НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

Дипломний проєкт

на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: Мобільний додаток на iOS для миттєвого обміну повідомленнями

Виконав : студент __4__ курсу, групи ІП-84
(шифр групи)

_____ Кришталь Дмитро Вікторович_____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____ асистент Пономаренко А. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) професор, д.т.н., Сімоненко В.П._____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проєкті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2022 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)
“__” _____2022 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Кришталя Дмитра Вікторович

1. Тема проєкту *Мобільний додаток на iOS для миттєвого обміну повідомленнями*
керівник проєкту Пономаренко Артем Миколайович, асистент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 11 травня 2022 року №1139-с

2. Термін здачі студентом закінченого проєкту *24 червня 2022 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні та статистичні дані*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Аналіз предметної області.

Розділ 2. Засоби для реалізації.

Розділ 3. Деталі розробки системи.

Розділ 4. Аналіз розробленого додатку.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень)
структурна схема системи, функціональна схема, принципова схема роботи

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Сімоненко В. П.		

7. Дата видачі завдання «25» грудня 2021 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*20.12.2021-30.12.2021*	
2.	*Вивчення та аналіз завдання*	*10.01.2022-30.03.2022*	
3.	*Розробка архітектури та загальної структури системи*	*30.03.2022-15.04.2022*	
4.	*Програмна реалізація системи*	*15.04.2022-30.04.2022*	
5.	*Виправлення помилок*	*30.04.2022-05.05.2022*	
6.	*Оформлення пояснювальної записки*	*05.05.2022-10.06.2022*	
7.	*Захист програмного продукту*	*27.05.2022*	
8.	*Передзахист*	*11.06.2022*	
9.	*Захист*	*24.06.2022*	

Студент-дипломник _____ Дмитро КРИШТАЛЬ
(підпис)

Керівник проєкту _____ Артем ПОНОМАРЕНКО
(підпис)

АНОТАЦІЯ

У даній бакалаврській роботі було розроблено мобільний додаток для миттєвого обміну повідомленнями для операційної системи iOS. Його реалізація складалася зі створення клаєнтської та сервеної частин.

Для розробки серверної частин використовувався NodeJS. Для завантаження старих даних з сервера було розроблено REST API. А функціонал додатку, який повинен відбуватись у реальному часі було реалізовано за допомогою бібліотеки Socket.IO

Для розробки клієнтської частини використовувалась мова програмування Swift та фреймворк SwiftUI. Як результат я отримав мобільний додаток за допомогою, якого можна додавати друзів та спілкуватися з ними у реальному часі.

Ключові слова: месенджер, Swift, JavaScript, NodeJS.

ANNOTATION

The bachelor’s thesis is described as a development of an instant messaging mobile application for iOS. The implementation is divided into two parts: development of the client side and development of the server side.

The server was developed using NodeJS. REST API was implemented in order to load old data from the database without pushing Socket.IO to its limit. All the communication that is supposed to happen between users in real time is implemented using Socket.IO, which is a library for NodeJS.

The client side was developed using Swift programming language and SwiftUI framework. As a result, I was able to create an application which allows users to add friends and start chatting with them.

Key words: messengers, Swift, JavaScript, NodeJS.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Мобільний додаток на iOS	3		
			для миттєвого обміну			
			повідомленням			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Мобільний додаток на iOS	75		
			для миттєвого обміну			
			повідомленнями			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Мобільний додаток на iOS	1		
			для миттєвого обміну			
			повідомленням			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Мобільний додаток на iOS	1		
			для миттєвого обміну			
			повідомленням			
			Функціональна схема			
	А4	***ІАЛЦ.467200.006 Д3***	Мобільний додаток на iOS	1		
			для миттєвого обміну			
			повідомленням			
			Принципова схема			
	А4	***ІАЛЦ.467200.007 Д4***	Мобільний додаток на iOS	44		
			для миттєвого обміну			
			повідомленням			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА			
Зм	***Лист***	***№ докум.***	***Підп***	***Дата***				
Розроб		Кришталь Д.В.			*Мобільний додаток на iOS для миттєвого обміну повідомленнями* Опис альбому	Літ.	Аркуш	Аркушів
Перев		Пономаренко А.М.					1	1
						НТУУ "КПІ" ФІОТ ІП-84		

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Мобільний додаток на iOS для миттєвого обміну повідомленнями*»

Київ – 2022

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Найменування: Мобільний додаток на iOS для миттєвого обміну повідомленнями.

Дане технічне завдання поширюється на розробку мобільного додатку на iOS для миттєвого обміну повідомленнями, а також на подальшу підтримку та вдосконалення розробленої системи.

Областю застосування є використання системи у некомерційних цілях.

2 ПІДСТАВИ ДО РОЗРОБКИ

Підставою для розробки даної системи є завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є розробка мобільного додатку на iOS для миттєвого обміну повідомленням з використанням найсучасніших технологій.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даної системи є публікації та статті у мережі інтернет.

5 ТЕХНІЧНІ ВИМОГИ

5.1 Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Максимально простий інтерфейс клієнтської частини системи, що дозволить максимально швидко розпочати роботу з нею.
- Можливість комунікувати у реальному часі.
- Максимально швидка робота серверної частини.
- Створити зрозумілу документацію для розробленого продукту.

5.2 Вимоги до програмного забезпечення

- ОС Mac.
- XCode 12 або вище.
- Visual Studio Code 2018 або вище.
- NodeJS 16.14.0 або вище.

5.3 Вимоги до апаратної частини

- ЦП не менше ніж Intel® Core (TM) i5- 4278U
- ROM не менше ніж 128 ГБ.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	20.12.2021-30.12.2021
Вивчення та аналіз завдання	10.01.2022-30.03.2022
Розробка архітектури та загальної структури системи	30.03.2022-15.04.2022
Програмна реалізація системи	15.04.2022-30.04.2022
Виправленя помилок	30.04.2022-05.05.2022
Оформлення пояснювальної записки	*05.05.2022-10.06.2022*

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Мобільний додаток на iOS для миттєвого обміну повідомленнями*»

Київ – 2022

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

ОС	Операційна система
ПЗ	Програмне забезпечення
MVVM	Model-View-ViewModel
JS	JavaScript
HTTP	Hypertext Transfer Protocol
IDE	Інтегроване середовище розробки
API	Прикладний програмний інтерфейс

ВСТУП

З моменту їх появи у кінці двадцятого сторіччя мобільні телефони пройшли довгий шлях та змінилися до невпізнання. Від продукту, який мала лише невелика кількість населення планети, вони доросли до продукту, без якого ми більше не можемо уявити нашого життя. Упродовж усього часу їх існування вони постійно розвивалися: збільшувалась кількість функціонала, зменшувались розміри тощо.

Раніше мобільні телефони мали обмежений функціонал через відсутність можливості власноруч встановлювати бажані додатки, оскільки не було доступу до інтернет-з'єднання та потужність апаратної частини була дуже низькою.

Сьогодні, навпаки, навіть найдешевший смартфон пропонує майже необмежений функціонал через можливість встановлювати додатки з вбудованих магазинів, а також, за бажанням, можливості власноруч розробляти їх..

Функціонал, який пропонують сучасні мобільні додатки є дуже широким. Від звичайних додатків для редагування тексту до надпотужних графічних редакторів.

Метою даної роботи є розробка власного додатка для обміну повідомленнями на операційній системі iOS.

Дана робота буде складатись з аналізу предметної області, вибору засобів для розробки, розробки системи та аналізу розробленої системи, перевірки її роботи.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Загальні відомості про мобільні додатки

Мобільний додаток[1] – тип програмного забезпечення, яке призначене для роботи на мобільних пристроях. Їхня основна ціль – забезпечення користувачів функціоналом, який доступний на персональних комп'ютерах, але з можливістю його використання у будь-який момент.

Загалом мобільні додатки розділяють на декілька видів:

- Ігрові додатки – мобільні програми, які створені, щоб замінити традиційні відеоігри для консолей та комп'ютерів. Сучасні смартфони настільки потужні, що дозволяють створювати ігри, які раніше можна було грати на потужних гаджетах.
- Lifestyle додатки – мобільні програми, що відіграють роль у особистому житті користувачів, наприклад месенджери, додатки для заняття спортом тощо.
- Productivity додатки – мобільні програми, що допомагають користувачам виконувати певні щоденні завдання.

За технологією розробки мобільні додатку розділяються на декілька типів: нативні, веб та гібридні.

1.1.1 Нативні мобільні додатки

Нативні мобільні додатки – програмне забезпечення, яке створене для роботи на окремій операційній системі. Сьогодні ринок майже повністю зайнятий двома операційними системами: iOS та Android. Для розробки їх нативного програмного забезпечення використовуються мови програмування Swift та Kotlin відповідно[2].

Дана технологія гарно підходить абсолютно для всіх типів додатків, якщо замовник має намір працювати лише з однією операційною системою або має фінансову можливість для створення програм на обох системах.

1.1.2 Веб додатки

Веб додатки – тип програмного забезпечення, яке працює на мобільних пристроях у якості вебсторінки у браузері. На відміну від нативних додатків, їх не потрібно завантажувати з магазину та встановлювати. Для роботи на смартфоні повинен бути лише браузер. Для розробки використовуються ті ж засоби, що і для звичайних сайтів: JavaScript, HTML та CSS [3].

Використання даної технології ідеально підходить для додатків, які побудовані лише для роботи онлайн, наприклад, інтернет-магазини. Якщо ж програма має функціонал, який повинен працювати офлайн, слід обрати іншу технологію.

1.1.3 Гібридні мобільні додатки

Гібридні мобільні додатки – програмне забезпечення, яке поєднує в собі переваги нативних і веб додатків. На перший погляд, користувач може подумати, що він працює з нативним додатком. Він навіть встановлюється з магазину та може мати офлайн функціонал. Але насправді гібридні додатки використовують певні контейнери, які завантажують сторінки. Також, як і в нативних додатках можливе використання апаратного забезпечення: камери, мікрофону тощо [4].

Вони зберігають усі переваги вебдодатку, але обернене в певну обгортку, що дозволяє йому виглядати як нативний додаток. Водночас зберігаються деякі недоліки, які мають веб додатки, а саме постійна залежність від інтернет-з’єднання.

1.1.4 Порівняння видів мобільних додатків

Таблиця 1.1 – Переваги та недоліки різних видів мобільних додатків.

Вид	Переваги	Недоліки
Нативні	1. Швидкість роботи є суттєво вищою. 2. Можливість створювати юзер інтерфейс, який буде максимально схожий на інтерфейс операційної системи. 3. Швидкість роботи є суттєво вищою. 4. Можливість створювати юзер інтерфейс, який буде максимально схожий на інтерфейс операційної системи. 5. Можливість використовувати весь функціонал смартфону. 6. Наявність офлайн функціоналу	1. Витрати на розробку, які суттєво збільшуються, якщо замовник бажає охопити аудиторію обох операційних систем. 2. Витрати на розробку, які суттєво збільшуються, якщо замовник бажає охопити аудиторію обох операційних систем.
Веб	1. Дані додатки можуть працювати майже на всіх девайсах, через відсутність вимог до операційної системи. 2. Набагато нижча ціна розробки. 3. Легка підтримка додатку, оскільки він має однаковий код для роботи на всіх системах.	1. Низька швидкість роботи в порівнянні з додатками, які розроблені за нативною технологією. 2. Відсутність можливості працювати у додатку офлайн.

Продовження таблиці 1.1

Вид	Переваги	Недоліки
Гібридні	1. Відсутність необхідності використовувати браузер.	1. Залежність від інтернет з’єднання.

Кінець таблиці 1.1

1.2 Аналіз клієнт-серверної архітектури

Клієнт-серверна архітектура – комп’ютерна модель, у якій декілька компонентів системи виконують чітко визначені розробниками їм ролі. З її допомогою користувачі можуть отримувати та відсилати дані через інтернет з’єднання[5]. Вона складається з двох основних компонентів:

- Клієнт – компонент, який виступає стороною, що дозволяє користувачу отримувати інформацію з сервера. В його якості може виступати будь-яка програма, з якою він взаємодіє. Залежно від дій користувача до сервера надсилаються відповідні запити, на які той повинен дати якусь відповідь. У системі може буде одночасно безліч клієнтів, які взаємодіють з одним і тим самим сервером.
- Сервер – компонент, який повинен обробляти запити, що надсилає клієнт. У його якості виступають зазвичай дуже потужні комп’ютери, які можуть з високою інтенсивністю справлятись з отриманими запитами.

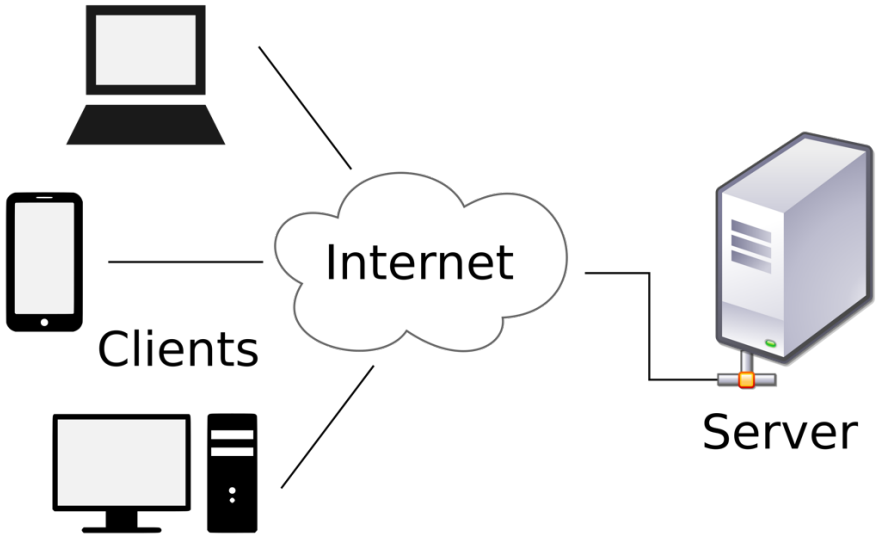


Рисунок 1.1 Приклад клієнт-серверної архітектури

Оскільки дана архітектура є централізованою, вона створює декілька очевидних переваг:

- Можливість легко контролювати усі процеси, що відбуваються на сервері.
- Дана архітектура є масштабованою. Кількість як клієнтів, так і серверів може постійно збільшуватись.

Дана архітектура має один суттєвий недолік. У випадку миттєвої зупинки сервера відбувається відмова усієї системи. Для уникнення такої проблеми будують системи з серверними кластерами, які складаються з мінімум двох серверів. Завдяки такому підходу побудови системи нівелюється можливість повної зупинки системи через проблеми у роботі окремого серверу. Також він допомагає пришвидшити роботу системи з великою кількістю користувачів через рівномірний розподіл навантаження завдяки балансувальнику, який спочатку отримує запит від клієнта та віддає серверу з найменшим навантаженням.

1.3 Характеристика мобільних операційних системи

1.3.1 Порівняння операційних систем

Попри велику кількість мобільних операційних систем у сучасному світі, майже весь ринок зайнятий двома. Більшу частину ринку займає Android, розробкою якого займається компанія Google та перша версія якого була випущена у 2008 році. Іншу частину займає компанія Apple з операційною системою iOS, яка з'явилася на 1 рік раніше.

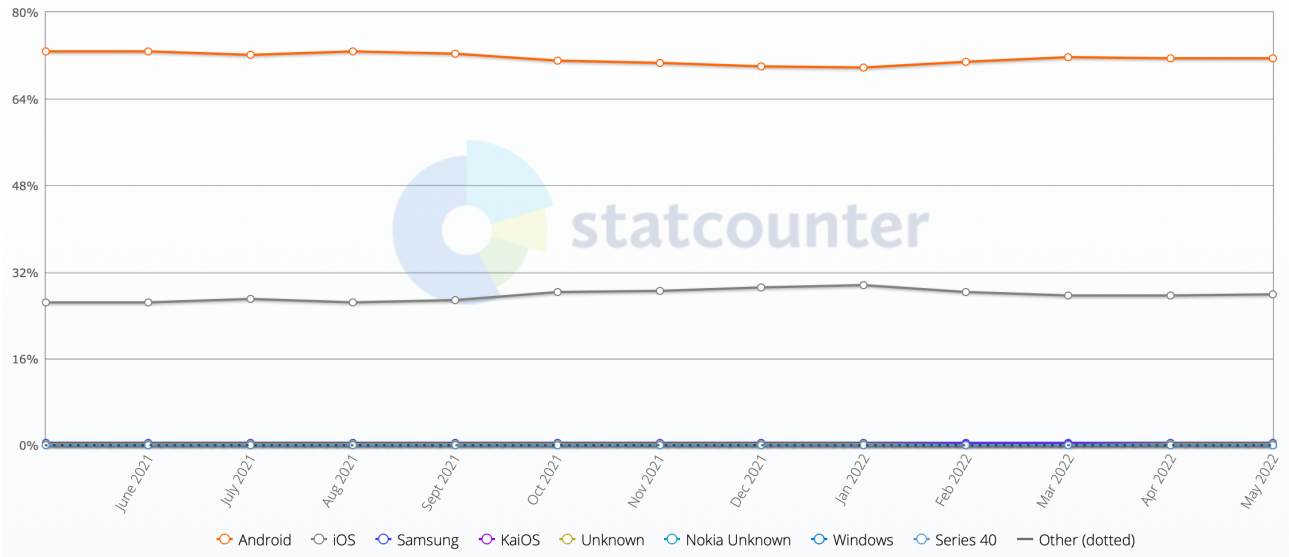


Рисунок 1.2 Частка ринку операційних систем[7]

Системи мають одну фундаментальну відмінність, яка робить їх дуже різними та змушує потенціальних користувачів робити важкий вибір при покупці смартфона.

З одного боку є iOS – закрита ОС, яка використовується лише на смартфонах від компанії Apple. Вона є набагато більш захищеною ніж Android, тому краще обирати цю систему, якщо в смартфоні зберігається надважлива персональна інформація. Для встановлення додаткового ПЗ дозволене використання лише вбудованого магазину App Store. Це нівелює незаконне розповсюдження ПЗ та забезпечує повний захист від шкідливого ПЗ.

З іншого боку є Android – відкрита ОС. Вона може встановлюватись на смартфони будь-яких виробників. Завдяки відкритому коду інтерфейс даної ОС може кардинально змінюватись від смартфонів одного виробника до іншого, що є неодмінно плюсом. Встановлення додаткового ПЗ може відбуватись будь-яким способом: з вбудованого магазину Google Play, сторонніми магазинами та завантаженими .apk файлами. Це робить систему вразливою до шкідливого ПЗ та збільшує кількість незаконного розповсюдження ПЗ [6].

Серед явних переваг iOS можна виділити довгу підтримку нових смартфонів: компанія Apple продовжує випускати оновлення навіть через 5-7 років після виходу гаджета. Наприклад, iPhone 7, який був випущений у 2016 році, дотепер отримує усі програмні оновлення. Від нових смартфонів він буде відрізнятись лише функціоналом, який обмежений апаратною частиною. Така довга підтримка дозволена завдяки максимально ефективній роботі апаратної частини з програмним забезпеченням, бо всі елементи розробляються інженерами компанії Apple.

1.3.2 Порівняння розробки програмного забезпечення

Обидві операційні системи відрізняються в плані розробки нативного програмного забезпечення.

Загалом створення програм для iOS вважається простішим та менш трудомістким, але вимагає знання мови Swift, яка вкрай рідко використовується для інших задач. Android, з іншого боку, потребує знання Kotlin, яка дуже схожа на Java.

При розробці на Android необхідно враховувати дуже велику кількість різних розмірів і співвідношень сторін екрана, що може створювати проблеми з сумісністю на окремих девайсах. На смартфонах від Apple використовуються екрани з двома різними співвідношеннями сторін екрана для деяких груп моделей: старіші моделі використовують 16:9, а більш сучасні 19.5:9. Також

вони мають певну вибірку розмірів, що забезпечують легку підтримку усіх девайсів.

Ще однією перевагою для вибору розробки на iOS є прибуток, який розробник отримує від магазину додатків. За 2020 загальний прибуток від App Store склав 72 мільярди доларів, а від Google Play Store близько 40 мільярдів доларів. Це показує, чому для розробника доцільніше спершу розробляти додаток під iOS[8].

1.4 Опис додатку для миттєвого обміну повідомленнями

Месенджер – додаток, що створений для миттєвого обміну повідомленнями, та файлами у деяких реалізаціях, через інтернет з'єднання. Цей спосіб посилання повідомлень використовується найчастіше та прийшов на заміну SMS повідомленням на початку 2000-х років. З моменту появи месенджерів їхня популярність значно виросла та сьогодні загальна кількість облікових засобів у всіх додатках цього типу сягає понад 7 мільярдів.

Основний функціонал месенджерів реалізується за допомогою кімнат, які можуть бути як приватними, де лише дві особи, так і груповими, де має бути не менше трьох осіб. Сьогодні функціонал месенджерів сильно виходить за рамки того, що було задумано спочатку. Крім можливості відправляти звичайні текстові повідомлення, ми також можемо обмінюватись файлами, здійснювати відео- та аудіодзвінки, створювати блоги тощо.

Месенджери створені не лише для швидкої персональної комунікації. Вони також суттєво пришвидшують продуктивність роботи у професіях, які потребують максимально швидкого і безпечного обміну даними та інформацією. Причиною цьому є шифрування повідомлень, яке може бути розшифроване лише співрозмовниками, навіть сам месенджер не знає інформацію, яку відправляє користувач[9].

1.5 Огляд існуючих рішень

Сьогодні існує безліч додатків для обміну повідомленнями, які мають популярність серед населення світу. Вони всі дуже схожі за функціоналом, який вони виконують, але кожен має свої особливості. Серед найпопулярніших можна виділити наступні: Telegram, iMessagея, WhatsApp та Slack.

1.5.1 Telegram

Telegram – безплатний месенджер, який виділяється своєю швидкістю, ,безпекою та функціоналом. Він доступний на більшості операційних систем. Також була створена вебверсія, яка дозволяє отримати можливість спілкуватись з людьми навіть за відсутності необхідної ОС.

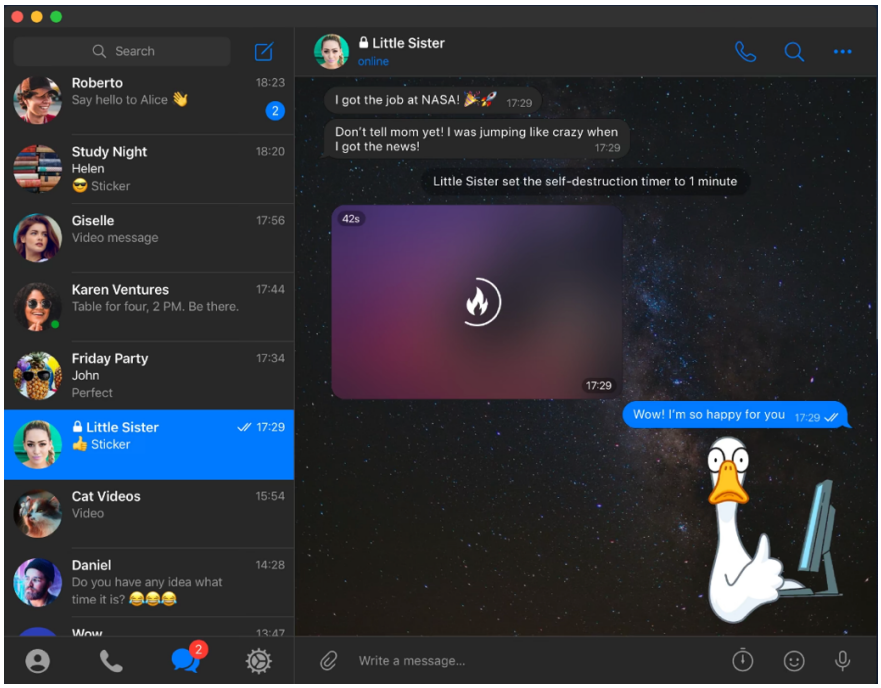


Рисунок 1.3 Інтерфейс Telegram на macOS

Telegram був створений на основі хмарних технологій, які дозволяють використовувати його одночасно на декількох девайсах майже без затримки.

Серед особливостей даного месенджера можна виділити наступні:

- Можливість створювати “секретні” чати, у яких буде використовуватись end-to-end шифрування. Повідомлення у цих чатах не будуть зберігатись на серверах, а доступ до них можна отримати тільки з того девайсу, якого було відправлене повідомлення.
- Наявність функціонала, який дозволяє користувачу змінювати дизайн програми.
- Створення групових аудіо- та відеодзвінків.
- Телеграм боти, які додають додатковий функціонал месенджеру та які можуть бути створені кожним користувачем.

Популярність даного месенджеру постійно зростає та, за останніми даними, кількість активних користувачів сягає понад 500 мільйонів[10].

1.5.2 iMessage

iMessage – додаток для миттєвого обміну повідомленнями від компанії Apple. Він вбудований на всіх девайсах, що працюють під управлінням операційних систем, розробкою яких займається Apple. Крім роботи як месенджер, він також використовується для отримання SMS та MMS повідомлень.

Рисунок 1.4 Інтерфейс iMessage на iOS

Даний месенджер має багато особливостей, через які користувачі віддають йому перевагу:

- End-to-end шифрування, що робить листування максимально безпечною: ніхто крім співрозмовників не зможе отримати доступ до інформації, яка зберігається в цих повідомленнях.
- Дизайн максимально схожий з операційною системою.
- Завдяки екосистемі Apple, яка має багато функцій та піднімає взаємодію між девайсами одного користувача на новий рівень, повідомлення, відправлені на одному гаджеті, автоматично з’являються на всіх інших.
- Можливість відправляти фотографії та відео без втрати якості.
- Усі SMS та повідомлення iMessage зберігаються у одному інтерфейсі, що є дуже зручним.

Не дивлячись на факт, що даний месенджер охоплює власників гаджетів лише однієї операційної системи, кількість користувачів є дуже високою. Близько 85% власників iPhone хоча іноді використовують його[11].

1.5.3 WhatsApp

WhatsApp – один з найвідоміших на сьогодні месенджерів. Доступний на всіх найпопулярніших операційних системах. Він має весь функціонал, який користувач очікує від сучасного месенджеру. Його розробкою займається компанія Meta, яке є однією з найвідоміших в IT-індустрії. Крім даного месенджеру Meta веде роботу над соціальними мережами: Facebook та Instagram, які користуються дуже широкою популярністю серед населення планети.

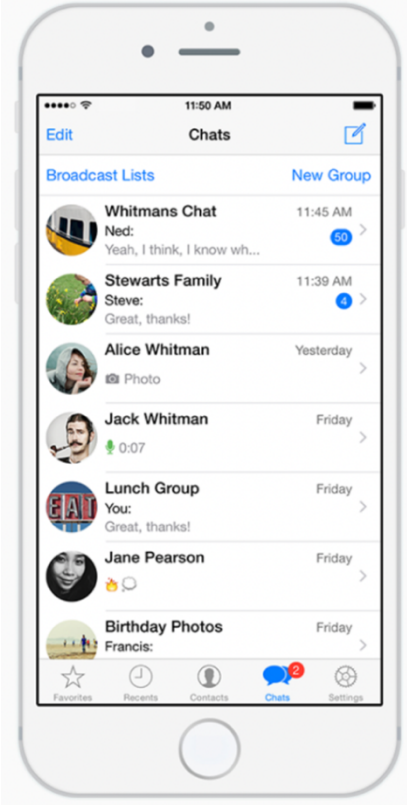


Рисунок 1.5 Інтерфейс WhatsApp на iPhone

Особливості WhatsApp:

- Можливість форматувати текст.
- Наявність функціонала, який, за бажанням користувача, автоматично видалить повідомлення через деякий час.
- Можливість збереження деякого чату для перенесення даних з WhatsApp до іншого месенджеру.
- Завжди використовується end-to-end шифрування.

На цей момент часу WhatsApp є найпопулярнішим месенджером. Кількість активних користувачів переступила поріг у 2 мільярди. Така кількість користувачів робить його третьою найпопулярнішою соціальною мережею, після Facebook та YouTube[12].

1.5.4 Slack

Slack – додаток для обміну повідомленнями, який зазвичай використовують на робочих місцях. Він має багато додаткових інструментів, які допомагають користувачам полегшити комунікацію з колегами. На відміну від більшості інших месенджерів, даний, крім безплатної підписки, має також платні, які дозволяють розширити його функціонал.

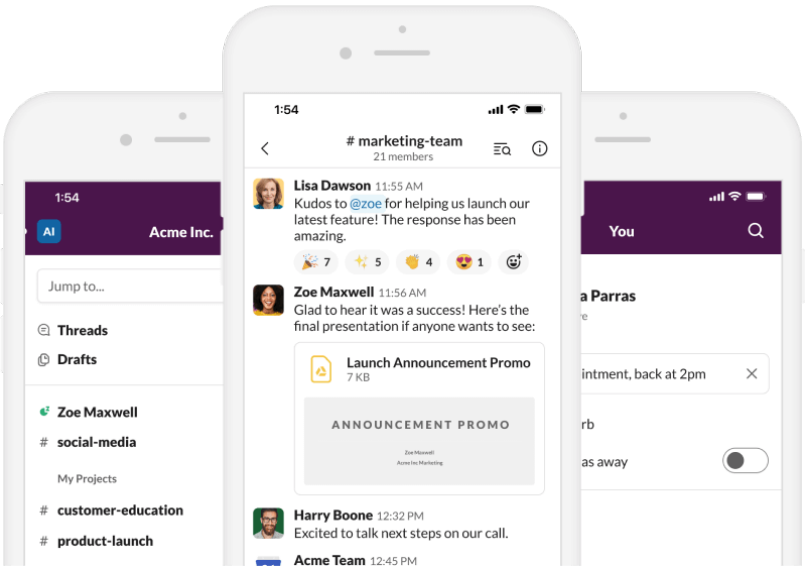


Рисунок 1.6 Інтерфейс Slack на iOS

Особливості Slack:

- Можливість створення каналів у межах однієї організації, що допомагає організувати список чатів та потік повідомлень.
- Створювання нагадувань на будь-яких повідомленнях, що будуть надсилати сповіщення у визначений час.
- Можливість підключення сторонніх сервісів до необхідних каналів.
- Список завдань, який буде нагадувати користувачам про пріоритетні задачі.

Оскільки Slack пропонує функціонал, який є корисним для відносно вузького кола людей, його популярність є значно нижчою ніж у традиційних месенджерів. Кількість активних щодня користувачів є близькою до десяти мільйонів[13].

1.6 Порівняння існуючих рішень

Таблиця 1.2 – Порівняльна характеристика існуючих месенджерів.

	Telegram	iMessage	WhatsApp	Slack
End-to-end шифрування	Лише у Secret chat	Так	Так	Ні
Можливість бачити чи співрозмовник у мережі	Так	Ні	Так	Так
Аудіо- та відеодзвінки	Так	Через FaceTime	Так	Так
Редагування відправлених повідомлень	Так	Ні	Ні	Так
Максимальний розмір файлів	1.5 GB	100 MB	16 MB	Доступні сховища з різною місткістю: Free – 5 GB (на всіх користувачів), Pro – 10 GB (на одного користувача) Business – 20 GB (на одного користувача)
Вартість(місяць)	Free	Free	Free	Free, Pro – 6.7$, Business – 12,5$
ОС	Android, iOS	iOS	Android, iOS	Android, iOS

З таблиці видно, що кожен з месенджерів має свої переваги та недоліки. Тому перед вибором слід проаналізувати для яких цілей він потрібен користувачу.

Для комунікації з колегами по роботі краще обирати Slack, оскільки він пропонує функціонал, який допомагає організувати роботу у командах, зберігати файли великих об’ємів та багато іншого. Для інших цілей краще віддавати перевагу більш традиційним месенджерам.

Якщо користувач планує використовувати месенджер для спілкування з друзями, то можна обрати і Telegram, і WhatsApp, і iMessage. Але останні два мають end-to-end шифрування у всіх чатах, тому є більш захищеними.

ВИСНОВОК ДО РОЗДІЛУ 1

Під час аналізу предметної області акцент було надано опису мобільних додатків, клієнт-серверної архітектури, поняття “месенджер” та порівнянню готових рішень.

Було проведено аналіз основних видів мобільних додатків: нативних, веб та гібридних. Визначив, які з цих методів розробки є кращими для розв'язання певних задач. Для розробки додатків з офлайн функціоналом краще використовувати нативний метод розробки. У іншому випадку можна вибирати як веб, так і гібридну технологію.

Також необхідно було провести огляд сучасних операційних систем. Виділив дві, які є найпоширенішими: iOS та Android. Навів переваги кожної з них. Основна перевага iOS над Android – строк підтримки девайсів, що її використовують.

Провів аналіз клієнт-серверної архітектури, яка буде використовуватись для подальшої розробки додатка. Була описана одна суттєва проблема, яка може виникати, у разі, якщо усі клієнтська частина додатку побудована на одному сервері.

Було визначено відмінності розробки нативних додатків під iOS та Android і вирішено, що розробка під iOS є дещо простішою через легшу мову програмування та обмежену кількість екранів. Також вона є більш прибутковою для розробника.

Проведено аналіз терміну “месенджер” та надано порівняльну характеристику різних представників даної категорії мобільних додатків. Визначено, що кожен з них має свої переваги та є більш зручним для певних задач.

РОЗДІЛ 2. ЗАСОБИ ДЛЯ РЕАЛІЗАЦІЇ

2.1 Засоби для реалізації клієнтської частини:

Для реалізації клієнтської частини я використовував засоби, розроблені компанією Apple, що використовуються для створення нативних програм для їхніх операційних систем, таких як iOS, macOS, iPadOS, watchOS та tvOS.

1.1 Swift.

Swift – потужна мова програмування, яка вперше була представлена компанію Apple на Worldwide Developer Conference (WWDC) у 2014 році.

Вона є багатопарадигмовою та об’єктно-орієнтованою та охоплює усі найсучасніші тренди з програмування.

Swift використовує надзвичайно швидкий компілятор LLVM, що дозволяє витискати максимум з сучасного апаратного забезпечення. Деякі дослідження показують, що дана мова є у майже 3 рази швидша ніж Objective-C, яка використовувалась компанією Apple у минулому, та у майже 9 разів швидша ніж Python.

Мова Swift має декілька особливостей, що виокремлюють її від інших мов програмування[14]:

- Усі перемінні ініціалізуються до того, як вони можуть бути використані, що попереджує помилки та небажаний результат роботи програми.
- Керування пам’яттю відбувається повністю автоматично, що дозволяє розробнику не думати про виділення пам’яті вручну.

- Swift використовує строгу типізацію, що означає наявність незмінного типу у кожної перемінної, дозволяє уникнути помилок та легше передбачити результат роботи програми.
- Вивід типів, що автоматично визначає тип виразу. Це означає, що розробник не повинен самостійно уточнювати тип перемінної, а компілятор робить все автоматично. Завдяки цій особливості код стає легшим для читання та швидшим.

Хоча Swift має дуже багато переваг над іншими мовами програмування, не обійшлось без недоліків. Серед основних можна виділити наступні:

- Оскільки ця мова з’явилась відносно нещодавно, вона постійно зазнає суттєвих змін.
- Кількість розробників, не зважаючи на її ріст, залишається все ще невеликою.
- Відсутність можливості розробки програм без техніки з операційною системою macOS.

Цю мову використовують у багатьох різних цілях, а саме: створення серверів, глибинне навчання. Але найчастіше Swift використовують для створення нативних застосунків для операційних систем від Apple. Оскільки ця компанія займається не лише розробкою програмного забезпечення, а й апаратного, усі компоненти систем працюють максимально ефективно. Це дозволяє давати гарантію, що гаджети, які користувачі купляють будуть релевантними через багато років.

Swift майже відразу після своєї появи почала набирати популярність серед розробників і стала однією мов, що найшвидше розвиваються. На даних момент вона займає 12 місце у рейтингу найпопулярніших мов програмування.

May 2022	May 2021	Change	Programming Language	Ratings	Change
1	2	↑	Python	12.74%	+0.86%
2	1	↓	C	11.59%	-1.80%
3	**3**		**Java**	**10.99%**	**-0.74%**
4	4		C++	8.83%	+1.01%
5	5		C#	6.39%	+1.98%
6	6		Visual Basic	5.86%	+1.85%
7	7		JavaScript	2.12%	-0.33%
8	8		Assembly language	1.92%	-0.51%
9	10	↑	SQL	1.87%	+0.16%
10	9	↓	PHP	1.52%	-0.34%
11	17	⇈	Delphi/Object Pascal	1.42%	+0.22%
12	18	⇈	Swift	1.23%	+0.08%

Рисунок 2.1 Рейтинг мов програмування за версією TIOBE[15].

2.1.2 SwiftUI

SwiftUI – новий фреймворк, представлений компанією Apple у 2019 році, що надає розробникам можливість розробляти користувацькі інтерфейси на операційних системах iOS, macOS, iPadOS, watchOS та tvOS. У його основі лежить фреймворк UIKit.

До появи SwiftUI основними фреймворками були UIKit та AppKit, що використовувалися для розробки на операційних системах iOS та macOS відповідно. Для побудови самих інтерфейсів використовували Interface Builder та Storyboard, у яких різноманітні елементи додавались на екран. У SwiftUI використовується інший підхід: побудова інтерфейсу відбувається повністю за допомогою коду.

Після виходу SwiftUI його переваги над старішими фреймворками стали очевидними. Серед основних можна виділити наступні:

- Використання декларативної парадигми програмування, на відміну від імперативної в UIKit, що дозволяє писати нам код, не описуючи потік

керування. Це означає, що більше не потрібно писати код, щоб контролювати кожну маленьку дію. Тепер просто необхідно вказати, що має відбутись, а фреймворк зробить все за нас. Як наслідок розробники можуть створювати програми, які мають такий самих функціонал, але код яких буде займати набагато менший об’єм та буде простішим для розуміння[16].

- Можливість розробляти одночасно на всі операційні системи Apple, без необхідності вивчати нові фреймворки або модифікувати написаний код.
- Наявність функціонала, що дозволяє переглядати зміни в інтерфейсі програми у реальному часі.

Крім явних переваг SwiftUI має і декілька недоліків. Частина з них пов’язані з там фактом, що він з’явився нещодавно. Варто зазначити наступні:

- Відсутність підтримки операційних систем, що були випущені до виходу SwiftUI.
- Мала кількість відповідей на питання, які можуть виникнути під час розробки. Водночас майже для кожної проблеми, яка може виникнути під час роботи з UIKit, користувач з легкістю знайде якісне рішення.
- Деякий функціонал старіших фреймворків ще не був реалізований у SwiftUI.

На даних момент UIKit та AppKit є все ще більш популярними ніж SwiftUI, оскільки їх використовували для написання старіших додатків. Але Apple постійно заохочує розробників переходити на новий та більш потужний фреймворк, тому ситуація може кардинально змінитись у наступні декілька років.

```swift
//
//  ContentView.swift
//  ExampleApp
//
//  Created by Dima Kryshtal on 21.05.2022.
//

import SwiftUI

struct ContentView: View {
    var body: some View {
        List {
            Text("First line")
            Text("Second line")
            Text("Third line")
            Text("Fourth line")

        }
    }
}

struct ContentView_Previews: PreviewProvider {
    static var previews: some View {
        ContentView()
    }
}

```

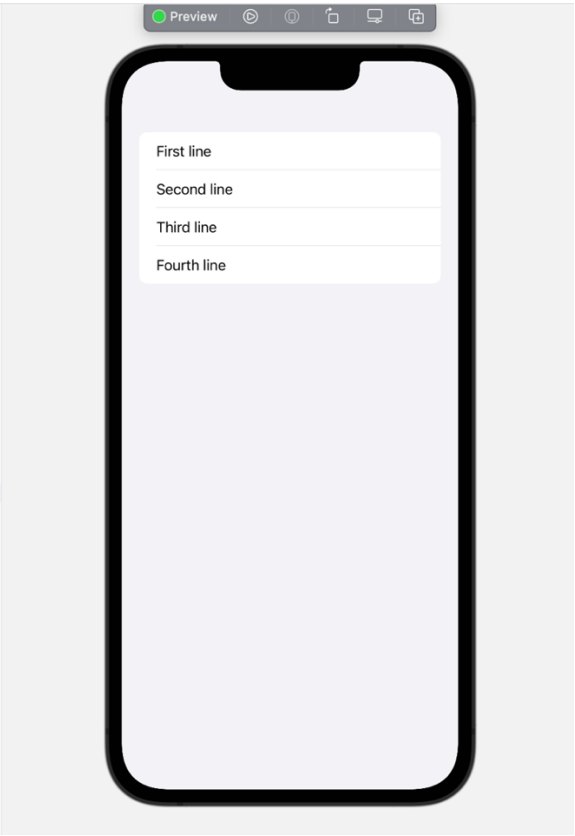


Рисунок 2.2. Приклад простої програми на SwiftUI

2.1.3 Swift протоколи

У мові програмування Swift існує багато протоколів, які дозволяються легко налаштовувати структури і класи.

- Codable – протокол, що дозволяє трансформувати типи даних Swift у JSON та навпаки. Він необхідний для форматування даних отриманих з сервера.
- Hashable – протокол, що дозволяє генерувати hash значення з об'єкта в залежності від його властивостей. Він необхідний якщо розробник планує використовувати об’єкт у сеті або як ключ у словнику(Dictionary). Це дозволяє цим колекціям бути максимально швидкими, оскільки hash завжди однаковий, та об’єкти, що зберігаються в них, завжди лежать в одному і тому самому місці.

- Identifiable – протокол, що дозволяє наявність тільки одного об’єкта з деяким полем, наприклад “id”. Його слід використовувати, якщо об’єкт цього типу може використовуватись у списках(List).
- Equatable – протокол, що дозволяє порівнювати значення об'єкта. У середині об'єкта, що використовує даний протокол, можна описати функцію “==”, яка буде порівнювати його не за всіма полями, а лише за тими, що бажає розробник.

2.1.4 Обгортки властивостей (Property Wrappers)

Обгортки властивостей[31] – особливість SwiftUI, яка дозволяє значно полегшити реалізацію взаємодії виглядів з даними, їх оновлення.

Для реалізації функціоналу даного проекту були використані наступні:

- @State – обгортка властивостей, яка використовується всередині вигляду. Її основна мета – автоматичне оновлення виглядів, які залежать від неї. Зазвичай використовується з простими типами даних.
- @Binding – обгортка властивостей, яка використовується всередині вигляду та здійснює прив’язку даних, що були передані іншим виглядом.
- @ObservedObject – обгортка властивостей, яка використовується всередині вигляду для об’єктів, що наслідують протокол ObservableObject.
- @Published – обгортка властивостей, що використовується для властивостей всередині об’єктів, які наслідують від протоколу ObservableObject. Викликає перезавантаження виглядів, які залежать від них.

- @StateObject – обгортка властивостей, що за функціями схожа на @ObservedObject. Її прийнято використовувати, коли ми створюємо об’єкт, що наслідує від Observable Object.
- @EnviromentObject – обгортка властивостей, що дозволяє отримати дані, які були створені раніше у ієрархії виглядів.

2.1.5 MVVM архітектура

MVVM – архітектурний патерн, суть якого полягає у відокремленні логіки юзер інтерфейсу від бізнес-логіки[24]. З його допомогою можна уникнути великої кількості коду у класах, де відбувається розробка інтерфейсу.

MVVM складається з 3 основних частин:

- Модель(Model) описує дані, що необхідні для роботи програми.
- ViewModel – елемент архітектури, який взаємодіє з моделлю. Він ніколи не повинен зберігати View.
- Вигляд(View) – графічний інтерфейс, з яким користувач взаємодіє напряму. Він підписується на ViewModel та з його допомогою оновлює елементи інтерфейсу.

Завдяки використанню цієї архітектури набагато легше здійснювати тестування.

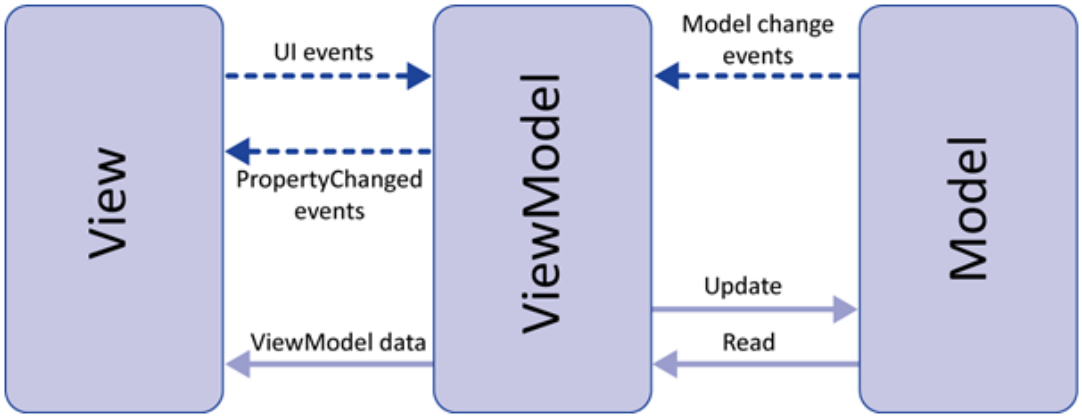


Рисунок 2.3 Діаграма MVVM архітектури.

2.2 Засоби для розробки серверної частини.

2.2.1 NPM

NPM – менеджер пакетів, що є повністю безплатним. Його встановлення відбувається разом з NodeJS. Менеджери пакетів необхідні для зберігання інформації про раніше встановлені пакети та для завантаження нових з хмарного сховища.

Керування відбувається через інтерфейс командного рядка. Усі дані про встановлені пакети зберігаються у файлі package.json. Завдяки цьому розробникам не потрібно зберігати файли, які можуть займати сотні мегабайт у хмарних сховищах. Після клонування репозиторію слід лише написати команду та npm автоматично завантажить всі пакети. Для встановлення пакетів слід використати команду “npm install”. Крім встановлення користувачі, які мають бажання поділитися з власними розробками зі спільнотою, можуть легко завантажити свої пакети до npm.

2.2.2 JavaScript

JavaScript – скриптова мова програмування, що використовується як для створення клієнтської, так і серверної частин програми. Скриптові мови програмування виконуються лінія за лінію за допомогою інтерпретатору під час роботи програми. З іншого боку, компільовані мови програмування, наприклад C++, виконують переклад синтаксису у машинний код перед виконанням програми[17].

З моменту своєї появи JavaScript дуже сильно змінився і від мови програмування, яка працювала лише для розробки клієнтської частини вебсторінок, виріс до мови програмування, за допомогою якої можна

розробляти майже все. Наприклад, фреймворк NodeJS дозволяє мові працювати не лише у браузері і розробляти сервера.

Як у всі мови програмування JavaScript має свої плюси та мінуси. Серед основних переваг можна виділити наступні[18]:

- Популярність відіграє чи не основною роль, чому розробники віддають перевагу JavaScript, а не іншим мовам. Майже усі проблеми, з якими може зіштовхнутись користувач, вже були вирішені, що дозволяє сильно прискорити розробку.
- Наявність багатьох пакетів, що допомагають значно полегшити розробку.
- Наявність підтримки у майже всіх сучасних браузерах.
- Мова є дуже легкою для розуміння, тому кожен може відразу розпочати працювати з нею.
- Можливість розробляти frond end і back end використовуючи лише JavaScript.

З іншого боку існує декілька недоліків, які відштовхують розробників від використання даної мови програмування:

- Може бути важко розробляти великі додатки з використанням чистого JS. Для таких задач слід використовувати TypeScript.
- Основним недоліком є те, що кожен користувач може переглянути код програми, написаної цією мовою.

JS є однією з найпопулярніших мов програмування серед розробників. За версією різних сервісів вона точно входить до 5 найпопулярніших мов. Більше ніж половина усіх програмістів вирішують ті чи інші задачі, використовуючи дану мову програмування. Основною причиною є можливість її легко інтегрувати з іншими мовами програмування та фреймворками. Загалом ріст популярності пов’язаний з виходом фреймворку NodeJS, що дозволив створювати серверну частину програм.

Rank	Change	Language	Share	Trend
1		Python	27.85 %	-2.5 %
2		Java	17.86 %	-0.1 %
3		JavaScript	9.17 %	+0.4 %
4		C#	7.62 %	+0.7 %
5		C/C++	7.0 %	+0.4 %
6		PHP	5.36 %	-1.0 %
7		R	4.34 %	+0.5 %
8	↑↑↑	TypeScript	2.39 %	+0.7 %
9	↓	Objective-C	2.25 %	+0.0 %
10		Swift	2.05 %	+0.3 %

Рисунок 2.4 Рейтинг мов програмування за версією PYPL[19].

2.2.3 NodeJS

NodeJS – середовище виконання для мови JavaScript, яке використовується для побудови швидких масштабованих мережевих систем. Воно працює на V8 JavaScript engine, що робить його дуже швидким[20].

Дане середовище працює на одному процесі без створення нових потоків для кожного запиту. Усі бібліотеки написані з використанням неблокуючої парадигми програмування, що надає можливість створювати масштабовані системи. Наприклад, коли сервер відправляє запит до бази даних, замість блокування та очікування на відповідь програма продовжує виконувати наступні запити. Операція продовжується тоді, коли сервер отримує відповідь.

Забезпечує таку роботу NodeJS event loop, який постійно слухає та обробляє події, які йому надходять. Він ініціалізується після запуску системи та його робота відбувається протягом усього циклу її життя. Після виклику

функції, система додає її до стека викликів. Коли результат її виконання отримано, система видаляє її зі стеку[21].

У разі, якщо ми викликаємо функцію обернену у setTimeout, вона додається до Web API, де очікує деяку кількість часу, що була задана другим аргументом. Після закінчення часу, функція передається у чергу та очікує на той момент, коли вона буде передана до стека виклику та виконана.

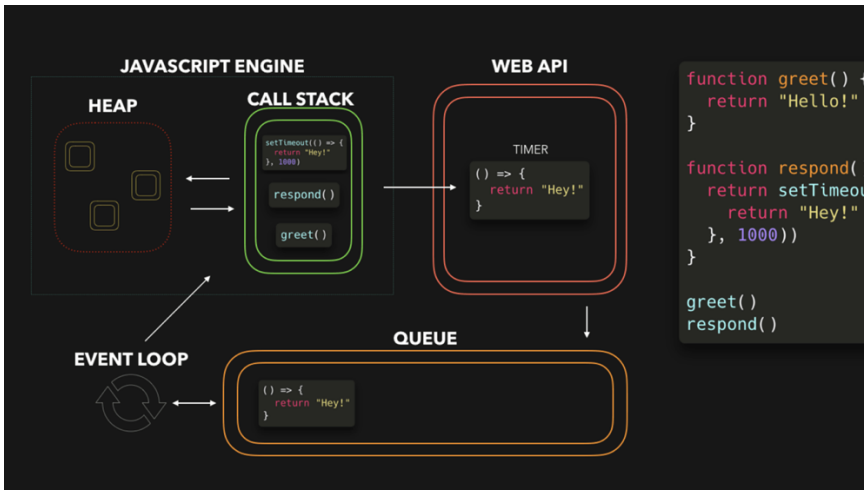


Рисунок 2.5 Приклад роботи event loop

Завдяки цим особливостям система може обробляти безліч запитів і надавати результат користувачу максимально швидко, а також бути стійкою до збільшення кількості користувачів.

2.2.4 MongoDB

MongoDB – не реляційна база даних, яка зберігає дані у вигляді JSON документів, що означає можливість зміни структури документів з плином часу. Замість традиційних таблиць, таких які використовуються у реляційних базах даних, наявні колекція та документи[22].

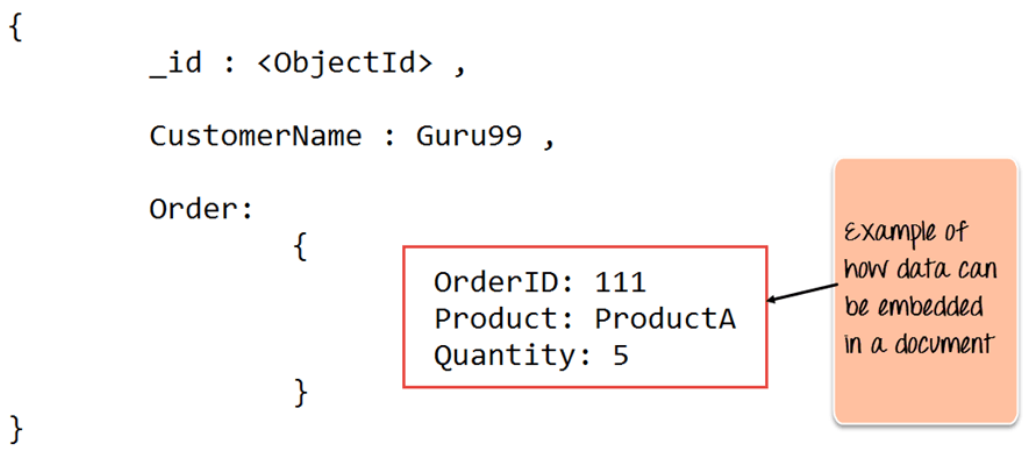


Рисунок 2.6 Приклад документу у MongoDB

Дана база даних має наступні особливості, які виокремлюють її від інших баз даних та заохочують розробників до її використання:

- Бази даних складаються з колекцій, які, в свою чергу, складаються з документів. Кожна колекція може мати різні документи. Вони не повинні мати обов'язково однакові поля.
- Висока швидкість роботи. За деякими дослідженнями база даних переважає за швидкістю реляційні бази даних у майже 100 разів, що має перевагу у таких додатках, як месенджери, через дуже велику кількість даних.
- З MongoDB набагато простіше розпочати роботу ніж з традиційними базами даних.

З недоліків можна виділити велику кількість пам'яті, яку займає база даних, та дублювання документів.

Для зберігання даних використовується MongoDB Atlas. Це хмарне середовище, в якому можна зберігати та легко керувати всіма функціями бази даних.

2.2.5 Socket.IO

Socket.IO – один з основних засобів, що використовувався для створення моєї програми. Він дозволяє користувачам обмінюватися даними між клієнтом та сервером у реальному часі та реалізований за допомогою протоколу WebSocket.

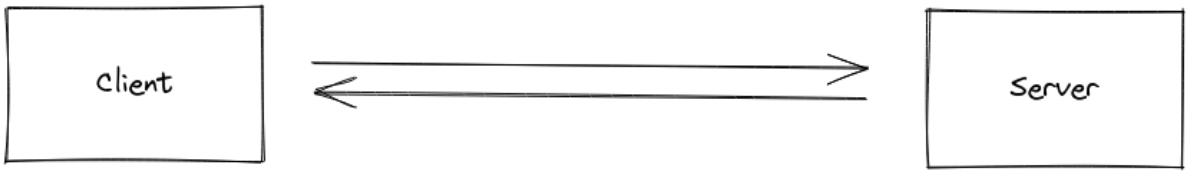


Рисунок 2.7 Комунікація між сервером та клієнтом через Socket.IO

Хоча Socket.IO побудований на WebSocket, до кожного пакету додаються додаткові дані, тому клієнт, який побудований на Socket.IO, не може підключатися до сервера, який побудований на WebSocket, і навпаки[26].

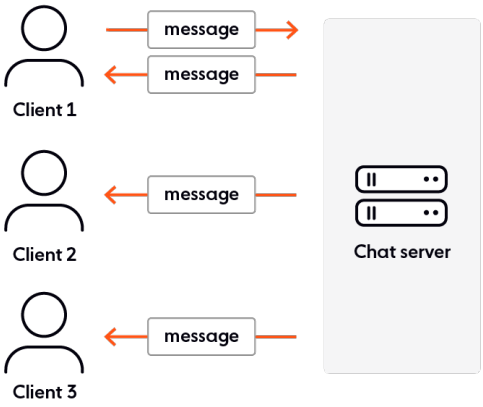


Рисунок 2.8 Приклад роботи Socket.IO у додатку для обміну повідомленнями

При створенні Socket.IO розробники приділили увагу реалізації деяких особливостей, які допомагають користувачам значно полегшити розробку:

- Підтримка автоматичного повторного підключення у випадку втрати з'єднання.
- Автоматичне виявлення втрати з'єднання між клієнтом та сервером завдяки "Механізму серцебиття", який відправляє запити з певним інтервалом для перевірки наявності комунікації.

2.2.6 ExpressJS

Побудова сервера на чистому NodeJS може бути складною та займати багато часу. Задля уникання цього використовується фреймворк ExpressJS, який побудований на NodeJS та являється одним з його найпопулярніших фреймворків.

```
const express = require('express' 4.18.1 )
const app = express()
const port = 3000

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

Рисунок 2.9 Приклад просто серверу на ExpressJS

Завдяки даному фреймворку можна дуже легко створювати сервера та налаштовувати маршрутизацію і кінцеві точки, які визначають як сервер відповідає на деякий запит. Також з допомогою ExpressJS зручно взаємодіяти з базами даних[27].

Налаштування маршрутів відбувається через функцію проміжної обробки(middleware)[28]. Ця функція приймає 3 аргументи:

- req(request) – аргумент запиту, що відповідає за дані, які були передані клієнтом.
- res(response) – аргумент відповіді, що відповідає за дані, які сервер відправить клієнту.
- next – функція, яка використовується у випадку, якщо функція проміжної обробки не закінчує виконання відправленням відповіді

клієнту. Тому, для переходу до обробки наступного запиту, необхідно викликати next.

2.2.7 REST API

REST – архітектурний стиль для створення мережевої системи, що записує та отримує інформацію за допомогою HTTP запитів. Нема єдиного стандарту як він має виглядати, тому його реалізація може виглядати по різному[23].

Взаємодія клієнта з сервером відбувається шляхом використання HTTP запитів, які здійснюють операції за допомогою наступних методів:

- GET – метод, який використовується для отримання певної інформації з серверу.
- POST – метод, який зазвичай використовується для створення нового елементу або оновлення старого.
- PUT – метод, який також призначений для створення та оновлення елементів. Але він завжди дає один і той самий результат. POST, з іншого боку, може мати побічні ефекти такі, як створення одного і того ж елементу багато разів.
- DELETE – метод, який використовується для видалення необхідних елементів.

Кожен запит крім методу складається з кінцевої точки, заголовків та даних, які він відправляє:

- Кінцева точка – це унікальне URL посилання, яке дозволяє клієнту запрошувати у сервера виконання деякої задачі. До нього можуть додаватися параметри, якщо вони необхідні для правильного виконання запиту.
- Заголовки зазвичай використовуються для відправки метаданих.

- Дані відправляються у “body” запиту та найчастіше мають JSON формат.

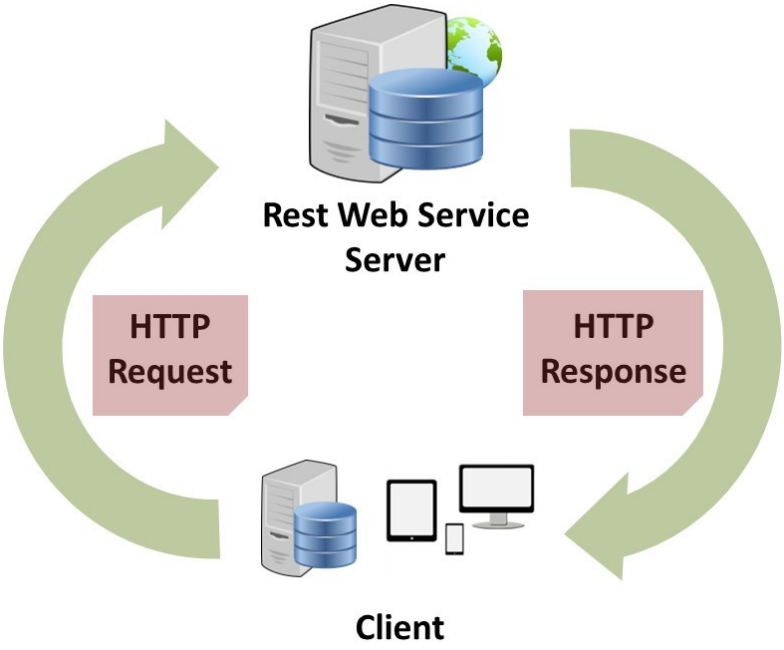


Рисунок 2.10 Приклад роботи REST API

2.3 Середовища для реалізації клієнтської та серверної частин.

2.3.1 XCode

XCode – це IDE від компанії Apple, що надає можливість розробляти програмне забезпечення на комп’ютерах з операційною системою macOS. Він забезпечує розробників всіма інструментами, які є необхідними для розробки програми: текстовий редактор, компілятор та симулятор для запуску програми на певній операційній системі. З його допомогою можна створювати, компілювати та налагоджувати програму. Також XCode використовується для завантаження програм до App Store.

Дане середовище також можна використовувати для редагування коду написаних на різних мовах програмування: C++, C#, Objective-C тощо.

XCode – єдине середовище, яке дозволяє розробляти нативне програмне забезпечення для операційних систем від Apple, тому розробник повинен використовувати його, якщо він хоче працювати у цій сфері.

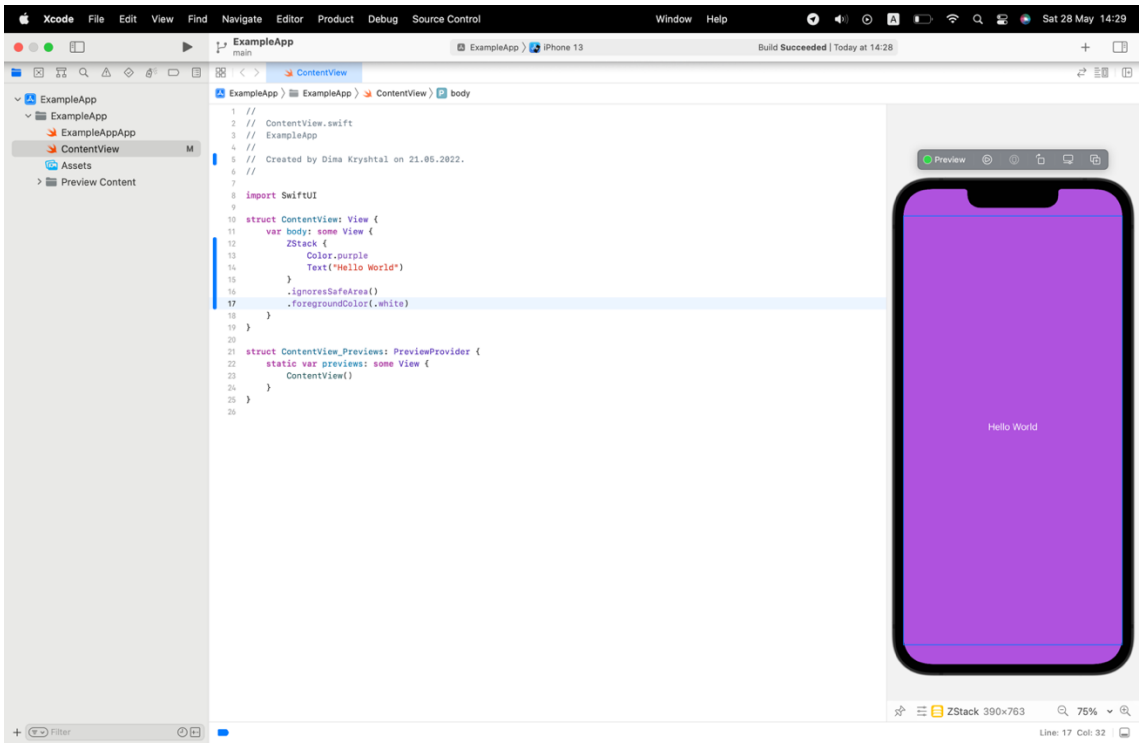


Рисунок 2.11 Інтерфейс XCode

Під час розробки на SwiftUI велику частину екрана займає текстовий редактор та симулятор, який показує зміни в інтерфейсі програми у реальному часі. У лівій частині розміщений Navigator, у якому розміщена різноманітна інформація, наприклад, файли проєкту або помилки.

2.3.2 Visual Studio Code

Visual Studio Code – безплатний текстовий редактор від компанії Microsoft. За статистикою він є одним з найпопулярніших редакторів. Причиною цьому є цілий ряд особливостей, які роблять його дуже потужним, але водночас він залишається легким та простим.

Він може працювати з більшістю сучасних мов програмування, що робить його дуже зручним для роботи з великими проєктами, коли треба переходити від роботи з однією мовою до іншої.

Також серед особливостей можна виділити його швидкість та наявність розширень, які дозволяють постійно розширювати функціонал редактору. Їх кількість постійно збільшується[30].

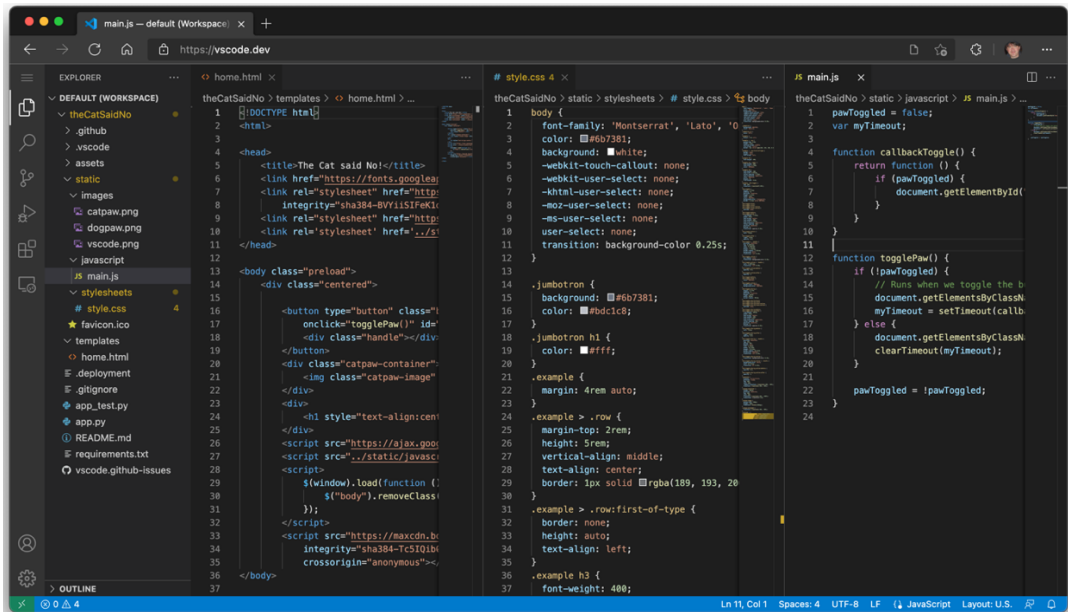


Рисунок 2.12 Інтерфейс Visual Studio Code

ВИСНОВОК ДО РОЗДІЛУ 2.

У другому розділі були описані технології, які використовувалися для написання даного проєкту.

Перший підрозділ розпочинається з опису технологій, які були обрані для створення клієнтської частини проєкту.

Для створення додатка була обрана мова програмування Swift. Були описані її особливості. Для проєктування інтерфейсу використовувався фреймворк SwiftUI. Були названі його переваги та недоліки у порівнянні з UIKit. Також був описаний архітектурний патерн MVVC, його структура та деякі переваги над іншими патернами.

У другому підрозділі були перераховані технології, що використовувалися для проєктування серверної частини.

У якості основної мови програмування була обрана JavaScript, яка є однією з найпопулярніших для розробки серверів. У якості середовища для виконання використовувався NodeJS, який показує високу швидкість в обробці великої кількістю одночасних запитів.

Одним з найважливіших елементів проєкту є база даних. Для даного додатку була обрана нереляційна база даних MongoDB, яка краще підходить для програм, у яких відбувається швидкий потік даних.

Для операцій, які не потребують постійної комунікацій з сервером було написано REST API. Воно здебільшого використовується для завантаження історії з бази даних. Для його реалізації застосовувався фреймворк ExpressJS.

Деякі з операцій у додатках для обміну повідомленнями потребують постійної комунікації між сервером та клієнтами. Для реалізації даного функціоналу використовувався Socket.IO.

У останньому розділі були описані середовища для розробки, які я використовував для написання клієнтської та серверної частин.

Спочатку було описане середовище XCode, яке є єдиним, що дозволяє написання нативних додатків на iOS. Для написання серверної частини використовувався текстовий редактор Visual Studio Code.

РОЗДІЛ 3. ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

Після аналізу предметної області та вибору засобів реалізації системи, можна приступити до її розробки. Вона складається з двох частин: розробка клієнтської частини та розробка серверної частини.

3.1 Розробка клієнтської частини

Під час реалізації клієнтської частини відбувається розробка інтерфейсів та сервісів для комунікації з сервером.

3.1.1 Створення проекту

Після створення проекту, у Project Navigator з'являються два файли:

- App.swift, який відповідає за створення програми. В ньому зберігається код необхідний для її запуску.
- ContentView.swift, у якому реалізовано стандартний інтерфейс SwiftUI програми. Він буде займати найвище місце у ієрархії виглядів у даному проекті.

```
import SwiftUI

@main
struct TestProjectApp: App {
    var body: some Scene {
        WindowGroup {
            ContentView()
        }
    }
}
```

Рисунок 3.1 Початковий зміст App.swift

```swift
import SwiftUI

struct ContentView: View {
    var body: some View {
        Text("Hello, world!")
            .padding()
    }
}
```

Рисунок 3.2 Початковий зміст ContentView.swift

3.1.2 Створення моделей

У якості сховища для інформації, яку необхідно зберігати навіть після завершення роботи програми використовується UserDefaults. Тут зберігається персональна інформація користувача та статус авторизації.

Для створення моделей використовуються додаткові протоколи, які дозволяють розширити їх функціонал з метою полегшення подальшого їх використання.

У клієнтській частині наявні 3 основних моделі, що відповідають за створення екземплярів об’єктів: User, ChatRoom, Message.

Об’єкт User відповідає за створення користувачів. Він використовує протоколи Codable, Hashable, Identifiable та складається з наступних властивостей:

- id – унікальний ідентифікатор об'єкта типу String
- firstName, lastName, username, email, password – ім’я, прізвище, ім’я користувача, електронна пошта та пароль користувача типу String.
- userChats – вибірка ідентифікаторів чатів користувача.
- userFriends – вибірка ідентифікаторів друзів користувача.

Об’єкт chatRoom відповідає за створення чату. Він використовує протоколи Codable, Equatable, Identifiable та складається з наступних властивостей:

- id – унікальний ідентифікатор типу String.
- chatName – назва чату типу String.
- chatType – вид чату типу String.
- user_ids – вибірка ідентифікаторів членів чату.

Об’єкт message відповідає за створення повідомлення. Він використовує протоколи Codable, Identifiable, Equatable та складається з наступних властивостей:

- id – унікальний ідентифікатор типу String.
- user_id –ідентифікатор відправника типу Int.
- chat_id – ідентифікатор чату типу String.
- message, date – повідомлення та дата типу String.

3.1.3 Створення виглядів

ContentView – вигляд, що запускається найпершим та показує один з виглядів: HomeWindow, якщо програма тільки запустилась, або MainView, якщо відбулась успішна авторизація або користувач підтвердив бажання продовжити роботи з цим обліковим записом.

```
struct ContentView: View {
    @StateObject private var authenticaion = Authentication()

    var body: some View {
        if (authenticaion.homeWindowIsShown == true) {
            HomeWindow(auth: authenticaion)
        } else {
            MainView()
                .environmentObject(authenticaion)
        }

    }
}
```

Рисунок 3.3 Демонстрація вигляду ContentView

Для збереження інформації про статус авторизації користувача, створюємо перемінну authentication типу Authentication, у якому зберігається властивість homeWindowIsShown та isLoggedIn з обгорткою @Published. При

ініціалізації даного класу відбувається отримання з UserDefaults статусу авторизації та передача його перемінній isLoggedIn.

У body описуємо умову, за якої мають показуватись відповідні вигляди. За зміни значення homeWindowIsShown властивість body повторно викликається та показує інших вигляд за необхідності.

MainView – основний вигляд програми, за допомогою якого можна перемикатися між виглядами, що зберігають інформацію про чати, друзів та профіль користувача. Для перемикання між ними використовується TabView, який показує вкладки, при натисненні на які можна перейти на потрібний вигляд.

Також у ньому лежать об’єкти, які зберігають інформацію про завантажені дані користувачем:

- chatsData – об’єкт, який зберігає інформацію про чати користувача. Дані завантажуються під час ініціалізації через сервіс ChatRoomNetworkManager.
- messagesData – об’єкт, який зберігає повідомлення. Дані завантажуються під час ініціалізації через сервіс MessagesNetworkManager.
- friendsData – об’єкт, який зберігає інформацію про друзів користувача. Дані завантажуються під час ініціалізації через сервіс UserNetworkManager.

```
var body: some View {
        TabView{
            ChatList()
                .tabItem {
                    Label("Chats", systemImage: "mail")
                }
                .tag(Tab.chats)

            Friends()
                .tabItem {
                    Label("Friends", systemImage: "person.2.fill")
                }
                .tag(Tab.friends)


            Profile()
                .tabItem {
                    Label("Profile", systemImage: "person.crop.circle")
                }
                .tag(Tab.profile)
        }
        .environmentObject(chatsData)
        .environmentObject(messagesData)
        .environmentObject(friendsData)
        .onAppear {
            initSocket()
        }

}
```

Рисунок 3.4 Реалізація MainView.swift

При першій появі вигляду відбувається підключення до сокету сервера через функцію initSocket, що дозволить спілкуватись в реальному часі.

Chat – вигляд, що відповідає за відображення чату та його повідомлень. Список повідомлень відображений у вигляді бульбашок, власні повідомлення відображаються з правого боку, я співрозмовників з лівого.

Friends – вигляд, що відображає список друзів. Через нього можна виконувати пошук та створювати нові чати. Доступ до нього можна отримати натиснувши другу вкладку.

Profile – вигляд, що відображує профіль користувача. Через нього можна вийти з профілю.

SignUp – вигляд, у якому реалізовано інтерфейс для створення нового користувача. Його поява викликається у HomeView при натисканні на відповідну кнопку.

3.1.4 Створення сервісів для комунікації з сервером

Одним з основних аспектів правильної роботи подібних додатків – правильна комунікація з сервером. Даний месенджер здійснює комунікацію з сервером за допомогою Socket.IO та REST API.

Для створення сервісів використовується патерн Singleton, що дозволяє існування лише одного екземпляру класу. Вони зазвичай використовуються для класів, які виконують комунікації з сторонніми серверами. Наприклад, для комунікації з базою даних слід використовувати даний патерн, щоб уникнути створення нового підключення багато разів.

SocketManager – сервіс, який відповідає за постійну комунікацію з сервером через Socket.IO. З його допомогою буде відбуватись посилання повідомлень та створення чатів у реальному часу.

Таблиця 3.1 – Функції сервісу SocketManager

Функція	Параметри	Опис
connectToSocketServer	Відсутні	Здійснює підключення до сокета та створює слухачі до подій “new message”, що виконується, коли співрозмовник присилає повідомлення, та “chat created”, що виконується, коли хтось створює чат з участю користувача.
sendMessage	message типу Message, що зберігає дані про	Посилання нового повідомлення.

Продовження таблиці 3.1

Функція	Параметри	Опис
	нове повідомлення, яке необхідно відправити	
createNewChat	chat типу ChatRoom, що зберігає дані про тільки створений чат, який необхідно відправити	Створення нового чату
disconnect	Відсутні	Відключення від сокета

Кінець таблиці 3.1

UserNetworkManager – сервіс, що покриває функціонал необхідний для отримання інформації, пов’язаної з користувачами.

Таблиця 3.2 - Функції сервісу UserNetworkManager

Функція	Параметри	Опис
getUsers	query – параметр типу String, що використовується для визначення кінцевої точки	Отримання даних користувача з сервера. Залежно від кінцевої точки запит може отримувати дані як одного, так і декількох
updateCurrentUser	username – параметр типу String, що вказує на логін користувача, дані якого слід оновити. requestBody – параметр, що є словником з ключем	Оновлення деяких даних користувача.

Продовження таблиці 3.2

Функція	Параметри	Опис
	типу String, а значення може бути значенням любого типу	
createNewUser	requestBody – параметр типу User, що отримується з вигляду SignUp	Створення нового користувача
login	requestBody – параметр, що є словником з ключем і значенням типу String	Авторизація користувача

Кінець таблиці 3.2

ChatRoomNetworkManager – сервіс, що відсилає запити на сервер для проведення дій над чатами користувача.

Він має функцію loadChatRooms, яка приймає у якості параметра arrayOfChatIds масив з ідентифікаторів чатів користувача. Дана функція відправляє запит на сервер, з метою отримати відповідь, яку буде містити масив з інформацією про чати.

MessagesNetworkManager – сервіс, що відсилає запити на сервер для отримання повідомлень.

Він має функцію loadMessages, яка приймає у якості параметру chat_id значення типу String, що ідентифікує чат, повідомлення якого має надати сервер.

3.1.5 Створення ViewModel

ViewModel використовується для відокремлення бізнес-логіки від логіки інтерфейсу. Його слід створювати, коли вигляд стає дуже великим та важким

для підтримки. Якщо ж кількість бізнес-логіки є невеликою, то можна залишити її у файлі з логікою інтерфейсу.

HomeViewViewModel – ViewModel, яка зберігає функціонал необхідний для правильної роботи HomeView, метою якого є авторизація чи підтвердження користувача.

Таблиця 3.3 – Перемінні класу HomeViewViewModel

Перемінна	Тип	Опис
showBackground	Bool	Якщо true, показ анімованого заднього фону. Якщо false, видалення анімованого заднього фону.
showingAlert	Bool	Якщо true, показ сповіщення про відсутність користувача у базі даних. Якщо false, видалення сповіщення.
authenticator	String	Збереження введеного автентифікатору.
password	String	Збереження введеного паролю
showSignUp	Bool	Якщо true, показ вікна реєстрації
showButtons	Bool	Якщо true, показ кнопок інтерфейсу

Таблиця 3.4 – Функції класу HomeViewViewModel

Функція	Параметри	Опис
showInterfaceButtons	Відсутні	Показ кнопок інтерфейсу. Викликається при появі HomeWindow
successfullLogin	Відсутні	Здійснює видалення кнопок інтерфейсу та фону після успішної авторизації.

Продовження таблиці 3.4

Функція	Параметри	Опис
hideHomeWindow	Відсутні	Закриває HomeWindow з певною затримкою після успішної авторизації.
logIn	Відсутні	Виконує авторизацію через функцію login сервісу UserNetworkManager.

Кінець таблиці 3.4

SignUpViewModel – ViewModel, яка зберігає функціонал вигляду SignUp, метою якого є створення нового користувача.

Таблиця 3.5 – Перемінні класу SignUpViewModel

Перемінна	Тип	Опис
showingAlert	Bool	Якщо true, показ сповіщення, про існування користувача у базі даних. Якщо false, видалення сповіщення.
authenticator	String	Збереження введеного автентифікатору.
email	String	Збереження введеної електронної пошти
password	String	Збереження введеного паролю
firstName	String	Збереження введеного ім’я користувача
lastName	String	Збереження введеного прізвища користувача

Вона має одну функцію sendNewUser, яка використовує раніше створений сервіс UserNetworkManager для відправки запиту на сервер для реєстрації користувача. У випадку, якщо користувач вже існує, додаток показує сповіщення “User already exists.”

ChatViewModel – ViewModel, в якій реалізований функціонал Chat вигляду, у якому здійснюється відправка та отримання повідомлень.

Таблиця 3.6 – Перемінні класу ChatViewModel

Перемінна	Тип	Опис
newMessageValue	String	Збереження нового повідомлення, введеного в поле.
sendIsTapped	Bool	Якщо значення стає true, відбувається відправка повідомлення.

Функціонал передачі повідомлення на сервер та відображення його на екрані реалізовується у функції sendMessage, яка використовує SocketManager для відправки повідомлення у реальному часі, а також додає його до масиву повідомлень відповідного чату.

FindFriendsViewModel – ViewModel, в якій розміщений функціонал для пошуку користувачів у базі даних.

Таблиця 3.7 – Перемінні класу FindFriendsViewModel

Перемінна	Тип	Опис
foundUsers	[User]	Зберігає масив зі знайдених користувачів
searchText	String	Зберігає текст, введений у поле для пошуку. При введені нового символу автоматично викликається функція searchResults.

У функції searchResults використовується функціонал сервісу UserNetworkManager, який відсилає запит на сервер для отримання масиву

користувачів. Після отримання даних, функція передає їх перемінній foundUsers.

3.2 Розробка серверної частини

Серверна частини є другим важливим елементом для роботи додатку. Складається з розробки бази даних та самого серверу.

3.2.1 Розробка бази даних

Оскільки для збереження інформації використовується не реляційна база даних mongoDB, то усі дані зберігається у вигляді JSON документу. На відміну від реляційних баз даних, вона не потребує попереднього створення таблиць, що дозволить додавати документи з різною структурою.

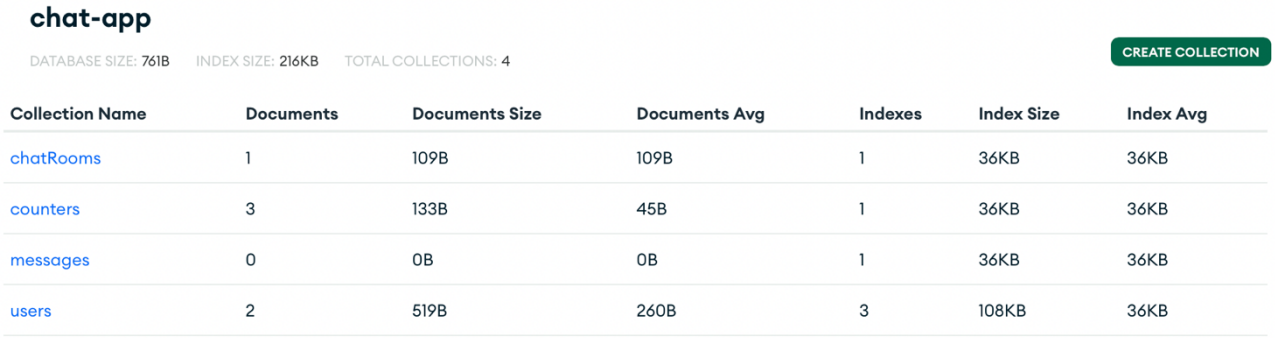


Рисунок 3.5 Структура бази даних

База даних mongoDB складається з колекцій:

- chatRooms – колекція, яка зберігає інформацію про усі чати, та складається з наступних ключів.
- messages – колекція, яка зберігає інформацію про усі повідомлення.
- users – колекція, яка зберігає інформацію про користувачів і має унікальні поля (email, username), щоб заперечує їх повторне використання для створення нового користувача.

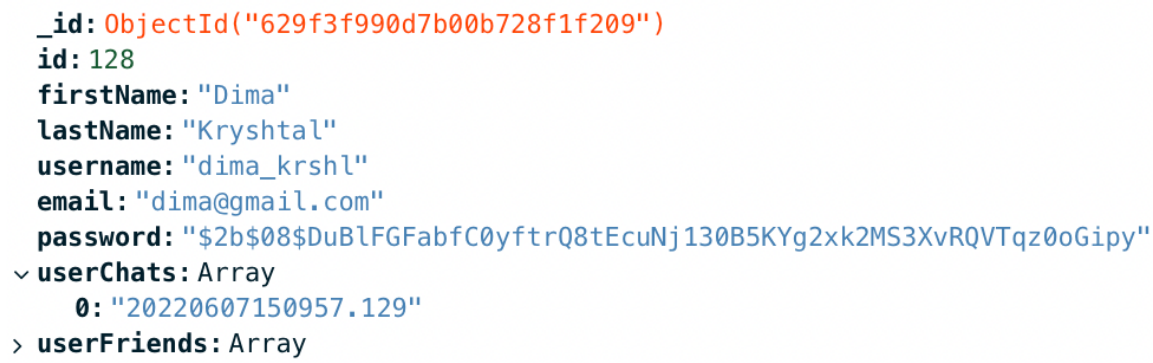


Рисунок 3.6 Структура документа у колекції users

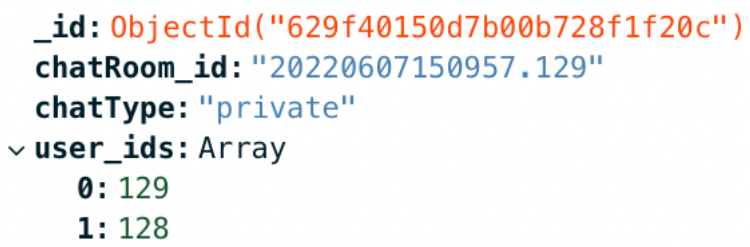


Рисунок 3.7 Структура документа у колекції chatRooms

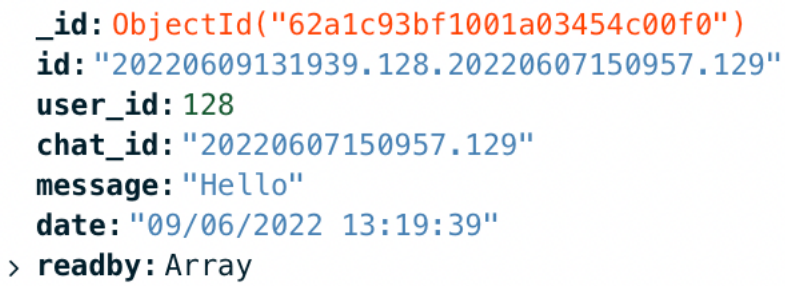


Рисунок 3.8 Структура документа у колекції messages

Для створення унікальних ключів колекції, після її створення у терміналі “mongosh” слід викликати відповідну функцію.

```
db.users.createIndex( { "username": 1 }, { unique: true } )
```

Рисунок 3.9 Функція для створення унікального ключа

3.2.2 Розробка сервера

Для розробки сервера використовуються фреймворки ExpressJS та Socket.IO, що дозволяють досить легко створювати та налаштовувати його.

Першим кроком є власне створення сервера та запуск його не певному порті.

```
import http from 'http';
import { Server } from 'socket.io';
import { connectToDB } from './db/connection.js';
import 'dotenv/config';
import { Socket } from './sockets/socketController.js';
import { app } from './app.js';

const server = http.createServer(app);
const io = new Server(server);
const socket = new Socket(io);

socket.socketEvents(socket);

server.listen(process.env.API_PORT, async () => {
    await connectToDB();
    console.log('Server started on port 8080');
});
```

Рисунок 3.10 Код, де відбувається запуск сервера

Створення сервера можна розділити на наступні кроки:

- Створюємо сервер за допомогою функції http.createServer(app), де app – express, за допомогою якого відбувається налаштування сервера: створення маршрутів(routes) та кінцевих точок(end points), підключення додаткових фреймворків тощо.
- Інтеграція Socket.IO до сервера відбувається через передачу його до класу Server з бібліотеки “Socket.IO”.
- Запуск сервера відбувається через функцію server.listen(port, callback), де port – порт, на якому відбувається запуск сервера, а callback – функція зворотного виклику, яка викликається при старті роботи сервера. У цьому випадку відбувається підключення до бази даних та виводиться повідомлення, що сервер успішно розпочав роботу.

Другим кроком у розробці сервера є створення кінцевих точок, за якими клієнт може виконувати певні запити на сервер та отримувати певний результат.

Усі кінцеві точки(end points) можна розділити на три види: для користувачів, для чатів, для повідомлень.

Таблиця 3.8 – Кінцеві точки для користувачів

Метод	Кінцева точка	Опис
GET	“/”	Метою даної кінцевої точки є отримання даних про користувачів. В залежності від query, який передає користувач. Може відбуватись як пошук лише одного користувача, так і декількох: • /?searchval=name, знаходить всіх користувачів, в логін яких має певний вираз. • /?searchid=id, знаходить користувача за певним ID. • /?searchids=id1,id2, знаходить вибірку користувачів з певними id.
POST	“/login/”	Метою даної кінцевої точки є проведення авторизації користувача. Спочатку проводиться пошук користувача на сервері. У випадку успішної операції перевіряється правильність введеного паролю. Якщо все вірно, програма дає дозвіл на авторизацію.

Продовження таблиці 3.8

Метод	Кінцева точка	Опис
POST	“/register”	Метою даної кінцевої точки є реєстрація користувача. Усі реєстраційні дані передаються з клієнта у вигляді JSON у тілі(body) запиту.
PUT	“/updatedata/:username”	Метою даної кінцевої точки є оновлення певних даних користувача. Пошук користувача у базі даних відбувається за допомогою параметру username, який передає ім’я користувача. Усі дані, що користувач має намір змінити передаються у тілі(body) запиту у вигляді JSON.
DELETE	“/delete/:username”	Метою даної кінцевої точки є видалення користувача з певним ім’ям, яке передається у параметрів username.

Кінець таблиці 3.8

Таблиця 3.9 – Кінцеві точки для користувачів

Метод	Кінцева точка	Опис
GET	“/loadchats”	Метою даної кінцевої точки є завантаження з бази даних усіх чатів користувача та відправка їх клієнту.

Таблиця 3.10 – Кінцеві точки для повідомлень

Метод	Кінцева точка	Опис
GET	“/messagesfrom/:chatid”	Метою даної кінцевої точки є отримання повідомлень деякого чату з бази даних, ідентифікатор якого передано у параметрі.

Далі слід налаштувати взаємодію сервера з клієнтом через Socket.IO. Для цього створимо файл socketController.js з класом Socket, у якому буде зберігатись метод socketEvents, який буде відповідати за весь функціонал та зберігати дані про підключених користувачів.

Суть даного методу полягає у слуханні(listening) на деякі події(events), що відправляє користувач.

Таблиця 3.11 – Події Socket.IO

Подія	Опис
“connection”	Відбувається, коли користувач підключається до сервера. У функції зворотного виклику, яка викликається після того як вона відбулась, описані інші події, слухання який розпочинається відразу після підключення клієнту до серверу.
“inituser’	Користувач додається до масиву підключених клієнтів та підключається до всіх кімнат(чатів), які він має.
“disconnect”	Відбувається, коли користувач відключається від сервера. Він відразу видаляється з масиву підключених користувачів.
“chat message”	Відбувається, коли підключений користувач відсилає нове повідомлення серверу. Викликається функція newMessage, визначена у messageListeners, яка генерує подію “new

Продовження таблиці 3.11

Подія	Опис
“chat message”	message”, та відсилає це повідомлення вже усім користувачам відповідної кімнати(чату), крім відправника, а також зберігає його у базі даних. Відправник отримує дату, коли повідомлення було відправлено співрозмовникам.
“new chat”	Відбувається, коли підключений користувач створює новий чат. Автоматично створюється кімната на сервері та відбувається підключення до неї співрозмовників, якщо вони підключені до сервера. Створюється екземпляр відповідного чату у базі даних.

Кінець таблиці 3.11

ВИСНОВОК ДО РОЗДІЛУ 3.

У третьому розділі відбулась розробка клієнтської та серверної частин додатка.

Для реалізації клієнтської частини була використана мова програмування Swift та фреймворк SwiftUI. Були описані обгортки властивостей та стандартні протоколи, які застосовувались при розробці. Створив сервіси для взаємодії між клієнтом та сервером.

Для реалізації серверної частини була використана мова JS та NodeJS. У якості бази даних застосував MongoDB. Були створені запити, які використовуються для отримання даних з бази даних та завантаження їх. Для безперервної комунікації застосував Socket.IO.

РОЗДІЛ 4. АНАЛІЗ РОЗРОБЛЕНОГО ДОДАТКУ.

Після розробки додатку треба впевнитись, що всі компоненти працюють правильно. Тому необхідно провести аналіз та перевірку роботи клієнтської та серверної частин.

4.1 Аналіз серверної частини.

Для перевірки роботи REST API буде використовуватись Postman – платформа, яка дозволяє відправляти запити на сервер та перевіряти правильність його роботи.

4.1.1 Перевірка запитів для user

1. Отримання вибірки усіх користувачів, ім’я яких складається з відповідного виразу.

GET http://localhost:8080/?searchval=Dim

Рисунок 4.1 Результат виконання запиту №1

2. Отримання користувача з відповідним ID.

GET http://localhost:8080/?searchid=132

```
[
    {
        "_id": "62a5decda02baf35b3a0d362",
        "firstName": "dima",
        "lastName": "krshl",
        "username": "dima_krshl"
    }
]
```


Рисунок 4.2 Результат виконання запиту №2

3. Отримання вибірки користувачів за декількома ID.

GET http://localhost:8080/searchids=128,132


```
[
    {
        "_id": "62a5decda02baf35b3a0d362",
        "firstName": "dima",
        "lastName": "krshl",
        "username": "dima_krshl"
    },
    {
        "_id": "62a87a794bb0c7c03e4b7af6",
        "firstName": "User",
        "lastName": "One",
        "username": "userone"
    }
]
```


Рисунок 4.3 Результат виконання запиту №3

4. Перевірка реєстрації користувача.

POST http://localhost:8080/register

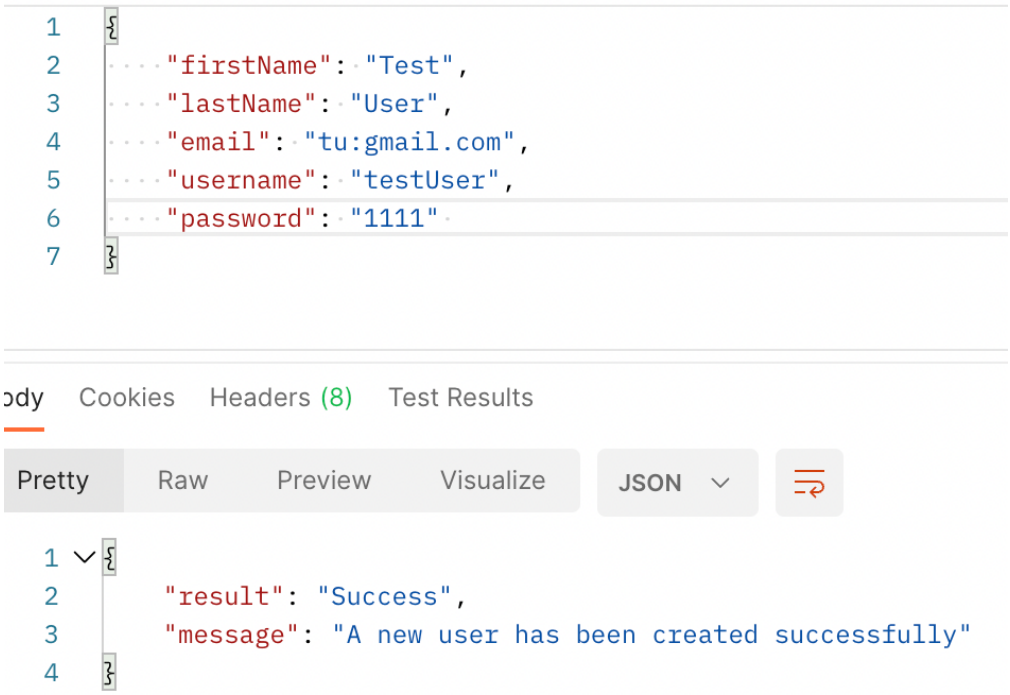


Рисунок 4.4 Результат виконання запиту №4

5. Перевірка авторизації користувача. Проведемо авторизацію для мого користувача.

POST http://localhost:8080/login/

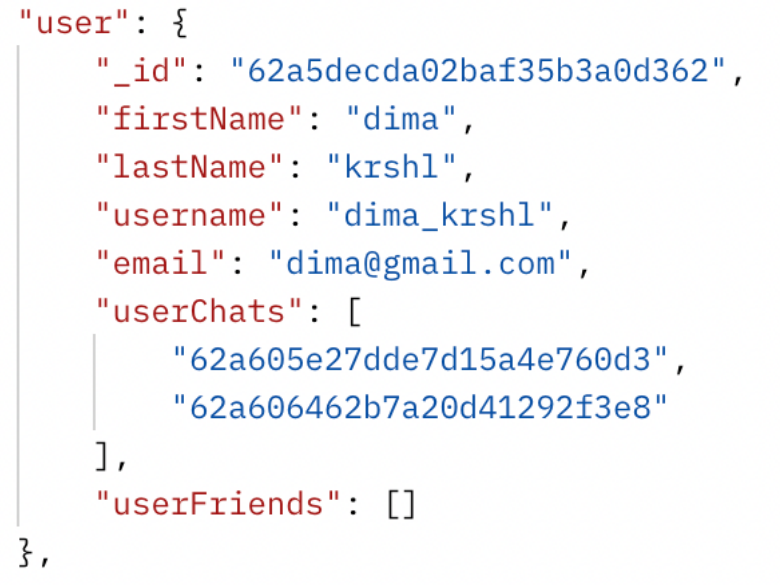


Рисунок 4.5 Результат виконання запиту №5

6. Перевірка оновлення даних користувача. Зміна ім'я користувача userone на user1

PUT http://localhost:8080/updatedata/userone


```
_id: ObjectId("62a87a794bb0c7c03e4b7af6")
firstName: "User"
lastName: "One"
username: "user1"
email: "uo:gmail.com"
password: "$2b$08$2OrXL57nfY7B/AohkLY4L.LZDLhCVmjr0dB/PSJYoSa5.CeoSAKOy"
userChats: null
userFriends: null
```


Рисунок 4.6 Результат виконання запиту №6

7. Перевірка видалення користувача.

DELETE http://localhost:8080/delete/user1


```
{
    "result": "Success",
    "message": "A user has been deleted successfully"
}
```


Рисунок 4.7 Результат виконання запиту №7

4.1.2 Перевірка запитів для chatRoom.

Перевірка отримання даних про чати.

GET http://localhost:8080/loadchats/?id=20220607150957.129

KEY	VALUE
id	62a606462b7a20d41292f3e8
Key	Value

Cookies Headers (8) Test Results

Raw Preview Visualize JSON


```
{
    "chats": [
        {
            "_id": "62a606462b7a20d41292f3e8",
            "chatType": "private",
            "user_ids": [
                "62a5e452f680410415b996f7",
                "62a5decda02baf35b3a0d362"
            ]
        }
    ],
    "message": "All chats have been fetched successfully"
}
```


Рисунок 4.8 Результат виконання запиту

4.1.3 Перевірка запитів для messages.

Отримання повідомлень певного чату.

GET http://localhost:8080/messagesfrom/62a606462b7a20d41292f3e8


```
"messages": [
    {
        "_id": "62a60fe63aa68e21a99084b8",
        "user_id": "62a5e452f680410415b996f7",
        "chat_id": "62a606462b7a20d41292f3e8",
        "message": "Hi",
        "date": "12/06/2022 19:10:14",
        "readby": []
    },
    {
        "_id": "62a87e817f6ba3664ee2e12e",
        "user_id": "62a5decda02baf35b3a0d362",
        "chat_id": "62a606462b7a20d41292f3e8",
        "message": " Hello",
        "date": "14/06/2022 15:26:40",
        "readby": []
    }
],
"message": "Messages have been fetcehd successfully"
```


Рисунок 4.9 Результат виконання запиту

4.2 Аналіз роботи мобільного додатку.

Після першого запуску додатку, користувач попадає на домашній екран, на якому він може бачити 2 поля, я яких він повинен ввести дані свого облікового запису: ім’я та пароль. Після введення своїх даних, він повинен натиснути на кнопку Log In для авторизації.

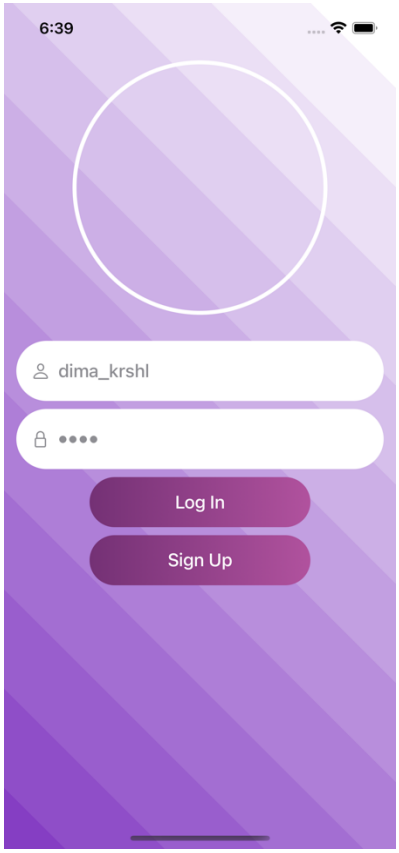


Рисунок 4.10 Домашній екран додатку

Якщо у користувача відсутній обліковий запис, йому потрібно його створити. Для цього необхідно натиснути на кнопку Sign Up. Далі користувач потрапляє на екран, у якому розміщені поля для введення усіх його даних: ім’я, прізвище, ім’я користувача, електронна пошта та пароль.

Після введення всіх даних, для реєстрації, користувачу необхідно натиснути кнопку Sign Up.

Створимо користувача test_user для подальшого тестування роботи додатку.

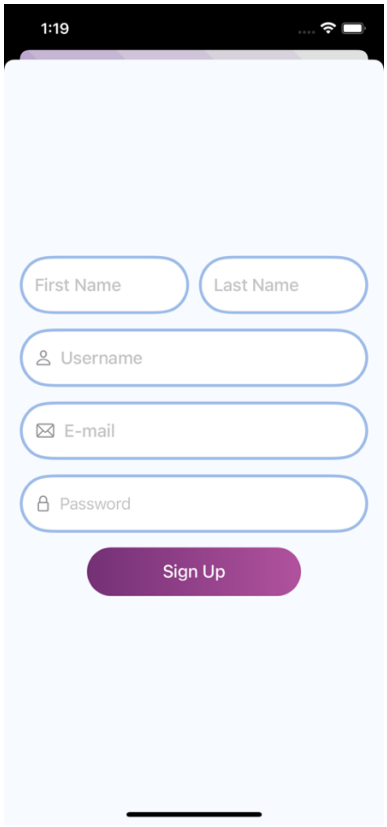


Рисунок 4.11 Вікно для реєстрації

Рисунок 4.12 Вікно MainView

Після реєстрації та авторизації користувач потрапляє у MainView. Для створення першого чату, йому слід перейти на вкладку Profiles, знайти користувача та додати його до списку друзів. Після цього можливий перегляд його облікового запису та створення чату.

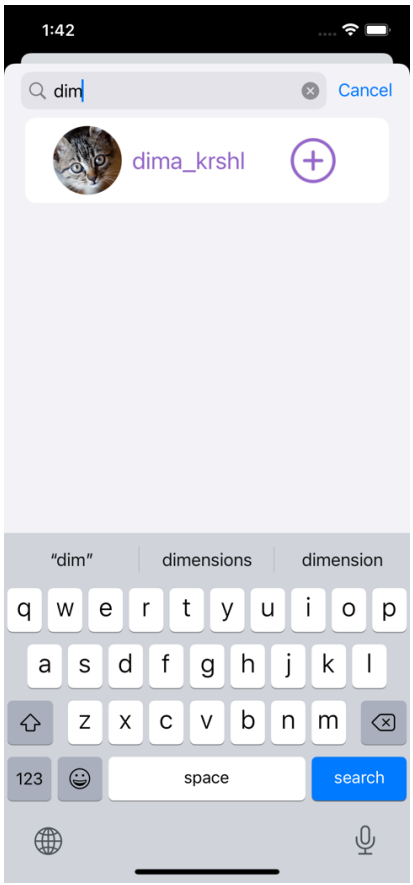


Рисунок 4.13 Вікно пошуку користувачів.

Після натискання на кнопку “+”, даного користувача можна переглядати у списку друзів. Після переходу на його обліковий запис побачимо кнопку “Create chat”, за допомогою якої відбувається створення нового чату. Він відразу додається у базу даних та відображається у обох користувачів у списку чатів, якщо другий користувач також знаходиться в додатку.

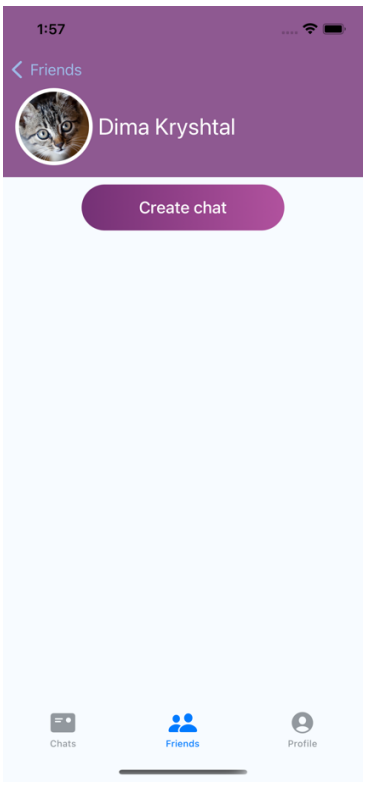


Рисунок 4.14 Профіль друга

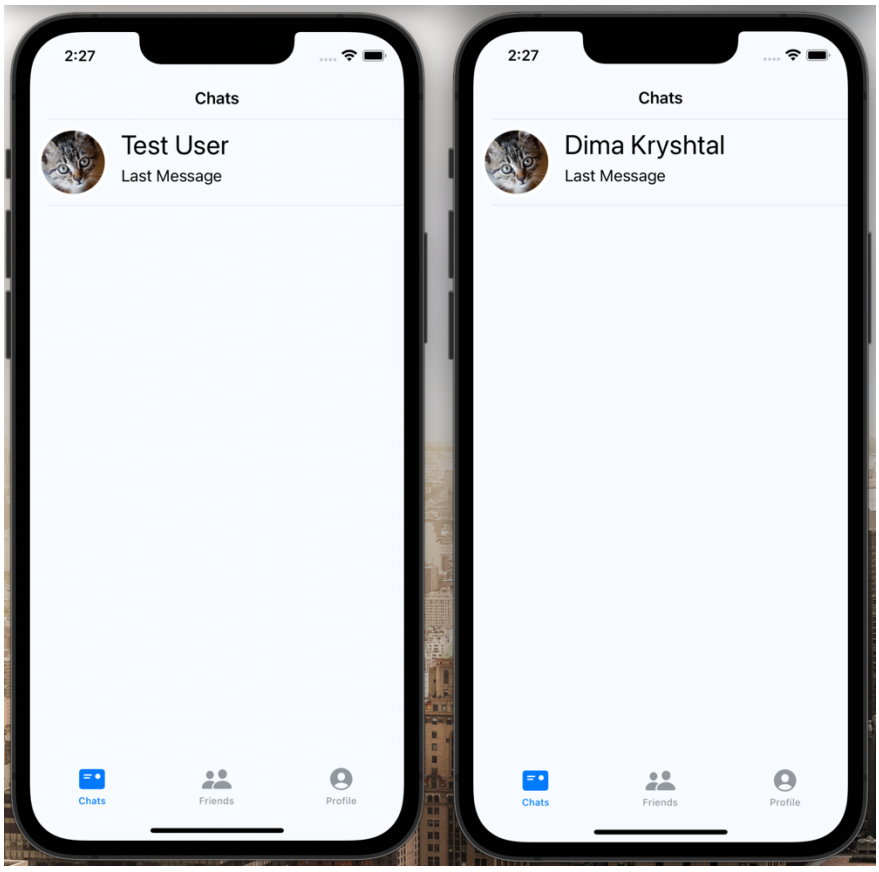


Рисунок 4.15 Екрани співрозмовників після створення чату

Можна перейти до перевірки роботи миттєвого обміну повідомленнями за допомогою Socket.IO. Введемо деяке повідомлення у спеціальне поле. Після натискання на кнопку ми побачимо, що повідомлення з’явилося на екранах відправника і отримувача та поле для введення повідомлення стало порожнім.

Рисунок 4.16 Приклад роботи чату

ВИСНОВОК ДО РОЗДІЛУ 4.

У четвертому розділі була проведена перевірка програми

Спочатку необхідно було перевірити роботу серверної частини. За допомогою Postman відправлялися запити до сервера та той повинен був відповісти певним результатом. Усі результати були очікувані та правильні. Це підтверджують наведені у цьому розділі скриншот.

Також було продемонстровано взаємодію клієнтської частини з серверною: створено користувача, виконано авторизацію та відправлено декілька повідомлень від одного користувача до іншого. Доведено, що даний функціонал програми працює повністю коректно.

ЗАГАЛЬНИЙ ВИСНОВОК.

В ході розробки цієї дипломної роботи був реалізований нативний додаток для миттєвого обміну повідомленнями на мобільній операційній системі iOS. Його робота базується на клієнт-серверній архітектурі.

Спочатку було проведено аналіз предметної області, де відбулось порівняння різних видів мобільних додатків та операційних систем, а також вже наявних месенджерів.

У наступному розділі було розглянуто обрані засоби для реалізації клієнтської та серверної частин додатка.

Для реалізації клієнтської частини була обрана мова програмування Swift та фреймворк SwiftUI.

Для реалізації серверу було обрано NodeJS та не реляційну базу даних MongoDB. У допомозі стали фреймворк ExpressJS та Socket.IO, які значно полегшили розробку.

У третьому розділі проводилась розробка клієнтської та серверної частин додатку. При розробці першої створено інтерфейс інтерфейс та сервіси для комунікації з сервером. Також був розроблений сервер, який надає клієнту необхідну інформацію, та база даних.

В кінці роботи було проведено аналіз розробленого додатку та проведене його тестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Mobile Application (Mobile App) [Електронний ресурс] – Режим доступу до ресурсу: https://www.techopedia.com/definition/2953/mobile-application-mobile-app
2. 5 Key Benefits of Native Mobile App Development [Електронний ресурс] – Режим доступу до ресурсу: https://clearbridgemobile.com/benefits-of-native-mobile-app-development/
3. What is a Web Application? [Електронний ресурс] – Режим доступу до ресурсу: https://blog.stackpath.com/web-application/
4. What are hybrid mobile apps? [Електронний ресурс] – Режим доступу до ресурсу: https://www2.stardust-testing.com/en/blog-en/hybrid-apps
5. What is Client Server Architecture? [Електронний ресурс] – Режим доступу до ресурсу: https://intellipaat.com/blog/what-is-client-server-architecture/
6. Android vs iOS [Електронний ресурс] – Режим доступу до ресурсу: https://www.diffen.com/difference/Android_vs_iOS
7. Mobile Operating System Market Share Worldwide [Електронний ресурс] – Режим доступу до ресурсу: https://gs.statcounter.com/os-market-share/mobile/worldwide
8. iOS vs Android App Development [Електронний ресурс] – Режим доступу до ресурсу: https://www.mooc.org/blog/ios-vs.-android-app-development
9. What is Instant Messaging? [Електронний ресурс] – Режим доступу до ресурсу: https://www.brosix.com/blog/what-is-instant-messaging/#Instant_Messaging_Helps_You_Work_Smarter
10. Telegram [Електронний ресурс] – Режим доступу до ресурсу: https://telegram.org/faq

11. iMessage [Електронний ресурс] – Режим доступу до ресурсу: https://appletoolbox.com/imessage-basic-guide/#5_Send_more_stuff_over_iMessage
12. WhatsApp [Електронний ресурс] – Режим доступу до ресурсу: https://www.cnet.com/tech/services-and-software/12-of-the-best-hidden-whatsapp-features-you-need-to-know/
13. 10 Slack Features That Make It a Perfect Project Management Tool [Електронний ресурс] - Режим доступу до ресурсу: https://blog.hubstaff.com/slack-project-management-2/
14. Swift. [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/swift/
15. TIOBE Index for June 2022. [Електронний ресурс] – Режим доступу до ресурсу: https://www.tiobe.com/tiobe-index/
16. SwiftUI. [Електронний ресурс] – Режим доступу до ресурсу: https://developer.apple.com/xcode/swiftui/
17. What is JavaScript? [Електронний ресурс] – Режим доступу до ресурсу: https://www.infoworld.com/article/3441178/what-is-javascript-the-full-stack-programming-language.html
18. Advantages and disadvantages of JavaScript. [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/advantages-and-disadvantages-of-javascript/
19. PYPL PopularitY Of Programming Language. [Електронний ресурс] – Режим доступу до ресурсу: https://pypl.github.io/PYPL.html
20. NodeJS. [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.dev/learn
21. Event Loop. [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif
22. What is MongoDB? [Електронний ресурс] – Режим доступу до ресурсу: https://www.guru99.com/what-is-mongodb.html

23. What is Rest API? [Електронний ресурс] – Режим доступу до ресурсу: https://www.cleo.com/blog/blog-knowledge-base-what-is-rest-api
24. MVVM Architecture. [Електронний ресурс] – Режим доступу до ресурсу: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1
25. What is NPM? [Електронний ресурс] – Режим доступу до ресурсу: https://www.freecodecamp.org/news/what-is-npm-a-node-package-manager-tutorial-for-beginners/
26. What is Socket.IO? [Електронний ресурс] – Режим доступу до ресурсу: https://socket.io/docs/v4/
27. What is ExpressJS? [Електронний ресурс] – Режим доступу до ресурсу: https://www.simplilearn.com/tutorials/nodejs-tutorial/what-is-express-js?source=sl_frs_nav_playlist_video_clicked
28. Using middleware [Електронний ресурс] – Режим доступу до ресурсу: https://expressjs.com/en/guide/using-middleware.html
29. What is XCode? [Електронний ресурс] – Режим доступу до ресурсу: https://www.zerotoappstore.com/what-is-xcode-and-why-do-i-need-it.html
30. 5 Reasons that Make Visual Studio Code one of the Most Popular Code Editors in 2021[Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/geekculture/5-reasons-that-make-visual-studio-code-one-of-the-most-popular-code-editors-in-2021-d8f95cffed73
31. All SwiftUI property wrappers explained and compared. [Електронний ресурс] – Режим доступу до ресурсу: https://www.hackingwithswift.com/quick-start/swiftui/all-swiftui-property-wrappers-explained-and-compared

ДОДАТОК 1

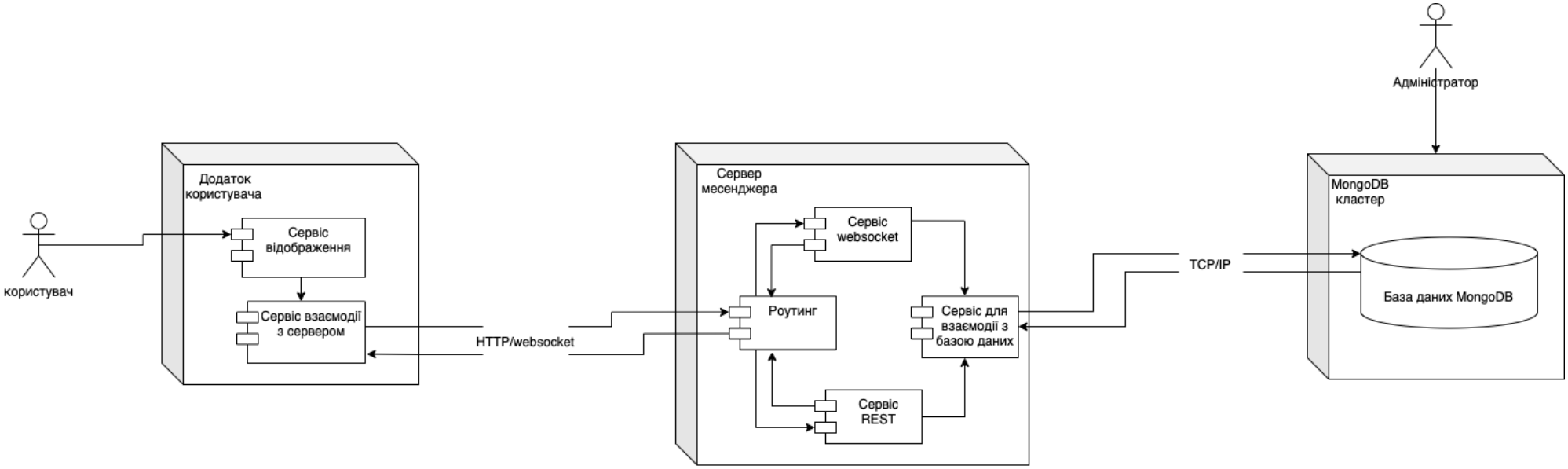
Мобільний додаток на iOS для миттєвого обміну повідомленнями

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.004 Д1
Зм.	Арк.	№ докум.	Підпис	Дата	
Розробив		Кришталь Д.В.			**Мобільний додаток на iOS для миттєвого обміну повідомленнями**
Перевірив		Пономаренко А.М.			Структурна схема системи
Реценз.					
Н. Контр.		Сімоненко В. П.			
Затвердив					

Літ.	Аркуш	Аркушів
	1	1

НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84

ДОДАТОК 2

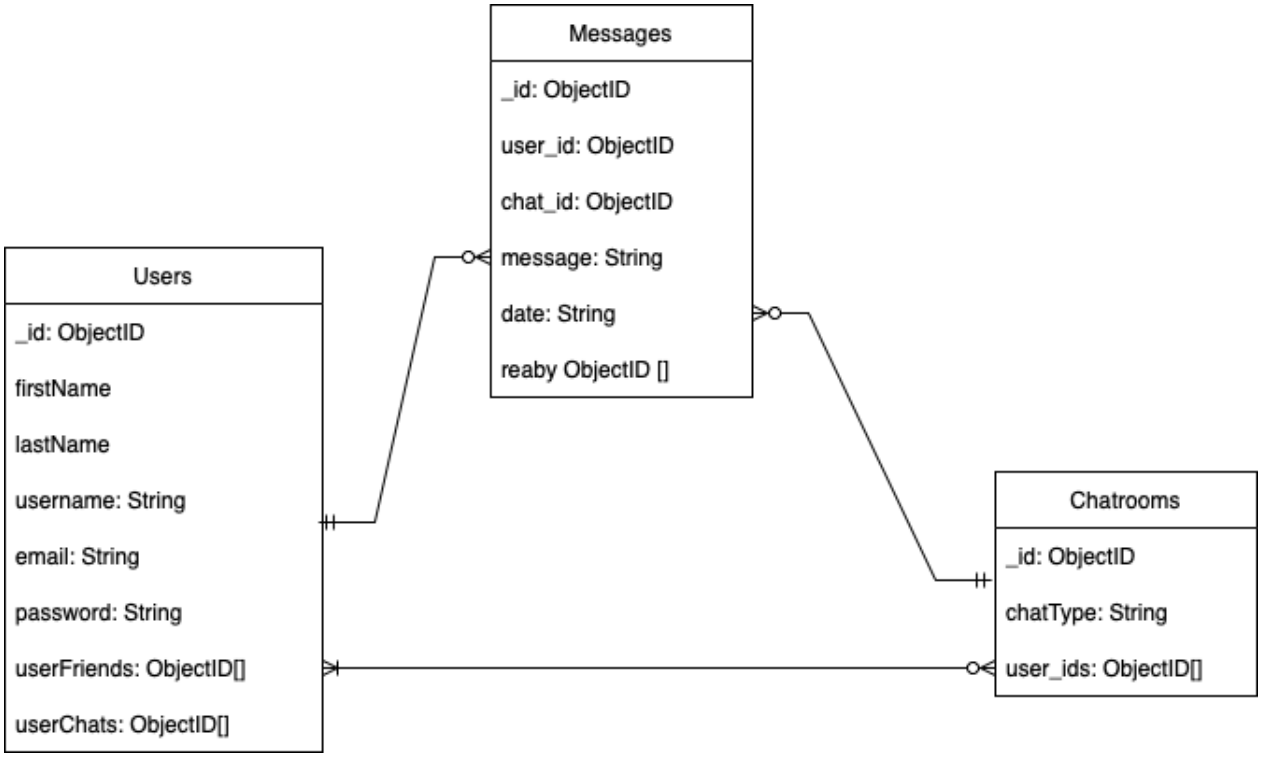
Мобільний додаток на iOS для миттєвого обміну повідомленнями

Функціональна схема

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Кришталь Д.В.			**Мобільний додаток на iOS для миттєвого обміну повідомленнями** Функціональна схема (діаграма даних)	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Пономаренко А.М.					
Реценз.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84	
Н. Контр.		Сімоненко В. П.					
Затвердив							

ДОДАТОК 3

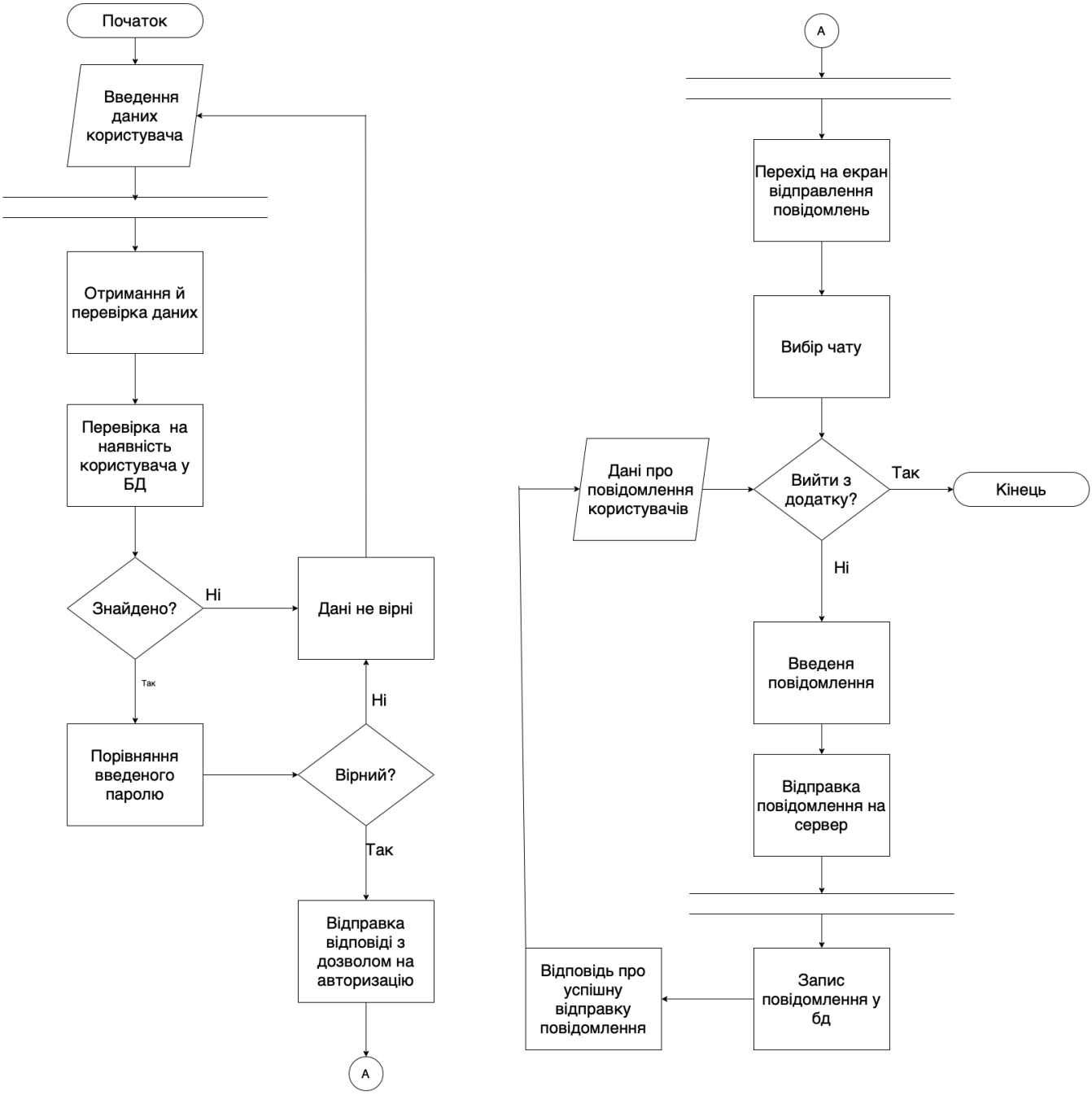
Мобільний додаток на iOS для миттєвого обміну повідомленнями

Принципова схема

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2022 р



					ІАЛЦ.467200.006 Д3	
Зм.	Арк.	№ докум.	Підпис	Дата		
Розробив		Кришталь Д.В.			**Мобільний додаток на iOS для миттєвого обміну повідомленнями** Принципова схема системи	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Пономаренко А.М.				
Реценз.						НТУУ КПІ ім. Ігоря Сікорського, ФІОТ, ІП-84
Н. Контр.		Сімоненко В. П.				
Затвердив						

ДОДАТОК 4

Мобільний додаток на iOS для миттєвого обміну повідомленнями

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 44

Київ 2022 р

ChatAppApp.swift

```
import SwiftUI

 @main
struct ChatAppApp: App {
   var body: some Scene {
      WindowGroup {
         ContentView()
      }
   }
}
```

ContentView.swift

```
import SwiftUI

struct ContentView: View {
   @StateObject private var authenticaion = Authentication()

   var body: some View {
      if (authenticaion.homeWindowIsShown == true) {
         HomeWindow(auth: authenticaion)
      } else {
         MainView()
            .environmentObject(authenticaion)
      }

   }
}

struct ContentView_Previews: PreviewProvider {
   static var previews: some View {
      ContentView()
   }
}
```

Background.swift

```
import SwiftUI

struct Background: View {
   @Binding var showBackground: Bool

   var parallelogramFrameHeight: CGFloat
   var sectionAngle: Double
   var screenWidth: CGFloat
   var screenHeight: CGFloat

   var body: some View {
      let sectionThickness: CGFloat = 70
      let offsetDeltaY = sectionThickness / sin(deg2rad(90.0 - sectionAngle))
      let offsetDeltaX = sectionThickness / cos(deg2rad(90.0 - sectionAngle))
      let positionX = screenWidth * 5 / 4

      let numberOfSections =  Int(screenWidth / offsetDeltaX + screenHeight / offsetDeltaY)
      ZStack {
         ForEach((0...numberOfSections), id: \.self) { index in
            BackgroundSection(
               shorterSide: screenWidth,
               sectionThickness: sectionThickness,
               angle: sectionAngle,
               color: .primaryColorPurple.opacity(Double(index) / Double(numberOfSections))
            )

            .position(x: -screenWidth * 3 / 4 + (showBackground ? positionX : 0),
                  y: -160 + (offsetDeltaY * Double(index)) + (showBackground ? screenWidth: 0) -
(screenWidth*tan(deg2rad(sectionAngle))))

            .animation(.ripple(index: index, backgroundIsShown: showBackground, numberOfSections:
numberOfSections).speed(0.8))
         }
      }

      .onAppear {
         showBackground = true
      }

      .ignoresSafeArea()

   }
}

struct Background_Previews: PreviewProvider {

   static var previews: some View {
      Background(showBackground: .constant(true),
            parallelogramFrameHeight: 390,
            sectionAngle: 45,
            screenWidth: 390,
            screenHeight: 844)
```

```
            .previewDevice(PreviewDevice(rawValue: "iPhone 13"))
            .previewDisplayName("iPhone 13")
    }
}
```

BackgroundSection.swift

```
import SwiftUI

func deg2rad(_ number: Double) -> Double {
    return number * .pi / 180
}

struct BackgroundSection: View {
    var shorterSide: CGFloat
    var sectionThickness: CGFloat
    var angle: Double
    var color: Color


    var body: some View {
        let length = shorterSide / sin(deg2rad(90 - angle))
        Rectangle()
            .fill(color)
            .frame(width: length * 1.2, height: sectionThickness, alignment: .leading)
            .rotationEffect(Angle(degrees: angle))

    }
}

struct BackgroundSection_Previews: PreviewProvider {
    static var previews: some View {
        BackgroundSection(shorterSide: 390, sectionThickness: 50, angle: 45, color: Color(0x853EC3, alpha: 0.1))

    }
}
```

SignUpViewModel.swift

```
import Foundation
import Combine
import SwiftUI

extension SignUp {
    class RegisterViewModel: ObservableObject {
        @Published var showingAlert: Bool = false
        @Published var authenticator: String = ""
        @Published var email: String = ""
        @Published var password: String = ""
        @Published var firstName: String = ""
        @Published var lastName: String = ""
```

```
    private var subscribers = Set<AnyCancellable>()

    func sendNewUser(user: User) {
      UserNetworkManager.shared.createNewUser(requestBody: user)
        .receive(on: DispatchQueue.main)
        .sink(receiveCompletion: { response in
          switch response {
          case .failure:
            self.showingAlert = true
            return
          case .finished:
            print("Succeesfully finished!")
          }
        }, receiveValue: { value in print ("\(value)")
        })
        .store(in: &subscribers)
    }
  }
}
```

HomeWindowViewModel.swift

```
import Foundation
import Combine
import SwiftUI

extension HomeWindow {
  class HomeWindowViewModel: ObservableObject {
    @Published var showBackground = false
    @Published var showingAlert = false
    @Published var authenticator: String = "dima_krshl"
    @Published var password: String = "2508"
    @Published var showSignUp = false
    @Published var showButtons = false

    var authentication: Authentication

    init(auth: Authentication) {
      authentication = auth
    }

    private var subscribers = Set<AnyCancellable>()

    func injection(auth: Authentication) {
      self.authentication = auth

    }



    func showInterfaceButtons() {
      DispatchQueue.main.async {
        withAnimation(.easeInOut(duration: 1).delay(2)) {
          self.showButtons.toggle()
```

```
            }
        }
    }

    func successfullLogin() {
        withAnimation {
            showButtons = false
        }
        showBackground = false
    }

    func hideHomeWindow() {
        DispatchQueue.main.asyncAfter(deadline: .now() + 2) {
            withAnimation {
                self.authentication.changeHomeWindowStatus()
            }
        }
    }

    func logIn() {
        if (authenticator == "") {
            self.successfullLogin()
            UserData.shared.saveData(user: User.defaultUser)
            self.authentication.changeStatus()
            self.hideHomeWindow()
        } else {
            UserNetworkManager.shared.login(requestBody: ["username": authenticator, "password":
password])
                .receive(on: DispatchQueue.main)
                .sink(receiveCompletion: { response in
                    print(response)
                    switch response {
                    case .failure:
                        print("\n\n\n\n \(response)")
                        self.showingAlert = true
                        return
                    case .finished:
                        self.successfullLogin()
                        self.authentication.changeStatus()
                        self.hideHomeWindow()
                        return

                    }
                }, receiveValue: {
                    value in print ("\(value)")
                    UserData.shared.saveData(user: value.user)
                })
                .store(in: &subscribers)
        }
    }
  }
}
```

ChatViewModel.swift

```
import Foundation

extension Chat {
  class ChatViewModel: ObservableObject {
    @Published var newMessageValue = ""
    @Published var sendIsTapped = false

    var chatData: ChatRoom
    var messages: MessagesData
    var isJustCreated: Bool

    init(chatRoom: ChatRoom, messages: MessagesData) {
      self.chatData = chatRoom
      self.messages = messages
      isJustCreated = false
    }

    func sendMessage() {
      if newMessageValue == "" { return }

      let userId = UserData.shared.getUserId()

      let message = Message(id: "temp_id",
                            user_id: userId,
                            chat_id: chatData.id,
                            message: newMessageValue,
                            pending: true)

      newMessageValue = ""
      sendIsTapped = true

      let newMessageIndex = (messages.messages[chatData.id] ?? []).count

      messages.messages[chatData.id]?.append(message)
      DispatchQueue.main.asyncAfter(deadline: .now() + 0.2) {
        self.sendIsTapped = false
      }
      Socket.shared.sendMessage(message: message) { id, date in
        self.messages.messages[self.chatData.id]![newMessageIndex].id = id
        self.messages.messages[self.chatData.id]![newMessageIndex].date = date
        self.messages.messages[self.chatData.id]![newMessageIndex].pending = nil
      }
    }
  }
}
```

FindFriendsViewModel.swift

```
import Foundation
import Combine

extension FindFriends {
```

```
    class ViewModel: ObservableObject {
        @Published var foundUsers: [User] = []
        @Published var searchText = "" {
            didSet {
                searchResults()
            }
        }
        private var subscribers = Set<AnyCancellable>()


        func searchResults(){
            if searchText.count >= 3 {
                UserNetworkManager.shared.getUsers(query: "?searchval=\(searchText)")
                    .receive(on: DispatchQueue.main)
                    .sink(receiveCompletion: { response in
                        print(response)
                    }, receiveValue: {value in print ("\(value)")
                        print(value)
                        self.foundUsers = value
                        print(value)
                    })
                    .store(in: &subscribers)
            } else {
                self.foundUsers = []
            }
        }
    }
}
```

ChatRowViewModel.swift

```
import Foundation
import Combine

extension ChatRow {
    class ViewModel: ObservableObject {
        var subscriptions = Set<AnyCancellable>()
        var chat: ChatRoom

        @Published var chatName: String?

        init(chatRoom: ChatRoom) {
            self.chat = chatRoom
        }

        func getChatName() {
            let user_id = chat.user_ids.first(where: {$0 != UserData.shared.getUserId()})
            UserNetworkManager.shared.getUsers(query: "?searchid=\(user_id!)")
                .receive(on: DispatchQueue.main)
                .sink(receiveCompletion: { response in
                    print("\(response)")
                }, receiveValue: {value in
                    print("\n\n\n\n\(UserData.shared.getUserChats())")
                    self.chatName = "\(value[0].firstName) \(value[0].lastName)"
```

```
                self.chat.chatName = "\(value[0].firstName) \(value[0].lastName)"
            })
            .store(in: &subscriptions)

        }
    }
}
```

SocketManager.swift

```
import Foundation
import SocketIO
import SwiftUI

struct UserInit: SocketData {
    var user_id: String
    var chats: [String]

    func socketRepresentation() -> SocketData {
        return ["user_id": user_id, "chats": chats]
    }
}

struct MessageSocketData: SocketData {
    var user_id: String
    var chat_id: String
    var message: String

    func socketRepresentation() throws -> SocketData {
        return ["user_id": user_id, "chat_id": chat_id, "message": message];
    }
}

struct ChatSocketData: SocketData {
    var chatType: String
    var user_ids: [String]

    func socketRepresentation() throws -> SocketData {
        return ["chatType": chatType, "user_ids": user_ids]
    }
}

class Socket {
    static let shared = Socket()
    var manager: SocketManager
    var socket:SocketIOClient
    var chats: ChatsData?
    var messages: MessagesData?

    init () {
        manager = SocketManager(socketURL: URL(string: "http://localhost:8080")!, config: [.log(true),
.compress])
        socket = manager.defaultSocket
    }
```

```
}


extension Socket {

   func initChats(chats: ChatsData, messages: MessagesData) {
      self.chats = chats
      self.messages = messages

   }

   func connectToSocketServer() {
      let user = UserInit(user_id: UserData.shared.getUserId(),
                          chats: UserData.shared.getUserChats() ?? [])



      socket.on(clientEvent: .connect) { [self]data, ack in
         print("socket connected")
         socket.emit("inituser", user)
      }
      socket.on("new message") { data, ack in
         if let arr = data as? [[String: Any]],
            let id = arr[0]["_id"] as? String,
            let chat_id = arr[0]["chat_id"] as? String,
            let message = arr[0]["message"] as? String,
            let user_id = arr[0]["user_id"] as? String {
             self.messages?.messages[chat_id]?.append(Message(id: id, user_id: user_id, chat_id: chat_id,
message: message))
         }

      }

      socket.on("chat created") { data, ack in
         if let arr = data as? [[String: Any]],
            let id = arr[0]["id"] as? String,
            let chatType = arr[0]["chatType"] as? String,
            let user_ids = arr[0]["user_ids"] as? [String] {
             self.chats?.chats.append(ChatRoom(id: id, chatName: nil, chatType: chatType, user_ids: user_ids))
             self.messages?.messages[id] = []
         }
      }

      socket.connect()
   }

   func sendMessage( message: Message, completion: @escaping (String, String) -> Void){
      let messageSocketData = MessageSocketData(user_id: message.user_id, chat_id: message.chat_id,
message: message.message);

      socket.emitWithAck("chat message", messageSocketData).timingOut(after: 1) { data in
         if let messageObj = data as? [[String: Any]],
            let id = messageObj[0]["id"] as? String,
            let date = messageObj[0]["date"] as? String {
```

```
            completion(id, date)
            print(messageObj)
          }

        }
    }

    func createNewChat(chat: ChatRoom, completion: @escaping(String) -> Void) {
        let chatSocketData = ChatSocketData(chatType: chat.chatType, user_ids: chat.user_ids)

        socket.emitWithAck("new chat", chatSocketData).timingOut(after: 1) { data in
            if let chatObj = data as? [[String: Any]],
               let id = chatObj[0]["id"] as? String {
                completion(id)
                print("\n\n\n \(id)")
            }
        }
    }

    func disconnect() {
        socket.disconnect()
    }
}
```

UserNetworkManager.swift

```
import Foundation
import Combine

protocol NetworkManagerDelegate {
    func createNewUser(requestBody: User) -> AnyPublisher <SuccessResponse, Error>
    func login(requestBody: [String: String]) -> AnyPublisher<LoginResponse, Error>
}

final class UserNetworkManager {
    static let shared = UserNetworkManager()
    let encoder = JSONEncoder()
    let decoder = JSONDecoder()

    init() {
        encoder.outputFormatting = .prettyPrinted
    }
}

extension UserNetworkManager: NetworkManagerDelegate {
    enum NetworkError: Error {
      case parsing(description: String)
      case network(description: String)
    }

    func getUsers(query: String) -> AnyPublisher<[User], Error> {
        var urlString = "http://localhost:8080/\(query)"
```

```
//      print(urlString)
        guard let components = URLComponents(string: urlString), let url = components.url else {
            return Fail(error:  NetworkError.network(description: "Error creating user")).eraseToAnyPublisher()
        }

        let request = URLRequest(url: url)

        return URLSession.shared.dataTaskPublisher(for: request)
            .print()
            .map(\.data)
            .decode(type: [User].self, decoder: decoder)
            .receive(on: RunLoop.main)
            .eraseToAnyPublisher()

    }

    func updateCurrentUser<T: Encodable>(username: String, requestBody: [String: T]) ->
AnyPublisher<SuccessResponse, Error> {
        let urlString = "http:/localhost:8080/updatedata/\(username)"

        guard let components = URLComponents(string: urlString), let url = components.url else {
            return Fail(error:  NetworkError.network(description: "Error creating user")).eraseToAnyPublisher()
        }

        var request = URLRequest(url: url)

        request.httpMethod = "PUT"
        request.addValue("application/json", forHTTPHeaderField: "Content-Type")
        request.addValue("application/json", forHTTPHeaderField: "Accept")
        request.httpBody = try? encoder.encode(requestBody)

        return URLSession.shared.dataTaskPublisher(for: request)
          .map(\.data)
          .decode(type: SuccessResponse.self, decoder: decoder)
          .eraseToAnyPublisher()
        }

    //REGISTER
    func createNewUser(requestBody: User) -> AnyPublisher<SuccessResponse, Error> {
          let urlString = "http:/localhost:8080/register"

          guard let components = URLComponents(string: urlString), let url = components.url else {
              return Fail(error:  NetworkError.network(description: "Error creating user")).eraseToAnyPublisher()
          }
          var request = URLRequest(url: url)
          request.httpMethod = "POST"
          request.addValue("application/json", forHTTPHeaderField: "Content-Type")
          request.addValue("application/json", forHTTPHeaderField: "Accept")
          request.httpBody = try? encoder.encode(requestBody)


           return URLSession.shared.dataTaskPublisher(for: request)
            .map(\.data)
```

```
            .decode(type: SuccessResponse.self, decoder: decoder)
            .eraseToAnyPublisher()
      }
    //LOGIN
    func login(requestBody: [String: String]) -> AnyPublisher<LoginResponse, Error> {
      let urlString = "http:/localhost:8080/login"

      guard let components = URLComponents(string: urlString), let url = components.url else {
        return Fail(error:  NetworkError.network(description: "Error validating the
user")).eraseToAnyPublisher()
      }

      var request = URLRequest(url: url)
      request.httpMethod = "POST"
      request.addValue("application/json", forHTTPHeaderField: "Content-Type")
      request.addValue("application/json", forHTTPHeaderField: "Accept")
      request.httpBody = try? encoder.encode(requestBody)

      return URLSession.shared.dataTaskPublisher(for: request)
        .print()
        .map(\.data)
        .decode(type: LoginResponse.self, decoder: JSONDecoder())
        .eraseToAnyPublisher()

    }
}
```

ChatRoomNetworkManager.swift

```
import Foundation
import Combine

protocol ChatRoomsNetworkManagerDelegate {
    func loadUserChatRooms(arrayOfChatIDs: [String]) -> AnyPublisher <ChatRoomsResponse, Error>
}

final class ChatRoomNetworkManager {
    static let shared = ChatRoomNetworkManager()
    let encoder = JSONEncoder()
    let decoder = JSONDecoder()

    init() {
      encoder.outputFormatting = .prettyPrinted
    }
}

extension ChatRoomNetworkManager: ChatRoomsNetworkManagerDelegate {
    enum NetworkError: Error {
      case parsing(description: String)
      case network(description: String)
    }

    func loadUserChatRooms(arrayOfChatIDs: [String]) -> AnyPublisher<ChatRoomsResponse, Error> {
```

```
        let urlString = "http:/localhost:8080/loadchats?id=\(arrayOfChatIDs.map({String($0)}).joined(separator:
",","))"

        guard let components = URLComponents(string: urlString), let url = components.url else {
            return Fail(error:  NetworkError.network(description: "Error validating the
user")).eraseToAnyPublisher()
        }

        let request = URLRequest(url: url)

        return URLSession.shared.dataTaskPublisher(for: request)
            .map(\.data)
            .decode(type: ChatRoomsResponse.self, decoder: decoder)
            .eraseToAnyPublisher()
    }
}
```

MessagesNetworkManager.swift

```
import Foundation
import Combine

protocol MessagesNetworkManagerDelegate {
    func loadMessages(chat_id: String) -> AnyPublisher<MessagesResponse, Error>
}

final class MessagesNetworkManager {
    static let shared = MessagesNetworkManager()

    let decoder = JSONDecoder()
}

extension MessagesNetworkManager:  MessagesNetworkManagerDelegate {
    enum NetworkError: Error {
      case parsing(description: String)
      case network(description: String)
    }
    func loadMessages(chat_id: String) -> AnyPublisher<MessagesResponse, Error> {
        let urlString = "http://localhost:8080/messagesfrom/\(chat_id)"

        guard let components = URLComponents(string: urlString), let url = components.url else {
            return Fail(error: NetworkError.network(description: "Error")).eraseToAnyPublisher()
        }

        let request = URLRequest(url: url)

        return URLSession.shared.dataTaskPublisher(for: request)
            .map(\.data)
            .decode(type: MessagesResponse.self, decoder: decoder)
            .eraseToAnyPublisher()
    }
}
```

Authentication.swift

```
import Foundation

class Authentication: ObservableObject {
    @Published var homeWindowIsShown: Bool
    @Published var isLoggedIn: Bool {
        didSet {
            UserDefaults.standard.set(isLoggedIn, forKey: "isLoggedIn")
            print(isLoggedIn)
        }
    }

    init() {
        isLoggedIn = UserDefaults.standard.bool(forKey: "isLoggedIn")
        homeWindowIsShown = true
    }

    func changeStatus() {
        isLoggedIn.toggle()
    }

    func changeHomeWindowStatus() {
        homeWindowIsShown.toggle()
    }
}
```

UserData.swift

```
import Foundation

protocol UserDataFunctions {
    func saveData(user: User?) -> Void
    func getUserId() -> String
    func getUserName() -> String
    func getName() -> String
    func getLastName() -> String
    func getUserChats() -> [String]
    func getUserFriends() -> [String]
}

final class UserData {
    static var shared = UserData()
}

extension UserData: UserDataFunctions {
    func getUserId() -> String {
        return UserDefaults.standard.string(forKey: "User_id") ?? "Unknown id"
    }

    func getUserName() -> String {
        return UserDefaults.standard.string(forKey: "Username") ?? "Unknown username"
    }
```

```
    func getName() -> String {
        return UserDefaults.standard.string(forKey: "FirstName") ?? "Unknown first name"
    }

    func getLastName() -> String {
        return UserDefaults.standard.string(forKey: "LastName") ?? "Unknown last name"
    }

    func getUserChats() -> [String] {
        return UserDefaults.standard.value(forKey: "UserChats") as! [String]
    }


    func getUserFriends() -> [String] {
        return UserDefaults.standard.value(forKey: "UserFriends") as! [String]
    }

    func saveData(user: User?) {
        UserDefaults.standard.set(user?.id, forKey: "User_id")
        UserDefaults.standard.set(user?.username, forKey: "Username")
        UserDefaults.standard.set(user?.firstName, forKey: "FirstName")
        UserDefaults.standard.set(user?.lastName, forKey: "LastName")
        UserDefaults.standard.set(user?.userChats, forKey: "UserChats")
        UserDefaults.standard.set(user?.userFriends, forKey: "UserFriends")
    }

}
```

ChatsData.swift

```
import Foundation
import Combine

final class ChatsData: ObservableObject {
    @Published var chats: [ChatRoom] = []

    var cancellable = Set<AnyCancellable>()

    init(chat_ids: [String]) {
        print("\n\n\n \(chat_ids)")
        if(UserData.shared.getUserName() != "") {
            ChatRoomNetworkManager.shared.loadUserChatRooms(arrayOfChatIDs: chat_ids)
                .receive(on: DispatchQueue.main)

                .sink(receiveCompletion: { response in
                    print(response)
                }, receiveValue: { [self] data in

                    self.chats = data.chats
                })
                .store(in: &cancellable)
        }
    }
```

```
}
```

MessagesData.swift

```
import Foundation
import Combine

final class MessagesData: ObservableObject {
   @Published var messages: [String: [Message]] = [:]{
      didSet {
         print("arrayChanged")
      }
   }


   var cancellable = Set<AnyCancellable>()

   init(chat_ids: [String]) {
      for (chat_id) in chat_ids {
         MessagesNetworkManager.shared.loadMessages(chat_id: chat_id)
            .receive(on: DispatchQueue.main)
            .sink(receiveCompletion: { response in
               print("\(response)")
            }, receiveValue: { [self] messages in
               self.messages[chat_id] = messages.messages
               print("\n\n\n \(messages)")
            })
            .store(in: &cancellable)
      }
   }
}
```

FriendsData.swift

```
import Foundation
import Combine

final class FriendsData: ObservableObject {
   var subscribers = Set<AnyCancellable>()
   @Published var friends: [User] = []

   init(friendsIds: [String]) {
      if friendsIds.count != 0 {
         UserNetworkManager.shared.getUsers(query: "?searchids=\(friendsIds.map{ String($0)
}.joined(separator: ", "))")
            .receive(on: DispatchQueue.main)
            .sink(receiveCompletion: { response in
               print(response)
            }, receiveValue: { value in
               print("\n\n\n\(value)")
               self.friends = value
            })
```

```
            .store(in: &subscribers)
        }
    }

    func addItem (item: User) {
        print("\n\n\n\n \(item)")
        self.friends.append(item)
        UserNetworkManager.shared.updateCurrentUser(username: UserData.shared.getUserName(),
                                    requestBody: ["$push": ["userFriends" : item.id]])
        .receive(on: DispatchQueue.main)
        .sink(receiveCompletion: { response in
            switch response {
            case .failure:
                return
            case .finished:
                print("Succeesfully finished!")
            }
        }, receiveValue: { value in print ("\(value)")
        })
        .store(in: &subscribers)
    }

}
```

LoginResponse.swift

```
import Foundation

struct LoginResponse: Codable {
    var user: User
    var token: String
    var refreshToken: String
}
```

CreateUserResponse.swift

```
import Foundation

struct SuccessResponse: Codable {
    var result: String
    var message: String
}
```

ChatRoomResponse.swift

```
import Foundation

struct ChatRoomsResponse: Codable {
    var chats: [ChatRoom]
    var message: String
}
```

LoadMessagesResponse.swift

```
import Foundation

struct MessagesResponse: Codable {
    var messages: [Message]
    var message: String
}
```

User.swift

```
import Foundation

struct User: Codable, Hashable, Identifiable {
    var id: String?
    var firstName: String
    var lastName: String
    var username: String
    var email: String?
    var password: String?
    var userChats: [String]?
    var userFriends: [String]?

    private enum CodingKeys: String, CodingKey {
        case id = "_id"
        case firstName
        case lastName
        case username
        case email
        case password
        case userChats
        case userFriends
    }

    static var defaultUser = User(id: "1111",
                                  firstName: "",
                                  lastName: "",
                                  username: "",
                                  email: "",
                                  password: "",
                                  userChats: ["00000.0"],
                                  userFriends: [])
}
```

Message.swift

```
import Foundation

struct Message: Codable, Identifiable, Equatable {
    var id: String
    var user_id: String
    var chat_id: String
    var message: String
    var date: String?
```

```
    var pending: Bool?

    private enum CodingKeys: String, CodingKey {
        case id = "_id"
        case user_id
        case chat_id
        case message
        case date
        case pending
    }

    static var defaultMessage = Message(id: "000000000.0.0", user_id: "1111", chat_id: "00000.0", message:
"test")

    mutating func changeMessageStatus() {
        pending = nil
    }
}
```

ChatRoom.swift

```
import Foundation

struct Message: Codable, Identifiable, Equatable {
    var id: String
    var user_id: String
    var chat_id: String
    var message: String
    var date: String?
    var pending: Bool?

    private enum CodingKeys: String, CodingKey {
        case id = "_id"
        case user_id
        case chat_id
        case message
        case date
        case pending
    }

    static var defaultMessage = Message(id: "000000000.0.0", user_id: "1111", chat_id: "00000.0", message:
"test")

    mutating func changeMessageStatus() {
        pending = nil
    }
}
```

FieldWithImage.swift

```
import SwiftUI

func rect(lineWidth: CGFloat) -> some View {
```

```
    return RoundedRectangle(cornerRadius: 37, style: .continuous)
        .strokeBorder(Color.primaryColorBlue, lineWidth: lineWidth)
        .background(RoundedRectangle(cornerRadius: 37)
        .fill(.white))

}

struct FieldWithImage: View {
    var imageName: String?
    var fieldName: String
    var lineWidth: CGFloat = 0
    @Binding var textValue: String
    var body: some View {
        HStack{
            if let imageName = imageName {
                Image(systemName: imageName)
                    .foregroundColor(.gray)
                    .aspectRatio(contentMode: .fit)

            }
            TextField(fieldName, text: $textValue)
                .foregroundColor(.gray)
                .font(.system(size: 18, weight: .medium, design: .default))
                .frame(maxWidth: .infinity, maxHeight: .infinity)
        }
        .padding(.leading, 15)
        .background(rect(lineWidth: lineWidth))

    }
}

struct FieldWithImage_Previews: PreviewProvider {
    static var previews: some View {
        FieldWithImage(imageName: "lock", fieldName: "UserName", lineWidth: 3, textValue: .constant("Test"))
            .previewLayout(.fixed(width: 390, height: 50))
    }
}
```

SecretFieldWithImage.swift

```
import SwiftUI

struct SecretFieldWithImage: View {
    var fieldName: String
    var lineWidth: CGFloat = 0
    @Binding var textValue: String
    var body: some View {
        HStack{
            Image(systemName: "lock")
                .foregroundColor(.gray)
                .aspectRatio(contentMode: .fit)

            SecureField(fieldName, text: $textValue)
                .foregroundColor(.gray)
```

```
            .frame(maxWidth: .infinity, maxHeight: .infinity)
        }
        .padding(.leading, 15)
        .background(rect(lineWidth: lineWidth))

    }
}

struct SecretFieldWithImage_Previews: PreviewProvider {
    static var previews: some View {
        SecretFieldWithImage(fieldName: "Password", textValue: .constant("test"))
    }
}
```

CustomStyles.swift

```
import Foundation
import SwiftUI

struct CustomButtonStyle: ButtonStyle {
    func makeBody(configuration: Configuration) -> some View {
        configuration.label
            .padding()
            .frame(width: 220, height: 50)
            .background(LinearGradient(colors: [Color.gradientColor1, Color.gradientColor2], startPoint: .leading,
endPoint: .trailing))
            .foregroundColor(.white)
            .font(.system(size: 18, weight: .medium, design: .default))
            .clipShape(Rectangle())
            .cornerRadius(30)
    }
}

struct CustomListButtonStyle: ButtonStyle {
    func makeBody(configuration: Configuration) -> some View {
        configuration.label
            .frame(maxWidth: .infinity, maxHeight: .infinity, alignment: .center)
            .contentShape(Rectangle())
            .background(Color.black.opacity(configuration.isPressed ? 0.1 : 0.0))
    }
}
```

CircleImage.swift

```
import SwiftUI

struct CircleImage: View {
    var image: Image

    var body: some View {
        image
            .resizable()
            .scaledToFill()
```

```
            .clipShape(Circle())
            .overlay {
                Circle().stroke(.white, lineWidth: 4)
            }
//          .shadow(color: .primaryColorPurple, radius: 2)
    }
}

struct CircleImage_Previews: PreviewProvider {
    static var previews: some View {
        CircleImage(image: Image("kitty"))
            .frame(width: 200, height: 200, alignment: .center)
    }
}
```

ChatBubble.swift

```
import SwiftUI

struct CircleImage: View {
    var image: Image

    var body: some View {
        image
            .resizable()
            .scaledToFill()
            .clipShape(Circle())
            .overlay {
                Circle().stroke(.white, lineWidth: 4)
            }
//          .shadow(color: .primaryColorPurple, radius: 2)
    }
}

struct CircleImage_Previews: PreviewProvider {
    static var previews: some View {
        CircleImage(image: Image("kitty"))
            .frame(width: 200, height: 200, alignment: .center)
    }
}
```

TextView.swift

```
struct TextView: View {
    var text: String
    var width: CGFloat?
    var height: CGFloat?

    var body: some View {
        Text(text)
            .padding()
            .frame(maxWidth: width, maxHeight: height, alignment: .leading)
            .background(rect(lineWidth: 0))
            .font(.system(size: 20, weight: .medium, design: .default))
```

```
            .foregroundColor(Color.primaryColorBlue)
    }
}

struct TextView_Previews: PreviewProvider {
    static var previews: some View {
        TextView(text: "Test", width: .infinity, height: 50)
    }
}
```

Friends.swift

```
import SwiftUI

struct Friends: View {
    @EnvironmentObject var friendsData: FriendsData
    @State private var showingSheet = false


    var body: some View {
        NavigationView {
            ZStack {
                Color.blue.opacity(0.03)
                    .ignoresSafeArea()
                Group {
                    if friendsData.friends.count != 0 {
                        List {
                            ForEach(friendsData.friends, id: \.self) { item in
                                NavigationLink {
                                    FriendDetails(user: item)
                                } label: {
                                    FriendsRow(userItem: item)
                                }
                            }
                        }
                    } else {
                        Text("No friends")
                    }

                }
                .navigationTitle("Friends")
                .navigationBarTitleDisplayMode(.inline)
                .toolbar {
                    Button("Add friends") {
                        showingSheet.toggle()
                    }
                    .sheet(isPresented: $showingSheet) {
                        FindFriends()
                    }
                }
            }
        }.accentColor(Color.primaryColorBlue)
    }
}
```

```
struct Friends_Previews: PreviewProvider {
   static var previews: some View {
      Friends()
         .environmentObject(FriendsData(friendsIds: ["111111"]))
   }
}
```

FriendsRow.swift

```
import SwiftUI

struct FriendsRow: View {
   @EnvironmentObject var friendsData: FriendsData
   @State var isAdded = false
   var showAddButton: Bool = false
   var userItem: User

   var body: some View {
      HStack(alignment: .center) {
         CircleImage(image: Image("kitty"))
            .frame(width: 70, height: 70)
            .padding(.leading, 5)
         Text(userItem.username)
            .font(.system(size: 23))
         Spacer()
         if showAddButton {
            Image(systemName: isAdded ? "checkmark.circle" : "plus.circle")
               .scaleEffect(2)
               .padding(.trailing, 40)
               .onTapGesture {
                  isAdded = true
                  friendsData.addItem(item: userItem)
               }
         }
      }
      .foregroundColor(.chatUserColor)
   }
}

struct FriendsRow_Previews: PreviewProvider {
   static var previews: some View {
      FriendsRow(showAddButton: true, userItem: User.defaultUser)
      .previewLayout(.fixed(width: 400, height: 80))
   }
}
```

FindFriends.swift

```
import SwiftUI
import Combine

struct FindFriends: View {
```

```
    @StateObject private var viewModel: ViewModel

    init() {
        _viewModel = StateObject(wrappedValue: ViewModel())
    }


    var body: some View {

        NavigationView {
            List {
                ForEach(viewModel.foundUsers, id: \.self) { item in
                    FriendsRow(showAddButton: true, userItem: item)
                }
            }
            .navigationTitle("Find friends")
            .searchable(text: $viewModel.searchText)
            .navigationBarTitleDisplayMode(.inline)
        }
    }
}

struct FindFriends_Previews: PreviewProvider {
    static var previews: some View {
        FindFriends()
    }
}
```

FriendDetails.swift

```
import SwiftUI

struct FriendDetails: View {
    @State private var chatPresented = false
    @EnvironmentObject var chatsData: ChatsData
    @EnvironmentObject var messages: MessagesData
    var user: User

    var body: some View {
        ZStack {
            Color.blue.opacity(0.03)
                .ignoresSafeArea()
            VStack {
                HStack {
                    CircleImage(image: Image("kitty"))
                        .frame(width: 80, height: 80)
                    Text("\(user.firstName) \(user.lastName)")
                        .font(.system(size: 25))
                    Spacer()
                }
                .padding([.leading,.trailing,.bottom], 15)
                .background(Color.gradientColor1.opacity(0.8))
                .foregroundColor(.white)
```

```
            Button("Create chat") {
               let newChat = ChatRoom(id: "temp.id",
                                      chatName: "\(user.firstName) \(user.lastName)",
                                      chatType: "private",
                                      user_ids: [UserData.shared.getUserId(), user.id!])
               let index = chatsData.chats.count
               chatsData.chats.append(newChat)
               Socket.shared.createNewChat(chat: newChat) { id in
                  chatsData.chats[index].id = id
               }

               self.chatPresented.toggle()
            }
            .fullScreenCover(isPresented: $chatPresented, content: {
               Chat(chatRoom: chatsData.chats.last ?? ChatRoom.defaultChatRoom,
                    messages: messages,
                    name: "\(user.firstName) \(user.lastName)")
            })
            .buttonStyle(CustomButtonStyle())
            Spacer()


         }
      }
   }
}

struct FriendDetails_Previews: PreviewProvider {
   static var previews: some View {
      FriendDetails(user: User.defaultUser)
   }
}
```

ChatList.swift

```
import SwiftUI
import Combine

struct ChatList: View {
   @EnvironmentObject var chatsData: ChatsData
   @EnvironmentObject var messagesData: MessagesData
   @State private var selectedChat: ChatRoom?

   var body: some View {
      NavigationView{
         ZStack {
            Color.blue.opacity(0.03)
               .ignoresSafeArea()
            List {
               ForEach(chatsData.chats) { chat in

                  Button {
```

```
                    selectedChat = chat
                } label: {
                    ChatRow(chatRoom: chat)

                }
                .listRowInsets(EdgeInsets())
                .buttonStyle(CustomListButtonStyle())


            }
            .listRowBackground(Color.clear)
            .fullScreenCover(item: $selectedChat, content: { item in

                Chat(chatRoom: item, messages: messagesData, name: item.chatName ?? "User")
            })


        }
        .frame(maxWidth: .infinity)
        .listStyle(.plain)
      }
      .navigationTitle("Chats")
      .navigationBarTitleDisplayMode(.inline)
    }
  }
}

struct ChatList_Previews: PreviewProvider {
  static var previews: some View {
    ChatList().preferredColorScheme(.light)
      .environmentObject(ChatsData(chat_ids: ["00000.0"]))
  }
}
```

Chat.swift

```
import SwiftUI
import SocketIO

struct Chat: View {
  @Environment(\.presentationMode) var presentationMode

  @ObservedObject var viewModel: ChatViewModel
  @Namespace var bottomID
  var userName: String

  init(chatRoom: ChatRoom, messages: MessagesData, name: String) {
    viewModel = ChatViewModel(chatRoom: chatRoom, messages: messages)
    userName = name
  }

  var body: some View {
    NavigationView{
```

```
        ZStack(alignment: .bottom) {
            Color.blue.opacity(0.03)
                .ignoresSafeArea()

            VStack{
                ScrollViewReader {proxy in
                    ScrollView {
                        LazyVStack {
                            ForEach(viewModel.messages.messages[viewModel.chatData.id] ?? []) { message in
                                ChatBubble(position: message.user_id == UserData.shared.getUserId() ? .right : .left,
                                           pending: (message.pending == nil) ? false : true, time: message.date ?? "10/10/10
00:00:00") {
                                    Text(message.message)
                                }
                            }
                            .onAppear {
                                proxy.scrollTo(viewModel.messages.messages[viewModel.chatData.id]!.last?.id, anchor:
.bottom)
                            }
                            .onChange(of: viewModel.messages.messages[viewModel.chatData.id]) { newValue in
                                proxy.scrollTo(viewModel.messages.messages[viewModel.chatData.id]!.last?.id, anchor:
.bottom)
                            }
                        }
                    }
                }

                HStack {
                    FieldWithImage(fieldName: "Message", textValue: $viewModel.newMessageValue)
                        .frame(width: 300, height: 40)
                        .padding(.trailing, 10)
                    Image(systemName: "arrow.up.message")
                        .resizable()
                        .aspectRatio(contentMode: .fit)
                        .frame(height: 30)
                        .scaleEffect(viewModel.sendIsTapped ? 1.3 : 1.0)
                        .animation(.spring(response: 0.4, dampingFraction: 0.6), value: viewModel.sendIsTapped)
                        .onTapGesture {

                            viewModel.sendMessage()

                        }


                }
                .frame(width: 390, height: 60)
                .background(Color.clear.background(.thinMaterial))
                .ignoresSafeArea()
            }
        }
        .navigationTitle(userName)
        .navigationBarTitleDisplayMode(.inline)
        .toolbar {
            Button("Hide") {
```

```
                presentationMode.wrappedValue.dismiss()
            }
        }
    }
  }
}

struct Chat_Previews: PreviewProvider {
  static var previews: some View {
    Chat(chatRoom: ChatRoom.defaultChatRoom, messages: MessagesData(chat_ids: ["00000.0"]), name:
"Test User")
  }
}
```

ChatRow.swift

```
import SwiftUI
import Combine

struct ChatRow: View {

  @ObservedObject var viewModel: ViewModel

  init(chatRoom: ChatRoom) {
    viewModel = ViewModel(chatRoom: chatRoom)
    if (viewModel.chat.chatType == "private") {
      viewModel.getChatName()
    }
  }

  var body: some View {
    ZStack {

      HStack (alignment: .top, spacing: 15) {
        CircleImage(image: Image("kitty"))
          .frame(width: 70, height: 70, alignment: .center)


        VStack(alignment: .leading, spacing: 5)  {
          Text(viewModel.chatName ?? "Chat user")
            .font(.title)
            .onAppear{
              if (viewModel.chat.chatType == "private") {
                viewModel.getChatName()
              }
            }
          Text("Last Message")


        }
        Spacer()
      }
      .padding(10)
```

```
        }
    }

}

struct ChatRow_Previews: PreviewProvider {
    static var previews: some View {
        ChatRow(chatRoom: ChatRoom.defaultChatRoom)
            .previewLayout(.fixed(width: 400, height: 100))
    }
}
```

HomeWindow.swift

```
import SwiftUI

struct HomeWindow: View {
//    @EnvironmentObject var authentication: Authentication
    @StateObject var viewModel: HomeWindowViewModel
    @State private var showingSheet = false
    @State private var showAuthInterface = true

    init(auth: Authentication) {
        _viewModel = StateObject(wrappedValue: HomeWindowViewModel(auth: auth))
    }

    var body: some View {
        ZStack (alignment: .top){
            Background(showBackground: $viewModel.showBackground,
                       parallelogramFrameHeight: 390,
                       sectionAngle: 45,
                       screenWidth: 390,
                       screenHeight: 844)


            VStack (alignment: .center, spacing: 8) {
                CircleImage(image: !showAuthInterface ? Image("kitty") : Image(""))
                    .frame(width: 250, height: 250)
                    .padding(.bottom, 20)

                if (showAuthInterface) {
                    FieldWithImage(imageName: "person", fieldName: "Username", textValue:
$viewModel.authenticator)
                        .frame(height: 60)
                    SecretFieldWithImage(fieldName: "Password", textValue: $viewModel.password)
                        .frame(height: 60)

                    Button("Log In") {
                        viewModel.logIn()
                    }
                    .alert("User does not exist", isPresented: $viewModel.showingAlert) {
                        Button("OK", role: .cancel) { }
                    }
```

```
                Button("Sign Up") {
                    showingSheet.toggle()
                }
                .sheet(isPresented: $showingSheet) {
                    SignUp()
                }
            } else {
                Button("Continue") {
                    viewModel.successfullLogin()
                    viewModel.hideHomeWindow()
                }
            }
        }
        .buttonStyle(CustomButtonStyle())
        .padding(12)
        .opacity(viewModel.showButtons ? 1 : 0)
        .onAppear {
            viewModel.showInterfaceButtons()
            showAuthInterface = !viewModel.authentication.isLoggedIn
        }
    }

  }
}

struct HomeWindow_Previews: PreviewProvider {
  static var previews: some View {
    HomeWindow(auth: Authentication())
  }
}
```

SignUp.swift

```
import SwiftUI
import Combine

struct SignUp: View {
  @StateObject private var viewModel = RegisterViewModel()

  var body: some View {
    ZStack{
      Color.blue.opacity(0.03)
        .ignoresSafeArea()
      VStack(alignment: .center, spacing: 15) {
        HStack(alignment: .center, spacing: 10) {
          FieldWithImage(imageName: nil, fieldName: "First Name", lineWidth: 3, textValue:
$viewModel.firstName)
          FieldWithImage(imageName: nil, fieldName: "Last Name", lineWidth: 3, textValue:
$viewModel.lastName)
        }
        .frame(height: 60)
```

```
            FieldWithImage(imageName: "person", fieldName: "Username", lineWidth: 3, textValue:
$viewModel.authenticator)
               .frame(height: 60)
            FieldWithImage(imageName: "envelope", fieldName: "E-mail", lineWidth: 3, textValue:
$viewModel.email)
               .frame(height: 60)
            SecretFieldWithImage(fieldName: "Password", textValue: $viewModel.password)
               .frame(height: 60)

            Button("Sign Up") {
               let user = User(firstName: viewModel.firstName,
                        lastName: viewModel.lastName,
                        username: viewModel.authenticator,
                        email: viewModel.email,
                        password: viewModel.password,
                        userChats: [],
                        userFriends: [])
               print("\n\n\n \(user)")
               viewModel.sendNewUser(user: user)

            }
            .buttonStyle(CustomButtonStyle())
            .alert("User already exists.", isPresented: $viewModel.showingAlert) {
               Button("OK", role: .cancel) { }
            }
         }
         .padding()
      }
   }
}

struct SignUp_Previews: PreviewProvider {
   static var previews: some View {
      SignUp()
   }
}
```

MainView.swift

```
import SwiftUI
import SocketIO
import Introspect

struct MainView: View {
   @StateObject private var chatsData = ChatsData(chat_ids: UserData.shared.getUserChats())
   @StateObject private var messagesData = MessagesData(chat_ids: UserData.shared.getUserChats())
   @StateObject private var friendsData = FriendsData(friendsIds: UserData.shared.getUserFriends())

   @State var tabSelection:Tab = .chats
   enum Tab {
      case chats
      case friends
      case profile
   }
```

```
    var body: some View {
        TabView{
            ChatList()
                .tabItem {
                    Label("Chats", systemImage: "mail")
                }
                .tag(Tab.chats)

            Friends()
                .tabItem {
                    Label("Friends", systemImage: "person.2.fill")
                }
                .tag(Tab.friends)


            Profile()
                .tabItem {
                    Label("Profile", systemImage: "person.crop.circle")
                }
                .tag(Tab.profile)
        }
        .environmentObject(chatsData)
        .environmentObject(messagesData)
        .environmentObject(friendsData)
        .onAppear {
            initSocket()
        }

    }

    func initSocket() {
        Socket.shared.connectToSocketServer()
        Socket.shared.initChats(chats: chatsData, messages: messagesData)
    }
}

struct MainView_Previews: PreviewProvider {
    static var previews: some View {
        MainView()
    }
}
```

Profile.swift

```
import SwiftUI

struct ViewOffsetKey: PreferenceKey {

    typealias Value = CGFloat
    static var defaultValue: Value = Value.zero

    static func reduce(value: inout Value, nextValue: () -> CGFloat) {
```

```
            value = nextValue()
        }
    }

    struct Profile: View {
        @State private var pictureSize: CGFloat = 100
        @State private var isDragging: Bool = false
        @EnvironmentObject var authentication: Authentication
        @EnvironmentObject var chatsData: ChatsData

        var drag: some Gesture {
            var previousGesture:CGFloat = 0.0

            return DragGesture()
                .onChanged { value in
                    let sizeDelta = value.translation.height - previousGesture
                    let newSize = pictureSize + sizeDelta

                    if sizeDelta > 0 {
                        pictureSize = newSize < 300 ? newSize : 300
                    } else if (sizeDelta < 0) {
                        pictureSize = newSize > 100 ? newSize : 100
                    }

                    previousGesture = value.translation.height

                }
                .onEnded { _ in
                    isDragging = false
                }
        }

        var body: some View {

            VStack {

                VStack {
                    Text("\(UserData.shared.getUserName())")
                        .font(.system(size: 25, weight: .medium, design: .default))
                        .foregroundColor(Color(0xF4ECFA))
                        .padding(.bottom, 30)

                    CircleImage(image: Image("kitty"))
                        .frame(width: pictureSize, height: pictureSize, alignment: .center)

    //                    .stroke(Color.primaryColorPurple, lineWidth: 4))
                }
                .background(Circle()
                    .fill(LinearGradient(colors: [.primaryColorPurple, .primaryColorPurple.opacity(0.1)], startPoint:
    .center, endPoint: .bottomTrailing))
                    .frame(width: UIScreen.screenWidth * 1.6, height: UIScreen.screenWidth * 1.6)
                    .offset(y:  -UIScreen.screenWidth * 0.7)
                    .ignoresSafeArea())
```

```
            List {
               ZStack{
                  VStack {
                     ForEach(1..<20) { index in
                        Text("\(index)")
                     }
                     Button ("Log Out") {
                        Socket.shared.disconnect()
                        withAnimation {
                           UserData.shared.saveData(user: nil)
                           authentication.changeStatus()
                           authentication.changeHomeWindowStatus()
                        }
                     }
                     .buttonStyle(CustomButtonStyle())
                  }
                  GeometryReader { proxy in
                     Color.clear.preference(key: ViewOffsetKey.self,
                                 value: -proxy.frame(in: .named("scroll")).origin.y)
                  }
               }
            }
            .listStyle(.plain)
            .onAppear(perform: {UITableView.appearance().isScrollEnabled = false})
            .gesture(drag)
            .coordinateSpace(name: "scroll")
            .onPreferenceChange(ViewOffsetKey.self) { value in
               if (value >= 0.0) {
                  isDragging = true
               }
            }

         }
      }
   }

   struct Profile_Previews: PreviewProvider {
      static var previews: some View {
         Profile()
            .previewInterfaceOrientation(.portrait)
      }
   }
```

Server.js

```
import http from 'http';
import { Server } from 'socket.io';
import { connectToDB } from './db/connection.js';
import 'dotenv/config';
import { Socket } from './sockets/socketController.js';
import { app } from './app.js';

const server = http.createServer(app);
```

```
const io = new Server(server);
const socket = new Socket(io);

socket.socketEvents(socket);

server.listen(process.env.API_PORT, async () => {
   await connectToDB();
   console.log('Server started on port 8080');
});
```

App.js

```
import express from 'express';
import cors from 'cors';
import morgan from 'morgan';
import swaggerUiExpress from 'swagger-ui-express';
import yaml from 'js-yaml';
import { readFileSync } from 'fs';
import { userRouter } from './routes/usersRoute.js';
import { authenticateToken, updateAccessToken } from './helpers/tokenHelpers.js';
import { chatRoomRouter } from './routes/chatRoomRouter.js';
import { messagesRouter } from './routes/messagesRouter.js';

export const app = express();

const doc = yaml.load(readFileSync('openapi.yml', 'utf8'));

app.use('/api-docs', swaggerUiExpress.serve, swaggerUiExpress.setup(doc));
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.use(cors());
app.use(morgan('dev'));

app.get('/testAuth', authenticateToken(process.env.JWT_SECRET_KEY));
app.post('/refreshToken', updateAccessToken);
app.use(userRouter);
app.use(chatRoomRouter);
app.use(messagesRouter);
```

ChatRoomController.js

```
import { ObjectId } from 'mongodb';
import { getDb } from '../db/connection.js';
import { getNextSequenceValue } from '../db/getNextId.js';
import { createChatRoom } from '../models/chatRoomModel.js';

export const chatRoomContoller = {
   createChat: async (req, res) => {
      const db = getDb();
      const {
         user_ids,
         chatType,
```

```
    } = req.body;
    const newChatRoom = createChatRoom(chatType === 'test' ? 0 : (await
getNextSequenceValue('chatroomid')), chatType, user_ids);
    try {
      await db.collection('chatRooms').insertOne(newChatRoom);
      res.status(200).json({ result: 'Success', message: 'A new chat room has been created successfully' });
    } catch (err) {
      res.json({ result: 'Error', message: err });
    }
  },

  findChat: async (req, res) => {
    const db = getDb();
    const { id } = req.params;

    try {
      const chat = await db.collection('chatRooms').findOne({ chatRoom_id: Number(id) });
      res.status(200).send({ chat, message: 'The chat is found' });
    } catch (err) {
      res.status(404).send({ result: 'Error', err });
    }
  },

  loadAllUsersChats: async (req, res) => {
    const db = getDb();

    const chatIDs = req.query.id.split(',').map((x) => new ObjectId(x));
    try {
      const chats = await db.collection('chatRooms').find({ _id: { $in: chatIDs } }).toArray();
      res.status(200).json({ chats, message: 'All chats have been fetched successfully' });
    } catch (err) {
      res.json({ result: 'Error', message: err });
    }
  },
  deleteUserChatById: async (req, res) => {
    const db = getDb();
    const chatRooms = db.collection('chatRooms');
    const { id } = req.params;

    try {
      await chatRooms.deleteOne({ chatRoom_id: Number(id) });
      res.status(202).json({ result: 'Success', message: 'The chat room has been deleted successfully' });
    } catch (err) {
      res.status(404).json({ result: 'Error', message: err });
    }
  },
  addUsersToChat: async (req, res) => {
    const db = getDb();
    const chatRooms = db.collection('chatRooms');
    const {
      chat_id,
      user_ids,
    } = req.body;
    try {
```

```
        await chatRooms.updateOne({
          chatRoom_id: Number(chat_id),
        }, {
          $push: { user_ids: { $each: user_ids } },
        });
        res.status(200).json({ result: 'Success', message: 'Users have been added successfully' });
      } catch (err) {
        res.status(404).json({ result: 'Error', message: err });
      }
    },
};
```

MessagesController.js

```
import { ObjectId } from 'mongodb';
import { getDb } from '../db/connection.js';

export const messageController = {
  loadMessages: async (req, res) => {
    const db = getDb();
    const messagesCollection = db.collection('messages');
    const { chatid } = req.params;

    try {
      const messages = await messagesCollection
        .find({ chat_id: new ObjectId(chatid) })
        .sort({ $natural: 1 })
        .limit(1000)
        .toArray();
      res.json({ messages, message: 'Messages have been fetcehd successfully' });
    } catch (err) {
      console.log(err);
      res.status(404).json({ result: 'Error', message: err });
    }
  },
};
```

UsersController.js

```
import { compare, hashSync } from 'bcrypt';
import { ObjectId } from 'mongodb';
import { getDb } from '../db/connection.js';
import { generateAccessToken, generateRefreshToken } from '../helpers/tokenHelpers.js';
import { createNewUser } from '../models/userModel.js';

export const userContoller = {
  getUsers: async (req, res, next) => {
    const db = getDb();
    const usersCollection = db.collection('users');
    const { query } = req;

    let body;
```

```
    if (query.searchval) {
      body = { username: { $regex: new RegExp(query.searchval, 'i') } };
    } else if (query.searchid) {
      const id = new ObjectId(query.searchid);
      body = { _id: id };
    } else if (query.searchids) {
      const user_ids = query.searchids.split(',').map((x) => new ObjectId(x));
      body = { _id: { $in: user_ids } };
    } else {
      res.status(400).json({ result: 'Error', message: 'Invalid query' });
      next();
    }

    try {
      const users = await usersCollection.find(body).project({
        username: 1,
        firstName: 1,
        lastName: 1,
        _id: 1,
      }).limit(15).toArray();
      res.status(200).json(users);
    } catch (err) {
      res.status(400).json({ result: 'Error', message: err });
    }
  },
  createNewUser: async (req, res) => {
    const db = getDb();
    const {
      firstName, lastName, username, email, password, userChats, userFriends,
    } = req.body;

    const hashedPassword = hashSync(password, 8);
    const newUser = createNewUser(firstName, lastName, email, hashedPassword, username, userChats,
userFriends);
    try {
      await db.collection('users').insertOne(newUser);
      res.status(201).json({ result: 'Success', message: 'A new user has been created successfully' });
    } catch (err) {
      res.status(400).json({ result: 'Error', message: err });
    }
  },
  // For authenticaition
  getParticularUser: async (req, res) => {
    const db = getDb();
    const usersCollection = db.collection('users');
    const {
      username,
      password,
    } = req.body;

    const user = await usersCollection.findOne({ username });
    if (user && await compare(password, user.password)) {
      delete user.password;
      const token = generateAccessToken(user);
```

```
            const refreshToken = generateRefreshToken(user);
            res.status(200).json({
               user,
               token,
               refreshToken,
            });
         } else {
            res.status(401).send("Error. User doesn't exist.");
            // res.status(200).json(fetched);
         }
      },
      updateParticularUser: async (req, res) => {
         const db = getDb();
         const users = db.collection('users');
         if (req.body.$push) {
            req.body.$push.userFriends = new ObjectId(req.body.$push.userFriends);
         }
         try {
            await users.updateOne({
               username: req.params.username,
            }, req.body);
            res.status(200).send();
         } catch (err) {
            console.log(err);
            res.status(400).send();
         }
      },
      deleteParticularUser: async (req, res) => {
         const db = getDb();
         const users = db.collection('users');
         try {
            await users.deleteOne({
               username: req.params.username,
            });
            res.status(202).json({ result: 'Success', message: 'A user has been deleted successfully' });
         } catch (err) {
            res.status(400).send();
         }
      },
   };
```

connection.js

```
import { MongoClient } from 'mongodb';
import 'dotenv/config';

const client = new MongoClient(process.env.MONGODB_URL, {
   useNewUrlParser: true,
   useUnifiedTopology: true,
});

let dbConnection;
```

```
export const connectToDB = () => new Promise((resolve, reject) => {
  client.connect((err, db) => {
    if (err) {
      console.error(`Error. ${err}`);
      reject(err);
    } else {
      console.log('Server has connected to the mongoDb');
      dbConnection = db.db('chat-app');
      resolve();
    }
  });
});

export const getDb = () => dbConnection;
```

chatsListeners.js

```
import { ObjectId } from 'mongodb';
import { getDb } from '../db/connection.js';
import { createChatRoom } from '../models/chatRoomModel.js';

export const chatListeners = {
  newChat: (socket, io, connectedUsers) => async (chatObject, callback) => {
    const db = getDb();
    const chatsCollection = db.collection('chatRooms');
    const usersCollection = db.collection('users');
    const {
      chatType,
      user_ids,
    } = chatObject;

    const newChat = createChatRoom(chatType, user_ids.map((x) => new ObjectId(x)));
    try {
      await chatsCollection.insertOne(newChat);
      const chatID = newChat._id;

      await user_ids.forEach(async (user_id) => {
        await usersCollection.updateOne({
          _id: new ObjectId(user_id),
        }, {
          $push: { userChats: chatID },
        });
        const userSocket = connectedUsers.find((sockData) => sockData.user_id === user_id);

        if (userSocket != null) {
          userSocket.socket.join(chatID);
        }
        console.log();
        console.log(io.sockets.adapter.rooms);
        socket.broadcast.to(chatID).emit('chat created', chatObject);
        callback({
          id: chatID,
        });
```

```
        });
      } catch (err) {
        console.log(err);
      }
    },
};
```

messagesListeners.js

```
import { ObjectId } from 'mongodb';
import { getDb } from '../db/connection.js';
import { createMessage } from '../models/messageModel.js';

export const messageListeners = {
  newMessage: (socket) => async (messageObject, callback) => {
    const db = getDb();
    const messagesCollection = db.collection('messages');
    const {
      user_id,
      chat_id,
      message,
    } = messageObject;

    const currentDate = new Date().toLocaleString().replace(',', '');

    const newMessage = createMessage(new ObjectId(user_id), new ObjectId(chat_id), message,
currentDate);
    try {
      await messagesCollection.insertOne(newMessage);
      const messageId = newMessage._id.toString();
      console.log(chat_id);
      socket.broadcast.to(chat_id).emit('new message', newMessage);
      callback({
        id: messageId,
        date: currentDate,
      });
    } catch (err) {
      socket.emit('error', err);
      console.log(err);
      callback({
        error: 'Error creating message',
      });
    }
  },
};
```

chatRoomModel.js

```
export const createChatRoom = (chatType, user_ids) => ({
  chatType,
  user_ids,
});
```

messageModel.js

```
export const createMessage = (user_id, chat_id, message, date) => ({
   user_id,
   chat_id,
   message,
   date,
   readby: [],
});
```

userModel.js

```
export const createNewUser = (firstName, lastName, email, password, username, userChats, userFriends) => ({
   firstName,
   lastName,
   username,
   email,
   password,
   userChats,
   userFriends,
});
```

chatRoomRouter.js

```
import express from 'express';
import { chatRoomContoller } from '../controllers/chatRoomController.js';

export const chatRoomRouter = express.Router();

chatRoomRouter
   .post('/newchat', chatRoomContoller.createChat)
   .get('/loadchats', chatRoomContoller.loadAllUsersChats)
   .get('/loadchat/:id', chatRoomContoller.findChat)
   .post('/addusers/', chatRoomContoller.addUsersToChat)
   .delete('/deletechat/:id', chatRoomContoller.deleteUserChatById);
```

messageRouter.js

```
import express from 'express';
import { messageController } from '../controllers/messagesController.js';

export const messagesRouter = express.Router();

messagesRouter
   .get('/messagesfrom/:chatid', messageController.loadMessages);
```

userRouter.js

```
import express from 'express';
import { userContoller } from '../controllers/usersContoller.js';

export const userRouter = express.Router();
```

```
userRouter
  .get('/', userContoller.getUsers)
  .post('/login/', userContoller.getParticularUser)
  .post('/register', userContoller.createNewUser)
  .put('/updatedata/:username', userContoller.updateParticularUser)
  .delete('/delete/:username', userContoller.deleteParticularUser);
```

SocketController.js

```
import { chatListeners } from '../listeners/chatsListeners.js';
import { messageListeners } from '../listeners/messageListeners.js';

export class Socket {
  constructor(io) {
    this.io = io;
    this.connectedUsers = [];
  }

  socketEvents() {
    this.io.on('connection', (socket) => {
      console.log('a user connected with id: ', socket.id);

      socket.on('inituser', (userObject) => {
        this.connectedUsers.push({
          socket,
          user_id: userObject.user_id,
        });
        userObject.chats.forEach((el) => {
          socket.join(el);
        });
        console.log(this.io.sockets.adapter.rooms);
      });

      socket.on('disconnect', () => {
        this.connectedUsers = this.connectedUsers.filter((x) => x.socket.id !== socket.id);
      });

      socket.on('chat message', messageListeners.newMessage(socket));

      socket.on('new chat', chatListeners.newChat(socket, this.io, this.connectedUsers));
    });
  }
}
```