Міністерство освіти і науки України

Луцький національний технічний університет

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

РОЗРОБКА МЕСЕНДЖЕРА З ВИКОРИСТАННЯМ REACT NATIVE ТА ASP.NET CORE

DEVELOPMENT OF A MESSENGER USING REACT NATIVE AND ASP.NET CORE

спеціальність 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

освітня програма «Інженерія програмного забезпечення»
(назва освітньої програми)

Виконав: здобувач вищої освіти
Групи ІПЗ-41
Удоденко Андрій Сергійович
_____
(підпис)

Керівник:
к.т.н., доцент
Ліщина Наталія Миколаївна
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«8» 06 2024 р.
к.т.н., доцент
Гарант освітньої програми:
Ліщина Наталія Миколаївна
_____
(підпис)

Луцьк – 2024 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

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

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Н.Ліщина
«2» 02 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Удоденку Андрію Сергійовичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: *Розробка месенджера з використанням React Native та ASP.NET Core*

Керівник роботи: *к.т.н., доцент, Ліщина Наталія Миколаївна*
затверджені наказом закладу вищої освіти від «*30*» *грудня 2023 р.* № *477/01-02*
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«5» червня 2024 р.*
3. Вихідні дані до роботи: *технічне та програмне забезпечення ЕОМ, вимоги до розробки програмного забезпечення, ергономічні вимоги до функціонування програмного засобу.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити):
Аналіз сучасного стану проблеми, існуючих методів і засобів її розв'язання, аналіз, визначення вимог до розроблюваного програмного забезпечення та проектування програмного забезпечення, опис функціонального наповнення об'єкта проектування, практична реалізація об'єкта проектування.

5. Перелік графічного матеріалу: *16 рисунків; 1 схема; 5 діаграм.*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	*Ліщина Н. М.*		
Специфікації вимог до розробленої системи	*Ліщина Н. М.*		
Розробка об'єкта проєктування	*Ліщина Н. М.*		
Нормоконтроль	*Повстяна Ю. С.*		
Гарант ОП	*Ліщина Н. М.*		
Показник запозичень тексту		5.5%	
Академічна доброчесність	*Яцук А. А.*		

7. Дата видачі завдання *«2» лютого 2024 р.*

КАЛЕНДАРНИЙ ПЛАН

№ З/п	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	*Огляд літературних джерел по темі кваліфікаційної роботи бакалавра*	*07.02.2024 р.*	
2	*Аналіз проблеми розробки та впровадження об'єкту проектування*	*27.02.2024 р.*	
3	*Обгрунтування вибору шляхів, технологій (алгоритмів) і засобів вирішення поставленого завдання*	*12.03.2024 р.*	
4	*Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних*	*17.04.2024 р.*	
5	*Практична реалізація об'єкта проектування та розробка бази даних*	*18.05.2024 р.*	
6	*Тестування та налагодження об'єкта проектування*	*01.06.2024 р.*	
7	*Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру*	*05.06.2024 р.*	

Здобувач вищої освіти _____ (підпис) *Удоденко А. С.* (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ (підпис) *Ліщина Н. М.* (прізвище, ініціали)

Міністерство освіти і науки України
Луцький національний технічний університет

Рецензія
на кваліфікаційну роботу

Здобувач вищої освіти: Удоденко Андрій Сергійович

Тема: розробка месенджера з використанням React Native та ASP.NET Core

Коротка характеристика кваліфікаційної роботи: кваліфікаційна робота присвячена розробці месенджера з використанням сучасних технологій, таких як: React, React Native та ASP.NET Core. Включає себе розробку клієнтського та серверного застосунку.

Самостійні розробки і пропозиції автора: автор самостійно розробив серверну частину месенджера, використавши популярну архітектуру та підходи та розробив клієнтські веб та мобільні застосунки із зручним та привабливим дизайном

Практичне значення роботи: практична значимість роботи полягає у створенні масштабованого месенджера, який здатний обробляти великі обсяги даних та забезпечувати швидку та надійну комунікацію між користувачами.

Недоліки: істотних недоліків не виявлено, проте недостатньо детально розглянуто аспекти безпеки.

Загальний висновок: роботу виконано у повному обсязі, у відповідності із завданням. Здобувач освіти Удоденко Андрій Сергійович заслуговує на позитивну оцінку.

Рецензент: к.пед.н., доцент, завідувач каф. ЧОТ Кабак В.Д.
(прізвище, ім'я, по-батькові, посада)

Рецензент кваліфікаційної роботи _____
(підпис)

«7» 06 2024 р.

Міністерство освіти і науки України
Луцький національний технічний університет

Факультет Комп'ютерних та інформаційних технологій
Кафедра Інженерії програмного забезпечення

ВІДГУК
КЕРІВНИКА НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач вищої освіти Удоденко Андрій Сергійович
(прізвище, ім'я, по батькові)

група ПЗ-41

Тема кваліфікаційної роботи — розробка месенджера з використанням React Native та ASP.NET Core.
Керівник Лінцина Наталія Миколаївна.
Актуальність теми — у сучасному світі месенджери стали невід'ємною частиною життя людей. Вони забезпечують миттєвий обмін інформацією. Розробка власного месенджера з використанням сучасних технологій дозволить отримати гнучкий та масштабований продукт, що відповідатиме потребам користувачів.
Об'єкт дослідження — процес розробки месенджера з використанням React, React Native та ASP.NET Core.

Характеристика теоретичного рівня, наявності самостійних розробок і практичної значимості роботи — робота демонструє високий теоретичний рівень, включає самостійні розробки. Розроблено гнучку та масштабовану систему. Детально наступні охоплює створення зручного клієнтського додатку та серверної частини.
Зауваження та недоліки — хотілося б щоб деякі моменти були розглянуті детальніше.

Загальний висновок — робота виконана у повному обсязі відповідно до поставлених завдань та заслуговує оцінку „добре" або „відмінно".

Керівник _____ (Лінцина Наталія Миколаївна)
(підпис) (прізвище, ім'я, по батькові)

«7» 06 2024 р.

АНОТАЦІЯ

Удоденко А. С. Розробка месенджера з використанням React Native та ASP.NET Core. Рукопис.

Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2024.

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

У першому розділі здійснено аналіз сучасного стану проблеми розробки месенджерів і сформульовано завдання. В другому розділі визначено вимоги до розроблюваної системи, а також засоби, методи і алгоритми розробки. У третьому розділі розроблено месенджер. У висновках узагальнено інформацію, відображену у попередніх частинах. Результати розробки демонструють можливості створення простого та зручного месенджера, який відповідає основним вимогам українських користувачів. Надалі планується додатково покращувати функціонал та проводити оптимізацію системи для забезпечення кращого користувацького досвіду.

Ключові слова: месенджер, безпека, конфіденційність, React, React Native, вебзастосунок, інтерфейс користувача, тестування, українські користувачі.

ABSTRACT

Udodenko A. S. Development of messenger using React, React Native and ASP.NET Core. Manuscript.

Bachelor's qualifying thesis of the educational program «Software Engineering». Lutsk National Technical University. Lutsk, 2024.

The bachelor's qualifying thesis consists of an introduction, three sections, conclusions, a list of used sources and annexes.

In the first chapter, an analysis of the current state of the problem of developing messengers was carried out and the task was formulated. The second section defines the requirements for the developed system, as well as means, methods and development algorithms. In the third chapter, messenger was developed and tested. The conclusions summarize the information presented in the previous parts. The development results demonstrate the possibilities of creating a simple and secure messenger that meets the basic requirements of Ukrainian users. Further plans include enhancing the functionality and optimizing the system to ensure a better user experience.

Keywords: messenger, security, privacy, React, web application, user interface, testing, ukrainian users

ЗМІСТ

ВСТУП

У сучасному світі месенджери стали невід'ємною частиною повсякденного життя людей. Вони забезпечують миттєвий обмін текстовими повідомленнями, файлами, фото та відео між користувачами, дозволяючи їм підтримувати зв'язок з друзями, родиною та колегами. Однак більшість популярних месенджерів є закритими системами, що не дозволяє користувачам повністю контролювати свої дані та конфіденційність. Крім того, вони можуть містити надлишкові функції, які не є необхідними для певних груп користувачів або вимагають додаткових ресурсів пристрою.

В Україні існує потреба у розробці месенджера, який би відповідав специфічним вимогам користувачів, забезпечуючи високий рівень безпеки, конфіденційності та зручності використання. Такий месенджер міг би використовуватися військовими, державними органами, а також усіма, хто цінує конфіденційність та безпеку своїх комунікацій. Для реалізації цього проєкту були використані сучасні технології та підходи, такі як React, React Native, ASP.NET Core, Domain-Driven Design, CQRS та інші. Це дозволило створити гнучку, масштабовану та легко модифіковану систему, яка задовольняє потреби користувачів та адаптується та може бути покращена у майбутньому.

Мета дослідження: створити високопродуктивний, масштабований та зручний месенджер, використовуючи React, React Native та ASP.NET Core, який забезпечить зручний обмін повідомленнями між користувачами.

Об'єкт дослідження: процес розробки месенджера з використанням React, React Native та ASP.NET Core.

Предмет дослідження: архітектура, підходи, технології та методи, необхідні для створення високопродуктивного, масштабованого та зручного месенджера з використанням React, React Native та ASP.NET Core.

Завдання дослідження:

1) аналіз існуючих рішень, їх переваг та недоліків, а також вимог користувачів до месенджерів;

2) проектування архітектури месенджера з використанням Domain-Driven Design та патерну CQRS для забезпечення гнучкості, масштабованості та можливості легко модифікувати систему в майбутньому;

3) розробка серверної частини (бекенд) месенджера з використанням ASP.NET Core, Entity Framework Core та інших сучасних бібліотек і фреймворків для забезпечення високої продуктивності та безпеки;

4) створення макету вебзастосунку та мобільного застосунку, реалізація клієнтської частини (фронтенд) месенджера з використанням React та React Native для створення зручного та інтуїтивно зрозумілого інтерфейсу користувача;

5) реалізація основних функцій месенджера, таких як обмін текстовими повідомленнями, додавання друзів, а також створення груп;

6) проведення тестування розробленого месенджера для виявлення та усунення потенційних помилок і недоліків.

РОЗДІЛ 1
АНАЛІЗ РОЗРОБКИ МЕСЕНДЖЕРА З ВИКОРИСТАННЯ REACT, REACT NATIVE ТА ASP.NET CORE

1.1 Аналіз сучасного стану проблеми

Актуальність теми. У сучасному світі месенджери стали невід'ємною частиною повсякденного життя людей. Вони забезпечують миттєвий обмін текстовими повідомленнями, файлами, фото та відео між користувачами, дозволяючи їм підтримувати зв'язок з друзями, родиною та колегами. Розробка власного месенджера з використанням сучасних технологій, таких як React, React Native та ASP.NET Core, дозволить створити гнучкий, масштабований та захищений продукт, який відповідатиме потребам користувачів та забезпечить конкурентоспроможність на ринку.

Існують різноманітні месенджери, такі як WhatsApp, Telegram, Signal, Viber, Facebook Messenger та інші. Кожен з них має свої унікальні можливості та особливості, проте всі вони прагнуть забезпе чити зручний, безпечний та надійний спосіб спілкування.

При проектуванні свого чат-застосунка в якості метода аналізу вимог я обрав аналіз вже існуючих систем. Це ефективний метод виявлення вимог для розробки чат-застосунка. Цей метод дає можливість вивчити вже розроблені та успішно функціонуючі системи, щоб визначити їхні сильні та слабкі сторони, а також тренди в дизайні та функціональності. Це допомагає уникнути повторного винаходу колеса та зосередитися на розробці нових та інноваційних функцій. Аналіз існуючих систем є потужним методом виявлення та уточнення вимог до програмного забезпечення, особливо у динамічній галузі чат-застосунків. Цей підхід дозволяє не тільки ідентифікувати стандартні функції та виявити прогалини на ринку, але й встановити чіткі пріоритети в розробці, зосереджуючись на найбільш критичних аспектах.

Переваги методу:

1) ідентифікація базових функцій – визначення ключових елементів, без яких месенджер не може функціонувати;

2) розуміння користувацьких очікувань – виявлення стандартних функцій, які користувачі вважають обов'язковими;

3) встановлення пріоритетів – допомагає розрізняти між «must-have» та «nice-to-have» функціями;

4) планування етапів розвитку – дозволяє створити дорожню карту від MVP до повнофункціонального продукту;

5) зниження ризиків – фокусування на перевірених функціях зменшує ризик відторгнення продукту.

Тепер застосуємо цей метод аналізу на практиці та розглянемо кожен з вище згаданих месенджерів більш детально:

1) WhatsApp [1] є одним з найпопулярніших месенджерів у світі. Його перевагами є крос-платформеність, підтримка голосових та відеодзвінків, обмін файлами, створення груп;

2) Telegram [2] також дуже популярний завдяки зручному інтерфейсу, підтримці каналів та ботів. Він пропонує шифрування «end-to-end» для секретних чатів. Проте, як згадувалось раніше, частина його серверів розташована в росії, що є потенційним ризиком безпеки для деяких користувачів;

3) Signal [3] вважається одним з найбезпечніших месенджерів через використання потужного шифрування. Він має відкритий вихідний код і пропонує голосові/відеодзвінки, обмін файлами. Але Signal не такий багатофункціональний як інші месенджери;

4) Viber [4] має схожі можливості до WhatsApp і популярний в деяких регіонах. Але він закритий і належить японській компанії Rakuten.

Більшість популярних месенджерів є закритими системами, що не дозволяє користувачам повністю контролювати свої дані та конфіденційність. Крім того, вони можуть містити надлишкові функції, які не є необхідними для певних груп користувачів або вимагають додаткових ресурсів пристрою.

Серед молоді особливо популярним є месенджер Telegram. Однак через те, що частина серверів Telegram знаходиться на території росії, це створює потенційні ризики для безпеки та конфіденційності даних користувачів. Зокрема, українські військові не можуть використовувати Telegram через ці побоювання.

За допомогою аналізу Telegram визначимо обов'язковий базовий функціонал для чат-застосунку. Розглянемо базовий функціонал для мого месенджера:

1) автентифікація та авторизація – реалізація автентифікації на основі JWT та Refresh токенів, власна реєстрація з хешуванням пароля за допомогою бібліотеки BCrypt. У Telegram використовується реєстрація за номером телефону та двофакторна автентифікація;

2) додавання друзів – пошук за ім'ям користувача. У Telegram використовується пошук за ім'ям користувача або номером телефону;

3) особисті чати – текстові повідомлення з індикаторами статусу (відправлено, доставлено, прочитано), аналогічно до Telegram;

4) групові чати – створення груп з можливістю призначення адміністраторів. У Telegram є створення груп, додавання/видалення учасників;

5) створення каналів – базове створення публічних каналів з можливістю підписки. У Telegram є публічні та приватні канали для мовлення;

6) надсилання повідомлень – надсилання текстових повідомлень з використанням SignalR для real-time комунікації. У Telegram можна надсилати текст, зображення, файли.

У цій роботі я зосереджуюсь на розробці месенджера з мінімальним необхідним функціоналом. Аналіз Telegram допоміг мені визначити ці ключові функції:

1) автентифікація та авторизація;
2) додавання друзів;
3) особисті та групові чати;
4) канали;

5) надсилання текстових повідомлень.

Ці елементи формують міцну основу, без якої жоден месенджер не може функціонувати. Цей підхід узгоджується з принципами Lean Startup та Agile, де MVP (Minimum Viable Product) створюється для швидкого отримання зворотного зв’язку від користувачів.

Важливо відзначити, що мій вибір базових функцій не означає простоту їх реалізації. Навпаки, я приділяю особливу увагу якості кожної функції:

– використання ASP.NET Core з Domain Driven Design та Чистою архітектурою для гнучкого та підтримуваного бекенду;

– застосування CQRS для розділення операцій на запити та команди;

– фокус на безпеці з JWT, Refresh токенами та BCrypt.

Такий підхід не тільки дозволяє створити надійний базовий продукт, але й закладає фундамент для майбутнього розширення. Кожна з обраних технологій та практик (React з TypeScript, DDD, CQRS) має потенціал для значного масштабування.

Існуючі месенджери мають певні переваги та недоліки. Нова розробка може поєднати в собі кращі практики та врахувати потреби українських користувачів в безпеці, конфіденційності та незалежності від зовнішніх чинників. Розробка власного месенджера з відкритим вихідним кодом, заснованого на сучасних технологіях та принципах безпеки, дозволить забезпечити незалежність від зовнішніх факторів ризику та захистити дані українських користувачів. Тому існує потреба у розробці нового месенджера, який буде відповідати специфічним вимогам користувачів та забезпечувати високий рівень безпеки, конфіденційності та зручності використання. Застосунку, який би не мав потенційних загроз безпеці, пов’язаних із розташуванням серверів на території держави-агресора. Такий месенджер міг би використовуватися військовими, державними органами, а також усіма, хто цінує конфіденційність та безпеку своїх комунікацій. Цей месенджер може бути

орієнтований на певні групи користувачів, наприклад, співробітників організацій або учасників спеціалізованих спільнот.

Для реалізації цього проекту будуть використані сучасні технології та підходи, такі як React, React Native, ASP.NET Core, Domain-Driven Design, CQRS та інші. Це дозволить створити гнучку, масштабовану та легко модифіковану систему, яка зможе задовольнити потреби користувачів та адаптуватися до змін у майбутньому.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

Основною метою кваліфікаційної роботи є розробка безпечного та зручного месенджера з використанням сучасних технологій та підходів. Для досягнення цієї мети необхідно виконати наступні завдання:

1) аналіз існуючих рішень, їх переваг та недоліків, а також вимог користувачів до месенджерів;

2) проектування архітектури месенджера з використанням Domain-Driven Design та патерну CQRS для забезпечення гнучкості, масштабованості та можливості легко модифікувати систему в майбутньому;

3) розробка серверної частини (бекенд) месенджера з використанням ASP.NET Core, Entity Framework Core та інших сучасних бібліотек і фреймворків для забезпечення високої продуктивності та безпеки;

4) створення макету вебзастосунку та мобільного застосунку, реалізація клієнтської частини (фронтенд) месенджера з використанням React та React Native для створення зручного та інтуїтивно зрозумілого інтерфейсу користувача;

5) реалізація основних функцій месенджера, таких як обмін текстовими повідомленнями, додавання друзів, а також створення груп;

6) проведення тестування розробленого месенджера для виявлення та усунення потенційних помилок і недоліків.

Висновки до розділу 1

У цьому розділі було проведено ґрунтовний аналіз технологій та підходів, необхідних для розробки сучасного месенджера. Ключові висновки:

1) проаналізовано сучасний стан проблеми та необхідність створення нового месенджера, який би задовольняв потреби користувачів у безпеці, конфіденційності та зручності використання. Особливо гостро це питання постає для українських державних установ та військових через ризики, пов’язані з використанням месенджерів, частина інфраструктури яких знаходиться на території росії;

2) досліджено існуючі популярні месенджери-аналоги, такі як WhatsApp, Telegram, Signal, Viber. Проаналізовано їхні переваги, недоліки та функціональні можливості. З’ясовано, що не всі з них повністю задовольняють вимоги щодо безпеки, конфіденційності та незалежності від зовнішніх факторів ризику;

3) сформульовано завдання на кваліфікаційну роботу бакалавра, які включають проектування архітектури, розробку серверної та клієнтської частин, інтеграцію механізмів шифрування, реалізацію основних функцій месенджера, тестування та документування;

4) визначено, що розробка нового месенджера буде здійснюватися з використанням сучасних технологій та підходів, таких як ASP.NET Core, React, React Native, Domain-Driven Design, CQRS та низки бібліотек для забезпечення високої продуктивності, безпеки та гнучкості системи.

Таким чином, виконано аналіз предметної області та обґрунтовано доцільність створення нового безпечного та зручного месенджера, орієнтованого на потреби українських користувачів.

РОЗДІЛ 2
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

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

Після аналізу різних моделей, таких як Entity-Relationship Diagram (ERD) та Business Process Model and Notation (BPMN), було обрано UML через ряд ключових переваг:

1) універсальність – UML пропонує широкий набір діаграм для моделювання структури, поведінки та архітектури системи;

2) відповідність DDD та CQRS – UML ідеально підходить для зображення агрегатів, сутностей, об’єктів значень та потоків команд/запитів;

3) зрозумілість – графічна нотація UML полегшує комунікацію з нетехнічними зацікавленими сторонами.

На відміну від ERD, обмеженої моделюванням структури даних, і BPMN, орієнтованої на бізнес-процеси, UML є універсальним інструментом для всебічного технічного проектування месенджера.

При належному застосуванні Domain-Driven Design для фокусування на ключових аспектах, UML забезпечить потужний і гнучкий інструментарій для моделювання месенджера на всіх рівнях архітектури. Для моделювання моєї системи месенджера обрано об’єктно-орієнтований підхід у поєднанні з принципами Domain-Driven Design (DDD). Цей вибір обумовлений наступними факторами:

1) допомагає точно змоделювати складну предметну область месенджера з численними сутностями та бізнес-правилами;

2) забезпечує гнучкість та здатність адаптуватися до змін вимог завдяки інкапсуляції та поліморфізму;

3) моделі відображають реальну бізнес-логіку через агрегати, сутності та об’єкти значень;

4) розділення на обмежені контексти сприяє масштабованості та незалежній роботі команд;

5) агрегати гарантують безпеку та цілісність даних, будучи «охоронцями» бізнес-правил;

6) локалізація змін завдяки інкапсуляції;

7) код стає зрозумілішим, відображаючи термінологію предметної області.

DDD модель для чат-застосунка може включати такі доменні об’єкти:

1) агрегат «Користувач» (User) – цей об’єкт описує користувача застосунку, включаючи його ім’я, email, пароль тощо та складається з сутностей refresh tokens (використовується для безпечного управління сеансами аутентифікації користувачів), друзі (Friends) (представляє зв’язки між користувачами всередині месенджера), запити на дружбу (Friend Requests) (керує вхідними та вихідними запитами на дружбу між користувачами), сповіщення (Notifications) (обробляє сповіщення або повідомлення, які користувачі отримують щодо активності в додатку);

2) агрегат «Чат» (Chat) – цей об’єкт описує чат між двома користувачами у вигляді діалогу, або більшою кількістю користувачів у вигляді групи, де вони можуть спілкуватися між собою та містить в собі сутності учасники чату (Chat Members), повідомлення (Messages), вкладення (Attachments), ролі учасників чату (Chat Members Roles) (визначає різні дозволи або ролі, призначені користувачам всередині контексту чату);

3) агрегат «Канал» (Channel) – цей об’єкт описує канал, до якого можуть приєднатися користувачі для створення постів та їх обговорення. Агрегат містить сутності учасники каналу (Channel Members), пости каналу (Channel Posts), вкладення (Attachments) (в контексті каналу), ролі членів каналу (Channel Members Roles) (конкретні дозволи або ролі, призначені користувачам всередині середовища каналу).

На рисунку 2.1 зображено діаграму з описаними вище агрегатами та сутностями.

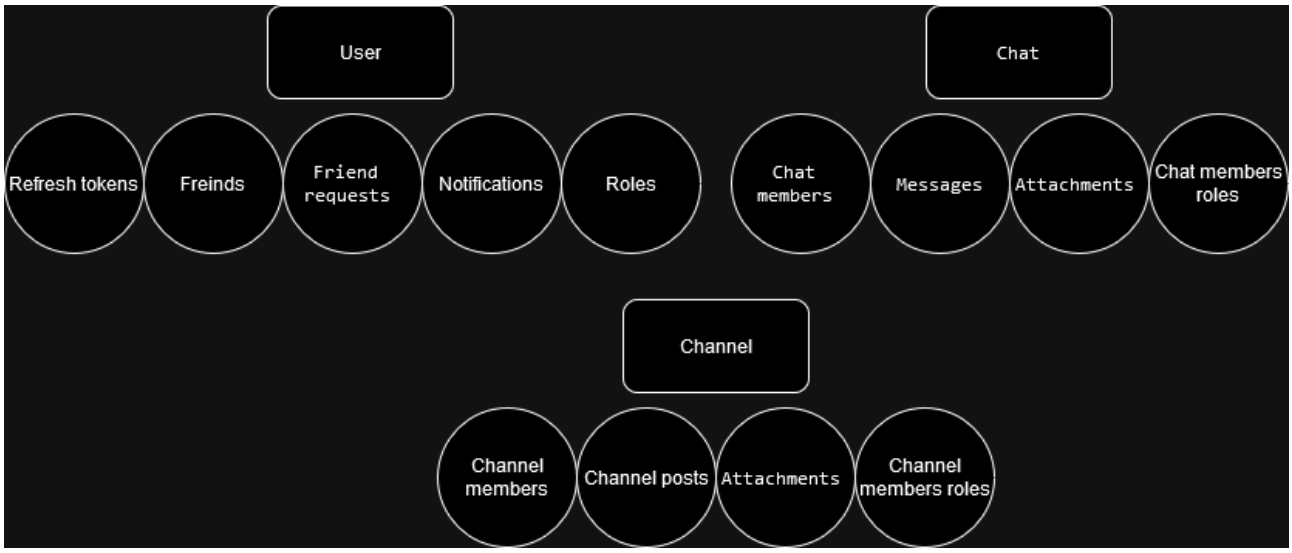

Рисунок 2.1 – Діаграма агрегатів та сутностей

Ці об’єкти мають зв’язки між собою, наприклад, повідомлення може бути пов’язано з користувачем, який його надіслав, а чат може містити багато повідомлень.

Отже, об’єктно-орієнтований підхід у поєднанні з Domain-Driven Design найкраще підходить для моделювання нашого месенджера. Він дозволяє точно відобразити складний домен, забезпечує безпеку та цілісність даних, полегшує розвиток та розуміння системи. Це особливо важливо, враховуючи високі вимоги до безпеки та конфіденційності у нашому проекті.

Перейдемо до огляду діаграми сценаріїв використання (Use Case Diagram). Вона є важливим інструментом для моделювання взаємодії користувачів з системою. Ця діаграма дозволяє візуально представити функціональність системи та її взаємодію з користувачами, що спрощує розуміння вимог та очікувань до програмного забезпечення. На цій діаграмі відображаються основні дії, які можуть виконувати різні користувачі системи, а також ролі користувачів.

Метою діаграми сценаріїв використання є визначення та документування всіх можливих взаємодій між користувачами та системою. Це допомагає зрозуміти, які функції повинні бути реалізовані, які користувачі будуть взаємодіяти з системою, та які їхні ролі й обов'язки. В результаті, діаграма допомагає забезпечити повне покриття вимог до системи та сприяє більш ефективному проектуванню та розробці програмного забезпечення. Отже, основною метою діаграми сценаріїв використання є представлення функціональних вимог до системи в зрозумілій і наочній формі. Вона використовується для:

1) виявлення вимог – діаграма допомагає визначити, які функції необхідні для реалізації системи, виходячи з потреб користувачів та інших зацікавлених сторін;

2) моделювання взаємодії користувачів із системою – діаграма відображає, як різні типи користувачів (акторів) взаємодіють із системою через сценарії використання;

3) документування вимог – діаграма є важливим елементом технічної документації, який описує функціональні можливості системи та її поведінку;

4) планування та розподіл завдань – діаграма допомагає визначити обсяг роботи та розподілити завдання серед членів команди розробників;

Ця діаграма є невід'ємною частиною процесу розробки програмного забезпечення, яка допомагає зрозуміти та задокументувати функціональні вимоги до системи. Вона забезпечує чітке уявлення про те, як користувачі взаємодіють із системою, і допомагає ефективно планувати та реалізовувати функціональні можливості. Для системи розроблюваної системи діаграма сценаріїв використання є важливим інструментом, який допомагає визначити ключові функції та забезпечити відповідність системи вимогам користувачів і зацікавлених сторін.

Основні сценарії використання включають реєстрацію, вхід до системи, надсилання повідомлень, перегляд чатів, управління дружбами та запитами в

друзі, управління каналами, отримання сповіщень та управління профілем. На рисунку 2.2 можна побачити діаграму сценаріїв використання застосунку.

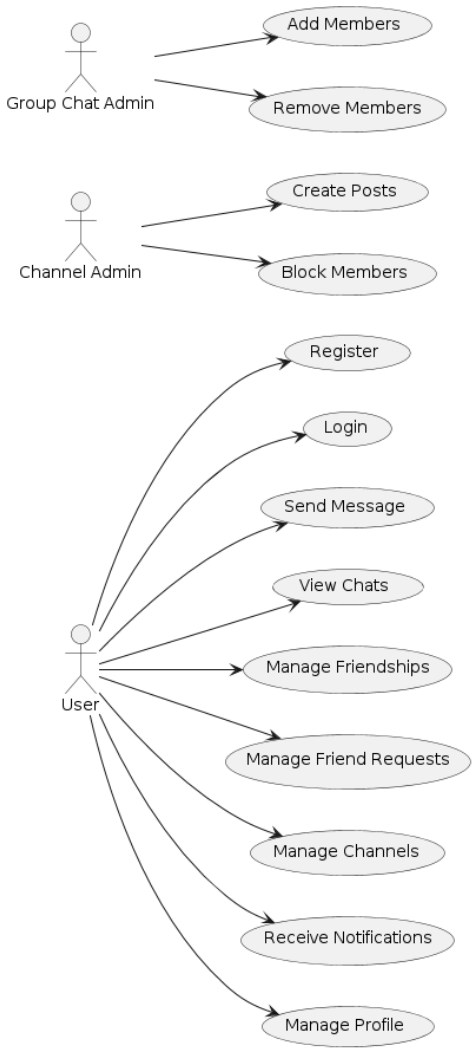

Рисунок 2.2 – Діаграма сценаріїв використання

Діаграма сценаріїв використання для системи чат-застосунку моделює ключові функціональні можливості, які система повинна надавати користувачам і адміністраторам.

Розглянемо наступну діаграму – діаграму послідовності. Це тип діаграми UML, який використовується для моделювання послідовності взаємодій між різними об’єктами або компонентами в системі впродовж певного процесу або сценарію. У цьому випадку діаграма послідовності ілюструє процес взаємодії

між користувачами (UA та UB), фронтенд-додатками (ReactAppA та ReactAppB), контролером повідомлень (MessageController), сервісом повідомлень (MessageService), сервісом сповіщень (NotificationService), а також обробником команд (CommandHandler) та репозиторієм чату(ChatRepository). На рисунку 2.3 можна побачити діаграму послідовностей, яка ілюструю процес відправки повідомлення.

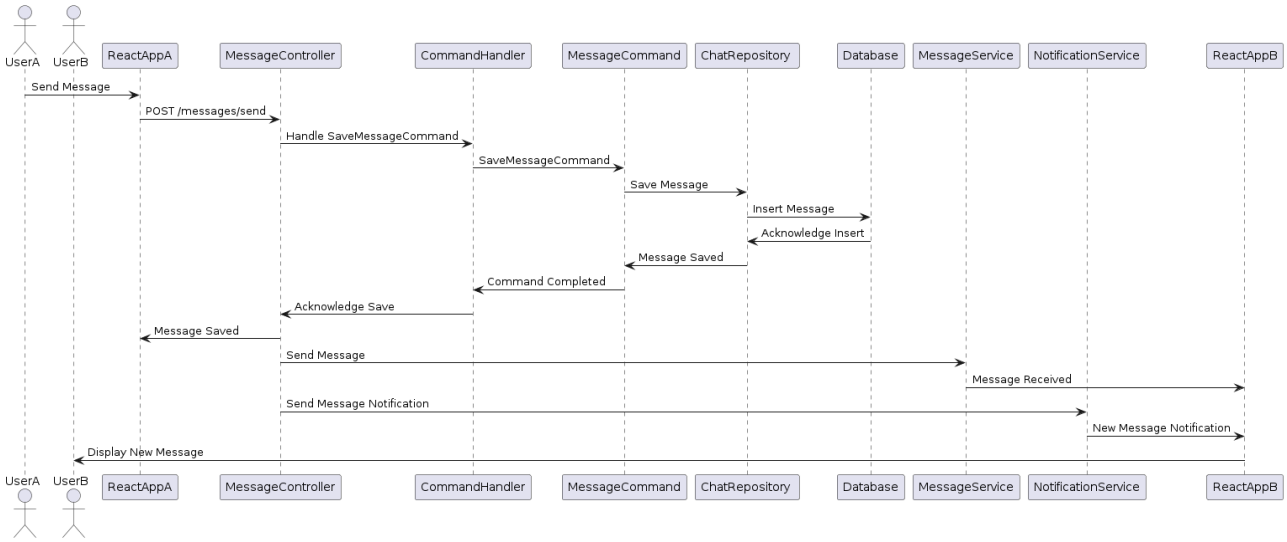

Рисунок 2.3 – Діаграма послідовності, яка ілюструє процес відправлення повідомлення

Процес починається з того, що користувач UA відправляє повідомлення через фронтенд-додаток ReactAppA. Фронтенд надсилає запит на створення повідомлення до контролера повідомлень (MessageController). Контролер обробляє запит та передає його обробнику команд (CommandHandler), який в свою чергу створює команду для збереження повідомлення (MessageCommand). Після цього команда обробляється, що включає збереження повідомлення в базі даних (ChatRepository), а також надсилання повідомлення за допомогою сервісу повідомлень (MessageService). Після успішного збереження та надсилання повідомлення, користувач UB через фронтенд-додаток ReactAppB отримує повідомлення та сповіщення про нього.

Така діаграма допомагає візуалізувати послідовність подій та взаємодій між різними частинами системи під час виконання конкретного сценарію.

Дизайн інтерфейсу є ключовим фактором успіху будь-якого мобільного додатку, і месенджер не є винятком. Кольорова схема та загальний стиль інтерфейсу повинні бути не лише візуально привабливими, але й сприяти зручному й ергономічному користуванню.

Наступний крок – створення макетів сторінок. Розглянемо використані мною інструменти.

Coolors.co – цей потужний генератор кольорових схем пропонує безліч гармонійних комбінацій. На рисунку 2.4 можна побачити сторінку з популярними кольоровими палітрами цього сайту.

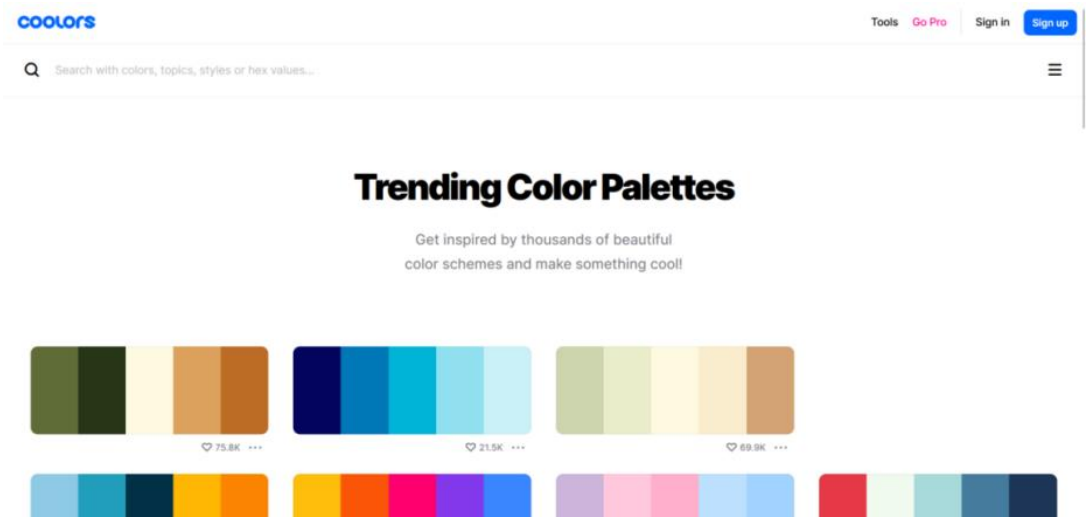

Рисунок 2.4 – Сторінка з палітрами на сайті Coolors [8]

Сайт дозволяє легко підбирати кольори, що добре поєднуються, враховуючи контраст та доступність. Це допомогло мені знайти приємну для очей і водночас функціональну кольорову схему.

Adobe Color (color.adobe.com) – цей інструмент від Adobe відомий своїм науковим підходом до теорії кольорів.Використовуючи «колірне колесо» (Color Wheel) у Adobe Color, я підібрав ідеальний яскравий, але не занадто агресивний

акцентний колір, який гармонійно поєднується з основною палітрою месенджера. Акцентний колір привертає увагу до ключових елементів (кнопки відправки, нові повідомлення). Для темного синьо-сірого інтерфейсу я вибрав пурпуровий акцент. На рисунку 2.5 можна побачити колірне колесо для підбору кольорової палітри.

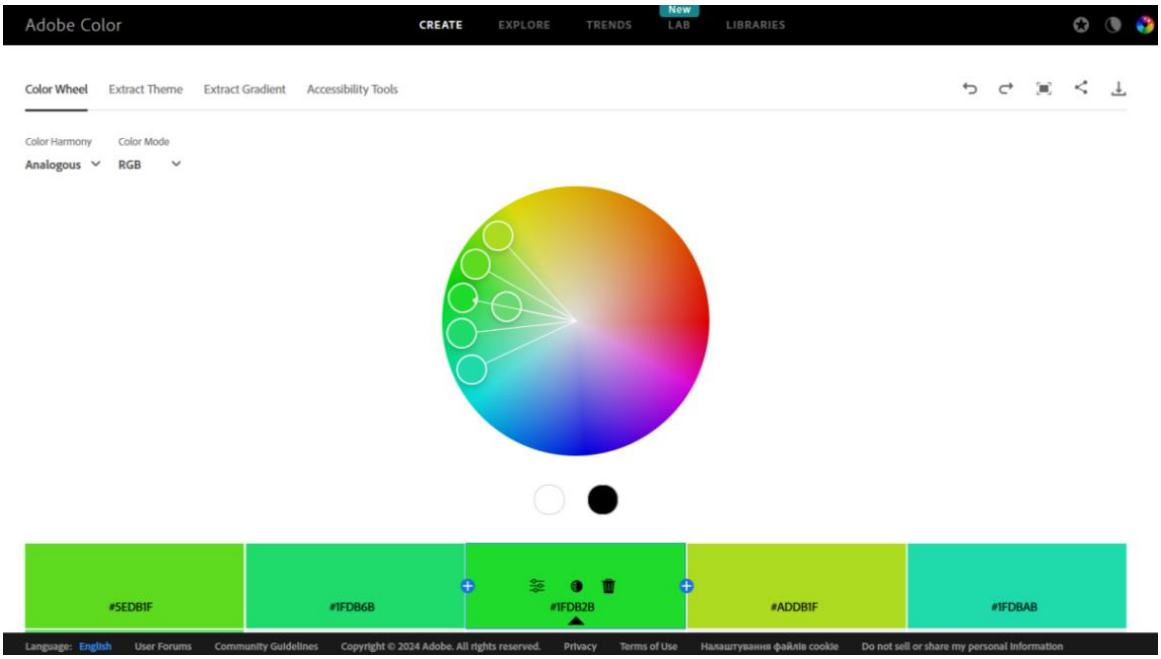

Рисунок 2.5 – Колірне колесо для підбору кольорової палітри на сайті Adobe [9]

Для створення макетів інтерфейсу я використовував Figma [10] – один з найпопулярніших і найпотужніших інструментів у сучасному UI/UX дизайні. Figma має ряд переваг, які роблять його ідеальним для проектування месенджера: спільну роботу в реальному часі, кросплатформеність, потужні інструменти дизайну, компонентний підхід, можливість прототипування, інтеграцію з розробкою та версійність.

Я почав зі створення головної сторінки. Figma надає набір зручних готових компонентів, які можна переналаштувати під свої потреби. Головна сторінка складається з компоненту бокового меню, на якому можна обрати вкладки зі списком чатів, дзвінків, друзів та груп або відкрити налаштування, списку чатів

і компонента чату з повідомленнями. На рисунку 2.6 можна побачити макет головної сторінки застосунку.

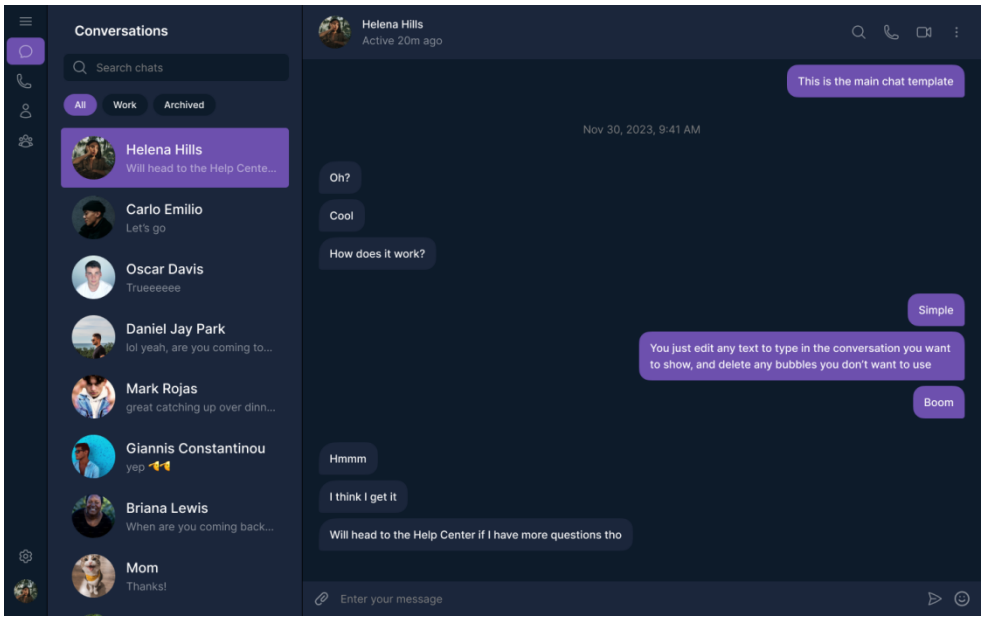

Рисунок 2.6 – Макет головної сторінки

Далі було створено макет сторінки для реєстрації нових користувачів. Його зображено на рисунку 2.7.

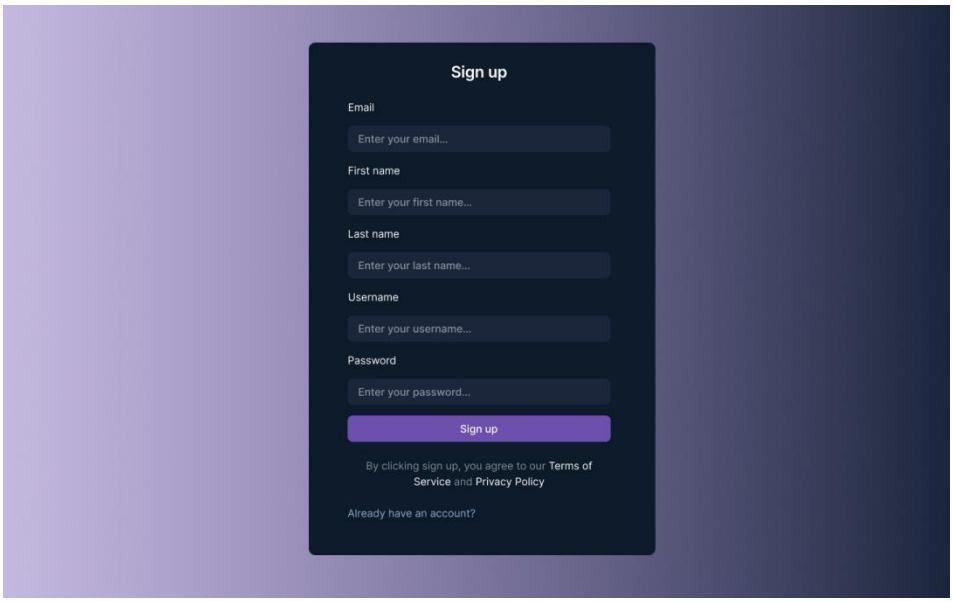

Рисунок 2.7 – Макет сторінки реєстрації

Дуже подібною до сторінки реєстрації було створено макет сторінки входу для вже зареєстрованих користувачів. Її можна побачити на рисунку 2.8.

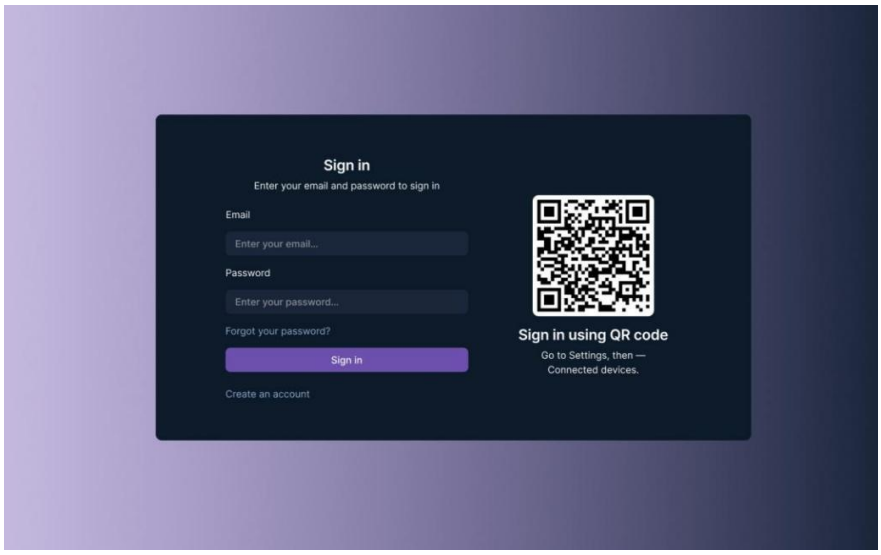

Рисунок 2.8 – Макет сторінки для входу

Дотримуючись цього ж дизайну і з використанням наявних компонентів було розроблено аналогічні компоненти для мобільної версії застосунку. Сторінку зі списком чатів зображено на рисунку 2.9.

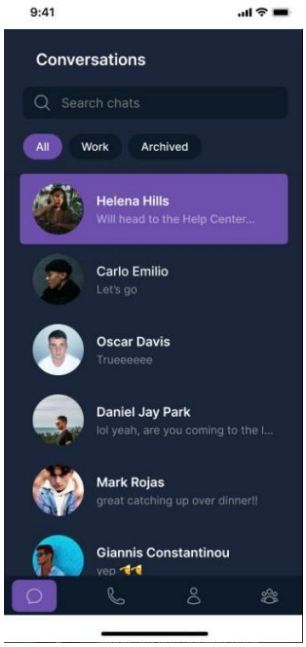

Рисунок 2.9 – Список чатів у макеті для мобільного пристрою

На рисунку 2.10 зображено вигляд діалогу на мобільному пристрої.

Рисунок 2.10 – Макет діалогу на мобільному пристрої

Подібними до веб версії застосунку було створено макети для реєстрації та входу. Перший можна побачити на рисунку 2.11.

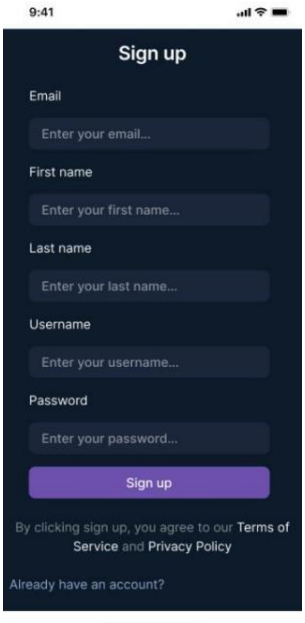

Рисунок 2.11 – Мобільна версія макету для сторінки реєстрації

На рисунку 2.12 можна побачити макет мобільної версії сторінки входу.

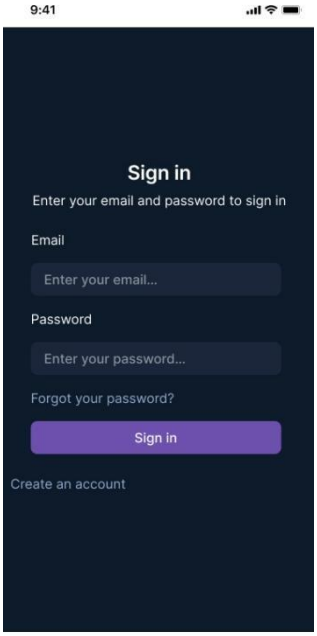

Рисунок 2.12 – Мобільна версія макету для сторінки входу

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір мов програмування є одним з найважливіших етапів розробки будь-якого програмного забезпечення. Від цього вибору залежить не лише продуктивність та масштабованість майбутнього додатку, але й те, скільки часу та ресурсів буде витрачено на його створення.

У моєму випадку, де я планую розробити месенджер, який поєднуватиме в собі фронтенд, бекенд та мобільний додаток, я маю ретельно підійти до вибору мов програмування, зваживши всі «за» та «проти» кожної з них.

JavaScript – популярна мова для розробки динамічних веб-інтерфейсів, проте через динамічну типізацію код може бути складним для читання та підтримки. TypeScript вирішує цю проблему, додаючи статичну типізацію до JavaScript. Це робить код надійнішим, більш читабельним та легшим у підтримці.

C# – потужна мова для розробки веб-сервісів та бекенду, з високою продуктивністю, модульністю та інтеграцією з .NET Framework. Чудовий вибір для бекенду месенджера завдяки зручним інструментам для роботи з базами даних.

Java – популярна мова для бекенду та мобільних додатків, з платформною незалежністю, високою продуктивністю та великою спільнотою. Альтернатива C# з аналогічними можливостями.

Python – проста, читабельна та універсальна мова. Може бути корисною для прототипування, автоматизації та скриптів тестування, хоча її продуктивність може бути нижчою, ніж у C# та Java для бекенду месенджера.

На рисунку 2.13 можна побачити діаграму з мовами програмування, які розміщенні за популярністю.

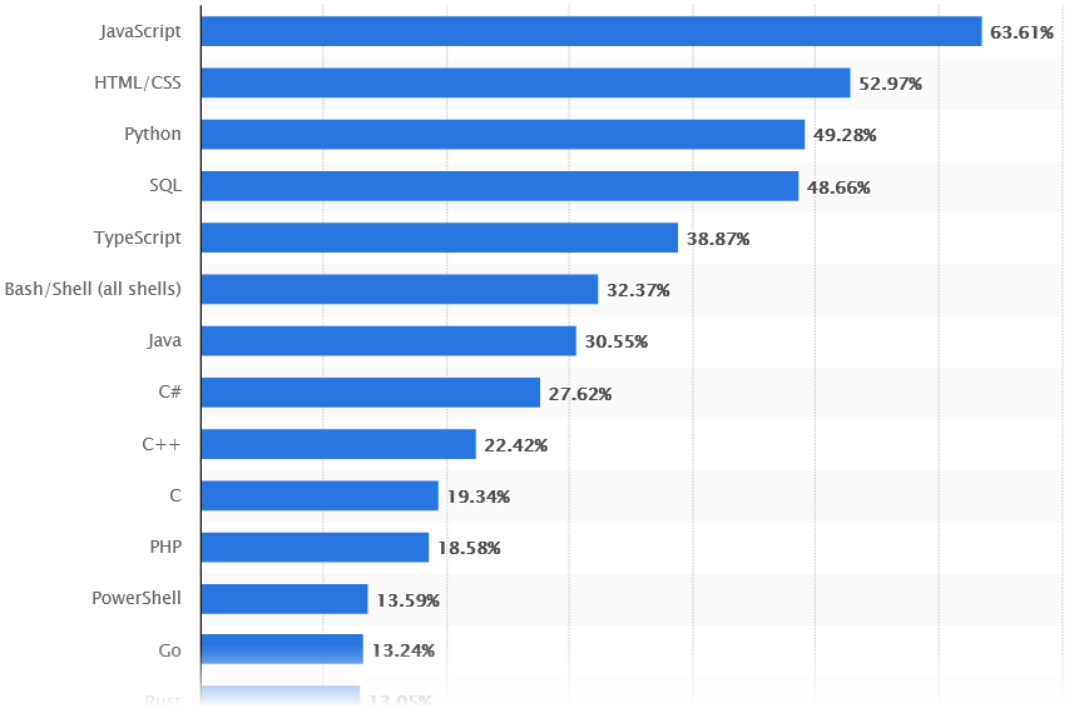

Рисунок 2.13 – Популярні мови програмування серед розробників [11]

Вибір мов програмування для розробки месенджера залежить від багатьох факторів, таких як особисті вподобання, досвід розробки, доступні ресурси та

майбутні плани розвитку проекту. В даний час я планую використовувати TypeScript для фронтенду та C# для бекенду мого месенджера. Цей вибір ґрунтується на моєму досвіді роботи з цими мовами, їхніх можливостях та доступності зручних інструментів для розробки.

Вибір правильного середовища розробки (IDE) є ключовим фактором для продуктивної та ефективної роботи над програмним проектом. IDE може значно спростити процес написання коду, тестування, відлагодження та розгортання. Зважаючи на те, що я планую використовувати C# для back-end та TypeScript для фронтенду, а також React для створення динамічних інтерфейсів, мені потрібна IDE, яка буде підтримувати всі ці технології.

Для розробки месенджера з використанням C#, TypeScript та React були розглянуті такі популярні IDE: Visual Studio, Visual Studio Code та Rider.

Зважаючи на всі фактори, я обрав Visual Studio Community Edition через такі причини:

1) безкоштовна IDE;
2) повна підтримка C# та .NET;
3) підтримка TypeScript та React;
4) вбудована інтеграція з Git.

Visual Studio Community Edition поєднує необхідні можливості для розробки всіх компонентів месенджера – бекенду на C#, фронтенду на TypeScript/React, а також забезпечує зручне середовище для командної роботи завдяки Git інтеграції. Цей вибір дозволяє ефективно розробляти повнофункціональний месенджер в одному потужному IDE.

Архітектура програмного забезпечення є ключовим фактором, який визначає його стійкість, масштабованість та загальну якість. У моєму випадку, де я планую розробити месенджер, який поєднує в собі фронтенд, бекенд та мобільний додаток, я ретельно підійшов до вибору архітектурного стилю, зваживши всі «за» та «проти» кожного з них. І ось чому я вирішив обрати Чисту архітектуру [15]:

1) структурованість та модульність – чітке розмежування компонентів за принципом розділених інтерфейсів полегшує розробку, тестування та підтримку різних частин системи (фронтенду, бекенду, мобільного додатку);

2) масштабованість – архітектура дозволяє легко додавати нові функції без переписування існуючого коду;

3) незалежність від технологій – можливість вільно обирати будь-які технології не впливаючи на загальну структуру;

4) легкість тестування – модульне тестування кожного компонента окремо;

5) зрозумілість та гнучкість – архітектурні принципи роблять код більш зрозумілим для розробників та гнучким для внесення змін.

На рисунку 2.14 можна побачити зв’язок між компонентами з використанням Чистої архітектури.

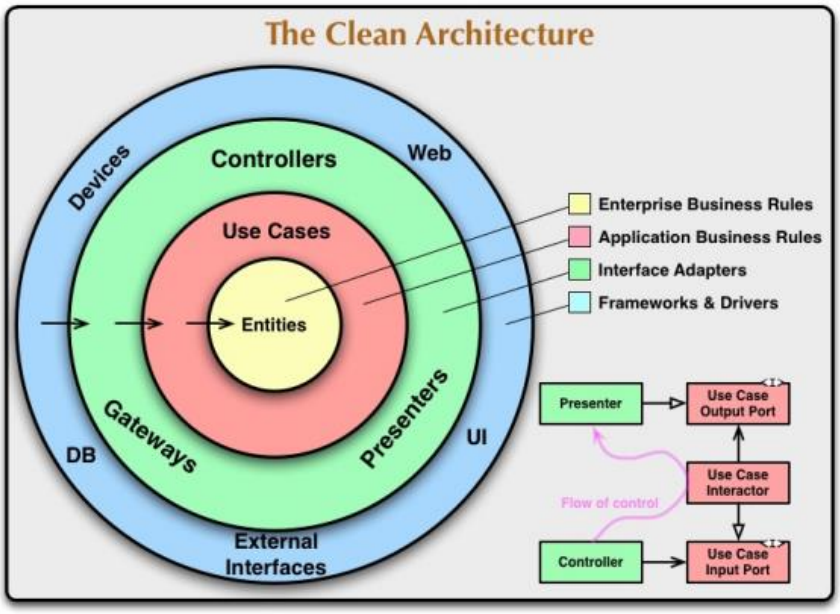

Рисунок 2.14 – Структура проекту з використанням Чистої архітектури [16]

Архітектура проекту побудована за принципами Чистої архітектури та включає чотири основні шари: Domain, Application, Infrastructure та Presentation.

Кожен з цих шарів виконує свою функцію і забезпечує високу якість, гнучкість і підтримуваність системи:

– Domain – містить основні бізнес-правила і логіку. Це серце системи, яке повинно залишатися незалежним від будь-яких зовнішніх шарів та фреймворків. Складається з таких компонентів:

1) Entities – основні бізнес-об’єкти, що представляють користувачів, повідомлення, дружні стосунки та канали;

2) Value Objects – незмінні об’єкти, що представляють цінності, такі як адреси або кількості;

3) Aggregates – групи пов’язаних об’єктів, що вважаються єдиною одиницею даних;

4) Domain Services – сервіси, що містять бізнес-логіку, яка не може бути реалізована у вигляді властивостей або методів об’єктів;

– Application – цей шар відповідає за реалізацію сценаріїв використання системи. Він орієнтований на виконання команд та обробку запитів:

1) Use Cases (Commands, Queries) – операції, що виконують бізнес-логіку, як-от відправлення повідомлення, додавання друзів, створення каналу тощо;

2) DTOs (Data Transfer Objects) – об’єкти, що використовуються для передачі даних між шарами системи;

3) Interfaces – контракти для реалізації в інших шарах (наприклад, репозиторії);

– Infrastructure – цей шар забезпечує технічну реалізацію та взаємодію з зовнішніми системами. Він містить реалізації інтерфейсів з шару Application та інфраструктурні сервіси:

1) Repositories – класи для роботи з базою даних, що реалізують інтерфейси з шару Application;

2) External Services – інтеграції з зовнішніми системами, такими як служби електронної пошти або бібліотеки для роботи з SignalR;

3) ORM (Object-Relational Mapping) – інструменти для роботи з базою даних, як-от Entity Framework;

– Presentation – відповідає за взаємодію з користувачем та представлення даних. Він містить інтерфейси користувача та API-контролери:

1) Controllers – контролери для обробки HTTP-запитів;

2) Views – представлення даних користувачу;

3) ViewModels – моделі даних, що використовуються у представленнях;

4) UI (User Interface) – інтерфейс користувача, що може бути реалізований за допомогою таких технологій, як React або Angular.

Зважаючи на всі вищезазначені фактори, я вважаю, що Чиста архітектура є найкращим вибором для розробки мого месенджера. Ця архітектура забезпечить стійкість, масштабованість, гнучкість та загальну якість коду, що є важливими факторами для успішного проекту.

Патерн CQRS [17] (Command Query Responsibility Segregation) був обраний через такі ключові переваги:

– масштабованість та продуктивність – роздільні моделі для операцій читання (запити) та запису (команди) дозволяють оптимізувати кожну модель для її завдання. Це критично для високонавантаженого месенджера з інтенсивними операціями читання

– структурованість – чітке розмежування обов’язків читання та запису даних робить код більш модульним та зрозумілим;

– покращена стійкість до помилок – відокремлені моделі мінімізують взаємний вплив помилок між операціями читання та запису, забезпечуючи високу доступність;

– можливість розширення – завдяки розділенню відповідальності, систему легше розширювати новими функціями без переписування існуючого коду.

Отже, CQRS ідеально підходить для створення високопродуктивного, масштабованого, стійкого до помилок та легко розширюваного месенджера з

чітко структурованою кодовою базою. У проекті реалізовано CQRS за допомогою таких компонентів:

– команди (Commands) – використовуються для зміни стану системи. Кожна команда відповідає за одну операцію і містить всі необхідні дані для її виконання;

– запити (Queries) – використовуються для отримання даних з системи. Кожен запит відповідає за отримання певного набору даних;

– обробники команд (Command Handlers) – реалізують логіку виконання команд. Вони приймають команди і викликають відповідні методи на рівні домену або інфраструктури;

– обробники запитів (Query Handlers) – реалізують логіку обробки запитів. Вони приймають запити і повертають дані у відповідь.

Domain-Driven Design [18] (DDD) — це підхід до розробки програмного забезпечення, який фокусується на глибокому розумінні домену (сфери діяльності) та тісній співпраці з експертами в цій сфері. Основна мета DDD — створення програмних систем, які відображають складність і логіку домену, для якого вони призначені.

Основні принципи DDD:

– відділення домену від інфраструктури – відділення бізнес-логіки від технічних аспектів, що дозволяє зосередитись на основній суті системи;

– спільна мова (Ubiquitous Language) – всі учасники проекту (розробники, аналітики, експерти домену) використовують однакову мову для опису бізнес-процесів і логіки;

– об’єктно-орієнтований підхід – використання об’єктів для моделювання елементів домену і їх поведінки.

У проекті використовується архітектура Domain-Driven Design (DDD) для чіткого відокремлення бізнес-логіки від інфраструктури та забезпечення гнучкості й підтримуваності системи. Ключові елементи DDD, такі як сутності,

об’єкти значень, агрегати, репозиторії, доменні сервіси та фабрики, використовуються для структурування бізнес-логіки.

Шар Domain містить основну бізнес-логіку й правила домену, включаючи моделі сутностей (користувачі, чати, повідомлення), агрегати (наприклад, агрегат «Чат») та інтерфейси репозиторіїв.

Шар Application координує діяльність між різними компонентами системи, використовуючи CQRS. Він містить команди та обробники для операцій, що змінюють стан системи, а також запити та обробники для операцій, що отримують дані.

Шар Infrastructure містить реалізації інтерфейсів для доступу до бази даних та інших зовнішніх систем, таких як репозиторії для доступу до бази даних і сервіси для інтеграції з іншими системами (електронна пошта, SignalR для обміну повідомленнями в реальному часі).

Шар Presentation відповідає за взаємодію з користувачем і містить контролери для взаємодії з фронтендом через API.

Ця архітектура забезпечує чітке розділення відповідальностей, модульність і можливість розширення системи в майбутньому.

Наступний крок – моделювання бази даних. Воно допомагає візуалізувати структуру даних і зв’язки між таблицями, що полегшує розуміння предметної області. Крім того, моделювання дозволяє виявити та уникнути потенційних проблем, таких як аномалії оновлення чи надлишковість даних, на ранній стадії проектування. Діаграма таблиць також забезпечує цілісність даних через визначення первинних і зовнішніх ключів та інших обмежень. Вона полегшує комунікацію між різними зацікавленими сторонами за допомогою наочного представлення структури даних і забезпечує узгодженість та стандартизацію, що полегшує подальше розширення та інтеграцію з іншими системами. На основі створеної діаграми агрегатів та сутностей з використанням інструментів dbdiagram [19] було створено діаграму таблиць, зображену на рисунку 2.15.

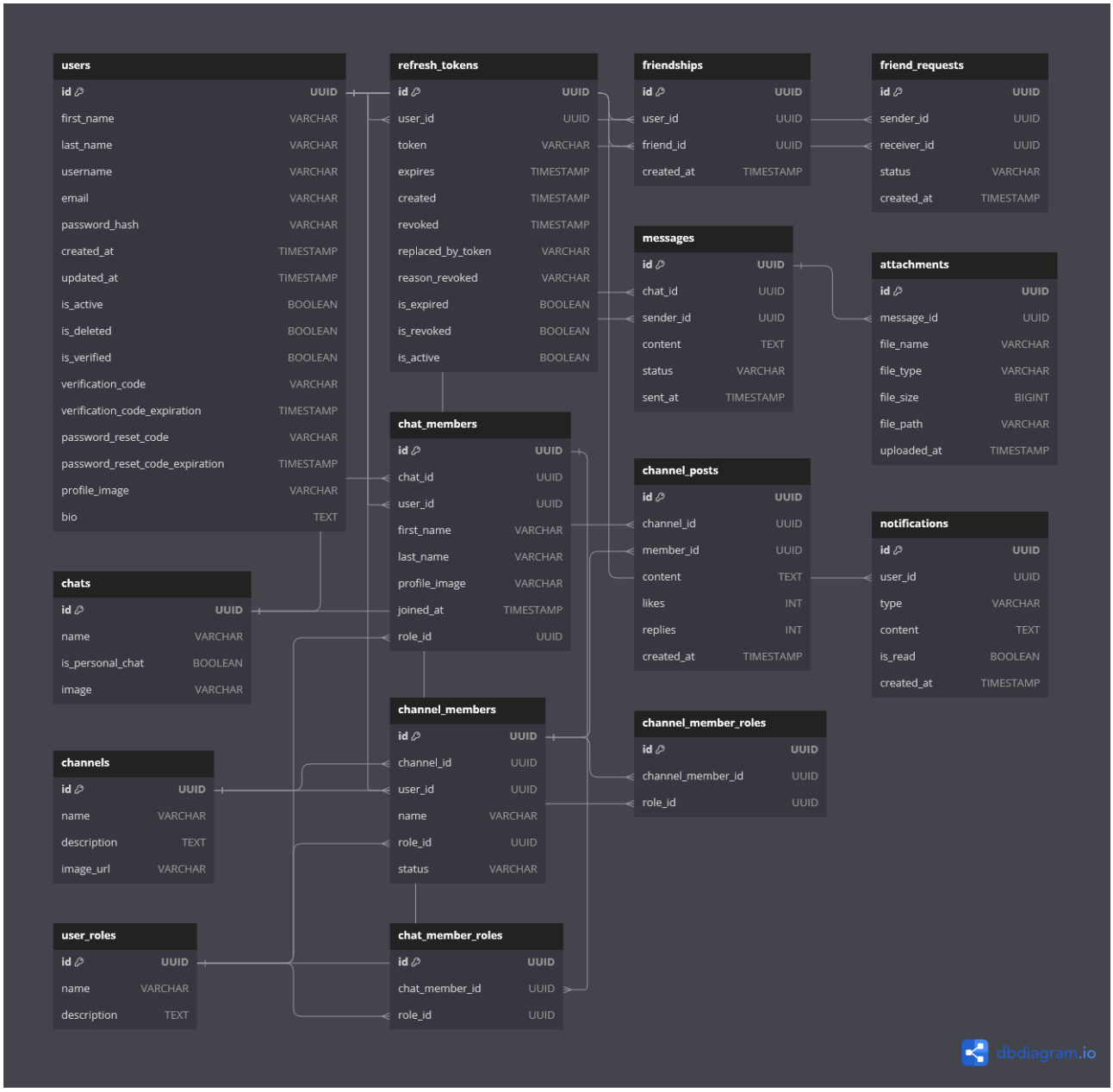

Рисунок 2.15 – Діаграма таблиць на основі агрегатів та сутностей

Я створив ось такі таблиці:

1) users – таблиця для зберігання інформації про користувачів, включаючи їх імена, електронні адреси, хешовані паролі та інші дані профілю. Ця таблиця має зв’язки з багатьма іншими таблицями через зовнішні ключі;

2) refresh_tokens – таблиця для зберігання рефреш-токенів, пов’язаних з користувачами через user_id;

3) friendships – таблиця для зберігання інформації про дружні зв’язки між користувачами, пов’язана з users через user_id та friend_id;

4) friend_requests – таблиця для зберігання запитів на дружбу, пов’язана з users через sender_id та receiver_id;

5) chats – таблиця для зберігання інформації про чати;

6) chat_members – таблиця для зберігання інформації про членів чатів, пов’язана з users через user_id та з chats через chat_id;

7) messages – таблиця для зберігання повідомлень в чатах, пов’язана з users через sender_id та з chats через chat_id;

8) attachments – таблиця для зберігання вкладень повідомлень, пов’язана з messages через message_id;

9) channels – таблиця для зберігання інформації про канали;

10) channel_members – таблиця для зберігання інформації про членів каналів, пов’язана з users через user_id та з channels через channel_id;

11) channel_posts – таблиця для зберігання постів у каналах, пов’язана з channel_members через member_id та з channels через channel_id;

12) notifications – таблиця для зберігання сповіщень, пов’язана з users через user_id;

13) user_roles – таблиця для зберігання ролей користувачів;

14) chat_member_roles – таблиця для зберігання ролей членів чатів, пов’язана з chat_members та user_roles;

15) channel_member_roles – таблиця для зберігання ролей членів каналів, пов’язана з channel_members та user_roles.

Більшість таблиць пов’язані з users через зовнішні ключі, що дозволяє зберігати дані про користувачів та їх взаємодію в системі.

Вибір СУБД є критичним етапом розробки. Для месенджера на ASP.NET Core було обрано Microsoft SQL Server через низку переваг: тісну інтеграцію з .NET екосистемою, високу продуктивність, відмінну сумісність з Entity Framework Core, надійні функції безпеки, підтримку JSON, доступність

безкоштовної версії SQL Server Express та багатий набір інструментів розробки. MSSQL забезпечує безпроблемну сумісність, оптимізовану продуктивність, зручне моделювання даних, міграції та запити, що є ідеальним рішенням для розробки чат-застосунку з інтенсивним обміном даними. В цілому, MSSQL пропонує всі необхідні характеристики для створення надійного, ефективного та масштабованого чат-застосунка на базі ASP.NET Core.

Діаграма компонентів є важливим інструментом в розробці програмного забезпечення, який дозволяє візуалізувати високорівневу архітектуру системи. Вона допомагає розробникам зрозуміти взаємозв'язки між різними компонентами системи, а також їх взаємодію. У даній діаграмі компонентів ми розглянемо архітектуру системи соціальної мережі, яка включає різні контролери, обробники, команди, запити, репозиторії, сервіси та хаби. Ця структура є основою для забезпечення функціональності додатку, такої як реєстрація, автентифікація, обмін повідомленнями, додаванням та видаленням друзів та каналами.

Система складається з інтерфейсу користувача ReactApp, розробленого на базі React, який забезпечує взаємодію користувача з системою. Контролери, такі як AuthController та інші контролери, виконують управління відповідними сутностями системи. Обробники команд (CommandHandler) та запитів (QueryHandler) виконують операції створення, видалення, оновлення та отримання даних з репозиторіїв, використовуючи відповідні команди та запити. Команди, такі як AuthCommand та інші, пов'язані з управлінням відповідними сутностями. Запити як от AuthQuery та інші використовуються для отримання даних з репозиторіїв UserRepository, ChatRepository та ChannelRepository відповідно. Сервіси EmailService та SignalRService забезпечують відправку електронних листів та зв'язок в реальному часі, а NotificationService керує сповіщеннями, взаємодіючи з SignalRService та EmailService. ChatHub забезпечує зв'язок в реальному часі між користувачами для обміну

повідомленнями через SignalRService. На рисунку 2.16 продемонстровано діаграму компонентів системи.

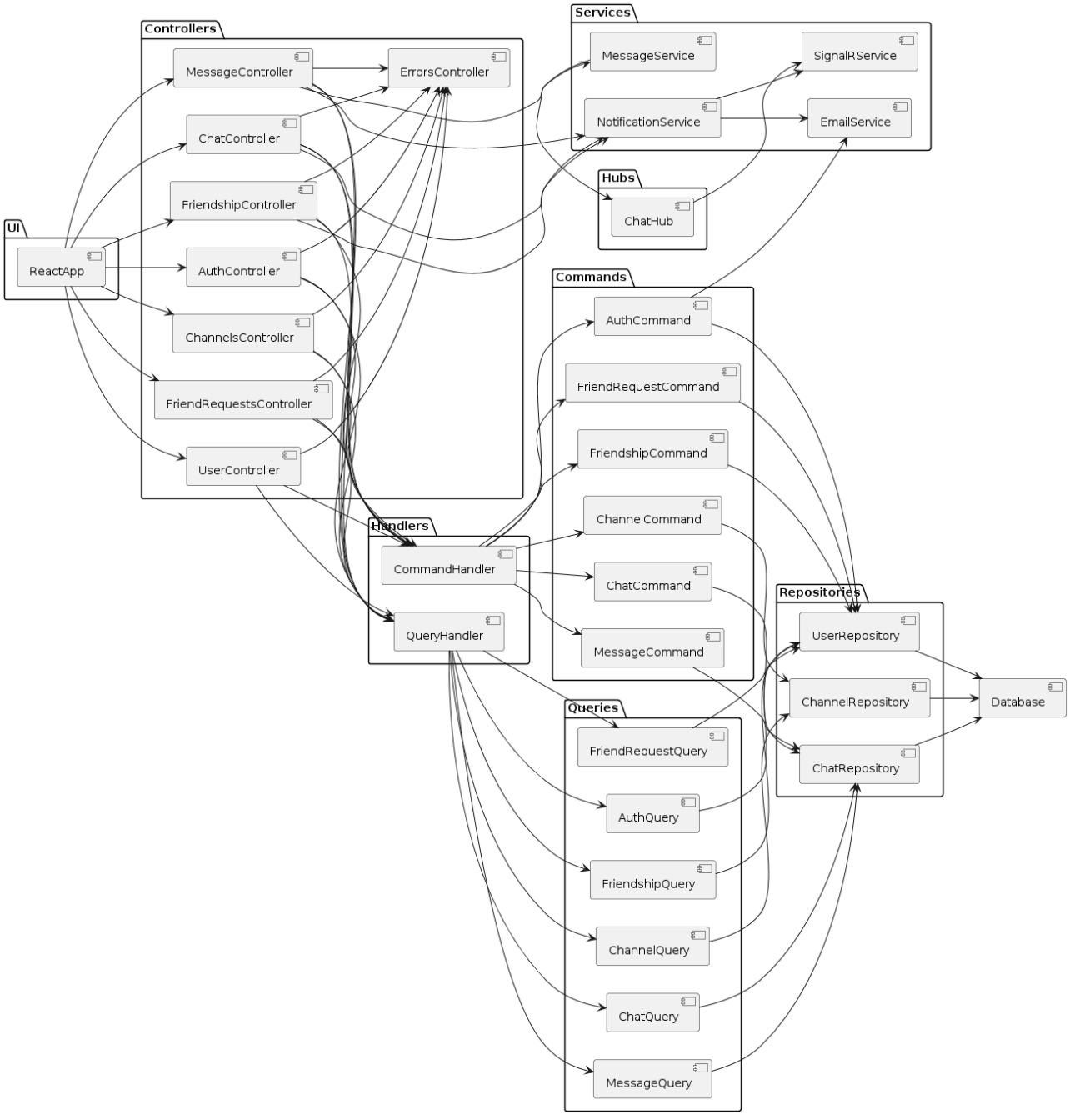

Рисунок 2.16 – Діаграма компонентів системи

Діаграма компонентів забезпечує структуроване уявлення про архітектуру системи, підкреслюючи ключові компоненти та їх взаємозв’язки. Вона допомагає розробникам та іншим зацікавленим сторонам зрозуміти, як різні

частини системи взаємодіють між собою, що є критично важливим для успішної розробки, підтримки та масштабування програмного забезпечення. Правильне визначення та реалізація цих компонентів забезпечить надійну та ефективну роботу системи.

Висновки до розділу 2

У цьому розділі було розглянуто технології та принципи, які використовуються при проектуванні застосунку та обґрунтовано їх використання.Я зробив декілька діаграм, таких як діаграма послідовності, діаграма компонентів та діаграма сценаріїв використання. Зробив діаграму таблиць для бази даних. Було зроблено аналіз існуючих систем та визначено вимоги до базового функціоналу, який необхідно реалізувати для месенджера. Були створені макети сторінок у Figma для мобільного застосунку та вебзастосунку.

Ця робота може розглядатися як перший етап довгострокового проекту. У подальшому, наприклад у магістерській роботі, можна розширити функціонал:

1) додати end-to-end шифрування для підвищення безпеки;
2) впровадити обмін медіа та іншими файлами;
3) впровадити голосові та відео дзвінки;
4) впровадити можливість обміну голосовими повідомленнями;
5) впровадити додавання пристрою до акаунту через QR код;
6) розробити кросплатформний клієнтський додаток для ПК.

РОЗДІЛ 3
СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

3.1 Практична реалізація об’єкта проєктування. Створення API для месенджера. Впровадження CQRS, DDD та Чистої архітектури

В якості IDE я обрав Visual Studio – це інтегроване середовище розробки (IDE) від Microsoft, яке підтримує широкий спектр мов програмування, включаючи C# та TypeScript. IDE пропонує безліч функцій, які полегшують процес розробки, таких як автозаповнення коду, відладка та тестування. Visual Studio, як інтегроване середовище розробки, значно прискорює мою роботу завдяки глибокій інтеграції з обраними технологіями. Від компіляції та відладки до рефакторингу та розгортання – весь цикл розробки оптимізовано.

Мій вибір C# з ASP.NET Core та TypeScript з React відображає прагнення до створення надійного, масштабованого та зручного у використанні месенджера. Використання принципів DDD, Чистої архітектури та CQRS забезпечує підтримуваність коду в довгостроковій перспективі. Потужні бібліотеки та інструменти, як-от MediatR, Entity Framework Core, допомагають ефективно вирішувати складні завдання.

У розробці програмного забезпечення архітектура проекту відіграє фундаментальну роль, особливо у великих і складних системах, таких як мій месенджер. Вона визначає не лише структуру коду, але й впливає на такі критичні аспекти, як масштабованість, підтримуваність та гнучкість системи.

В контексті мого проекту – розробки сучасного, високонавантаженого месенджера, мені потрібно обрати правильну архітектуру. Месенджер – це не просто додаток для обміну текстовими повідомленнями. Це складна система, яка повинна обробляти тисячі або навіть мільйони транзакцій у реальному часі, зберігати величезні обсяги даних, забезпечувати безпеку особистої інформації та залишатися швидкою навіть під час пікових навантажень.

Тому я вирішив застосувати принципи «чистої архітектури» (Clean Architecture) та предметно-орієнтованого проектування (Domain-Driven Design, DDD). Ці підходи допомагають створити систему, в якій:

– бізнес-логіка (домен) ізольована і не залежить від зовнішніх факторів;

– компоненти слабко пов’язані, що дозволяє легко замінювати або оновлювати частини системи;

– код організований навколо реальних бізнес-процесів, а не технічних деталей.

На рисунку 3.1 зображено структуру проєкту API .

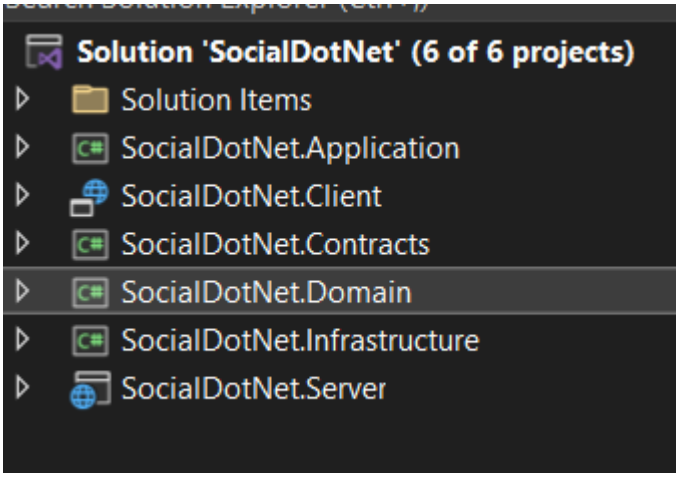

Рисунок 3.1 – Огляд архітектури проекту

На зображенні, яке відкрите у моєму Visual Studio, ми бачимо практичне втілення цих принципів. Проект розділений на кілька модулів, кожен з яких має чітко визначену відповідальність:

– SocialDotnet.Domain – це серце застосунку, «ядро» в термінах «чистої архітектури». Тут знаходяться бізнес-сутності (User, Message, Chat) та бізнес-правила. Не має залежностей від інших модулів, що забезпечує його стабільність;

– SocialDotnet.Application – шар, що містить бізнес-логіку, яка оперує об’єктами домену. Тут знаходяться сервіси, команди та запити (CQRS). Залежить лише від Domain, не знаючи про Infrastructure, Application та Presentation;

– SocialDotnet.Infrastructure – відповідає за «технічні деталі» бази даних, зовнішні сервіси. Реалізує інтерфейси з Application (Dependency Inversion). Наприклад, тут буде код для роботи з MSSQL через Entity Framework;

– SocialDotnet.Contracts – модуль, який містить Data Transfer Objects (DTOs). Я використовую record типи, що ідеально підходять для DTOs.

Після створення структури проекту, яка відображає принципи Чистої архітектури, наступним логічним кроком є встановлення правильних залежностей між проектами. Це не просто технічне завдання – це відображення архітектурних меж та напрямків залежностей, які ми визначили.

Після створення структури проекту, у Visual Studio я переходжу до кожного проекту і налаштовую посилання:

– SocialDotnet.Domain – не має посилань на інші проекти. Це фундаментальний принцип: домен не повинен залежати від зовнішніх деталей;

– SocialDotnet.Application – посилання на SocialDotnet.Domain. Це відповідає правилу Dependency Inversion: бізнес-логіка залежить тільки від абстракцій;

– SocialDotnet.Infrastructure – посилання на SocialDotnet.Domain та SocialDotnet.Application. Реалізує інтерфейси з Application, використовуючи типи з Domain;

– SocialDotnet.Contracts – може мати посилання на SocialDotnet.Domain для використання його типів у DTOs. Часто залишається незалежним, щоб клієнт не мав транзитивних залежностей;

– SocialDotnet.Server – посилання на всі проекти: Domain, Application, Infrastructure, Contracts. Це «клей», що з’єднує всі компоненти;

– SocialDotnet.Client – у моєму випадку не має посилань.

Цей процес у Visual Studio дуже наочний. При спробі додати неправильне посилання (наприклад, з Domain на Infrastructure) IDE попереджає про

порушення архітектури. Це додатковий захист, що допомагає дотримуватися чистоти дизайну.

Тепер, коли архітектура встановлена, я переходжу до розширення функціональності моїх модулів за допомогою ретельно підібраних бібліотек. Кожна з них вирішує конкретну проблему в розробці месенджера. У Visual Studio я використовую NuGet Package Manager для встановлення:

– Application:

1) BCrypt.Net-Next – безпечне хешування паролів;
2) Fluent Validation: – валідація вхідних даних;
3) MediatR – реалізація патерну CQRS;

– Domain:

1) ErrorOr – елегантна обробка помилок;

– Infrastructure:

1) Microsoft.AspNetCore.Authentication.JwtBearer – автентифікація JWT;
2) Microsoft.EntityFrameworkCore – робота з базою даних;
3) Microsoft.EntityFrameworkCore.Design – інструменти EF Core в консолі диспетчера пакетів;
4) Microsoft.EntityFrameworkCore.SqlServer – робота з базою даних SQL Server;
5) Microsoft.Extensions.Configuration – конфігурація;
6) Microsoft.Extensions.DependencyInjection.Abstractions – DI;
7) Microsoft.Extensions.Options.ConfigurationExtensions – розширення конфігурації для DI;
8) System.IdentityModel.Tokens.Jwt – створення та перевірка JWT;

– Server:

1) BCrypt.Net-Next – доступ до функцій хешування;
2) Mapster – об’єктне маппінг;
3) MapsterDependencyInjection – DI для Mapster;
4) MediatR – відправка команд і запитів;

5) Microsoft.AspNetCore.SpaProxy – інтеграція з React-фронтендом;

6) Microsoft.EntityFrameworkCore.Design – інструменти EF Core в консолі диспетчера пакетів;

7) Microsoft.Extensions.DependencyInjection – налаштування DI;

8) Serilog.AspNetCore – структуроване логування;

9) Swashbuckle.AspNetCore – документація Swagger.

Кожна бібліотека ретельно вибрана для вирішення конкретних завдань у розробці месенджера. Наприклад, ErrorOr і Fluent Validation забезпечують надійну валідацію і обробку помилок, що критично для запобігання спаму або витоку даних. MediatR з CQRS допомагає розділити операції запису (відправка повідомлень) та читання (отримання історії чату), що оптимізує навантаження на базу даних.

Entity Framework Core з MSSQL забезпечує надійне зберігання повідомлень. JWT і BCrypt захищають автентифікацію, а Serilog допомагає виявляти проблеми і аналізувати поведінку користувачів.

Ці бібліотеки не лише розширюють функціональність, але й підтримують архітектурні принципи. Вони сприяють слабкому зв’язуванню (MediatR) і чіткому розділенню відповідальності.

Далі переходимо до створення базового контролера для всіх інших контролерів. У моїй системі використовується базовий API контролер для стандартизації обробки помилок у всіх контролерах. Контролер містить методи для обробки різних типів помилок, таких як валідаційні помилки, конфлікти та помилки, пов’язані з відсутністю ресурсу. Метод «Problem(List<Error> errors)» визначає, який тип помилки обробляти, і відповідно повертає належний код статусу HTTP. Це забезпечує консистентну та зрозумілу обробку помилок, покращуючи підтримку та масштабованість додатку. Використання CQRS, DDD та Чистої архітектури дозволяє підтримувати чистоту коду та розділення відповідальностей між різними рівнями системи. Лістинг 3.1 з демонстрацією контролера наведений нижче.

Лістинг 3.1 – Демонстрація базового API контролера

```
[ApiController]
[Authorize]
public class ApiController : ControllerBase
{
    protected IActionResult Problem(List<Error> errors)
    {
        if (errors.Count is 0)
        {
            return Problem();
        }

        if (errors.All(error => error.Type ==
ErrorType.Validation))
        {
            return ValidationProblem(errors);
        }
        HttpContext.Items[HttpContextItemKeys.Errors] = errors;

        return Problem(errors[0]);
    }
```

Кінець лістингу 3.1

Після цього створюємо новий контролер – ErrorsController. ErrorsController – це спеціальний контролер для обробки глобальних помилок, які виникають у застосунку. Він визначений з атрибутом Route, що вказує шлях, на якому він буде доступний. Цей контролер використовується для того, щоб централізовано обробляти винятки та повертати стандартну відповідь про помилку.

Коли у застосунку виникає некерований виняток, він перенаправляється до цього контролера, де метод Error отримує виняток з HttpContext і викликає метод Problem() для генерації стандартної відповіді про помилку. Це дозволяє централізовано обробляти всі винятки, забезпечуючи узгоджену обробку та логування помилок, а також покращує підтримку і масштабованість додатку. Лістинг 3.2 наведений нижче.

Лістинг 3.2 – Демонстрація контролера для оброблення помилок

```
[Route("/error")]
    public class ErrorsController : ControllerBase
    {
        [HttpPost]
        public IActionResult Error()
        {
            Exception? exception =
HttpContext.Features.Get<IExceptionHandlerFeature>()?.Error;
              return Problem();
        }
    }
```

Кінець лістингу 3.2

Далі потрібно налаштувати точку входу в програму – файл Program.cs . Він містить код, що визначає точку входу в застосунок ASP.NET Core. У цьому файлі налаштовуються всі сервіси та проміжні програмні компоненти (middleware), які будуть використовуватись у застосунку. Потрібно додати налаштування для контролера, який обробляє помилки та інші middleware, які потрібні для авторизації та автентифікації. Далі, в цьому ж файлі, нам потрібно налаштувати сервіси. Тепер треба створити новий статичний клас DependencyInjection в проекті Infrastructure і додати налаштування для автентифікації. Лістинг 3.3 приводиться нижче.

Лістинг 3.3 – Налаштування автентифікації на JWT

```
public static IServiceCollection AddAuth(
            this IServiceCollection services,
            ConfigurationManager configuration)
        {
            var JwtSettings = new JwtSettings();
            configuration.Bind(JwtSettings.SectionName,
JwtSettings);

            services.AddSingleton(Options.Create(JwtSettings));
```

```
services.Configure<JwtSettings>(configuration.GetSection(JwtSettings.Sect
ionName));
                services.AddSingleton<IJwtTokenGenerator,
JwtTokenGenerator>();

                services.AddAuthentication(defaultScheme:
JwtBearerDefaults.AuthenticationScheme)
                    .AddJwtBearer(options =>
options.TokenValidationParameters = new TokenValidationParameters()
                    {
                        ValidateIssuer = true,
                        ValidateAudience = true,
                        ValidateLifetime = true,
                        ValidateIssuerSigningKey = true,
                        ValidIssuer = JwtSettings.Issuer,
                        ValidAudience = JwtSettings.Audience,
                        IssuerSigningKey = new SymmetricSecurityKey(

Encoding.UTF8.GetBytes(JwtSettings.Secret)),
                    });
                return services;
            }
```

Кінець лістингу 3.3

Далі потрібно створити в проекті Application інтерфейс з методом для генерації JWT та Refresh токенів Нижче наведений лістинг 3.4, де демонструється інтерфейс з методами для генерації JWT та Refresh токенів.

Лістинг 3.4 – Демонстрація інтерфейсу для генерації JWT та Refresh токенів

```
    public interface IJwtTokenGenerator
        {
            string GenerateToken(User user);
            RefreshToken GenerateRefreshToken();
            RefreshToken RotateRefreshToken(RefreshToken refreshToken);
            void RemoveOldRefreshTokens(User user);
            void              RevokeDescendantRefreshTokens(RefreshToken
refreshToken, User user, string reason);
```

```
            void RevokeRefreshToken(RefreshToken token, string reason =
null, string replacedByToken = null);
        }
```

Кінець лістингу 3.4

Тепер необхідно створити базові класи для Value Objects, Entities та Aggregates. Нижче приведений лістинг коду 3.5 для базового класу агрегату, який буде використовуватись для створення агрегатів.

Лістинг 3.5 – Налаштування автентифікації на JWT

```
    public abstract class AggregateRoot<TId> : Entity<TId>
            where TId : notnull
        {
            protected AggregateRoot(TId id) : base(id)
            {
            }

            protected AggregateRoot()
            {
            }
        }
```

Кінець лістингу 3.5

Цей клас є частиною DDD підходу, де агрегатний корінь використовується для забезпечення цілісності та консистентності групи об’єктів. Він контролює доступ до інших об’єктів всередині агрегата і гарантує, що інваріанти бізнес-логіки зберігаються.

Після цього на основі цих базових класів можна почати створювати агрегати, залежні від них сутності та об’єкти значень, які я визначив в попередньому розділі. Код агрегата користувача можна знайти в додатку А. Після цього потрібно налаштувати таблицю з користувачами та інші пов’язані таблиці. Фрагмент коду наведений нижче у лістингу 3.6.

Лістинг 3.6 – Налаштування таблиці з користувачами

```
private void ConfigureUsersTable(EntityTypeBuilder<User> builder)
        {
            builder.ToTable("Users");
            builder.HasKey(u => u.Id);
            builder.Property(u => u.Id)
                .HasConversion(
                    id => id.Value,
                    value => UserId.Create(value));

            builder.Property(u => u.FirstName)
                .IsRequired()
                .HasMaxLength(50);

            builder.Property(u => u.LastName)
                .IsRequired()
                .HasMaxLength(50);

            builder.Property(u => u.Email)
                .IsRequired()
                .HasMaxLength(100);

            builder.Property(u => u.PasswordHash)
                .IsRequired()
                .HasMaxLength(100);
        }
```

Кінець лістингу 3.6

Цей код визначає метод ConfigureUsersTable, який налаштовує конфігурацію для таблиці Users у базі даних за допомогою Fluent API в Entity Framework Core. Метод налаштовує властивості та ключі для сутності User, що буде відповідати записам у таблиці Users.

Тепер потрібно створити новий контролер, який буде відповідати за автентифікацію та містити ендпоінти для логіну, реєстрації, створення і відкликання токенів. Нижче приведений лістинг 3.7 з методом для реєстрації користувачів.

Лістинг 3.7 – Демонстрація метода для реєстрації

```
    [AllowAnonymous]
            [HttpPost("register")]
            public  async  Task<IActionResult>  Register(RegisterRequest
request)
            {
                var command = _mapper.Map<RegisterCommand>(request);
                ErrorOr<AuthenticationResult>    authResult    =    await
_mediator.Send(command);
                return authResult.Match(
                    authResult => {
                        var                  response                  =
_mapper.Map<AuthenticationResponse>(authResult);
                        SetTokenCookie(response.RefreshToken);
                        return Ok(response);
                    },
                    errors => Problem(errors));
            }
```

Кінець лістингу 3.7

Цей метод обробляє процес реєстрації нових користувачів. Він приймає дані від користувача через HTTP POST запит, перетворює їх у відповідну команду, яка виконує логіку реєстрації, і повертає відповідь, що містить результат реєстрації або помилки. Використовуючи CQRS, Mapster і ErrorOr, метод забезпечує чітке розділення запитів і команд, просту обробку помилок і ефективну роботу з даними. Повний код контролера знаходиться у додатку А.

Тепер необхідно створити команду, обробник команди і валідатор для команди. Нижче наведений лістинг3.8 з командою для реєстрації.

Лістинг 3.8 – Демонстрація команди для реєстрації

```
    public record RegisterCommand(
            string FirstName,
            string LastName,
            string Username,
            string Email,
```

```
        string Password) : IRequest<ErrorOr<AuthenticationResult>>;
```

Кінець лістингу 3.8

Цей код визначає клас-запис, який представляє команду для реєстрації нового користувача в системі. Він використовує сучасні можливості C# для створення іммутабельних об’єктів, а також інтеграцію з CQRS (Command Query Responsibility Segregation) паттерном і бібліотекою MediatR для обробки команд і запитів.

Далі створюємо обробник команди, де буде знаходитися вся основна логіка для створення користувача. Нижче переведено лістинг коду 3.9 для обробника команди реєстрації.

Лістинг 3.9 – Демонстрація обробника команди реєстрації

```
    public async Task<ErrorOr<AuthenticationResult>>
Handle(RegisterCommand command, CancellationToken cancellationToken)
        {
            await Task.CompletedTask;
            if (await
_userRepository.GetUserByEmailAsync(command.Email) is not null)
            {
                return Errors.User.DuplicateEmail;
            }
        var hashedPassword =
BC.EnhancedHashPassword(command.Password, 13);
            var refreshToken =
_jwtTokenGenerator.GenerateRefreshToken();

            var user = User.Create(command.FirstName,
command.LastName, command.Username, command.Email, hashedPassword,
[refreshToken]);

            var token = _jwtTokenGenerator.GenerateToken(user);

            await _userRepository.AddAsync(user);

            return new AuthenticationResult(
```

```
                user,
                token,
                refreshToken.Token);
        }
```

Кінець лістингу 3.9

Метод виконує асинхронну обробку команди реєстрації користувача, перевіряючи наявність користувача з вказаною електронною поштою. Якщо користувач з такою поштою вже існує, метод повертає помилку про дублювання адреси електронної пошти. Пароль вводиться через захешування з використанням BCrypt з фактором солі 13 для підвищеної безпеки. Генерується оновлюваний токен для сесії користувача, а також токен автентифікації за допомогою JWT. Створюється об’єкт користувача з введеними даними реєстрації та згенерованим токеном оновлення. Новий користувач додається до репозиторію користувачів. Метод повертає результат аутентифікації, який містить інформацію про користувача, токен автентифікації та токен оновлення.

Далі необхідно реалізувати репозиторій для користувачів, в яких визначимо операції над таблицею з користувачами. Лістинг коду 3.10 приведений нижче.

Лістинг 3.10 – Демонстрація репозиторію для користувачів

```
    public async Task AddAsync(User user)
        {
            _context.Users.Add(user);
            await _context.SaveChangesAsync();
        }

        public async Task<User?> GetUserByEmailAsync(string email)
        {
            return    await    _context.Users.SingleOrDefaultAsync(u    =>
u.Email == email);
        }
```

```
        public async Task<User?> GetUserByIdAsync(UserId id)
        {
            return await _context.Users.FindAsync(id);
        }

        public async Task<User?> GetUserByRefreshTokenAsync(string
refreshToken)
        {
            return await _context.Users.FirstOrDefaultAsync(u =>
u.RefreshTokens.Any(t => t.Token == refreshToken));
        }

        public async Task UpdateAsync(User user)
        {
            _context.Update(user);
            await _context.SaveChangesAsync();
        }
```

Кінець лістингу 3.10

Цей код представляє набір асинхронних методів для взаємодії з базою даних, спрямованих на роботу з користувачами. Метод AddAsync додає нового користувача до контексту бази даних та зберігає зміни асинхронно. GetUserByEmailAsync повертає користувача за його електронною поштою. GetUserByIdAsync здійснює пошук користувача за його ідентифікатором. GetUserByRefreshTokenAsync знаходить користувача за його токеном оновлення. Нарешті, метод UpdateAsync оновлює існуючого користувача в базі даних. Всі ці методи працюють асинхронно для ефективної взаємодії з базою даних та покращення продуктивності застосунку.

3.2 Тестування та налагодження інформаційно-комп’ютерної системи

Тестування є важливим етапом у розробці будь-якого програмного забезпечення, оскільки воно дозволяє виявити і виправити помилки, забезпечити стабільність та відповідність вимогам. У процесі розробки месенджера для

тестування API я використовував Swagger [21] – потужний інструмент для тестування API я використовував Swagger [21] – потужний інструмент для документування та тестування веб-сервісів. Swagger дозволяє створювати інтерактивну документацію API, де можна перевіряти роботу кожного ендпоінта, відправляти запити та отримувати відповіді без необхідності написання додаткового коду. Це значно спрощує процес тестування та налагодження API. На рисунку 3.2 демонструється процес тестування API за допомогою Swagger.

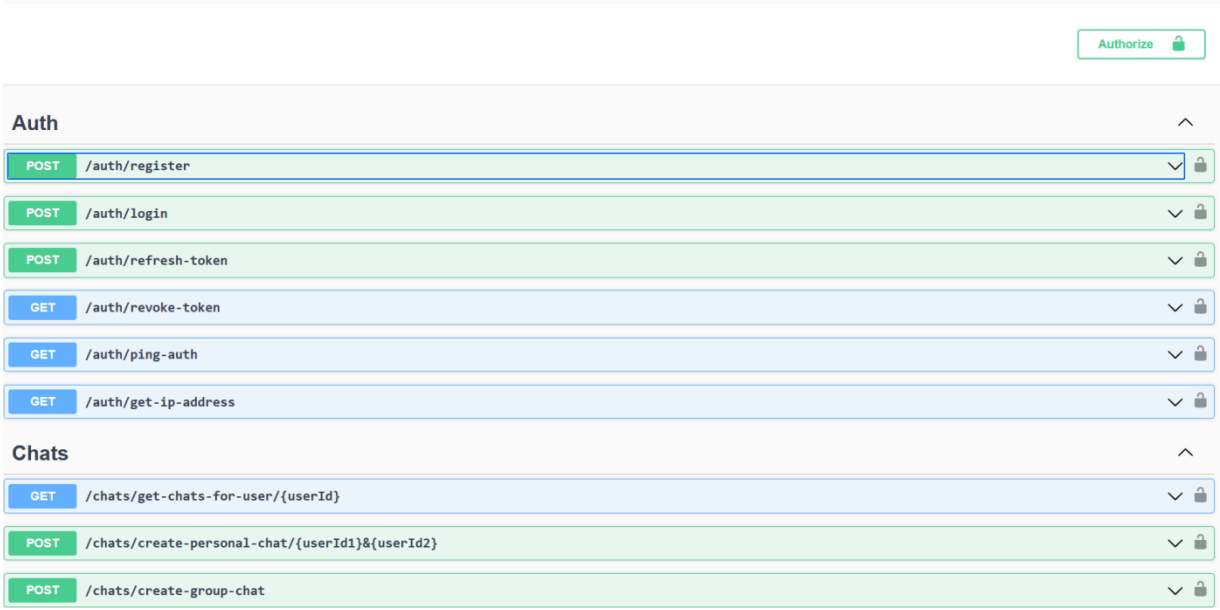

Рисунок 3.2 – Демонстрація Swagger

Для ефективного налагодження та моніторингу роботи системи я інтегрував Serilog – бібліотеку для структурованого логування. Serilog дозволяє зберігати детальну інформацію про помилки, попередження та інформаційні повідомлення. Це допомагає швидко виявляти і аналізувати проблеми, що виникають під час виконання програми. Serilog також підтримує різні рівні логування (debug, info, warning, error, fatal), що дозволяє контролювати рівень деталізації логів в залежності від потреб та середовища (розробка, тестування, продакшн). На рисунку 3.3 можна побачити консоль з інформацією від Serilog.

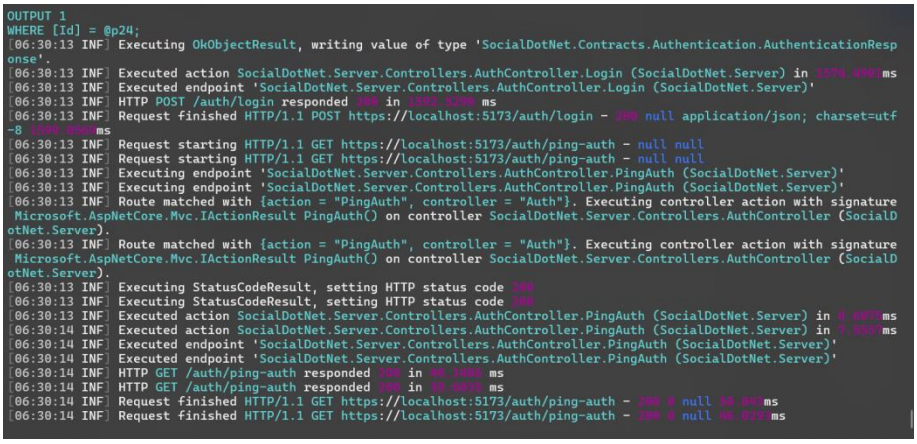

Рисунок 3.3 – Демонстрація роботи Serilog

Для налагодження коду використовувалися стандартні інструменти, що надаються середовищем розробки Visual Studio. Процес налагодження програми зображено на рисунку 3.4 нижче.

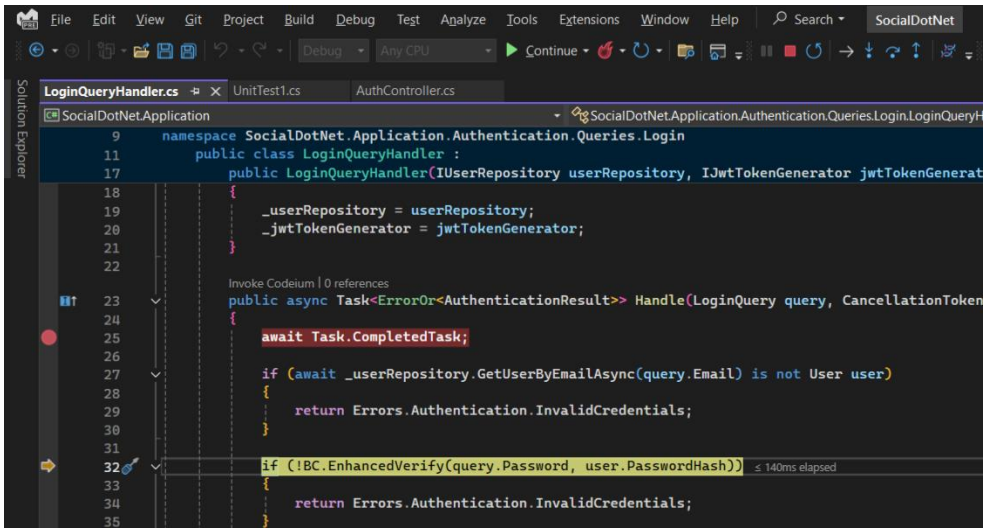

Рисунок 3.4 – Демонстрація процесу налагодження програми

Серед них такі інструменти, як точкові зупинки (breakpoints), крокування (step over, step into, step out), огляд змінних у різні моменти часу виконання, а також можливість аналізу стеку викликів (call stack). Використання цих інструментів дозволило швидко ідентифікувати і виправляти помилки на ранніх етапах розробки.

Для забезпечення стабільної роботи інформаційної системи були визначені мінімальні та рекомендовані системні вимоги.

Мінімальні системні вимоги:

– процесор – двоядерний з тактовою частотою 2.0 ГГц;
– оперативна пам’ять – 4 ГБ;
– місце на диску – 50 ГБ вільного місця;
– операційна система – Windows 10 або новіша, macOS 10.15 або новіша, Linux з ядром 4.15 або новішим;
– .NET Core Runtime 3.1 або новіший.

Рекомендовані системні вимоги:

– процесор – чотириядерний з тактовою частотою 3.0 ГГц;
– оперативна пам’ять – 8 ГБ;
– місце на диску – 100 ГБ вільного місця;
– операційна система – Windows 10 або новіша, macOS 11 або новіша, Linux з ядром 5.0 або новішим;
– .NET Core Runtime 5.0 або новіший.

Тестування та налагодження є невід’ємною частиною процесу розробки програмного забезпечення. Тестування API за допомогою Swagger, дозволило виявити і виправити більшість помилок, забезпечуючи високу якість та стабільність роботи месенджера. Інтеграція Serilog для логування сприяла швидкому виявленню та усуненню проблем, що виникали під час виконання програми. Встановлення мінімальних та рекомендованих системних вимог гарантує стабільну роботу інформаційної системи на різних платформах.

3.3 Розробка інтерфейсу користувача з використанням React, React Native, TypeScript та Tailwind

У цьому розділі ми зосередимося на практичній реалізації інтерфейсу користувача для нашого месенджера, використовуючи React та React Native на

TypeScript та Tailwind CSS. Як було зазначено раніше, цей вибір технологій забезпечує нам гнучкість, безпеку типів і швидкість розробки.

Я почав розробку з веб версії застосунку, використовуючи React. Ключовим аспектом успішної розробки React-застосунку є добре організована структура проекту. У моєму месенджері я дотримуюсь принципів модульності та розділення відповідальності, що відображається в наступній структурі:

– директорія services містить axiosWithAuth.ts – цей модуль інкапсулює логіку автентифікації для HTTP-запитів. Він розширює axios, автоматично включаючи JWT-токен у заголовки. Це забезпечує безпечний зв’язок з ASP.NET Core бекендом, гарантуючи, що кожен запит від автентифікованого користувача правильно ідентифікується;

– assets директорія містить зображення та іконки;

– components директорія – це серце UI. Ця папка містить модульні, багаторазові компоненти React:

1) компоненти автентифікації (AuthorizeView.tsx, LoginForm.tsx, RegisterForm.tsx) забезпечують безпечний і зручний процес входу;

2) чат-компоненти (ChatList.tsx, ChatSearch.tsx, ChatTabs.tsx, Sidebar.tsx, Sidebar.tsx) формують основний інтерфейс, дозволяючи користувачам ефективно та зручно керувати своїми розмовами;

3) компоненти повідомлень (MessageItem.tsx, MessageTextarea.tsx) відповідають за відображення та створення повідомлень, підтримуючи різні типи контенту;

4) службові компоненти (InputField.tsx) покращують загальний UX, забезпечуючи послідовність і інтуїтивність у всьому застосунку;

– директорія pages – ці компоненти високого рівня визначають основні маршрути та макети:

1) Home.tsx – центральний вузол, де користувачі проводять більшість часу, об’єднуючи ключові компоненти чату;

2) Login.tsx і Register.tsx – сторінки входу та реєстрації;

– директорія hooks – містить хуки. Хуки дозволяють використовувати стан та інші можливості React без написання класів. На рисунку 3.5 зображено структуру проєкту на React.

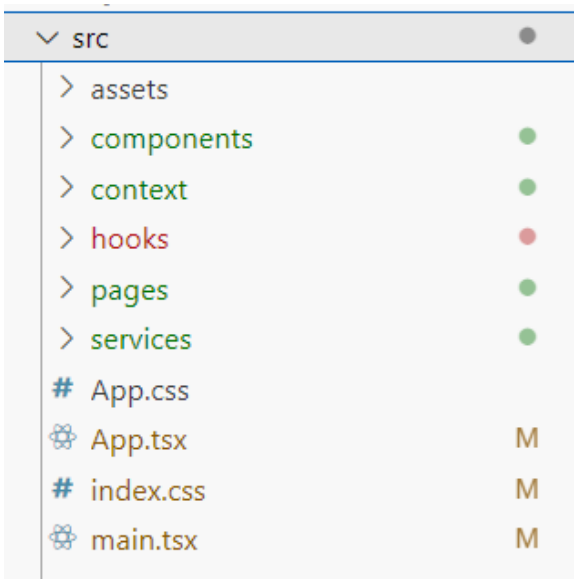

Рисунок 3.5 – Демонстрація структури React проекту

Після створення структури для проекту можна перейти до визначення React компонентів. У рамках розробки інтерфейсу користувача для месенджера, особливу увагу було приділено питанням безпеки та авторизації. У сучасних вебзастосунках, особливо в таких, де обробляються конфіденційні дані, як у моєму месенджері, критично важливо гарантувати, що доступ до функціональності та даних надається тільки автентифікованим і авторизованим користувачам.

Для вирішення цього завдання я розробив компонент AuthorizeView. Повний код компонента можна побачити у додатку Б. Цей компонент гарантує, що доступ до приватних маршрутів мають лише автентифіковані користувачі. У випадку неавторизованого доступу, користувача буде автоматично перенаправлено на сторінку входу. У лістингу 3.11 представлено фрагмент коду.

Лістинг 3.11 – Демонстрація використання TypeScript у компоненті

```
const UserContext = createContext({});

interface User {
    email: string;
}

function AuthorizeView(props: { children: React.ReactNode }) {
    const [authorized, setAuthorized] = useState<boolean>(false);
    const [loading, setLoading] = useState<boolean>(true);
              const [user, setUser] = useState<User>({ email: "" });
    const [redirect, setRedirect] = useState<boolean>(false);
```

Кінець лістингу 3.11

Ми активно використовуємо TypeScript для забезпечення безпеки типів. Визначення інтерфейсу User і явне типування станів (useState<boolean>, useState<User>) допомагає уникнути багатьох поширених помилок. Наприклад, компілятор TypeScript не дозволить нам випадково присвоїти числове значення змінній authorized або забути включити поле email при встановленні стану user.

Ми використовуємо React Hook useEffect для виконання побічних ефектів, у цьому випадку – для перевірки статусу автентифікації користувача через запит до нашого ASP.NET Core бекенду. Особливо варто відзначити реалізацію механізму повторних спроб. У реальних умовах мережеві запити можуть іноді завершуватися невдачею через тимчасові проблеми. Наша функція fetchWithRetry елегантно обробляє такі ситуації, повторюючи запит після короткої затримки. Цей підхід значно підвищує надійність нашого застосунку, особливо в умовах нестабільного з'єднання, що часто трапляється в мобільних мережах. У лістингу 3.12 приведено фрагмент коду з компоненту авторизації.

Лістинг 3.12 – Демонстрація коду для автентифікації з повторними спробами

```
const fetchWithRetry = async (url: string, options: any) => {
            try {
```

```
                const response = await fetchWithAuth(url, options);
                if (response.status === 200) {
                    const userData = response.data;
                    setUser({ email: userData.email });
                    setAuthorized(true);
                    return response;
                } else if (response.status === 401) {
                    setRedirect(true);
                    return response;
                } else {
                    throw new Error("" + response.status);
                }
            } catch (error) {
                retryCount++;
                if (retryCount > maxRetries) {
                    setRedirect(true);
                    throw error;
                } else {
                    await wait(delay);
                    return fetchWithRetry(url, options);
                }
            }
        };

        fetchWithRetry("auth/ping-auth", { method: "GET" })
            .catch(error => {
                console.log(error.message);
            })
            .finally(() => {
                setLoading(false);
            });
    }, []);
```

Кінець лістингу 3.12

Ми використовуємо React Context для безпечної передачі інформації про користувача вниз по дереву компонентів без необхідності явно передавати пропси через кожен рівень. Це не лише робить наш код чистішим, але й запобігає випадковому витоку даних до непризначених компонентів. Компонент

AuthorizeView також демонструє кращі практики умовного рендерингу в React. Під час перевірки автентифікації ми показуємо індикатор завантаження. У разі неавторизованого доступу або вичерпання спроб автентифікації, ми програмно перенаправляємо користувача на сторінку входу за допомогою компонента Navigate з react-router-dom.

Перед початком створення основних компонентів, які і будуть визначати собою вигляд застосунку, я підібрав кольорову палітру та додав її до конфігурації Tailwind. Лістинг 3.13 файлу конфігурації продемонстровано нижче.

Лістинг 3.13 – Лістинг файлу конфігурації Tailwind

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {
      colors: {
        'navyBlue': '#1B263B',
        'darkBlue': '#0D1B2A',
        'white': '#FFFFFF',
        'royalPurple': '#6D50AE',
        'lightBlue': '#95ADCF',
      }
    },
  },
  plugins: [],
}
```

Кінець лістингу 3.13

Далі було створено компонент для реєстрації та входу. У лістингу 3.14 представлено функцію для реєстрації користувачів.

Лістинг 3.14 – Демонстрація функції для реєстрації

```
    function RegisterForm() {
        const [email, setEmail] = useState('');
        const [firstName, setFirstName] = useState('');
        const [lastName, setLastName] = useState('');
        const [username, setUsername] = useState('');
        const [password, setPassword] = useState('');

        const handleSubmit = async (e: React.FormEvent<HTMLFormElement>)
=> {
            e.preventDefault();
            try {
                const response = await axios.post('auth/register', {
email, password, firstName, lastName, username }
    );
                console.log('Register successful:', response.data);
            } catch (error) {
                console.error('Register failed:', error);
            }
        };
```

Кінець лістингу 3.14

Після цього я розпочав роботу над елементами месенджера. Було створено компонент для відображення діалогу з користувачем. Нижче у лістингу 3.15 демонструється код.

Лістинг 3.15 – Демонструється частина компоненту діалога

```
    {messages.map((message: Message, index: int) => (
                        <MessageItem
                            content={message.content}
                            isSent={message.isSent}
                         />
                    ))}
                </div>
            </div>
            <div className="input-panel flex p-4">
                <input
```

```
                        type="text"
                        className="flex-1 mr-2 py-2 px-3 rounded-lg
dark:bg-gray-800 bg-gray-100 focus:outline-none caret-white text-white"
                        placeholder="Type your message..."
                        value={newMessage}
                        onChange={handleMessageChange}
                        onKeyDown={handleKeyDown}
                    />
                    <button
                        className="py-2 px-4 bg-blue-900 text-white
rounded-lg hover:bg-blue-950"
                        onClick={handleSendMessage}
```

Кінець лістингу 3.15

До діалогу було створено компонент для відображення повідомлень між користувачами. Код компонента знаходиться нижче у лістингу 3.16.

Лістинг 3.16 – Демонструється компонент для повідомлення

```
    interface Message {
        id: string;
        sender: User;
        text: string;
        timestamp: Date;
      }
    function MessageItem({ content, isSent }: Message) {
        const messageContainerClass = isSent ? 'sent' : 'received';
        const messageColor = isSent ? 'dark:bg-blue-700 text-white' :
'dark:bg-gray-800 text-white';
        return (
            <div className={`flex ${messageContainerClass}`}>
                <div className={`message ${messageColor}`}>
                    {content}
                </div>
            </div>
        );
    }
    export default MessageItem;
```

Кінець лістингу 3.16

Далі я перейшов до створення мобільного застосунку, використовуючи React Native з Expo та по аналогії з описаним вище вебзастосунком розробив інтерфейс користувача для месенджера. Я використав шаблон від Expo з налаштованою навігацією вкладками, що значно спростило розробку. Далі я почав впровадження автентифікації у свій застосунок. Створив хук для збереження інформації про користувача, який пройшов процес автентифікації, який зберігає jwt та ідентифікатор користувача в локальному сховищі браузера або в сховищі Expo в залежності від платформи, на якій запущено проєкт.

Далі я створив компонент контексту автентифікації, де створив функції для реєстрації, входу та виходу з системи. Нижче приведено лістинг 3.17, де можна побачити функцію для виконання входу в систему.

Лістинг 3.17 – Функція для входу в систему

```
const onLogin = async (email: string, password: string) => {
    try {
        const response = await httpService.post('auth/login', { email, password });
      console.log(response.data);
      const token = response.data.token;
      const userId = response.data.id;
      setToken(token);
      setUserId(userId);
      axios.defaults.headers.common['Authorization'] = `Bearer ${token}`;
      return { success: true };
    } catch (error: any) {
      return { success: false, error: error.response.data };
    }
  };
```

Кінець лістингу 3.17

Також виніс налаштування бібліотеки Axios для здійснення HTTP запитів до вказаного API URL в окремий файл – httpService. Нижче приведено лістинг 3.18, де демонструється налаштування бібліотеки Axios.

Лістинг 3.18 – Налаштування бібліотеки Axios

```
import axios from 'axios';

const API_URL = 'https://localhost:7008/';
const httpService = axios.create({
  baseURL: API_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});

export default httpService;
```

Кінець лістингу 3.18

Далі у файлі _layout.tsx я додав перевірку на те, чи порожній контекст автентифікації, щоб перевірити, чи автентифікований користувач, і якщо ні – відбувається переспрямування користувача на сторінку входу, яку я розробив наступною. Нижче приведено лістинг 3.19, де продемонстровано фрагмент коду для сторінки входу.

Лістинг 3.19 – Сторінка входу

```
            <Text className='text-base text-gray-400 w-full items-
start'>Email</Text>
            <AuthInput placeholder="Enter your email..." value={email}
onChangeText={setEmail} />
            <Text className='text-base text-gray-400 w-full items-
start'>Password</Text>
            <AuthInput placeholder="Enter your password..."
value={password} onChangeText={setPassword} secureTextEntry />
            <Text className='text-base text-gray-400 w-full items-start
mt-2 mb-8'>Forgot your password?</Text>
            <AuthButton
              title="Sign in"
              onPress={async () => {
                try {
                  debugger;
                  let response = await onLogin(email, password);
                  if (response.success) {
                    router.replace('/');
                  }
                } catch (error) {
```

```
                console.log(error);
              }
            }}
          />
          <Text className='text-base text-gray-400 w-full items-start
mt-8'>Create an account</Text>
        </View>
```

Кінець лістингу 3.19

В даному коді використано попередньо розроблені компоненти AuthButton – кнопка для виконання входу в систему та поле AuthInput для введення даних.

Після виконання входу, користувач потрапляє до вкладок. Тому наступним кроком стало створення вкладок для списку чатів, дзвінків, друзів та групових чатів. На лістингу 3.20 приведено фрагмент коду для вкладки з чатами.

Лістинг 3.20 – Сторінка входу

```
    <View className="flex-1 bg-gray-900 p-5">
          <Text className="text-2xl font-bold text-white mb-
5">Conversations</Text>
          <TextInput
            placeholder="Search chats"
            className="bg-gray-800 rounded-lg px-3 mb-5 text-white"
            placeholderTextColor="grey"
          />
          <View className="flex-row justify-between mb-5">
            <Button label="All" className="flex-1 mx-1 bg-gray-800
rounded-full" labelStyle={{ color: 'white' }} />
            <Button label="Work" className="flex-1 mx-1 bg-gray-800
rounded-full" labelStyle={{ color: 'white' }} />
            <Button label="Archived" className="flex-1 mx-1 bg-gray-800
rounded-full" labelStyle={{ color: 'white' }} />
          </View>
          <FlatList
            data={chats}
            renderItem={renderItem}
            keyExtractor={(item) => item.id}
            contentContainerStyle={{ paddingBottom: 20 }}
          />
        </View>
```

Кінець лістингу 3.20

3.4 Захист інформаційно-комп’ютерної системи

Алгоритм BCrypt призначений для безпечного хешування паролів. Він використовує адаптивний підхід, дозволяючи збільшувати кількість ітерацій хешування для додаткового захисту. Основні етапи:

1) генерація солі;
2) комбінування пароля та солі;
3) застосування алгоритму Blowfish з заданою кількістю ітерацій;
4) генерація хешу, який зберігається в базі даних.

Blowfish – симетричний алгоритм шифрування, розроблений Брюсом Шнайєром у 1993 році. Він використовує 64-бітні блоки даних та ключі довжиною від 32 до 448 біт. Нижче на рисунку 3.6 зображено функціональну схему алгоритма шифрування.

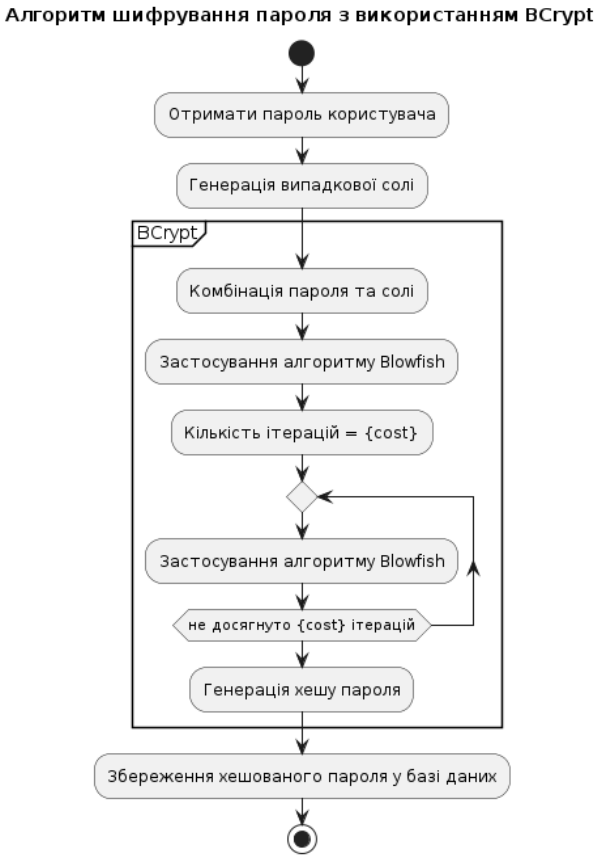

Рисунок 3.6 – Функціональна схема шифрування паролю за допомогою BCrypt

Blowfish використовує генератор псевдовипадкових чисел (ГПВЧ) для генерування ключа шифрування. ГПВЧ ініціалізується вхідними даними та використовується для генерування підключів, які задіяні в процесі шифрування та дешифрування. Параметр cost визначає, скільки разів ГПВЧ буде використовуватися для генерування підключів, впливаючи на стійкість та швидкість генерування ключа.Алгоритм Blowfish використовує генератор псевдовипадкових чисел (ГПВЧ) для генерування ключа шифрування. ГПВЧ ініціалізується вхідними даними P', а потім використовується для генерування 18 підключів P[0] ... P[17]. Ці підключі використовуються в процесі шифрування та дешифрування.

Нижче наведено лістинг коду 3.21, де використовується хешування пароля з використанням BCrypt.

Лістинг 3.21 – Приклад використання хешування пароля через BCrypt

```
public async Task<ErrorOr<AuthenticationResult>> Handle(RegisterCommand command, CancellationToken cancellationToken)
        {
            await Task.CompletedTask;

            if (await _userRepository.GetUserByEmailAsync(command.Email) is not null)
            {
                return Errors.User.DuplicateEmail;
            }

            var hashedPassword = BC.EnhancedHashPassword(command.Password, 13);
```

Кінець лістингу 3.21

3.5 Розробка документації для програмного продукту

Для того, щоб розгорнути веб-версію месенджера та api, потрібно виконати наступні кроки:

– встановлення .NET Core SDK:

1) завантажте останню версію .NET Core SDK з офіційного сайту Microsoft;

2) запустіть інсталятор та виконайте всі кроки, дотримуючись інструкцій;

3) після встановлення перевірте успішне встановлення, відкривши термінал та ввівши команду dotnet –version;

– клонування репозиторію проекту:

1) відкрийте термінал або командний рядок;

2) перейдіть до бажаної директорії, де ви хочете розмістити проект;

3) виконайте команду git clone https://github.com/UAndo/SocialDotNet.git для клонування репозиторію;

– встановлення Docker:

1) завантажте Docker для вашої операційної системи з офіційного сайту або інсталюйте через пакетний менеджер вашої системи;

2) після встановлення перевірте успішне встановлення, відкривши термінал та ввівши команду docker version;

– встановлення образу Microsoft SQL Server в Docker:

1) відкрийте термінал або командний рядок;

2) виконайте команду docker pull mcr.microsoft.com/mssql/server:2022-latest;

3) дочекайтесь успішного завершення завантаження образу;

4) виконайте команду «docker run -e “ACCEPT_EULA=Y” -e “SA_PASSWORD=ваш_пароль” -p 1433:1433 --name sql_server -d mcr.microsoft.com/mssql/server:2022-latest», замінивши «ваш_пароль» на безпечний пароль за вашим вибором;

5) дочекайтесь успішного запуску контейнера;

– оновлення бази даних за допомогою Entity Framework Core:

1) відкрийте термінал або командний рядок;

2) перейдіть до директорії SocialDotNet;

3) виконайте команду dotnet ef migrations add назва_міграції -p .\SocialDotNet.Infrastructure\ -s .\SocialDotNet.Server\ для створення нової міграції;

4) виконайте команду «dotnet ef database update -p .\SocialDotNet.Infrastructure\ -s .\SocialDotNet.Server\ --connection “Server=localhost;Database=SocialDotNet;User Id=sa;Password=ваш_пароль;Encrypt=false;”» для оновлення бази даних, замінивши «ваш_пароль» на встановлений вами пароль;

5) дочекайтесь успішного завершення процесу оновлення бази даних;

– встановлення Node.js:

1) перейдіть на офіційний сайт Node.js та завантажте інсталятор або використовуйте термінал і пакетний менеджер вашої системи;

2) після встановлення перезавантажте ваш комп'ютер;

– запуск проекту ASP.NET Core:

1) відкрийте термінал або командний рядок;

2) перейдіть до директорії SocialDotNet/SocialDotNet.Server;

3) виконайте команду dotnet run для запуску проекту ASP.NET Core;

4) дочекайтесь успішного завантаження та запуску проекту;

5) відкрийте веб-браузер та перейдіть за адресою http://localhost:7008/swagger для перевірки роботи Swagger UI та API.

Після виконання всіх цих кроків ви матимете повністю налаштоване та працююче середовище для розробки і тестування месенджера на основі ASP.NET Core та React. База даних буде створена та ініціалізована в контейнері Docker з Microsoft SQL Server, а ваш проект ASP.NET Core буде запущено локально.

Висновки до розділу 3

У цьому розділі було продемонстровано процес розробки API для месенджера, використовуючи CQRS, DDD та Чисту архітектуру. Також було застосування цих підходів дозволяє створити гнучку та масштабовану архітектуру месенджера. Впровадження CQRS забезпечує розділення команд (запис) та запитів (читання), що підвищує продуктивність системи та спрощує її модифікацію в майбутньому.

Також було розглянуто створення інтерфейсу користувача з використанням React та React Native, що забезпечує створення сучасного та зручного інтерфейсу, який легко адаптується до різних платформ. Використання Tailwind CSS дозволяє швидко та ефективно стилізувати компоненти інтерфейсу, забезпечуючи привабливий та функціональний дизайн.

Особливу увагу було приділено тестуванню API, що є критично важливим для забезпечення його надійної роботи. Було розроблено детальну інструкцію для розгортання програми, що спрощує її введення в експлуатацію.

Використання сучасних технологій та підходів дозволило створити продукт, який відповідає сучасним вимогам користувачів щодо продуктивності та зручності використання.

У цьому розділі було продемонстровано, що обрані підходи та технології забезпечують створення високоякісного продукту, який задовольняє вимоги сучасного ринку та має значний потенціал для подальшого розвитку та масштабування.

ВИСНОВКИ

Ця робота зосереджувалась на проектуванні та реалізації месенджера з використанням сучасних підходів та технологій. Перш за все, було проведено аналіз існуючих рішень, їх переваги та недоліки, а також вимоги користувачів до месенджерів, що визначило актуальність обраної теми.

Основною метою кваліфікаційної роботи було створення месенджера з використанням Domain-Driven Design та патерну CQRS для забезпечення гнучкості, масштабованості та можливості легкої модифікації системи в майбутньому. Цей підхід дозволяє ефективно керувати доменом додатка та розділяти операції читання та запису даних для підвищення продуктивності.

У рамках кваліфікаційної роботи було визначено об’єкт дослідження (месенджер) та його ключові концепції (повідомлення, користувачі, групи тощо), а також розроблена модель предметної області, яка відображає ці концепції та їх взаємозв’язки згідно з принципами DDD.

На практичному рівні було реалізовано серверну частину месенджера з використанням ASP.NET Core, Entity Framework Core, MediatR та інших сучасних технологій для забезпечення високої продуктивності, безпеки та зручного API взаємодії. Налаштування Docker та використання безпечних механізмів автентифікації дозволяють забезпечити стабільну та безпечну роботу месенджера.

Для клієнтської частини було розроблено макети веб та мобільного застосунків з використанням React і React Native для створення зручного та інтуїтивно зрозумілого інтерфейсу користувача на різних платформах.

Застосунок було протестовано за допомогою Swagger та інструментів Visual Studio. Було знайдено та виправлено недоліки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. WhatsApp. URL: https://web.whatsapp.com/ (дата звернення: 10.05.2024).
2. Telegram Desktop. URL: https://desktop.telegram.org/?setln=uk (дата звернення: 10.05.2024).
3. Signal Messenger: Speak Freely. URL: https://signal.org/uk/ (дата звернення: 10.05.2024).
4. Home | Viber. URL: https://www.viber.com/ua/ (дата звернення: 10.05.2024).
5. Unified Modeling Language (UML) Diagrams - GeeksforGeeks. URL: https://www.geeksforgeeks.org/unified-modeling-language-uml-introduction/ (дата звернення: 12.05.2024).
6. What is an Entity Relationship Diagram (ERD)? URL: https://www.lucidchart.com/pages/er-diagrams (дата звернення: 12.05.2024).
7. The Basics of Business Process Modeling and Notation (BPMN) – IBM Blog. URL: https://www.ibm.com/blog/bpmn/ (дата звернення: 12.05.2024).
8. Coolors – The super fast color palettes generator!. URL: https://coolors.co/ (дата звернення: 13.05.2024).
9. Adobe. URL: https://color.adobe.com/create/color-wheel (дата звернення: 13.05.2024).
10. Figma: The Collaborative Interface Design Tool. URL: https://www.figma.com (дата звернення: 13.05.2024).
11. Most used languages among software developers globally 2023 | Statista. URL: https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used/ (дата звернення: 13.05.2024).
12. Visual Studio: IDE and Code Editor for Software Developers and Teams. URL: https://visualstudio.microsoft.com/ (дата звернення: 16.05.2024).
13. Visual Studio Code – Code Editing. Redefined. URL: https://code.visualstudio.com/ (дата звернення: 16.05.2024).

14. Rider: The Cross-Platform .NET IDE from JetBrains. URL: https://www.jetbrains.com/rider/ (дата звернення: 16.05.2024).

15. Martin R. Clean Architecture: A Craftsman's Guide to Software Structure and Design (Robert C. Martin Series). Prentice Hall, 2017. 432 с.

16. Clean Coder Blog. URL: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html (дата звернення: 19.05.2024).

17. CQRS pattern – Azure Architecture Center. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs (дата звернення: 19.05.2024).

18. Vernon V. Implementing Domain-Driven Design. Pearson Education, Limited.

19. dbdiagram.io – Database Relationship Diagrams Design Tool. URL: https://dbdiagram.io/ (дата звернення: 19.05.2024).

20. Microsoft Data Platform | Microsoft. URL: https://www.microsoft.com/en-us/sql-server (дата звернення: 22.05.2024).

21. API Documentation & Design Tools for Teams | Swagger. URL: https://swagger.io/ (дата звернення: 22.05.2024).

22. Download .NET (Linux, macOS, and Windows). URL: https://dotnet.microsoft.com/en-us/download (дата звернення: 22.05.2024).

23. Docker: Accelerated Container Application Development. URL: https://www.docker.com/ (дата звернення: 22.05.2024).

24. Node.js — Run JavaScript Everywhere. URL: https://nodejs.org/en (дата звернення: 22.05.2024).

ДОДАТКИ

Додаток А

Лістнг А.1 – Агрегат користувача

```
using SocialDotNet.Domain.Common.Models;
using SocialDotNet.Domain.UserAggregate.Entities;
using SocialDotNet.Domain.UserAggregate.ValueObjects;
using System.Linq;

namespace SocialDotNet.Domain.UserAggregate
{
    public sealed class User : AggregateRoot<UserId>
    {
        public string FirstName { get; private set; }
        public string LastName { get; private set; }
        public string Username { get; private set; }
        public string Email { get; private set; }
        public string PasswordHash { get; private set; }
        public DateTime CreatedAt { get; private set; }
        public DateTime UpdatedAt { get; private set; }
        public bool IsActive { get; private set; }
        public bool IsDeleted { get; private set; }
        public bool IsVerified { get; private set; }
        public string? VerificationCode { get; private set; }
        public DateTime VerificationCodeExpiration { get; private
set; }
        public string? PasswordResetCode { get; private set; }
        public DateTime PasswordResetCodeExpiration { get; private
set; }
        public string? ProfileImage { get; private set; }
        public string? Bio { get; private set; }
        public List<RefreshToken> RefreshTokens { get; private set;
}
        private readonly List<Friendship> _friendships = [];
        public IReadOnlyList<Friendship> Friendships =>
_friendships.AsReadOnly();

        private readonly List<FriendRequest> _friendRequests = [];
        public IReadOnlyList<FriendRequest> FriendRequests =>
_friendRequests.AsReadOnly();

        private User(
            UserId Id,
            string firstName,
            string lastName,
            string username,
            string email,
```

```
            string passwordHash,
            List<RefreshToken> refreshToken) : base(Id)
        {
            FirstName = firstName;
            LastName = lastName;
            Username = username;
            Email = email;
            PasswordHash = passwordHash;
            CreatedAt = DateTime.UtcNow;
            RefreshTokens = refreshToken;
        }

        private User()
        {
        }

        public static User Create(
            string firstName,
            string lastName,
            string username,
            string email,
            string passwordHash,
            List<RefreshToken> refreshTokens)
        {
            var user = new User(UserId.CreateUnique(), firstName,
lastName, username,
                email, passwordHash, refreshTokens);

            return user;
        }

        public void AddFriendRequest(FriendRequest friendRequest)
        {
            if (!_friendRequests.Contains(friendRequest))
            {
                _friendRequests.Add(friendRequest);
            }
        }

        public void AcceptFriendRequest(FriendRequest
friendRequest)
        {
            _friendRequests.FirstOrDefault(x => x.Id ==
friendRequest.Id)?.Accept();
        }
```

```
            public void RejectFriendRequest(FriendRequest
friendRequest)
            {
                _friendRequests.FirstOrDefault(x => x.Id ==
friendRequest.Id)?.Reject();
            }

            public void AddFriendship(Friendship friendship)
            {
                if (!_friendships.Contains(friendship))
                {
                    _friendships.Add(friendship);
                }
            }

            public void RemoveFriendship(Friendship friendship)
            {
                if (_friendships.Contains(friendship))
                {
                    _friendships.Remove(friendship);
                }
            }
        }
    }
```

Кінець лістингу А.1

Лістинг А.2 – AuthControler

```
    using ErrorOr;
    using MapsterMapper;
    using MediatR;
    using Microsoft.AspNetCore.Authorization;
    using Microsoft.AspNetCore.Mvc;
    using SocialDotNet.Application.Authentication.Commands.Register;
    using SocialDotNet.Application.Authentication.Commands.RevokeToken;
    using
SocialDotNet.Application.Authentication.Commands.UpdateRefreshToken;
    using SocialDotNet.Application.Authentication.Common;
    using SocialDotNet.Application.Authentication.Queries.Login;
    using SocialDotNet.Contracts.Authentication;
    using SocialDotNet.Domain.Common.Errors;

    namespace SocialDotNet.Server.Controllers
    {
        [Route("auth")]
        public class AuthController : ApiController
        {
```

```
        private readonly ISender _mediator;
        private readonly IMapper _mapper;

        public AuthController(IMediator mediator, IMapper mapper)
        {
            _mediator = mediator;
            _mapper = mapper;
        }

        [AllowAnonymous]
        [HttpPost("register")]
        public async Task<IActionResult> Register(RegisterRequest
request)
        {
            var command = _mapper.Map<RegisterCommand>(request);
            ErrorOr<AuthenticationResult> authResult = await
_mediator.Send(command);
            return authResult.Match(
                authResult => {
                    var response =
_mapper.Map<AuthenticationResponse>(authResult);
                    SetTokenCookie(response.RefreshToken);
                    return Ok(response);
                },
                errors => Problem(errors));
        }

        [AllowAnonymous]
        [HttpPost("login")]
        public async Task<IActionResult> Login(LoginRequest
request)
        {
            var query = _mapper.Map<LoginQuery>(request);
            var authResult = await _mediator.Send(query);

            if (authResult.IsError && authResult.FirstError ==
Errors.Authentication.InvalidCredentials)
            {
                return Problem(
                    statusCode: StatusCodes.Status401Unauthorized,
                    title: authResult.FirstError.Description);
            }

            return authResult.Match(
                authResult => {
                    var response =
_mapper.Map<AuthenticationResponse>(authResult);
```

```
                        SetTokenCookie(response.RefreshToken);
                        return Ok(response);
                    },
                    errors => Problem(errors));
            }

            [AllowAnonymous]
            [HttpPost("refresh-token")]
            public async Task<IActionResult> RefreshToken()
            {
                var refreshToken = Request.Cookies["refreshToken"];
                var request = new RefreshTokenRequest(refreshToken);
                var command =
_mapper.Map<UpdateRefreshTokenCommand>(request);
                var authResponse = await _mediator.Send(command);
                if (authResponse.IsError && authResponse.FirstError ==
Errors.Token.InvalidToken)
                {
                    return Problem(
                        statusCode: StatusCodes.Status401Unauthorized,
                        title: authResponse.FirstError.Description);
                }

                return authResponse.Match(
                    authResponse => {
                        var response =
_mapper.Map<AuthenticationResponse>(authResponse);
                        SetTokenCookie(response.RefreshToken);
                        return Ok(response);
                    },
                    errors => Problem(errors));
            }

            [HttpGet("revoke-token")]
            public async Task<IActionResult> RevokeToken()
            {
                // accept refresh token in request body or cookie
                var token = Request.Cookies["refreshToken"];
                var request = new RevokeTokenRequest(token);
                var command = _mapper.Map<RevokeTokenCommand>(request);
                var response = await _mediator.Send(command);

                return response.Match(
                    response => Ok(new { message = "Token revoked" }),
                    errors => Problem(errors));
            }
```

```
            [HttpGet("ping-auth")]
            public IActionResult PingAuth()
            {
                return Ok();
            }

            [AllowAnonymous]
            [HttpGet("get-ip-address")]
            public IActionResult GetIpAddress()
            {
                // get source ip address for the current request
                if (Request.Headers.ContainsKey("X-Forwarded-For"))
                    return Ok(Request.Headers["X-Forwarded-
For"].FirstOrDefault());
                else
                    return
Ok(HttpContext.Connection.RemoteIpAddress.MapToIPv4().ToString());
            }

            private void SetTokenCookie(string token)
            {
                // append cookie with refresh token to the http
response
                var cookieOptions = new CookieOptions
                {
                    HttpOnly = true,
                    Expires = DateTime.UtcNow.AddDays(7)
                };
                Response.Cookies.Append("refreshToken", token,
cookieOptions);
            }
        }
```

Кінець лістингу А.2

Лістинг А.3 – Обробник команди на надсилання запиту в друзі

```
    using ErrorOr;
    using MediatR;
    using SocialDotNet.Application.Common.Interfaces.Persistence;
    using SocialDotNet.Application.FriendRequests.Common;
    using SocialDotNet.Domain.UserAggregate.Entities;
    using SocialDotNet.Domain.UserAggregate.ValueObjects;

    namespace
SocialDotNet.Application.FriendRequests.Commands.CreateFriendRequest
    {
```

```
        public class SendFriendRequestCommandHandler :
IRequestHandler<SendFriendRequestCommand, ErrorOr<FriendRequestResult>>
        {
            private readonly IUserRepository _userRepository;

            public SendFriendRequestCommandHandler(IUserRepository
userRepository)
            {
                _userRepository = userRepository;
            }

            public async Task<ErrorOr<FriendRequestResult>>
Handle(SendFriendRequestCommand request, CancellationToken
cancellationToken)
            {
                var friendRequest =
FriendRequest.Create(UserId.Create(request.SenderId),
UserId.Create(request.ReceiverId));
                var user = await
_userRepository.GetByIdAsync(friendRequest.SenderId);
                user!.AddFriendRequest(friendRequest);
                await _userRepository.UpdateAsync(user);
                return new FriendRequestResult();
            }
        }
    }
```

Кінець лістингу А.3

Додаток Б

Лістинг Б.1 – Компонент, який перевіряє чи авторизований користувач

```
import React, { useState, useEffect, createContext } from 'react';
import { Navigate } from 'react-router-dom';
import axiosWithAuth from '../services/axiosWithAuth';

const UserContext = createContext({});

interface User {
    email: string;
}

function AuthorizeView(props: { children: React.ReactNode }) {
    const [authorized, setAuthorized] = useState<boolean>(false);
    const [loading, setLoading] = useState<boolean>(true);
    const [user, setUser] = useState<User>({ email: "" });
    const [redirect, setRedirect] = useState<boolean>(false);

    useEffect(() => {
        let retryCount = 5;
        const maxRetries = 1;
        const delay = 1000;

        const wait = (delay: number) => new Promise(resolve =>
setTimeout(resolve, delay));

        const axiosWithRetry = async (url: string, options: any) =>
{
            try {
                const response = await axiosWithAuth(url, options);
                if (response.status === 200) {

                    const userData = response.data;
                    setUser({ email: userData.email });
                    setAuthorized(true);
                    return response;
                } else if (response.status === 401) {

                    setRedirect(true);
                    return response;
                } else {
                    throw new Error("" + response.status);
                }
```

```
                } catch (error) {
                    retryCount++;
                    if (retryCount > maxRetries) {
                        setRedirect(true);
                        throw error;
                    } else {
                        await wait(delay);
                        return axiosWithRetry(url, options);
                    }
                }
            };

            axiosWithRetry("auth/ping-auth", { method: "GET" })
                .catch(error => {
                    console.log(error.message);
                })
                .finally(() => {
                    setLoading(false);
                });
        }, []);

        if (loading) {
            return <p>Loading...</p>;
        } else {
            if (redirect) {
                return <Navigate to="/login" />;
            }
            if (authorized) {
                return <UserContext.Provider
value={user}>{props.children}</UserContext.Provider>;
            } else {
                return <Navigate to="/login" />;
            }
        }
    }

    export function AuthorizedUser(props: { value: string }) {
        const user: any = React.useContext(UserContext);
        if (props.value === "email") return <>{user.email}</>;
        else return <></>;
    }

    export default AuthorizeView;
```

Кінець лістингу Б.1

Лістинг Б.2 – Компонент бокового меню

```
import React from 'react';
import { Menu, MenuButton, MenuItem, MenuItems, Transition } from
'@headlessui/react';
import axiosWithAuth from '../services/axiosWithAuth';
import { ChatBubbleOvalLeftIcon, PhoneIcon, UserIcon, UserGroupIcon,
Cog6ToothIcon, Bars3Icon } from '@heroicons/react/24/outline';
import { useNavigate } from 'react-router-dom';

interface SidebarProps {
    isSidebarVisible: boolean;
    toggleSidebar: () => void;
    entityType: string;
    setSelectedOption: (option: string) => void;
}
const Sidebar: React.FC<SidebarProps> = ({ isSidebarVisible,
toggleSidebar, entityType, setSelectedOption }) => {
    const navigate = useNavigate();
    const logout = async () => {
        await axiosWithAuth('/auth/revoke-token');
        localStorage.removeItem('jwt');
        navigate("/login");
    };
    return (
                    <div    className={`transition-all    duration-300
${isSidebarVisible ? 'w-[4%]' : 'w-[0%]'} text-gray-500 h-screen flex flex-
col items-center justify-between py-5`}>
                <div className="flex flex-col w-full items-center">
                                <div   className="mb-4   cursor-pointer"
onClick={toggleSidebar}>
                        <Bars3Icon className="h-6 w-6" />
                    </div>
                    <div className={`flex flex-col items-center justify-
center cursor-pointer h-10 ${entityType === 'chats' ? 'bg-royalPurple
rounded-md w-[80%]' : ''}`} onClick={() => setSelectedOption('chats')}>
                        <ChatBubbleOvalLeftIcon className="h-6 w-6" />
                    </div>
                    <div className={`flex flex-col items-center justify-
center cursor-pointer h-10 ${entityType === 'calls' ? 'bg-royalPurple
rounded-md w-[80%]' : ''}`} onClick={() => setSelectedOption('calls')}>
                        <PhoneIcon className="h-6 w-6" />
                    </div>
                    <div className={`flex flex-col items-center justify-
center cursor-pointer h-10 ${entityType === 'friends' ? 'bg-royalPurple
rounded-md w-[80%]' : ''}`} onClick={() => setSelectedOption('friends')}>
                        <UserIcon className="h-6 w-6" />
                    </div>
                    <div className={`flex flex-col items-center justify-
center cursor-pointer h-10 ${entityType === 'groups' ? 'bg-royalPurple
rounded-md w-[80%]' : ''}`} onClick={() => setSelectedOption('groups')}>
```

```
                        <UserGroupIcon className="h-6 w-6" />
                    </div>
                </div>
                <div className="">
                    <Menu>
                       <MenuButton className="flex items-center justify-
center focus:outline-none">
                            <Cog6ToothIcon className="h-6 w-6" />
                        </MenuButton>
                        <Transition
                            enter="transition ease-out duration-75"
                            enterFrom="opacity-0 scale-95"
                            enterTo="opacity-100 scale-100"
                            leave="transition ease-in duration-100"
                            leaveFrom="opacity-100 scale-100"
                            leaveTo="opacity-0 scale-95"
                        >
                            <MenuItems
                                anchor="top end"
                                  className="absolute right-0 w-52 mt-2
origin-top-right   bg-white   dark:bg-gray-800   border   border-gray-200
dark:border-gray-700   divide-y   divide-gray-100   dark:divide-gray-700
rounded-md shadow-lg outline-none"
                            >
                                <div className="px-1 py-1 text-white">
                                    <MenuItem>
                                        {({ active }) => (
                                            <button className={`${active
? 'bg-gray-100 dark:bg-gray-700' : ''} group flex rounded-md items-center
w-full px-2 py-2 text-sm`}>
                                                Settings
                                            </button>
                                        )}
                                    </MenuItem>
                                    <MenuItem>
                                        {({ active }) => (
                                            <button className={`${active
? 'bg-gray-100 dark:bg-gray-700' : ''} group flex rounded-md items-center
w-full px-2 py-2 text-sm`} onClick={() => logout()}>
                                                Logout
                                            </button>
                                        )}
                                    </MenuItem>
                                </div>
                            </MenuItems>
                        </Transition>
                    </Menu>
                </div>
            </div>
```

```
    );
};
export default Sidebar;
```

Кінець лістингу Б.2