестування: однобічний потік даних сприяє легкості тестування. Кожен крок у потоці даних може бути протестований окремо, що полегшує написання юніт-тестів та забезпечує покриття коду тестами.

Мінуси Unidirectional Data Flow:

- збільшений обсяг коду: використання Unidirectional Data Flow може призвести до збільшення обсягу коду в порівнянні з іншими архітектурними підходами. Це може бути особливо помітно в великих проектах, де потрібно багато додаткового коду для управління потоком даних.
- збільшена складність: у великих та складних програмах реалізація Unidirectional Data Flow може вимагати більшої уваги до деталей та обумовлювати потребу у додатковому плануванні та організації коду. Це може стати викликом для менш досвідчених розробників;
- завищена абстракція: у деяких випадках, використання Unidirectional Data Flow може призвести до занадто абстрактного коду, який може бути складним для розуміння та підтримки. Важливо збалансувати рівень абстракції та простоту розуміння.

В рамках застосунку, дана архітектура дозволяє відображати вікна з різним контентом, та за допомогою хуків оновлювати стан вікон у реальному часі.

2.3.2. Клієнт-серверна архітектура

При взаємодії клієнта з даними в React Native застосунку було використана архітектура клієнт-сервер (Client-Server Architecture). В даній архітектурі клієнт (мобільний додаток) взаємодіє з сервером (node.js сервер), що знаходиться на іншому комп'ютері або веб-сервері. Клієнт та сервер комунікують між собою за допомогою протоколу HTTP або WebSocket.

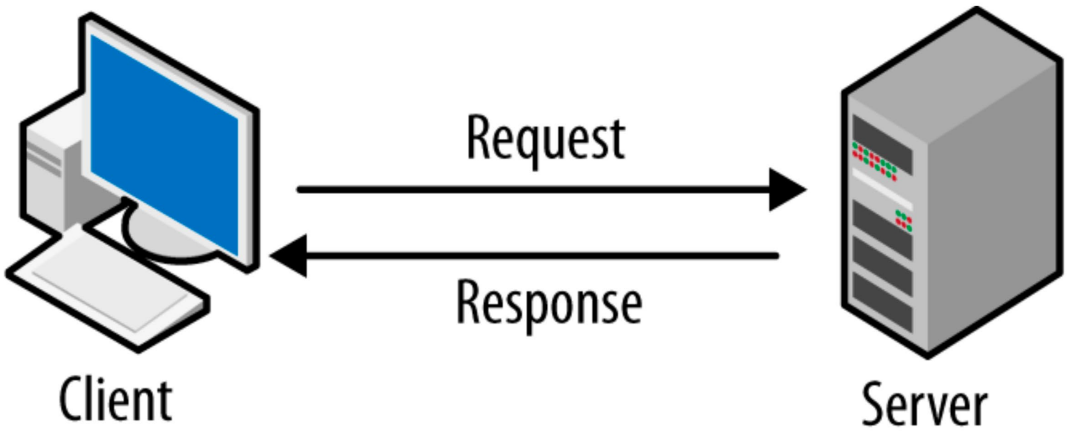

Рис. 2.2. Графічне представлення клієнт-серверної архітектури

Переваги клієнт-серверної архітектури:

- масштабованість: клієнт-серверна архітектура дозволяє масштабувати систему, розміщуючи серверні компоненти на окремих машинах або використовуючи хмарні рішення. Це дозволяє обробляти більшу кількість запитів та забезпечувати більшу кількість користувачів;
- розділення відповідальності: архітектура дозволяє розділити логіку додатку між клієнтом і сервером. Клієнтська частина відповідає за користувацький інтерфейс, взаємодію з користувачем та локальне зберігання даних. Серверна частина відповідає за обробку бізнес-логіки, збереження та доступ до даних;
- більша безпека: за допомогою клієнт-серверної архітектури можна забезпечити більшу безпеку шляхом обмеження доступу до серверних ресурсів. Сервер може встановлювати правила авторизації та аутентифікації, що забезпечує захист конфіденційної інформації;
- покращена керованість: клієнт-серверна архітектура дозволяє керувати додатком більш ефективно. Зміни в бізнес-логіці можуть бути внесені на серверному рівні, що дозволяє оновлювати клієнтські додатки без необхідності оновлення їх на кожному пристрої окремо;

Недоліки клієнт-серверної архітектури:

- залежність від мережі: клієнтські додатки вимагають постійного з'єднання з сервером для отримання даних та оновлень. Це означає, що в разі

відсутності мережі або поганого з'єднання користувачі можуть стикнутись з проблемами доступу до функціональності додатку;

- навантаження на сервер: клієнтські додатки, особливо з великою кількістю користувачів, можуть створювати значне навантаження на сервер. Це може призводити до потреби в масштабуванні серверних ресурсів та забезпечення ефективної обробки запитів;
- синхронізація даних: при клієнт-серверній взаємодії потрібно дбати про синхронізацію даних між клієнтом і сервером. Це може бути складним завданням, особливо при обробці одночасних змін даних на різних пристроях або конфліктах;
- залежність від сервера: при клієнт-серверній архітектурі користувачі залежать від доступності та надійності сервера. Якщо сервер недоступний або стикається з проблемами, це може призвести до обмежень у функціональності додатку.

У месенджері основна взаємодія React Native клієнта та Node.js може бути описана наступним чином:

- аутентифікація та реєстрація: клієнт надсилає запити на сервер для аутентифікації користувача або реєстрації нового користувача. Сервер перевіряє дані, виконує необхідні перевірки та відповідає клієнту залежно від результату;
- отримання та оновлення даних чатів: клієнт отримує список чатів користувача з сервера. При отриманні нових повідомлень сервер використовує WebSocket для надсилання оновленої інформації клієнту у реальному часі;
- відправлення повідомлень: клієнт надсилає повідомлення до сервера, який пересилає його відповідному чату або користувачу. Сервер зберігає повідомлення та розсилає його всім зацікавленим сторонам за допомогою WebSocket;

- отримання та оновлення даних профілю користувача: клієнт отримує дані профілю користувача з сервера та може змінювати ці дані. Сервер зберігає зміни та оновлює їх у відповідних записах.

Ця клієнт-сервер архітектура дозволяє ефективно передавати дані між клієнтом та сервером, забезпечує швидку відповідь та оновлення у реальному часі за допомогою WebSocket. Вона також дозволяє легко масштабувати систему, додавати нові функціональності та забезпечує зручне управління взаємодією з базою даних на SQLite.

2.4. Опис використаних технологій та мов програмування

Для розробки React Native месенджера, який взаємодіє з сервером на Node.js та базою даних SQLite, були використані різні технології та мови програмування. Нижче наведений огляд цих технологій та їх обґрунтування на підставі проведених аналізів.

- React Native: є основною технологією для розробки мобільних додатків для платформ Android і iOS. Використання React Native дозволяє писати код один раз і запускати його на обох платформах, що прискорює процес розробки та забезпечує більшу переносимість. Був обраний через доступність та росповсюдженність, а також він простий у використанні. React Native працює зі станами, коли змінюються стани – змінюється і наповнення сторінки, яку бачить користувач. Такий динамічний рендерінг зображення дуже вдало підходить для такого програмного забезпечення, як месенджер, де важливо постійно оновлювати данні та оновлений результат одразу відображати на сторінці;
- JavaScript: є мовою програмування, яка використовується для розробки React Native додатків. Вона є широко відомою та має велику спільноту розробників, що забезпечує багато ресурсів та підтримку для розробки додатків;

- Node.js: є платформою для розробки серверних додатків з використанням JavaScript. У месенджері, Node.js використовується для створення серверної частини, яка взаємодіє з React Native додатком через HTTP та Socket.io;
- Express.js: є легким і гнучким веб-фреймворком для Node.js, який використовується для створення серверного застосунку. Express.js дозволяє легко визначати роути та обробляти HTTP запити від клієнта;
- SQLite: є легким і вбудованим реляційним СУБД, який використовується для зберігання даних на сервері. Вибір SQLite обґрунтований його простотою використання та портативністю, що підходить для вашої додатку месенджера;
- Socket.io: є бібліотекою JavaScript, яка забезпечує можливість реалізації двостороннього зв'язку між клієнтом і сервером за допомогою веб-сокетів. Використання Socket.io дозволяє оновлювати дані у реальному часі на клієнті при отриманні повідомлень з сервера. Socket.io був обраний, тому що він надає зручні інструменти для взаємодії з протоколом Websocket;
- React Navigation: є навігаційною бібліотекою для React Native, яка дозволяє легко керувати навігацією між екранами вашого додатку. Використання React Navigation допомагає створити зручний інтерфейс для користувача та полегшує навігацію по різним екранам додатку.

Вибір цих технологій обґрунтовується їхньою популярністю, підтримкою спільноти розробників, легкістю використання та здатністю задовольняти потреби вашого проекту месенджера.

2.5. Опис структури програми та алгоритмів її функціонування

React Native клієнт, що розроблений з використанням архітектури Unidirectional Data Flow, має особливості, які дозволяють ефективно керувати станом додатку та динамічно оновлювати його інтерфейс у реальному часі.

У цій архітектурі програма складається з різних вікон, кожне з яких включає зовнішній вигляд вікна, а також стани, які пов'язані з цим вікном. Стани визначаються як змінні, що відображають різні аспекти стану вікна, такі як дані, налаштування чи статус.

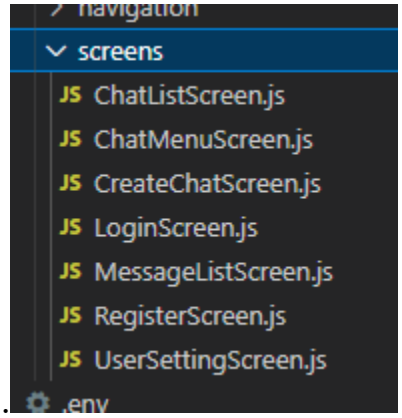

Рис. 2.3. Папка з вікнами програми

React Native відстежує зміни станів і автоматично оновлює відповідне вікно, коли стан змінюється. Це означає, що при внесенні змін у стан програми, активне вікно перефарбовується, а користувач спостерігає оновлені дані чи зміни в інтерфейсі. Такий підхід дозволяє розробникам створювати різноманітні вікна, які мають можливість динамічного оновлення в реальному часі.

Ця архітектура сприяє чіткому розділенню відповідальностей і покращує керованість станом програми. Зміни стану відбуваються через виклик функцій, які оновлюють стани вікон, забезпечуючи однонаправлене поширення даних у програмі. Це дозволяє зручно керувати станами і спрощує налагодження та розробку.

Завдяки архітектурі Unidirectional Data Flow React Native клієнт зможе ефективно працювати зі станами вікон і забезпечити динамічне оновлення інтерфейсу в реальному часі, що важливо для месенджера, де користувачі отримують повідомлення та оновлення у режимі реального часу.

Для забезпечення функціонування системи вікон у нашому React Native додатку необхідний механізм для переміщення між вікнами. Цю функціональність надає бібліотека React Navigation Native. Використовуючи цю бібліотеку, ми можемо легко переходити від одного вікна до іншого, просто

вказавши його назву (name). Головна умова для переходу полягає в тому, щоб вікно, до якого ми хочемо перейти, було в тому самому Stack.Navigator, що й вікно, з якого ми переходимо.

```
return (
    <NavigationContainer>
    {isAuth ? (
        <Stack.Navigator screenOptions={{ headerTitleAlign: 'center' }}>
            {/* <Stack.Screen name='Chat Menu' component={ChatMenuScreen} /> */}
            <Stack.Screen name='Chat List' component={ChatListScreen}/>
            <Stack.Screen name='Message List' component={MessageListScreen}/>
            <Stack.Screen name='User Setting' component={UserSettingScreen}/>
            <Stack.Screen name='Create Chat' component={CreateChatScreen}/>
            <Stack.Screen name='Chat Menu' component={ChatMenuScreen}/>
        </Stack.Navigator>
    ) : (
        <Stack.Navigator>
            <Stack.Screen name='Login' component={LoginScreen} options={{ headerShown: false }}/>
            <Stack.Screen name='Register' component={RegisterScreen} options={{ headerShown: false }}/>
        </Stack.Navigator>
    )}
    </NavigationContainer>
)
```

Рис. 2.4. Головна навігація програми navigation

На рисунку 2.4 представлено головну навігацію програми. Зауважимо, що вікна Login та Register розташовані окремо від усіх інших вікон. Залежно від значення стану isAuth, користувач може перебувати в навігації для авторизації або в навігації з контентом. Це зроблено для того, щоб користувач не мав можливості переходити від вікон авторизації до вікон, які відображають контент, за допомогою звичайної навігації. Таке рішення забезпечує більший рівень безпеки, оскільки без авторизації користувач не матиме доступу до контенту.

Також у ході виконання дипломної роботи було прийнято рішення замінити стан isAuth на контекст isAuth. Контекст - це механізм React Native, який дозволяє встановлювати область бачення для станів. Для цього ми просто огорнули відповідну область в спеціальний тег, що дозволяє нам зберігати та змінювати стан isAuth у вікнах-родичах. Такий підхід дозволяє полегшити подальшу роботу зі станом аутентифікації, оскільки цей стан може змінюватись у будь-якому вікні програми.

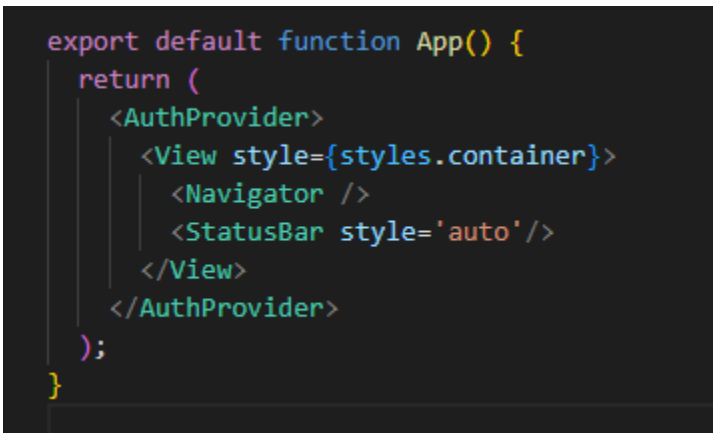

Рис. 2.5. Головна навігація програми navigation

На початку виконання програми значення isAuth встановлено на false, тому користувач потрапляє до вікон "Login" та "Registration", де він може пройти авторизацію. Якщо авторизація пройшла успішно, значення isAuth змінюється на true, і користувач переходить до головного меню - "Chat List". Тепер давайте розглянемо детальніше, як саме клієнт проходить процес авторизації.

Клієнт взаємодіє з сервером, щоб отримати аутентифікаційні дані. Ця взаємодія побудована на основі клієнт-серверної архітектури. Усі дані, що пов'язані з аутентифікацією, передаються клієнтом та отримуються за допомогою HTTP-запитів. Це забезпечує вищий рівень безпеки для розробленої інформаційної системи. Клієнт формує тіло запиту на основі станів форми авторизації та відправляє його на сервер, очікуючи відповіді. Однією з основних відмінностей між HTTP-запитами та веб-сокет-зв'язком є те, що HTTP завжди отримує відповідь від сервера, що також підвищує надійність нашої системи аутентифікації.

```
//Аутентифікація на сервері
axios.post(`${SERVER_URL}/login`, loginData)
.then(response => {
    const { success, message, token, login } = response.data;
    if (success) {
        SecureStore.setItemAsync('token', token)
        .then(() => {
            setIsAuth(true)
            setMyLogin(login)
            console.log('Token:', token);

        })
        .then(() => {
            console.log('Login: ', myLogin)
        })
        .catch(error => {
            console.log('SecureStore error:', error.message);
        });
    }
})
.catch(error => {
    if (error.response.data.message === 'Login error: wrong login') {
        setErrorLogin(true)
    }
    else if (error.response.data.message === 'Login error: wrong password') {
        setErrorPassword(true)
    }
    else {
      console.error(error);
    }
})
```

Рис. 2.6. Процес аутентифікації

Після успішної аутентифікації, клієнт зберігає токен, який він надалі буде використовувати для наступних запитів. Також токен дозволяє зберігати статус аутентифікація навіть після закриття застосунку, тому користувачу не потрібно щоразу авторизовуватись – він відразу потрапить до вікна Chat List. Це була реалізація http запитів.

Коли користувач потрапляє до "Chat List", спочатку він надсилає HTTP-запит до сервера, щоб отримати свої чати. Після успішного отримання та відображення чатів, клієнт підписується на WebSocket-прослуховувач. WebSocket - це протокол двостороннього спілкування, який дозволяє встановити постійне з'єднання між клієнтом та сервером. За допомогою

WebSocket можна реалізувати логіку передачі повідомлень: клієнт може надсилати повідомлення на сервер, а сервер передає це повідомлення всім іншим клієнтам. Це досягається за допомогою прослуховувача подій on. В якості аргументу прослуховувача передається маршрут і функція-зворотний виклик, яка виконує обробку отриманого повідомлення.

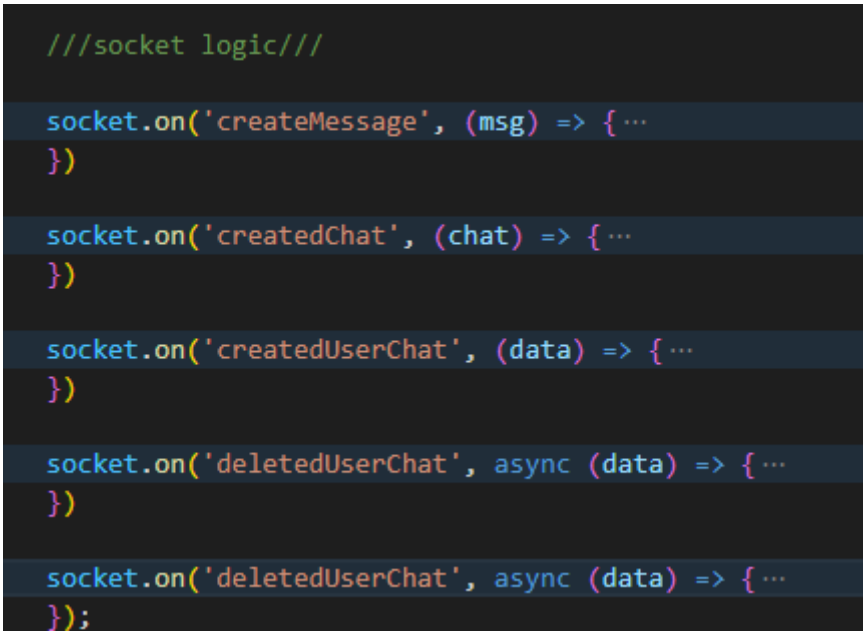

Рис. 2.7. Прослуховувачі подій

Події в програмі можуть бути різного характеру, але більшість з них пов'язані з операціями створення (Create), отримання (Read), оновлення (Update) та видалення (Delete) даних в таблицях бази даних. Наприклад, для відображення списку чатів нам потрібна інформація з таблиці "chat", для відображення останнього повідомлення в чаті - з таблиці "message", а для перевірки, чи є користувач учасником чату, потрібна інформація з таблиці "UserChat", ми можемо побачити, що всі прослуховувачі у вікні з чатами обробляють дані з таблиць chat, user та UserChat, з яких вікно брало для себе дані. Аналогічна схема використовується й у інших вікнах програми - вони завантажують певні таблиці бази даних та слідкують за змінами в цих таблицях.

У базі даних програми використовуються чотири таблиці: "chat", "user", "UserChat" та "message". Кожна з цих таблиць відповідає за зберігання конкретної категорії даних. Наприклад, таблиця "chat" містить інформацію про чати, таблиця "user" містить дані про користувачів, а таблиця "message" зберігає

повідомлення. Таблиця "UserChat" виконує функцію зберігання зв'язку між користувачами та чатами. Вона містить пари user_id-chat_id, що дозволяє ідентифікувати, який користувач знаходиться в якому чаті. Цю таблицю також можна назвати "members", оскільки вона відображає членів чату. Нижче наведено схему бази даних для кращого уявлення про структуру програми:

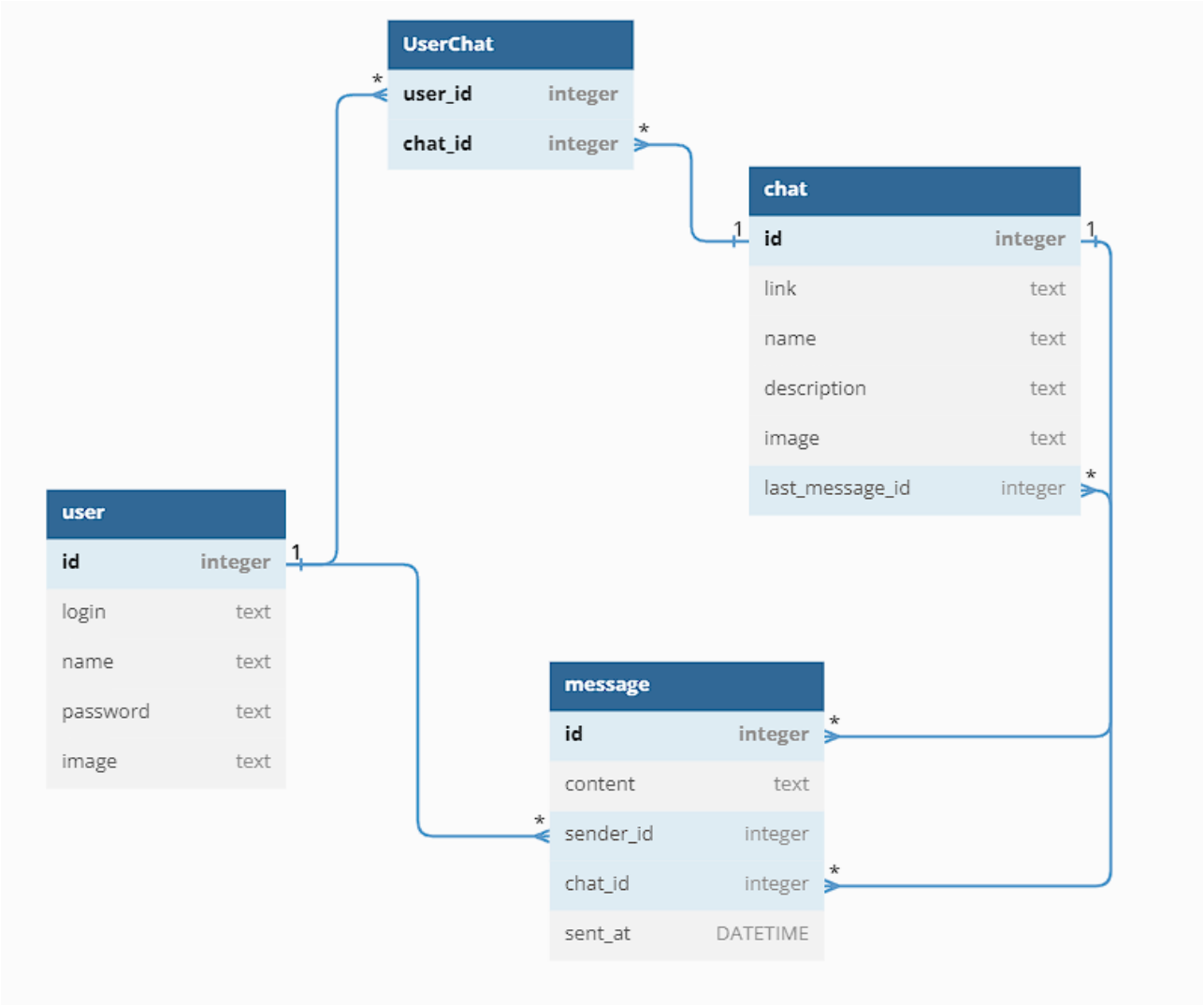

Рис. 2.8. Структура бази даних програми

Серверна частина програми побудована з використанням фреймворку Express. Все, що реалізовано на сервері, - це набір маршрутів, які виконують SQL-команди до бази даних. Важливо звернути увагу на масив "clients", який існує на сервері. Його завдання - зберігати відповідність між "socket.id" (ідентифікатор сокета) і "user.id" (ідентифікатор користувача). Це необхідно, щоб сервер міг отримати "socket.id" за "user.id" і відправити повідомлення конкретному клієнту. Оскільки базовий варіант ідентифікації за "socket.id" не

забезпечує повноцінну ідентифікацію користувачів, "clients" дозволяє використовувати "user.id" для спрямування повідомлень до конкретних клієнтів за їх "socket.id".

```
1   import express from 'express'
2   import sqlite3 from 'sqlite3'
3   import jwt from 'jsonwebtoken'
4   import { Server } from 'socket.io'
5
6   const PORT = process.env.PORT ?? 3000
7   const db = new sqlite3.Database('database/messengerapp.db')
8   const SECRET_KEY = process.env.SECRET_KEY ?? '88888888'
9
10  var clients = []
11
12  const app = express()
13
14  app.use(express.json())
15
16
17  app.post('/register', (req, res) => {···
47  })
48
49  app.post('/login', (req, res) => {···
76  })
77
78  app.post('/check-token', (req, res) => {···
89  })
90
91  app.get('/chat/:chat_id', (req, res) => {···
105 })
106
107 app.get('/chats', (req, res) => {···
174 })
```

Рис. 2.9. Структура серверної частини програми

Короткий опис інших компонентів проекту:

- client/src/components – папка з кастомними компонентами;
- client/assets – папка з зображенями для лого месенджеру та фону чату;
- client/.env – файл з змінами середовища, тут записано SERVER_URL та TIME_ZONE;
- server/database – папка з базою даних, розташовується локально на сервері.

2.6. Обгрунтування та організація вхідних та вихідних даних програми

У розділі "Обґрунтування та організація вхідних та вихідних даних програми" розглянуті такі аспекти:

Характер, організація і попередня підготовка вхідних даних:

У React Native месенджері, розробленому у рамках цього дипломного проекту, характер вхідних даних визначається різними екранами та їх функціями. Наприклад, на екрані аутентифікації (логін та реєстрація) користувач вводить свої облікові дані. На екрані налаштувань, користувач може переглянути та змінити свій профіль. Вхідні дані також включають повідомлення, які користувач отримує від сервера у реальному часі, включаючи дані про чати та повідомлення у цих чатах.

Організація вхідних даних полягає у взаємодії з сервером на Node.js. Запити клієнта до сервера виконуються за допомогою протоколу HTTP для отримання та відправлення даних. Крім того, використовується бібліотека Socket.io для забезпечення постійного оновлення даних у реальному часі за допомогою веб-сокетів.

Формат, опис і спосіб кодування вхідних даних залежать від типу даних. Наприклад, облікові дані користувача можуть передаватися у форматі JSON, який легко оброблюється сервером. Повідомлення можуть бути у форматі тексту або структурованого JSON, що залежить від потреб додатку та його функцій.

Характер і організація вихідних даних:

Вихідні дані програми React Native месенджера включають відображення різних екранів та їх вмісту. Наприклад, на екрані чату вихідні дані включають повідомлення, які користувачі відправляють один одному, інформацію про чат, включаючи список учасників чату, а також профілі користувачів.

Організація вихідних даних включає відображення даних на екранах за допомогою компонентів React Native та управління станом даних. Вихідні дані

оновлюються у реальному часі завдяки використанню протоколу HTTP та веб-сокетів для отримання та передачі даних між клієнтом та сервером.

Формат, опис і спосіб кодування вихідних даних:

Формат та опис вихідних даних залежать від типу вмісту, який відображається на екранах. Наприклад, повідомлення можуть бути в текстовому або JSON форматі, що залежить від потреб додатку. Інформація про користувачів та їх профілі також може бути представлена у структурованому форматі JSON.

Щодо способу кодування вихідних даних, React Native використовує JavaScript для створення компонентів та управління вмістом екранів. Дані можуть бути передані у вигляді об'єктів або рядків JSON, які потім можна обробляти та відображати на відповідних екранах.

Таким чином, вхідні та вихідні дані програми React Native месенджера організовані з урахуванням взаємодії з сервером на Node.js та базою даних SQLite. Формат та спосіб кодування даних визначаються відповідно до типу даних, які передаються та відображаються на різних екранах за допомогою компонентів React Native.

2.7. Опис розробленого програмного продукту

Розроблений програмний продукт - це мобільний месенджер, який призначений для забезпечення зручного та швидкого обміну повідомленнями між користувачами. Програмний продукт реалізує широкий функціонал чату, що дозволяє користувачам спілкуватися один з одним в реальному часі.

Розроблений програмний продукт використовує сучасні технології та інструменти, такі як Node.js для серверної частини та React Native для клієнтської частини. Для зберігання даних використовується база даних SQLite. Продукт також використовує пакет Expo для спрощення розробки та тестування на мобільних пристроях.

2.7.1. Використані технічні засоби

Мій персональний комп’ютер:

1) Центральний процесор (ЦП): Ryzen 5 3600.
2) Відеокарта: RX 5500 XT.
3) Відеопам'ять: 8 ГБ.
4) Накопичувач: 1 ТБ.
5) Оперативна пам'ять: 16 ГБ.
6) Клавіатура, миша, монітор.

Мій смартфон: Google Pixel 4 128G

2.7.2. Використані програмні засоби

Для забезпечення функціонування розробленої системи були використані наступні операційні системи та програмні засоби:

1. Редагування та розробка програмного коду:
   - VS Code: Інтегроване середовище розробки (IDE), яке надає зручні можливості для написання, відлагодження та керування програмним кодом.
2. Робота з базою даних:
   - DB Browser for SQLite: Графічний інструмент для роботи з базою даних SQLite. Використовувався для створення, редагування та перегляду даних в базі даних на основі SQLite.
3. Виконання та тестування мобільного додатку:
   - Expo Go: Додаток, який дозволяє запускати та тестувати мобільні додатки, розроблені з використанням платформи Expo. Використовувався для тестування розробленого мобільного додатку на різних пристроях.

Зазначені програмні засоби були використані з метою полегшення розробки, управління кодом, роботи з базою даних та тестування мобільного додатку.

2.7.3. Виклик та завантаження програми

Для запуску та виклику розробленої системи, яка включає серверний застосунок на платформі node.js та клієнтський застосунок на платформі React Native, використовуються наступні способи виклику та завантаження програми:

Для запуску серверного застосунку, розробленого на платформі node.js, необхідно виконати команду node index.js. Це запустить сервер та розпочне його роботу.

Для запуску клієнтського застосунку, розробленого на платформі React Native, використовується пакет Expo. Для початку розробки та тестування застосунку виконується команда expo start. Ця команда запускає розробничий сервер Expo, який дозволяє завантажити та виконати застосунок на фізичних пристроях або емуляторах.

Важливо враховувати, що для успішного виконання системи необхідно наявність підтримуваного середовища для виконання node.js та Expo для клієнтського застосунку, а також коректно введений SERVER_URL у .env файлі.

2.7.4. Опис інтерфейсу користувача

При першому запуску клієнт потрапить на екран аутентифікації. З екрана аутентифікації користувач може перейти на екран реєстрації. Після успішної реєстрації або аутентифікації, користувач буде перенаправлений на екран списку чатів.

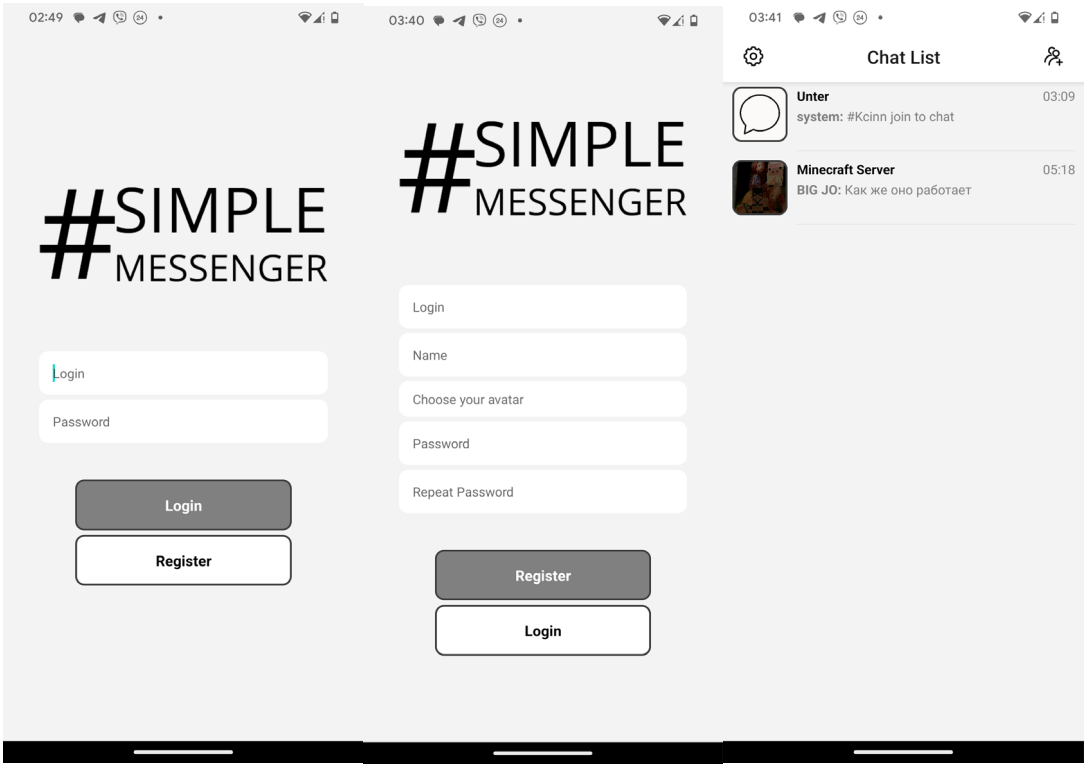

Рис. 2.10. Аутентифікація

В месенджері користувач має можливість виконувати наступні дії. Він може натиснути на кнопку, розташовану угорі ліворуч, щоб увійти в налаштування та змінити свої особисті дані. Також, натиснувши на кнопку, розташовану угорі праворуч, користувач може перейти до вікна створення нового чату.

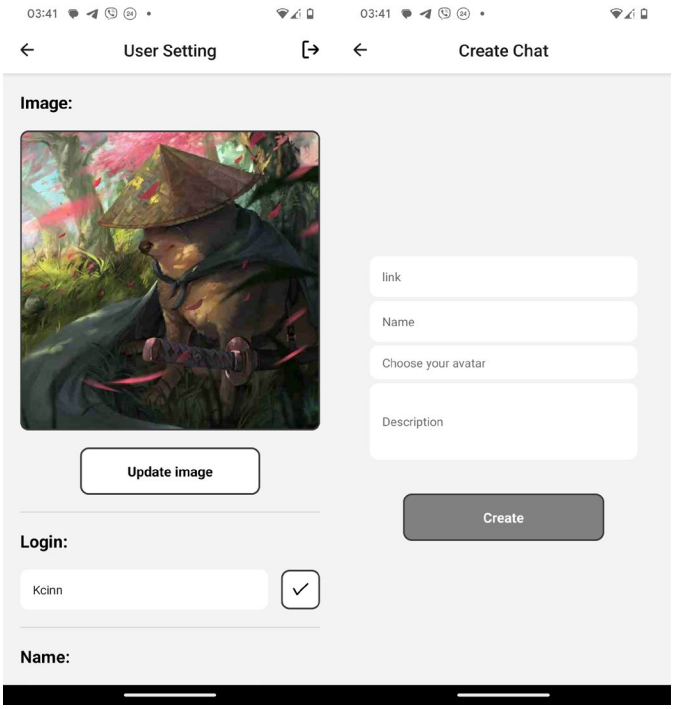

Рис. 2.11. Налаштуванн Рис. 2.12. Створення чату

Якщо у користувача вже є чати, він може натиснути на відповідний чат у списку, щоб перейти до списку повідомлень цього чату. У вікні з повідомленнями користувач може вводити нові повідомлення у полі внизу та надсилати їх. Він також може повернутися до списку чатів, натиснувши кнопку угорі ліворуч. Крім того, він може перейти до вікна з інформацією про чат, натиснувши кнопку угорі праворуч.

У вікні з інформацією про чат користувач може додати нового учасника до чату або вийти з чату. Якщо користувач бажає вийти зі свого облікового запису, він може натиснути кнопку угорі праворуч у налаштуваннях та здійснити вихід з аккаунту.

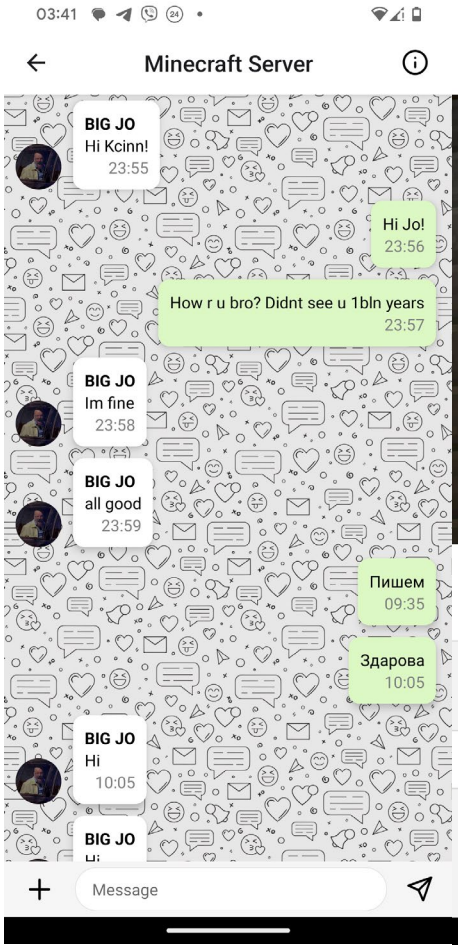

рис. 2.13. Чат

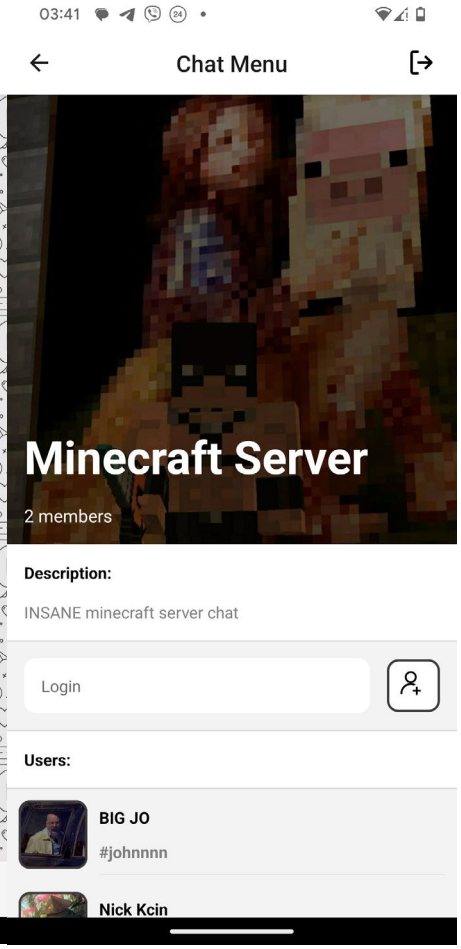

рис. 2.14. Інформація про чат

РОЗДІЛ 3
ЕКОНОМІЧНИЙ РОЗДІЛ

3.1. Розрахунок трудомісткості та вартості розробки програмного продукту

Вихідні дані:

1. передбачуване число операторів програми – 1632;

2. коефіцієнт складності програми – 1,25;

3. коефіцієнт корекції програми в ході її розробки – 0,2;

4. годинна заробітна плата розробника – 115 грн/год (за версією сайта WorkUA) - https://www.work.ua/salary-dnipro-it/;

5. коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі – 1,2;

6. коефіцієнт кваліфікації програміста, обумовлений від стажу роботи з даної спеціальності – 1,1;

7. вартість машино-години ЕОМ – 18 грн/год (1 грн – е/е, 10 грн – ПЗ, 7 грн -амортизація).

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки. Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д \text{, людино-годин,}$$

де t_o – витрати праці на підготовку й опис поставленої задачі (приймається 50);

t_u – витрати праці на дослідження алгоритму рішення задачі;

t_a – витрати праці на розробку блок-схеми алгоритму;

t_n – витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ – витрати праці на налагодження програми на ЕОМ;

$t_д$ – витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється.

Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1 + p),$$

де q – передбачуване число операторів (1632);

C – коефіцієнт складності програми (1,25);

p – коефіцієнт корекції програми в ході її розробки (0,2).

$$Q = 1632 \cdot 1{,}25 \cdot (1 + 0{,}2) = 2448;$$

Витрати праці на вивчення опису задачі t_u визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75...85) \cdot K}, \text{ людино-годин}$$

де В – коефіцієнт збільшення витрат праці внаслідок недостатнього опису задачі (1,2);

К – коефіцієнт кваліфікації програміста, обумовлений стажем роботи з даної спеціальності (1,1);

$$t_u = \frac{2448 \cdot 1{,}2}{85 \cdot 1{,}1} = 31{,}42 \text{ людино-годин.}$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20...25) \cdot K};$$

$$t_a = \frac{2448}{20 \cdot 1{,}1} = 111{,}27 \text{ людино-годин.}$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20...25)\cdot K};$$

$$t_n = \frac{2448}{25\cdot 1,1} = 89 \text{ людино-годин.}$$

Витрати праці на налагодження програми на ЕОМ:

– за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(4...5)\cdot K};$$

$$t_{отл} = \frac{2448}{5\cdot 1,1} = 445 \text{ людино-годин.}$$

– за умови комплексного налагодження завдання:

$$t_{отл}^{к} = 1,2 \cdot t_{отл};$$

$$t_{отл}^{к} = 1,2 \cdot 445 = 534 \text{ людино-годин.}$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до};$$

де $t_{др}$ – трудомісткість підготовки матеріалів і рукопису;

$$t_д = \frac{Q}{(15...20)\cdot K};$$

$$t_{др} = \frac{2448}{20\cdot 1,1} = 111,27 \text{ людино-годин.}$$

де $t_{до}$ – трудомісткість редагування, печатки й оформлення документації;

$$t_{до} = 0,75 \cdot t_{др};$$

$$t_{до} = 0{,}75 \cdot 111{,}27 = 83{,}45 \text{ людино-годин.}$$

$$t_{д} = 111{,}27 + 83{,}45 = 194{,}72 \text{ людино-годин.}$$

Отримаємо трудомісткість розробки програмного забезпечення:

$$t = 50 + 31{,}42 + 111{,}27 + 89 + 445 + 194{,}72 = 921{,}41 \text{ людино-годин.}$$

У результаті ми розрахували, що в загальній складності необхідно 921,41 людино-годин для розробки даного програмного забезпечення.

3.2. Розрахунок витрат на створення програми

Витрати на створення ПЗ $Кпо$ включають витрати на заробітну плату виконавця програми $З_{ЗП}$ витрат машинного часу, необхідного на налагодження програми на ЕОМ.

$$Кпо = З_{ЗП} + З_{МВ}, \text{ грн,}$$

$З_{ЗП}$ – заробітна плата виконавців, яка визначається за формулою:

$$З_{ЗП} = t \cdot С_{ПР}, \text{ грн,}$$

де t – загальна трудомісткість, людино-годин;

$С_{ПР}$ – середня годинна заробітна плата програміста, грн/година.

З урахуванням того, що середня годинна зарплата розробника становить 115 грн/год, то отримаємо:

$$З_{ЗП} = 921{,}41 \cdot 115 = 105\,962{,}15 \text{ грн}$$

Вартість машинного часу $З_{МВ}$, необхідного для налагодження програми на ЕОМ, визначається за формулою:

$$З_{МВ} = t_{отл} \cdot С_М \text{, грн,}$$

де $t_{отл}$ – трудомісткість налагодження програми на ЕОМ, год;

$С_М$ – вартість машино-години ЕОМ, грн/год.

$$З_{МВ} = 445 \cdot 18 = 8010 \text{ грн}$$

Звідси витрати на створення програмного продукту:

$$Кпо \quad 105\,962{,}15 + 8010 = 113\,972{,}15 \text{ грн}$$

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p} \text{, мес.}$$

де B_k – число виконавців;

F_p – місячний фонд робочого часу (при 40 годинному робочому тижні F_p = 176 годин).

Витрати на створення програмного продукту:

$$T = \frac{921{,}41}{1 \cdot 176} = 5{,}23 \text{ міс.}$$

Вартість розробки інтернет-магазину становить 113 972,15 грн. Час розробки очікується приблизно 5,23 місяці при стандартному 40-годинному робочому тижні і 176-годинному робочому місяці. Цей термін включає час, необхідний для дослідження, розробки алгоритму, дизайну і документування. Загальна кількістю людино-годин, яку буде витрачено на розробку – 921,41.

ВИСНОВКИ

У рамках даної дипломної роботи був розроблений мобільний месенджер на платформі React Native. Програмний продукт був створений з метою надання користувачам можливості спілкування за допомогою текстових повідомлень та реалізації функціоналу чату.

Результати роботи над проектом були оцінені відносно аналогів на ринку месенджерів, і виявилося, що розроблений мобільний додаток відповідає основним вимогам та функціональності, яку надають інші популярні месенджери. Він забезпечує зручний і безпечний спосіб спілкування для користувачів.

Новизна розробленого месенджера полягає в його імплементації на платформі React Native, що дозволяє запускати додаток на різних мобільних пристроях. Практичне значення розробленого месенджера полягає в його можливості бути використаним як основний засіб комунікації для користувачів мобільних пристроїв. Він надає зручність та швидкість спілкування, що важливо в сучасному цифровому світі.

Наукове значення даної роботи полягає у використанні сучасних технологій розробки мобільних додатків та вивченні практичного застосування платформи React Native. Результати дослідження можуть бути корисними для інших розробників, які планують створювати мобільні додатки з використанням подібної технології.

Щодо подальшого розвитку об'єкта дослідження, можливість розширення функціоналу месенджера шляхом додавання додаткових можливостей, таких як відео- та аудіодзвінки, може покращити його конкурентоспроможність на ринку месенджерів.

У економічному розділі були розраховані трудомісткість розробки програмного забезпечення та витрати на його створення. Прогнозні показники свідчать про те, що розроблений месенджер може мати певну економічну цінність та потенціал для успішної комерціалізації.

Отже, розробка мобільного месенджера на платформі React Native відповідає поставленим цілям та завданням дипломної роботи. Результати дослідження та розробки вказують на успішну реалізацію проекту та його потенціал для подальшого розвитку та використання у практичних ситуаціях.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. React Native. URL: https://reactnative.dev/
2. Learning React Native. - Bonnie Eisenman, 2015. - 322 pages.
3. React Native in Action. - Nader Dabit, 2018. - 408 pages.
4. React Native Cookbook. - Dan Ward, 2017. - 442 pages.
5. Fullstack React Native: The Complete Guide to React Native. - Devin Abbott, 2017. - 244 pages.
6. React Native for iOS Development. - Akshat Paul, 2016. - 200 pages.
7. Node.js. URL: https://nodejs.org/en/docs
8. React Native Blueprints. - Emilio Rodriguez Martinez, 2016. - 366 pages.
9. React Native By Example. - Richard Kho, 2016. - 298 pages.
10. Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development. - Azat Mardan, 2014. - 372 pages.
11. Express.js Guide: The Comprehensive Book on Express.js. - Azat Mardan, 2016. - 154 pages.
12. Web Development with Node and Express: Leveraging the JavaScript Stack. - Ethan Brown, 2014. - 336 pages.
13. Express. URL: https://expressjs.com/
14. Learning Node.js: A Hands-On Guide to Building Web Applications in JavaScript. - Marc Wandschneider, 2013. - 296 pages.
15. Node.js in Action. - Mike Cantelon, 2014. - 416 pages.
16. Practical Node.js: Building Real-World Scalable Web Apps. - Azat Mardan, 2014. - 400 pages.
17. Beginning Node.js. - Basarat Ali Syed, 2014. - 296 pages.
18. Learning SQLite for iOS. - Gene Da Rocha, 2015. - 192 pages.
19. SQLite. URL: https://www.sqlite.org/docs.html
20. SQLite Database Programming for Xamarin: Cross-platform C# Database Development for iOS and Android using SQLite.NET. - Greg Shackles, 2015. - 168 pages.

21. SQL Cookbook: Query Solutions and Techniques for Database Developers. - Anthony Molinaro, 2019. - 574 pages.
22. notJust-dev/whatsapp. URL: https://github.com/notJust-dev/whatsapp
23. Beginning SQL Queries: From Novice to Professional. - Clare Churcher, 2008. - 560 pages.

ДОДАТОК А

КОД ПРОГРАМИ

server/index.js:

```
import express from 'express'
import sqlite3 from 'sqlite3'
import jwt from 'jsonwebtoken'
import { Server } from 'socket.io'

const PORT = process.env.PORT ?? 3000
const db = new sqlite3.Database('database/messengerapp.db')
const SECRET_KEY = process.env.SECRET_KEY ?? '88888888'

var clients = []

const app = express()

app.use(express.json())


app.post('/register', (req, res) => {
    const { login, name, password } = req.body
    const image = req.body.image == '' ?
'https://static.vecteezy.com/system/resources/previews/008/442/086/original/illustratio
n-of-human-icon-user-symbol-icon-modern-design-on-blank-background-free-vector.jpg'
                                       : req.body.image
    const checkUserQuery = 'SELECT * FROM user WHERE login = ?';
    const checkUserParams = [login];

    db.get(checkUserQuery, checkUserParams, (err, row) => {
        if (err) {
            console.error(err);
            res.status(500).json({ success: false, message: 'Register error: cant
select user' });
        } else if (row) {
            // Користувач з даним логіном вже існує
            res.status(200).json({ success: false, message: 'Register error: this user
is already in db' });
        } else {
            // Додаємо нового користувача до таблиці
            const insertUserQuery = 'INSERT INTO user (login, name, password, image)
VALUES (?, ?, ?, ?)';
            const insertUserParams = [login, name, password, image];

            db.run(insertUserQuery, insertUserParams, (err) => {
                if (err) {
                    console.error(err);
                    res.status(500).json({ success: false, message: 'Register error:
cant insert user' });
                } else {
                    // Користувача додано вдало
                    res.status(200).json({ success: true, message: 'Register is
successful' });
                }
            });
        }
    });
})
```

```
app.post('/login', (req, res) => {
    const { login, password } = req.body;

    const findUserQuery = `SELECT * FROM user WHERE login = ?`;
    db.get(findUserQuery, [login], (err, row) => {
        if (err) {
            console.error(err);
            res.status(500).json({ success: false, message: 'Login error: cant select
user' });
        } else if (!row) {
            // Користувача з таким логіном не знайдено
            res.status(401).json({ success: false, message: 'Login error: wrong login'
});
        } else {
            // Користувач знайдений, перевіряємо правильність пароля
            if (row.password === password) {
                // Пароль вірний, виконайте необхідні дії для авторизації користувача
                const token = jwt.sign({ login: row.login }, SECRET_KEY);

                console.log(token)

                // Повертаємо успішну відповідь з токеном доступу
                res.status(200).json({ success: true, message: 'Login is successful',
token: token, login: login });
            } else {
                // Невірний пароль
                res.status(401).json({ success: false, message: 'Login error: wrong
password' });
            }
        }
    });
})

app.post('/check-token', (req, res) => {

    const { token } = req.body;
    console.log("Проверяю токен: "+token)

    jwt.verify(token, SECRET_KEY, (err, decoded) => {
        if (err)
            res.status(401).json({ success: false, message: 'Token verification failed'
});
        else
            res.status(200).json({ success: true, message: 'Token verification
successful', login: decoded.login });
    });
})

app.get('/chat/:chat_id', (req, res) => {
    const chat_id = req.params.chat_id;

    const getChat = `SELECT * FROM chat WHERE id = ?`;

    db.get(getChat, [chat_id], (err, row) => {
        if (err) {
            console.error(err);
            res.status(500).json({ success: false, message: 'Error getting chat' });
        } else {
            const chat = row
            res.status(200).json(chat);
        }
    });
```

```
})

app.get('/chats', (req, res) => {
    const token = req.headers.authorization;
    jwt.verify(token, SECRET_KEY, (err, decoded) => {
        if (err)
            res.status(401).json({ success: false, message: 'Token verification failed'
});
        else {
            const userLogin = decoded.login;

            const getUserIdQuery = `SELECT id FROM user WHERE login = ?`;
            db.get(getUserIdQuery, [userLogin], (err, row) => {
                if (err) {
                    console.error(err);
                    res.status(500).json({ success: false, message: 'Error fetching
user' });
                } else {
                    const userId = row.id;

                    const getChatsQuery = `
                    SELECT
                        c.id AS chat_id,
                        c.link,
                        c.name AS chatname,
                        c.image,
                        m.id AS last_message_id,
                        m.content,
                        m.sender_id,
                        m.chat_id,
                        m.sent_at,
                        u.name AS username
                    FROM
                        chat AS c
                        LEFT JOIN message AS m ON c.last_message_id = m.id
                        JOIN UserChat AS uc ON c.id = uc.chat_id
                        JOIN user AS u ON m.sender_id = u.id
                    WHERE
                        uc.user_id = ?
                    `;

                    db.all(getChatsQuery, [userId], (err, rows) => {
                        if (err) {
                            console.error(err);
                            res.status(500).json({ success: false, message: 'Error
fetching chats' });
                        } else {
                            const chats = rows.map((row) => {
                                return {
                                    id: row.chat_id,
                                    link: row.link,
                                    name: row.chatname,
                                    image: row.image,
                                    lastMessage: {
                                        id: row.last_message_id,
                                        content: row.content,
                                        sender_id: row.sender_id,
                                        chat_id: row.chat_id,
                                        sent_at: row.sent_at,
                                        user: {
                                            name: row.username
                                        }
```

```
                                    }
                                };
                            });
                            res.status(200).json(chats);
                        }
                    });
                }
            });
        }
    });
})

app.get('/messages/:chat_id', (req, res) => {
    const chat_id = req.params.chat_id;

    const getMessagesQuery = `
    SELECT
      m.id,
      m.content,
      m.sender_id,
      m.chat_id,
      m.sent_at,
      u.login,
      u.name,
      u.image
    FROM
      message AS m
      JOIN user AS u ON m.sender_id = u.id
    WHERE
      m.chat_id = ?
    `;

    db.all(getMessagesQuery, [chat_id], (err, rows) => {
        if (err) {
            console.error(err);
            res.status(500).json({ success: false, message: 'Error fetching messages'
});
        } else {
            const messageList = rows.map((row) => ({
                chat_id: row.chat_id,
                content: row.content,
                id: row.id,
                sender_id: row.sender_id,
                sent_at: row.sent_at,
                user: {
                    login: row.login,
                    name: row.name,
                    image: row.image,
                },
            }));

            res.status(200).json(messageList);
        }
    });
})

app.get('/user', (req, res) => {
    const token = req.headers.authorization;

    jwt.verify(token, SECRET_KEY, (err, decoded) => {
        if (err)
```

```
            res.status(401).json({ success: false, message: 'Token verification failed'
});
        else {
            const login = decoded.login

            const findUserQuery = `SELECT * FROM user WHERE login = ?`;
            db.get(findUserQuery, [login], (err, row) => {
                if (err) {
                    console.error(err);
                    res.status(500).json({ success: false, message: 'Login error: cant
select user' });
                } else if (!row) {
                    // Користувача з таким логіном не знайдено
                    res.status(401).json({ success: false, message: 'Login error: wrong
login' });
                } else {
                    const user = {
                        id: row.id,
                        login: row.login,
                        name: row.name,
                        image: row.image
                    }
                    res.status(200).json(user);
                }
            });
        }
    });
})

app.get('/user/:user_id', (req, res) => {
    const user_id = req.params.user_id;
    const token = req.headers.authorization;

    jwt.verify(token, SECRET_KEY, (err, decoded) => {
        if (err)
            res.status(401).json({ success: false, message: 'Token verification failed'
});
        else {
            const login = decoded.login

            const findUserQuery = `SELECT * FROM user WHERE id = ?`;
            db.get(findUserQuery, [user_id], (err, row) => {
                if (err) {
                    console.error(err);
                    res.status(500).json({ success: false, message: 'Login error: cant
select user' });
                } else if (!row) {
                    // Користувача з таким логіном не знайдено
                    res.status(401).json({ success: false, message: 'Login error: wrong
login' });
                } else {
                    const user = {
                        id: row.id,
                        login: row.login,
                        name: row.name,
                        image: row.image
                    }
                    res.status(200).json(user);
                }
            });
        }
    });
```

```
})

app.get('/users/:chat_id', (req, res) => {
    const chat_id = req.params.chat_id;

    const query = `
    SELECT u.* FROM user AS u
    JOIN UserChat AS uc ON uc.user_id = u.id
    WHERE uc.chat_id = ?
    `;

    db.all(query, [chat_id], (err, rows) => {
        if (err) {
            console.error(err);
            res.status(500).send('Select Users Error');
            return;
        }

        res.json(rows);
    });
})

app.put('/user/avatar', (req, res) => {
    const token = req.headers.authorization;

    jwt.verify(token, SECRET_KEY, (err, decoded) => {
        if (err)
            res.status(401).json({ success: false, message: 'Token verification failed'
});
        else {
            const login = decoded.login
            const image = req.image

            const updateQuery = `UPDATE user SET image = ? WHERE login = ?`;
            db.run(updateQuery, [image, login], (err) => {
                if (err) {
                    console.error(err);
                    res.status(500).json({ success: false, message: 'Failed to update
user image' });
                } else {
                    res.status(200).json({ success: true, message: 'User image updated
successfully' });
                }
            })
        }
    });
})

app.post('/chat', (req, res) => {
    const token = req.headers.authorization;
    const chatData = req.body.chatData;

    console.log(chatData)


})

const server = app.listen(PORT, () => {
    console.log(`Server has been started on port ${PORT}...`)
})

const io = new Server(server)
```

```
io.on('connection', (socket) => {
    console.log(`Client ${socket.id} just connected`)

    socket.on('idMe', (token) => {
        jwt.verify(token, SECRET_KEY, (err, decoded) => {
            if (err)
                console.log(err)
            else {
                const getUserId = `SELECT id FROM user WHERE login = ?`
                db.get(getUserId, [decoded.login], (err, row) => {
                    if (err) {
                        console.log(err)
                    } else {
                        const client = {
                            socketId: socket.id,
                            userId: row.id
                        }

                        var sockeNotFound = true

                        clients.forEach((cl) => {
                            if(cl.socketId === socket.id){
                                sockeNotFound = false
                                cl.userId = client.userId
                                console.log(clients)
                                return
                            }

                        })

                        if(sockeNotFound) {
                            clients.push(client)
                            console.log(clients)
                        }
                    }
                })
            }
        });
    })

    socket.on('joinToRoom', (id) => {
        socket.join(id)
        console.log(`Client ${socket.id} - join to room ${id}`)
    })

    socket.on('leaveRoom', (id) => {
        socket.leave(id)
        console.log(`Client ${socket.id} - leaves the room ${id}`)
    })

    socket.on('createMessage', (message) => {
        const {content, chat_id, sender_token} = message
        console.log('Получил месседж:')
        console.log(JSON.stringify(message))

        jwt.verify(sender_token, SECRET_KEY, (err, decoded) => {
            if(!err){
                const login = decoded.login;

                const getUserIdQuery = `SELECT id, name, image FROM "user" WHERE login
= ?`;
```

```
                db.get(getUserIdQuery, [login], (err, row) => {
                    if (err) {
                        console.error(err);
                    } else {
                        const userId = row.id;
                        const userName = row.name
                        const image = row.image

                        // Добавление нового сообщения в таблицу message
                        const addMessageQuery = `
                            INSERT INTO message (content, sender_id, chat_id, sent_at)
                            VALUES (?, ?, ?, DATETIME('now'))
                        `;
                        db.run(addMessageQuery, [content, userId, chat_id], function
(err) {
                            if (err) {
                                console.error(err);
                            } else {
                                const messageId = this.lastID;

                                // Обновление поля last_message_id в таблице chat
                                const updateChatQuery = `
                                    UPDATE chat
                                    SET last_message_id = ?
                                    WHERE id = ?
                                `;
                                db.run(updateChatQuery, [messageId, chat_id], function
(err) {
                                    if (err) {
                                        console.error(err);
                                    } else {
                                        // Отправка сообщения всем пользователям в
чатруме
                                        const messageObject = {
                                            id: messageId,
                                            content: content,
                                            sender_id: userId,
                                            chat_id: chat_id,
                                            sent_at: new Date(),
                                            user: {
                                                name: userName,
                                                login: login,
                                                image: image
                                            }
                                        };

                                        io.to(chat_id).emit('createMessage',
messageObject);
                                    }
                                });
                            }
                        });
                    }
                });
            }
        })
    })

    socket.on('updateUserImage', (message) => {
        const {image, token} = message
        console.log('Выполняю запрос на апдейт имедж')
        jwt.verify(token, SECRET_KEY, (err, decoded) => {
```

```
            if(!err){
                const login = decoded.login;
                const updateQuery = `UPDATE user SET image = ? WHERE login = ?`;
                db.run(updateQuery, [image, login], (err) => {
                    if (err) {
                        console.error(err);
                    } else {


                        const selectUserQuery = 'SELECT id FROM user WHERE login = ?';
                        db.get(selectUserQuery, [login], (err, user) => {
                            if (err) {
                                console.error(err);
                            } else {
                                if (user) {
                                    const userId = user.id;

                                    const selectUserQuery = 'SELECT * FROM user WHERE
login = ?'
                                    db.get(selectUserQuery, [login], (err,userUpdated)
=> {
                                        if (err) {
                                            console.error(err);
                                        } else {
                                            const userResult = {
                                                id: userUpdated.id,
                                                login: userUpdated.login,
                                                name: userUpdated.name,
                                                image: image
                                            }

                                            // Получить все chat_id по user_id=user.id
из таблицы UserChat
                                            const selectChatIdsQuery = 'SELECT chat_id
FROM UserChat WHERE user_id = ?';
                                            db.all(selectChatIdsQuery, [userId], (err,
rows) => {
                                                if (err) {
                                                    console.error(err);
                                                } else {
                                                    // Для каждого chat_id выполнить
io.to(chat_id).emit('updateUserImage', image);
                                                    rows.forEach((row) => {
                                                        const chatId = row.chat_id;
                                                        console.log('Отправляю имедж в
комнату ' + chatId)

io.to(chatId).emit('updateUserImage', userResult);
                                                    });
                                                }
                                            });
                                        }
                                    })
                                }
                            }
                        });
                    }
                })
            }
        })
    })
```

```
    /*socket.on('updateUserLogin', (message) => {
        const {login, token, id} = message
        console.log('Выполняю запрос на апдейт логин')
        jwt.verify(token, SECRET_KEY, (err, decoded) => {
            if(!err){
                const realLogin = decoded.login

                const checkLoginQuery = 'SELECT id FROM user WHERE login = ?'

                db.get(checkLoginQuery, [login], (err, user) => {
                    if (err) {
                        console.error(err);
                    } else {
                        console.log(user)
                        if (!user) {
                            const updateLoginQuery = `UPDATE user SET login = ? WHERE
id = ?`;

                            db.get(updateLoginQuery, [login, id], (err, user) => {
                                if (err) {
                                    console.error(err);
                                } else {
                                    const selectUserQuery = 'SELECT * FROM user WHERE
login = ?'
                                    db.get(selectUserQuery, [login], (err,userUpdated)
=> {
                                        if (err) {
                                            console.error(err);
                                        } else {
                                            const userResult = {
                                                id: userUpdated.id,
                                                login: userUpdated.login,
                                                name: userUpdated.name,
                                                image: image
                                            }

                                            // Получить все chat_id по user_id=user.id
из таблицы UserChat
                                            const selectChatIdsQuery = 'SELECT chat_id
FROM UserChat WHERE user_id = ?';
                                            db.all(selectChatIdsQuery, [id], (err,
rows) => {
                                                if (err) {
                                                    console.error(err);
                                                } else {
                                                    // Для каждого chat_id выполнить
io.to(chat_id).emit('updateUserImage', image);
                                                    rows.forEach((row) => {
                                                        const chatId = row.chat_id;
                                                        console.log('Отправляю имедж в
комнату ' + chatId)

io.to(chatId).emit('updateUserImage', userResult);
                                                    });
                                                }
                                            });
                                        }
                                    })
                                }
                            })
                        }
                    }
```

```
                })
            }
        })
    })*/

    socket.on('updateUserName', (message) => {
        const {name, token} = message
        console.log('Выполняю запрос на апдейт имедж')
        jwt.verify(token, SECRET_KEY, (err, decoded) => {
            if(!err){
                const login = decoded.login;
                const updateQuery = `UPDATE user SET name = ? WHERE login = ?`;
                db.run(updateQuery, [name, login], (err) => {
                    if (err) {
                        console.error(err);
                    } else {
                        const selectUserQuery = 'SELECT id FROM user WHERE login = ?';
                        db.get(selectUserQuery, [login], (err, user) => {
                            if (err) {
                                console.error(err);
                            } else {
                                if (user) {
                                    const userId = user.id;

                                    const selectUserQuery = 'SELECT * FROM user WHERE
login = ?'
                                    db.get(selectUserQuery, [login], (err,userUpdated)
=> {
                                        if (err) {
                                            console.error(err);
                                        } else {
                                            const userResult = {
                                                id: userUpdated.id,
                                                login: userUpdated.login,
                                                name: userUpdated.name,
                                                image: userUpdated.image
                                            }

                                            // Получить все chat_id по user_id=user.id
из таблицы UserChat
                                            const selectChatIdsQuery = 'SELECT chat_id
FROM UserChat WHERE user_id = ?';
                                            db.all(selectChatIdsQuery, [userId], (err,
rows) => {
                                                if (err) {
                                                    console.error(err);
                                                } else {
                                                    // Для каждого chat_id выполнить
io.to(chat_id).emit('updateUserImage', image);
                                                    rows.forEach((row) => {
                                                        const chatId = row.chat_id;
                                                        console.log('Отправляю name в
комнату ' + chatId)

io.to(chatId).emit('updateUserName', userResult);
                                                    });
                                                }
                                            });
                                        }
                                    })
                                }
                            }
```

```
                    });
                }
            })
        }
    })
})

    socket.on('createChat', (data) => {
        const {link, name, description, token} = data
        const image = req.body.image == '' ?
'https://static.vecteezy.com/system/resources/previews/005/337/802/original/icon-
symbol-chat-outline-illustration-free-vector.jpg'
                                    : req.body.image

        jwt.verify(token, SECRET_KEY, (err, decoded) => {
            if (err) {console.log(err)} else {
                const login = decoded.login

                const getUserIdQuery = `SELECT id FROM user WHERE login = ?`;
                db.get(getUserIdQuery, [login], (err, row) => {
                    if (err) {console.log(err)} else {
                        const userId = row.id;

                        const checkLinkQuery = 'SELECT * FROM chat WHERE link = ?';
                        db.get(checkLinkQuery, [link], (err, row) => {
                            if (err) {console.log(err)} else {
                                if (row) {
                                    // Значение link уже существует в таблице
                                    res.status(400).json();
                                    socket.emit('createChatAnswer', { success: false,
message: 'Link already exists' })
                                } else {
                                    const addChatQuery = `
                                        INSERT INTO chat (link, name, image,
description)
                                        VALUES (?, ?, ?, ?)
                                    `;
                                    db.run(addChatQuery, [link, name, image,
description], function (err) {
                                        if (err) {console.log(err)} else {
                                            const chatId = this.lastID;

                                            const addUserToChatQuery = `
                                                INSERT INTO UserChat (chat_id, user_id)
VALUES (?, ?)
                                            `;
                                            db.run(addUserToChatQuery, [chatId,
userId], function (err) {
                                                if (err) {console.log(err)} else {
                                                    const chatWasCreatedMessage = `
                                                        INSERT INTO message (content,
chat_id, sender_id, sent_at)
                                                        VALUES ('Chat was created', ?,
5, datetime('now'))
                                                    `
                                                    db.run(chatWasCreatedMessage,
[chatId], function (err) {
                                                        if (err) {console.log(err)}
else {
                                                            const messageId =
this.lastID;
```

```
                                                                const setMessageId = `
                                                                    UPDATE chat SET
last_message_id = ?
                                                                    WHERE chat.link = ?
                                                                `
                                                                db.run(setMessageId,
[messageId, link], (err) => {
                                                                    if (err)
{console.log(err)} else {

socket.emit('createChatAnswer', { success: true, message: 'Chat create is successful'
})

                                                                        socket.join(chatId)
                                                                        console.log(`Client
${socket.id} - join to room ${id}`)


                                                                        const
createdChatQuery = `
                                                                        SELECT
                                                                            c.id AS
chat_id,
                                                                            c.link,
                                                                            c.name AS
chatname,
                                                                            c.image,
                                                                            m.id AS
last_message_id,
                                                                            m.content,
                                                                            m.sender_id,
                                                                            m.chat_id,
                                                                            m.sent_at,
                                                                            u.name AS
username
                                                                        FROM
                                                                            chat AS c
                                                                            LEFT JOIN
message AS m ON c.last_message_id = m.id
                                                                            JOIN UserChat
AS uc ON c.id = uc.chat_id
                                                                            JOIN user AS u
ON m.sender_id = u.id
                                                                        WHERE
                                                                            c.id = ?
                                                                        `;




db.get(createdChatQuery, [chatId], (err, row) => {
                                                                            if (err)
{console.log(err)} else {
                                                                                const chat
=  {
                                                                                    id:
row.chat_id,
                                                                                    link:
row.link,
                                                                                    name:
row.chatname,
                                                                                    image:
row.image,
```

```
lastMessage: {
                                                                                id:
row.last_message_id,
content: row.content,
sender_id: row.sender_id,
chat_id: row.chat_id,
sent_at: row.sent_at,
user: {
name: row.username
                                                                                }
                                                                            }
                                                                        };


socket.emit('createdChat', chat)
                                                                                }
                                                                            });
                                                                        }
                                                                    })
                                                                }
                                                            })
                                                        }
                                                    })
                                                }
                                            }
                                        })
                                    }
                                })
                            }
                        })
                    }
                })

    socket.on('createUserChat', (data) => {
        const { login, chat_id } = data

        const getUserQuery = `SELECT id FROM "user" WHERE login = ?`;
        db.get(getUserQuery, [login], (err, row) => {
            if (err) {
                console.error(err);
            } else if (!row) {
                console.log(`User with login ${login} not found`);
            } else {
                const user_id = row.id;
                console.log('Added user id: '+ user_id)

                const insertUserChatQuery = `INSERT INTO UserChat (user_id, chat_id)
VALUES (?, ?)`;
                db.run(insertUserChatQuery, [user_id, chat_id], function(err) {
                    if (err) {
                        console.error(err);
                    } else {
                        const userIsJoinToChat = `
                        INSERT INTO message (content, chat_id, sender_id, sent_at)
```

```
                        VALUES (?, ?, 5, datetime('now'))
                        `
                        const content = '#' + login + ' join to chat';
                        console.log(content)

                        db.run(userIsJoinToChat, [content, chat_id], function (err) {
                            if (err) {console.error(err)} else {
                                const messageId = this.lastID;
                                console.log('Hello Message id: ', messageId)
                                const setMessageId = `
                                    UPDATE chat SET last_message_id = ?
                                    WHERE chat.id = ?
                                `
                                db.run(setMessageId, [messageId, chat_id], (err) => {
                                    if (err) {console.error(err)} else {
                                        const createdChatQuery = `
                                        SELECT
                                            c.id AS chat_id,
                                            c.link,
                                            c.name AS chatname,
                                            c.image,
                                            m.id AS last_message_id,
                                            m.content,
                                            m.sender_id,
                                            m.chat_id,
                                            m.sent_at,
                                            u.name AS username,
                                            u.login,
                                            u.image AS logo
                                        FROM
                                            chat AS c
                                            LEFT JOIN message AS m ON c.last_message_id
= m.id
                                            JOIN UserChat AS uc ON c.id = uc.chat_id
                                            JOIN user AS u ON m.sender_id = u.id
                                        WHERE
                                            c.id = ?
                                        `;


                                        db.get(createdChatQuery, [chat_id], (err, row)
=> {
                                            if (err) {console.error(err)} else {
                                                const chat =  {
                                                    id: row.chat_id,
                                                    link: row.link,
                                                    name: row.chatname,
                                                    image: row.image,
                                                    lastMessage: {
                                                        id: row.last_message_id,
                                                        content: row.content,
                                                        sender_id: row.sender_id,
                                                        chat_id: row.chat_id,
                                                        sent_at: row.sent_at,
                                                        user: {
                                                            name: row.username,
                                                            login: row.login,
                                                            image: row.logo
                                                        }
                                                    }
                                                };
```

```
                                                    console.log('chatdata: '+
chat.lastMessage.content)

                                                    io.to(chat_id).emit('createdUserChat',
{chatFromData: chat, userIdFromData: user_id})
                                                    clients.forEach((client) => {
                                                        if (client.userId === user_id) {
                                                            console.log('Юзер ' +
client.socketId + ' получил приглашение на вступление в чат ' + chat_id)

io.to(client.socketId).emit('createdUserChat', {chatFromData: chat, userIdFromData:
user_id})
                                                        }
                                                    })
                                                }
                                            });
                                        }
                                    )
                                }
                            )
                        }
                    );
                }
            );
        }
    )

    socket.on('deleteUserChat', (data) => {
        const { chat_id, token } = data

        jwt.verify(token, SECRET_KEY, (err, decoded) => {
            if (err)
                console.error(err)
            else {
                const userLogin = decoded.login;

                const getUserIdQuery = `SELECT id FROM user WHERE login = ?`;
                db.get(getUserIdQuery, [userLogin], (err, row) => {
                    if (err) {
                        console.error(err);
                    } else {
                        const user_id = row.id;

                        const deleteUserChatQuery = `DELETE FROM UserChat WHERE chat_id
= ? AND user_id = ?`;
                        db.run(deleteUserChatQuery, [chat_id, user_id], function(err) {
                            if (err) {
                                console.error(err);
                            } else {

                                const userIsJoinToChat = `
                                INSERT INTO message (content, chat_id, sender_id,
sent_at)
                                VALUES (?, ?, 5, datetime('now'))
                                `
                                const content = '#' + userLogin + ' leaves to chat';
                                console.log(content)

                                db.run(userIsJoinToChat, [content, chat_id], function
(err) {
                                    if (err) {console.error(err)} else {
                                        const messageId = this.lastID;
```

```
                                    console.log('Hello Message id: ', messageId)
                                    const setMessageId = `
                                        UPDATE chat SET last_message_id = ?
                                        WHERE chat.id = ?
                                    `
                                    db.run(setMessageId, [messageId, chat_id],
(err) => {
                                        if (err) {console.error(err)} else {
                                            const createdChatQuery = `
                                            SELECT
                                                c.id AS chat_id,
                                                c.link,
                                                c.name AS chatname,
                                                c.image,
                                                m.id AS last_message_id,
                                                m.content,
                                                m.sender_id,
                                                m.chat_id,
                                                m.sent_at,
                                                u.name AS username,
                                                u.login,
                                                u.image AS logo
                                            FROM
                                                chat AS c
                                                LEFT JOIN message AS m ON
c.last_message_id = m.id
                                                JOIN UserChat AS uc ON c.id =
uc.chat_id
                                                JOIN user AS u ON m.sender_id =
u.id
                                            WHERE
                                                c.id = ?
                                            `;
                                            db.get(createdChatQuery, [chat_id],
(err, row) => {
                                                if (err) {console.error(err)} else
{
                                                    const chat =  {
                                                        id: row.chat_id,
                                                        link: row.link,
                                                        name: row.chatname,
                                                        image: row.image,
                                                        lastMessage: {
                                                            id:
row.last_message_id,
                                                            content: row.content,
                                                            sender_id:
row.sender_id,
                                                            chat_id: row.chat_id,
                                                            sent_at: row.sent_at,
                                                            user: {
                                                                name: row.username,
                                                                login: row.login,
                                                                image: row.logo
                                                            }
                                                        }
                                                    };
                                                    console.log('chatdata: '+
chat.lastMessage.content)
```

```
io.to(chat_id).emit('deletedUserChat', {chatFromData: chat, userIdFromData: user_id});
                                                        }
                                                    });
                                                }
                                            })
                                        }
                                    })
                                }
                            });
                        }
                    });
                }
            });
    })
})

process.on('SIGINT', () => {
    db.close((error) => {
      if (error) {
        console.error(error.message);
      } else {
        console.log('Database connection closed.');
      }
      process.exit(0);
    });
});


navigation/index.js:
import React, { useContext, useEffect } from 'react'
import { createNativeStackNavigator } from '@react-navigation/native-stack'
import { NavigationContainer } from '@react-navigation/native';
import LoginScreen from '../screens/LoginScreen';
import RegisterScreen from '../screens/RegisterScreen';
import { AuthContext } from '../../AuthContext';
import ChatListScreen from '../screens/ChatListScreen';
import axios from 'axios';
import { SERVER_URL } from '@env'
import * as SecureStore from 'expo-secure-store';
import MessageListScreen from '../screens/MessageListScreen';
import ChatMenuScreen from '../screens/ChatMenuScreen';
import UserSettingScreen from '../screens/UserSettingScreen';
import CreateChatScreen from '../screens/CreateChatScreen';

const Stack = createNativeStackNavigator();

const Navigator = () => {

    const {isAuth, setIsAuth, myLogin, setMyLogin} = useContext(AuthContext)

    useEffect(() => {
        //Перевіряємо токен при старті застосунку
        const checkToken = async () => {
            try {
                const token = await SecureStore.getItemAsync('token');

                if (token) {
                    const response = await axios.post(`${SERVER_URL}/check-token`, {
token });

                    const { success, login } = response.data;
```

```
                    setIsAuth(success);

                    if(success) {
                        setMyLogin(login)
                    }
                } else {
                    setIsAuth(false);
                }
            } catch (error) {
                console.error('no token bad token '+error);
            }
        };

        checkToken();
    }, []);

    return (
        <NavigationContainer>
        {isAuth ? (
            <Stack.Navigator screenOptions={{ headerTitleAlign: 'center' }}>
                {/* <Stack.Screen name='Chat Menu' component={ChatMenuScreen} /> */}
                <Stack.Screen name='Chat List' component={ChatListScreen}/>
                <Stack.Screen name='Message List' component={MessageListScreen}/>
                <Stack.Screen name='User Setting' component={UserSettingScreen}/>
                <Stack.Screen name='Create Chat' component={CreateChatScreen}/>
                <Stack.Screen name='Chat Menu' component={ChatMenuScreen}/>
            </Stack.Navigator>
        ) : (
            <Stack.Navigator>
                <Stack.Screen name='Login' component={LoginScreen} options={{
headerShown: false }}/>
                <Stack.Screen name='Register' component={RegisterScreen} options={{
headerShown: false }}/>
            </Stack.Navigator>
        )}
        </NavigationContainer>
    )
}

export default Navigator


LoginScreen.js:
import { KeyboardAvoidingView, StyleSheet, Text, TextInput, TouchableOpacity, View,
Image, Platform } from 'react-native'
import React, { useContext, useState } from 'react'
import { useNavigation } from '@react-navigation/native'
import { AuthContext } from '../../AuthContext';
import axios from 'axios';
import { SERVER_URL } from '@env'
import * as SecureStore from 'expo-secure-store';

const LoginScreen = () => {
    const navigation = useNavigation();

    const {isAuth, setIsAuth, myLogin, setMyLogin} = useContext(AuthContext)

    const [login, setLogin] = useState('')
    const [password, setPassword] = useState('')

    const [errorLogin, setErrorLogin] = useState(false)
```

```
    const [errorPassword, setErrorPassword] = useState(false)

    const handleLoginChange = (text) => {
        setLogin(text);
        if (text.length < 5 || text.length > 16 || !/^[a-zA-Z0-9]+$/.test(text)) {
            setErrorLogin(true);
        } else {
            setErrorLogin(false);
        }
    };

    const handlePasswordChange = (text) => {
        setPassword(text);
        setErrorPassword(text.length < 8);
    };

    const tryAuth = () => {
        if(!errorLogin && !errorPassword &&
            login.length != 0 && password.length != 0) {
            const loginData = {
                login: login,
                password: password,
            };

            //Аутентифікація на сервері
            axios.post(`${SERVER_URL}/login`, loginData)
            .then(response => {
                const { success, message, token, login } = response.data;
                if (success) {
                    SecureStore.setItemAsync('token', token)
                    .then(() => {
                        setIsAuth(true)
                        setMyLogin(login)
                        console.log('Token:', token);

                    })
                    .then(() => {
                        console.log('Login: ', myLogin)
                    })
                    .catch(error => {
                        console.log('SecureStore error:', error.message);
                    });
                }
            })
            .catch(error => {
                if (error.response.data.message === 'Login error: wrong login') {
                    setErrorLogin(true)
                }
                else if (error.response.data.message === 'Login error: wrong password')
{
                    setErrorPassword(true)
                }
                else {
                  console.error(error);
                }
            })
        }
    };

    return (
        <KeyboardAvoidingView
            style={styles.container}
```

```
            behavior={Platform.OS == 'ios' ? 'padding' : 'height'}
        >
            <View style={styles.logoImageContainer}>
                <Image
                    source={require('../../assets/logo.png')}
                    style={styles.logoImage}
                />
            </View>

            <View style={styles.inputContainer}>
                <TextInput
                    placeholder='Login'
                    value={ login }
                    onChangeText={text => handleLoginChange(text)}
                    style={[styles.input, errorLogin ? {borderWidth: 2, borderColor:
'salmon'} : null]}
                />
                <TextInput
                    placeholder='Password'
                    value={ password }
                    onChangeText={text => handlePasswordChange(text)}
                    style={[styles.input, errorPassword ? {borderWidth: 2, borderColor:
'salmon'} : null]}
                    secureTextEntry
                />
            </View>

            <View style={styles.buttonContainer}>
                <TouchableOpacity
                    onPress={tryAuth}
                    style={styles.button}
                >
                    <Text style={styles.buttonText}>Login</Text>
                </TouchableOpacity>
                <TouchableOpacity
                    onPress={() => {navigation.navigate('Register')}}
                    style={[styles.button, styles.buttonOutline]}
                >
                    <Text style={styles.buttonOutlineText}>Register</Text>
                </TouchableOpacity>
            </View>
        </KeyboardAvoidingView>
    )
}

export default LoginScreen

const styles = StyleSheet.create({
    container:{
        flex: 1,
        justifyContent: 'center',
        alignItems: 'center',
    },
    logoImageContainer:{
        width: '80%',
    },
    logoImage:{
        width: "100%",
        resizeMode: 'contain',
    },
    inputContainer:{
        marginTop: 40,
```

```
        width: '80%'
    },
    input:{
        backgroundColor: 'white',
        paddingHorizontal: 15,
        paddingVertical: 10,
        borderRadius: 10,
        marginTop: 5
    },
    buttonContainer:{
        marginTop: 40,
        width: '60%',
        justifyContent: 'center',
        alignItems: 'center',
    },
    button:{
        width: '100%',
        backgroundColor: '#808080',
        padding: 15,
        borderRadius: 10,
        borderWidth: 2,
        borderColor: '#3c3c3c',
        alignItems: 'center',
    },
    buttonOutline:{
        marginTop: 5,
        backgroundColor: 'white',
    },
    buttonText:{
        color: 'white',
        fontWeight: 'bold',
        fontSize: 16,
    },
    buttonOutlineText:{
        color: 'black',
        fontWeight: 'bold',
        fontSize: 16,
    }
})
```

ДОДАТОК Б

ВІДГУК КЕРІВНИКА ЕКОНОМІЧНОГО РОЗДІЛУ

ДОДАТОК В

ПЕРЕЛІК ДОКУМЕНТІВ НА ОПТИЧНОМУ НОСІЇ

Ім’я файлу	**Опис**
Пояснювальні документи	
Диплом_ Лохвицький.docx	Пояснювальна записка до дипломного проекту. Документ Word.
Диплом_ Лохвицький.pdf	Пояснювальна записка до дипломного проекту в форматі PDF
Програма	
messenger.rar	Архів. Містить коди програми і відкомпільовану програму
Презентація	
Презентація_Лохвицький.ppt	Презентація дипломного проекту