Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра комп'ютерних наук
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка програми для обміну текстовими повідомленнями на основі платформи .Net.

Виконав: студент IV курсу, групи СНс-42
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)

_____________ Дацик С.В.
(підпис) (прізвище та ініціали)

Керівник _____________ Готович В.А.
(підпис) (прізвище та ініціали)

Нормоконтроль _____________ Шимчук Г.В.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____________ Боднарчук І.О.
(підпис) (прізвище та ініціали)

Рецензент _____________ Тиш Є.В.
(підпис) (прізвище та ініціали)

Тернопіль
2022

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
______ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«__» ________ 2022 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавр
(назва освітнього ступеня)

за спеціальністю 122 Комп'ютерні науки
(шифр і назва спеціальності)

Студенту Дацику Станіславу Васильовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програми для обміну текстовими повідомленнями на основі платформи .Net.

Керівник роботи Готович Володимир Анатолійович, кандидат технічних наук, доцент кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 16 » березня 2022 року № 4/7-161

2. Термін подання студентом завершеної роботи 22.06.2022 р.

3. Вихідні дані до роботи Перелік літературних та інтернет джерел

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1. Поставка задачі. 1.1 Вибір архітектури проекту. 1.2 Вибір технології розробки. 1.3 Аналіз поставленого завдання. 1.4 Моделювання архітектури програмного забезпечення 1.5 Розробка моделей даних. 1.6 Висновок до першого розділу. 2. Проектування та реалізація програмного рішення 2.1 Проектування інтерфейсу програмного забезпечення. 2.2 Реалізація інтерфейсу бази даних. 2.3 Проектування методів для операцій з базою даних. 2.4 Тестування та встановлення програмного забезпечення. 2.5 Висновок до другого розділу. 3. Безпека життєдіяльності, основи охорони праці. 3.1 Соціальні та психологічні фактори ризику. 3.2 Загальні вимоги безпеки з охорони праці для користувачів ПК. 3.3 Соціальне значення охорони праці. 3.4 Висновок до третього розділу. Перелік джерел. Додатки.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Вступ, актуальність теми, мета та задачі дослідження, вибір архітектури, роль платформи .NET, технологія WebSocket, програмна реалізація застосунку, підсумки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи хорони праці	Гурик О.Я., доцент кафедри МТ		

7. Дата видачі завдання 24 січня 2022 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	24.01.2022	*Виконано*
2.	Підбір джерел про технології розробки програми для обміну текстовими повідомленнями на основі платформи .Net	04.01.2022-30.01.2022	*Виконано*
3.	Переклад та опрацювання джерел про технології розробки програми для обміну текстовими повідомленнями на основі платформи .Net	31.01.2022-06.02.2022	*Виконано*
4.	Виконання дослідження щодо актуальності розроблюваного проекту. Розробка програми для обміну текстовими повідомленнями на основі платформи .Net	07.02.2022-13.02.2022	*Виконано*
5.	Оформлення розділу «1. Аналіз поставленого завдання»	14.02.2022-06.03.2022	*Виконано*
6.	Оформлення розділу «2. Проектування та реалізація програми»	07.03.2022-03.04.2022	*Виконано*
7.	Виконання завдання до підрозділу «Безпека життєдіяльності»	04.04.2022-17.04.2022	*Виконано*
8.	Виконання завдання до підрозділу «Основи охорони праці»	18.04.2022-01.05.2022	*Виконано*
9.	Оформлення кваліфікаційної роботи	02.05.2022-15.05.2022	*Виконано*
10.	Нормоконтроль	16.05.2022-22.05.2022	*Виконано*
11.	Перевірка на плагіат	08.06.2022	*Виконано*
12.	Попередній захист кваліфікаційної роботи	09.06.2022	*Виконано*
13.	Захист кваліфікаційної роботи	22.06.2022	

Студент ____________ Дацик С.В.
(підпис) (прізвище та ініціали)

Керівник роботи ____________ Готович В.А.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка програми для обміну текстовими повідомленнями на основі платформи .Net. // Кваліфікаційна робота освітнього рівня «Бакалавр» // Дацик Станіслав Васильович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп’ютерно-інформаційних систем і програмної інженерії, кафедра комп’ютерних наук, група СНс-42 // Тернопіль, 2022 // С. – 52, рис. – 25, кресл. – , табл. – 0, додат. – 3 , бібліогр. 33.

Ключові слова: програма-месенджер, клієнт-сервер, API, веб-сокет.

Кваліфікаційна робота присвячена розробці програми для обміну текстовими повідомленнями на основі платформи .Net.

Метою роботи є закріплення теоретичних знань та практичних навичок по використанню можливостей платформи .Net при розробці програмного рішення для обміну текстовими повідомленнями.

В першому розділі кваліфікаційної роботи здійснено аналіз та вибір найбільш ефективних технологій для реалізації поставленого завдання. Здійснено моделювання архітектури програмних частин проекту. В результаті отримано можливіть практичної реалізації програми з відповідними функціональними можливостями.

В другому розділі кваліфікаційної роботи здійснено проектування та реалізацію клієнтської та серверної частин архітектури програми для обміну текстовими повідомленнями на основі платформи .Net. В результаті отримано програму з відповідним функціоналом, яку перевірено на коректність роботи в цілому.

В третьому розділі кваліфікаційної роботи розглянуто питання з безпеки життєдіяльності та охорони праці. Розглянуто три питання пов’язані з поданою тематикою, яка відповідає темі кваліфікаційної роботи.

ANNOTATION

Application development for text messaging based on the .Net platform // Qualification work of the educational level «Bachelor» // Datsyk Stanislav Vasilovich // Ternopil Ivan Pul'uj National Technical University, Faculty of Computer Information Systems and Software Engineering, group SNs–42 // Ternopil, 2022 // p. – 52, fig. – 25, sheets – , tab. – 0, add. – 3, ref. – 33.

Keywords: messenger, client-server, API, web socket.

Qualification work is dedicated application development for text messaging based on the .Net platform.

The purpose of the work is to consolidate theoretical knowledge and practical skills to use the capabilities of the .Net platform with the relevant project, namely application for text messaging based on the .Net platform.

The first section of the qualification work considers the concept of using the most effective technologies for project implementation. It is the choice of the architecture on which it will be based, then choosing the necessary tools for development. As a result, we get a certain concept of what we can develop with the appropriate functionality. Modeling the architecture of the program and its data while gaining the opportunity for practical implementation.

The second section of the qualification work considers the design and implementation of client and server architectural part application for text messaging based on the .Net platform. In the end, we received a program with the appropriate functionality, which was tested for correctness of the work as a whole.

In the third section of the qualification work, the issues of life safety and labor protection are considered. Three issues related to the given topic, which corresponds to the topic of the qualification work, are considered.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

БД – база даних;

Месенджер – програма для обміну повідомленнями між користувачами системи;

ПЗ – програмне забезпечення;

СУБД (укр. Система Управління Базами Даних) – набір програм, які дозволяють створювати базу даних та маніпулювати даними;

DFD (англ. Data Flow Diagrams) – діаграма потоків даних;

Entity – об'єкти для яких важлива ідентичність, а життєвий цикл яких зазвичай ширший ніж робота аплікації;

EF Core (англ. Core Entity Framework Core) – технологія доступу до даних;

ER (англ. Entity-Relationship model) – модель даних сутність-зв’язок;

.Net – платформа для розробки застосунків різного призначення;

Web Socket – протокол передачі даних;

WPF (анг. Windows Presentation Foundation) – технологічне середовище яке призначене для користувача інтерфейсу.

ЗМІСТ

ВСТУП 7
РОЗДІЛ 1. АНАЛІЗ ПОСТАВЛЕНОГО ЗАВДАННЯ 8
1.1 Поставка задачі 8
1.2 Вибір архітектури проекту 9
1.3 Вибір технології розробки 12
1.4 Моделювання архітектури програмного забезпечення 15
1.5 Розробка моделей даних 17
1.6 Висновок до першого розділу 20
РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО РІШЕННЯ 21
2.1 Проектування інтерфейсу програмного забезпечення 21
2.2 Реалізація інтерфейсу бази даних 30
2.3 Проектування методів для операцій з базою даних 37
2.4 Тестування та встановлення програмного забезпечення 38
2.5 Висновок до другого розділу 43
РОЗДІЛ 3. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ 44
3.1 Соціальні та психологічні фактори ризику 44
3.2 Загальні вимоги безпеки з охорони праці для користувачів ПК 46
3.3 Соціальне значення охорони праці 48
3.4 Висновок до третього розділу 49
ВИСНОВКИ 50
ПЕРЕЛІК ДЖЕРЕЛ 51
ДОДАТКИ

ВСТУП

Актуальність теми роботи полягає в тому, що вплив мережі Інтернет на різні аспекти соціальної взаємодії на сьогоднішній день не викликає заперечень. Можливості додатків дозволяють інтернет-користувачам споживати і передавати великий обсяг інформації в режимі реального часу за допомогою своїх пристроїв. Тенденції, що формуються в рамках широко розповсюджених інформаційних технологій, свідчать про актуальність програм для обміну текстовими повідомленнями.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Бакалавр» є розробка програми для обміну текстовими повідомленнями на основі платформи .Net.

Для досягнення поставленої мети потрібно розв’язати наступні **задачі**:

– аналіз відомих засобів та методів розробки програми для обміну текстовими повідомленнями на основі платформи .Net;

– релізація та обгрунтування засобів і методів розробки програми для обміну текстовими повідомленнями на основі платформи .Net;

– проектування та конфігурація програми для обміну текстовими повідомленнями на основі платформи .Net;

– використання та тестування програми для обміну текстовими повідомленнями на основі платформи .Net.

Об’єкт дослідження – засоби та методи розробки програмного забезпечення для обміну текстовими повідомленнями на основі платформи .Net.

Предмет дослідження – процес розробки програмного забезпечення для обміну програми текстовими повідомленнями на основі платформи .Net.

Практичне значення отриманих результатів полягає в створенні програмного рішення для обміну текстовими повідомленнями.

РОЗДІЛ 1. АНАЛІЗ ПОСТАВЛЕНОГО ЗАВДАННЯ

1.1 Поставка задачі

Під час пошуку існуючих програм для обміну текстовими повідомленнями було виявлено, що потрібно:

– розробити простий та зрозумілий інтерфейс;

– використати актуальні на даний момент технології;

– забезпечити надійним захистом від зовнішніх чинників;

– надати повний функціонал програми без обмежень.

Оскільки, після аналізу було виявлено, що більшість програм цієї тематики не відповідають поданим особливостям, перейдемо до вибору вимог щодо програми для обміну текстовими повідомленнями.

Поставлено завдання розробити програму для обміну текстовими повідомленнями на основі платформи .Net, використовуючи мову програмування C# – проста, потужна, статично типізована, об’єктно орієнтована мова програмування [1]. Програма повинна виконувати основні функції для обміну повідомленнями, а також мати можливість виконувати наступні операції:

– аутентифікація;

– авторизація;

– створення, додавання кімнат;

– обмін текстовими повідомленнями;

– оновлювання вікно програми.

Програмний продукт повинен відповідати наступним вимогам надійності:

– HTTP шифрування передачі даних;

– HMAC 256 – одностороннє шифрування з побітовим вміщенням даних для зберігання паролів [2];

– EF core захист від SQL-ін'єкції щодо програми.

1.2 Вибір архітектури проекту

Для реалізації будь-якої програми необхідно чітко визначити яку саме архітектуру доречно використовувати згідно поставленого завдання. Вибір архітектури проекту відіграє важливу роль у розробці програми, як основа на котрій буде реалізуватись. Оптимальний вибір архітектури надає можливість більш ефективно використовувати функціонал програми. При цьому в подальшому маючи можливість модифікувати проект без додаткових проблем.

Архітектура програмного забезпечення полягає в способі структурування програмної або обчислювальної системи, абстракції елементів на певній фазі роботи. Система може складатись з кількох рівнів абстракції і мати багато фаз роботи, кожна з яких може мати окрему архітектуру [3].

Згідно поставленого завдання було обрано клієнт-серверну архітектуру, яка найбільш підходить для реалізації програми для обміну текстовими повідомленнями.

Клієнт-серверна архітектура широко використовується під час роботи з базами даних, в яких зосереджена значна частина інформації. Дану архітектуру можна означити як концепцію інформаційної мережі, в якій основна частина її ресурсів зосереджена в серверах, які обслуговують своїх клієнтів. Така архітектура визначає такі типи компонентів [4]:

– набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;

– набір клієнтів, які використовують сервіси, що надаються серверами;

– мережа, яка забезпечує взаємодію між клієнтами та серверами.

На рисунку 1.1 зображено структурну схему концепції архітектури.

Модель такої системи визначається перш за все розподілом обов’язків між клієнтом та сервером. Можна відокремити наступні рівні операції:

– рівень представлення даних;

– прикладний рівень;

– рівень управління даними.

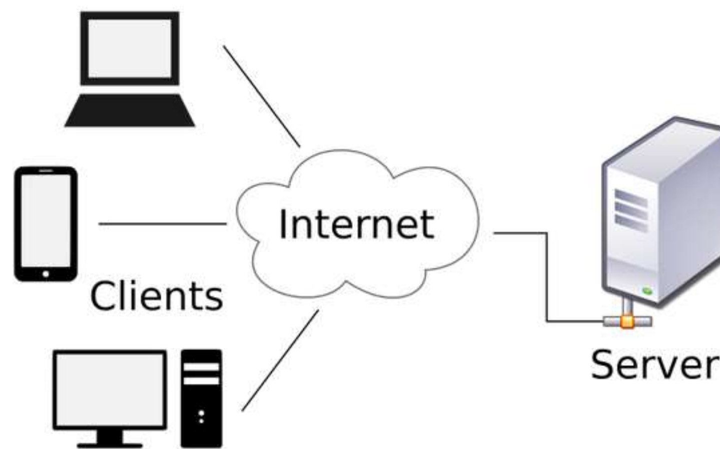

Рисунок 1.1 – Структурна схема клієнт-серверної архітектури

Перший рівень відповідає за представлення даних користувачеві і введення від нього керуючих команд, по суті являє собою інтерфейс користувача. На прикладному рівні здійснюється необхідна обробка інформації, яка реалізує основну логіку застосунку. Останній рівень зберігає данні та надає доступ до них [5].

Мережі клієнт-серверної архітектури містять наступні переваги:

– організацію мережі з певною кількість робочих станцій;

– забезпечення управління доступу та безпеки облікових записів;

– доступ до мережевих ресурсів;

– введення одного паролю надає доступ до всіх ресурсів, на які поширюється доступ користувача;

Без недоліків тут не обійтись, таких як:

– несправність сервера сприяє виникненню проблем з працездатності мережі;

– необхідність кваліфікованого персоналу для адміністрування;

– забезпечення дорогим обладненням мереж.

Для взаємодії між клієнтом і сервером необхідно використовувати протоколи передачі даних, так званні правила, завдяки яким здійснюється обмін даними. Серед таких протоколів часто зустрічається HTTP (або HTTPS). Недоліком цього протоколу є наявність повторюваних запитів з метою

отримання нової інформації. Щоб вирішити недолік використаємо протокол WebSocket, який створювався для того, щоб можна було підтримувати тривалий зв'язок між клієнтом та сервером [6].

Веб-сокети можна використовувати для проектування:

- додатків реального часу;
- чат-додатків;
- IoT-додатків.

На рисунку 1.2 зображено функціональну схему протоколів передачі даних.

Протокол не містить шифрування даних, тому відсутність будь-яких доповнень у протоколі WebSocket, який забезпечує двонаправлений канал зв'язку через один TCP-сокет [7]. Отримавши швидкість та тривалість роботи зв'язку, яка необхідна для розроблюваного проекту, а саме – програми для обміну тестовими повідомленнями на основі платформи .Net.

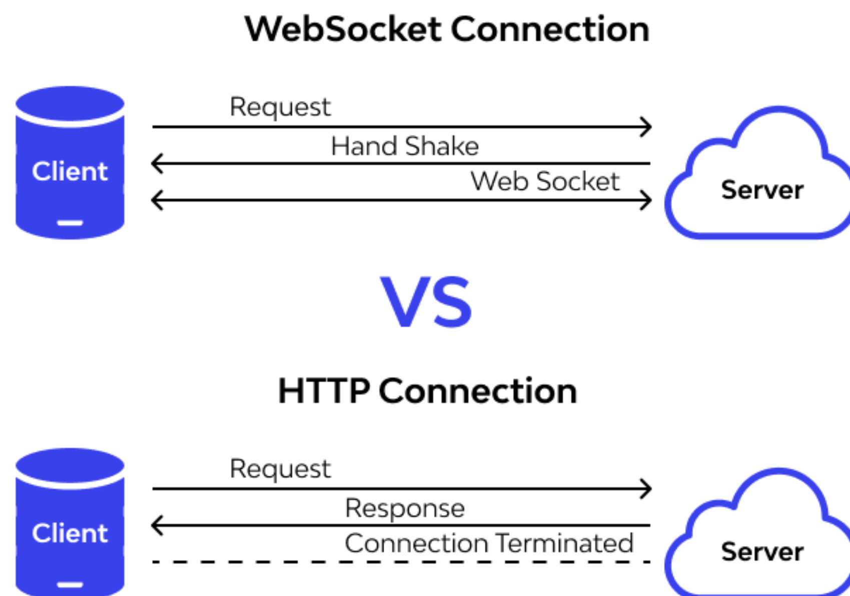

Рисунок 1.2 – Функціональна схема протоколів передачі даних

WebSocket – зручна технологія передачі динамічно оновлюваних даних. Її впровадження дозволяє понизити навантаження на мережу і серверне

устаткування, прискорити роботу програми. Перейдемо до переваг та недоліки використання протоколу передачі даних WebSocket.

Переваги використання:

– стандартизація рішень. Простіша розробка кросплатформних рішень;

– необмежений час з’єднання. Не потрібно періодино оновлювати;

– швидка передача динамічних даних. Без лишнього навантаження на сервер;

– безліч одночасно відкритих сесій.

Недоліки використання:

– відвал мережі без повідомлення. Іноді потрібно використовувати додаткові зв’язкими між клієнтом та сервером;

– зміна мережі клієнтом. Інформація про зміну адреси не надається серверу, коли клієнт не закрив з’єднання;

– підвищені вимоги до серверного устаткування.

1.3 Вибір технології розробки

Для реалізації цієї програми було обране середовище розробки Microsoft Visual Studio 2022, який дозволяє створювати програми, які використовують у своїй роботі платформу .Net. Вона дозволяє використовувати великий набір сервісів, що реалізуються у вигляді проміжного, незалежного від базової архітектури, коду.

Основною метою створення платформи .Net є можливість реалізації розробниками спеціальних сервісно-орієнтованих програм, що працюють на будь-яких платформах [8]. До цієї платформи краще підходить мова програмування C#, яка й буде використовуватися в проекті.

.NET Framework складається із загальномовного середовища виконання (CLR) та бібліотек класів .NET Framework. У основі .NET Framework лежить середовище CLR. Середовище виконання можна розглядати як агент, що

управляє кодом під час виконання та надає базові служби, такі як управління пам'яттю, багатопоточність та віддалену взаємодію.

В той же час середовище накладає строго типізовані умови та інші типи перевірок точності коду для забезпечення безпеки та надійності. По суті, основне завдання середовища виконання – керування кодом. Код, який звертається до середовища виконання, називається керованим кодом, а код, який не звертається до середовища виконання, називається некерованим кодом.

Платформу .NET Framework можна використовувати для створення таких програм і служб:

– консольні додатки;

– застосунки з графічним інтерфейсом користувача для системи Windows (Windows Forms);

– застосунки NET.CORE;

– мобільні застосунки;

– веб-служби JSON;

– служби Windows [9].

У якості систем управління базами даних було обрано Microsoft SQL Server 2019. Для управління сервером і створення запитів використовується утиліта SQL Management Studio 2018.

MS SQL Management Studio для PostgreSQL — це комплексне рішення для розробки та керування базами даних PostgreSQL. Завдяки компонентам, розробленим для вирішення важливих завдань адміністрування бази даних PostgreSQL, SQL Studio надасть незамінні інструменти для адміністрування бази даних та управління об’єктами, а також створення бази даних.

PostgreSQL, виконання міграції, вилучення бази даних, створення SQL запитів, імпорт і експорт та порівняння даних. SQL Studio поєднує всі ці інструменти керування PostgreSQL в потужне та зручне середовище робочого місця, яке може працювати завжди. Використовуючи SQL Management Studio, можна автоматизувати різноманітні завдання з управління базами даних, такі як

міграція, завантаження та синхронізація даних, резервне копіювання та видалення баз даних.

SQL Server характеризується такими особливостями, як:

– продуктивність. SQL Server працює дуже швидко;

– надійність і безпека. SQL Server надає шифрування даних;

– простота. З цієї СУБД відносно легко працювати і вести адміністрування [10].

Центральним аспектом у MS SQL Server, як і в будь-якій СУБД, є база даних. База даних – сховище даних, структуризоване певним чином. В цілому база даних надає можливості створення, збереження, оновлення та пошуку інформації в базах даних, з контролем доступу до даних [11]. Для зберігання та адміністрування баз даних застосовуються системи керування базами даних (database management system) або СУБД. MS SQL Server є однією з таких СУБД.

Для реалізації програми використовуємо принципи SOLID – набір принципів об'єктно-орієнтованого програмування. Їхня ідея полягає в тому, що необхідно уникати залежностей між компонентами коду. Якщо є велика кількість залежностей, такий код важко підтримувати [12].

SOLID містить наступні принципи:

– принцип єдиної відповідальності;

– принцип відкритості, закритості;

– принцип підстановки Лісков;

– принцип розділення інтерфейсу;

– принцип інверсії залежностей.

Перший принцип вказує на те, що клас повинен виконувати лише одну дію для його зміни має бути тільки одна причина. Наступний принцип вказує, щоб класи були відкриті для розширення, але закриті для модифікації. Принцип підстановки Барбари Лісков вказує, що підкласи повинні замінювати свої базові класи. Розділення має на увазі, що речі треба зберігати окремо один від одного, а принцип розділення інтерфейсу торкається розділення інтерфейсів. Принцип

інверсії залежностей вказує, що наші класи повинні залежати від інтерфейсів або абстрактних класів, а не від конкретних класів і функцій.

При розробці програми для обміну текстовими повідомленнями було використано технологію Web Socket – передова технологія, яка дозволяє створювати інтерактивні зв'язки між клієнтом і сервером для обміну повідомленнями в режимі реально часу. Веб-сокети, на відміну від HTTP, дозволяють працювати з двонаправленим потоком даних, що робить цю технологію абсолютно унікальною [13].

Наступною технологією є Entity Framework Core, яка являє собою об'єктно-орієнтовану легку і розширювану технологію від компанії Microsoft для доступу до даних. EF Core є ORM-інструментом, тобто EF Core дозволяє працювати з базами даних, але являє собою більш високий рівень абстракції. EF Core дозволяє абстрагуватися від самої бази даних і таблиць та працювати з даними незалежно від типу сховища. Якщо на фізичному рівні оперуємо таблицями, індексами, первинними і зовнішніми ключами, то на концептуальному рівні, який нам пропонує Entity Framework, вже працюємо з об'єктами [14].

По суті .NET Core – це практично повне перезавантаження стека .NET Framework. З нової платформи з різних причин було виключено низку технологій. Слід розуміти, що платформа .NET Core розрахована насамперед на розробку для серверних та хмарних рішень. Для десктопних програм краще підходять класичний .NET для Windows з підтримкою WPF та Windows Forms [15].

1.4 Моделювання архітектури програмного забезпечення

Початковим етапом у розробці певного продукту на основі БД є здійснення аналізу предметної області, в якій буде реалізований проект. Для цього необхідно визначити структуру та основні компоненти, включаючи зв'язки.

Враховуючи дану предметну область, основним користувачами розроблюваної бази даних програми для обміну текстовими повідомленнями будуть соціальні групи людей. Так як основне завдання бази даних забезпечити всією необхідною інформацією користувача, якому може знадобитися можливість комунікації з іншими користувачами.

Основні концепції застосунку, які будуть взаємодіяти з БД, поділимо на наступні етапи:

– авторизація та аутентифікація. Надає користувачу можливість використання програмного продукту на повній основі вказавши необхідні параметри для користування повним функціоналом програми;

– створення та додавання кімнат. Надається після здійснення аунтентифікації. Суть функції відповідає самій назві. Створення кімнат здійснюється шляхом натискання на кнопку «Create room», після чого відкривається вікно, в якому необхідно вказати ім’я кімнати та підтвердити кнопкою «Add»;

– обмін повідомленнями. Дозволяє користувачам здійснювати комунікацію між собою. Для цього лише потрібно виконати попередні етапи, щоб отримати таку можливість;

– оновлення вікна програми. Відповідає за отримання останніх змін, які відбувалися зі сторони користувачів, на зразок отримання повідомлення.

Діаграми потоків даних містить чотири типи графічних елементів це процеси які являють собою трансформацію даних в рамках описуваної системи, сховища даних, зовнішні по відношенню до системи сутності, потоки даних між елементами трьох попередніх типів.

Наступним кроком буде розробка схеми потоку даних, яка забезпечить цілісну картину виконання запиту та структури обробки даних у системі. DFD є основним інструментом для моделювання функціональних вимог майбутнього програмного забезпечення [16]. Тому вимоги поділяються на процеси і зображаються як мережі, з'єднані потоками даних (Див. рисунок 1.3).

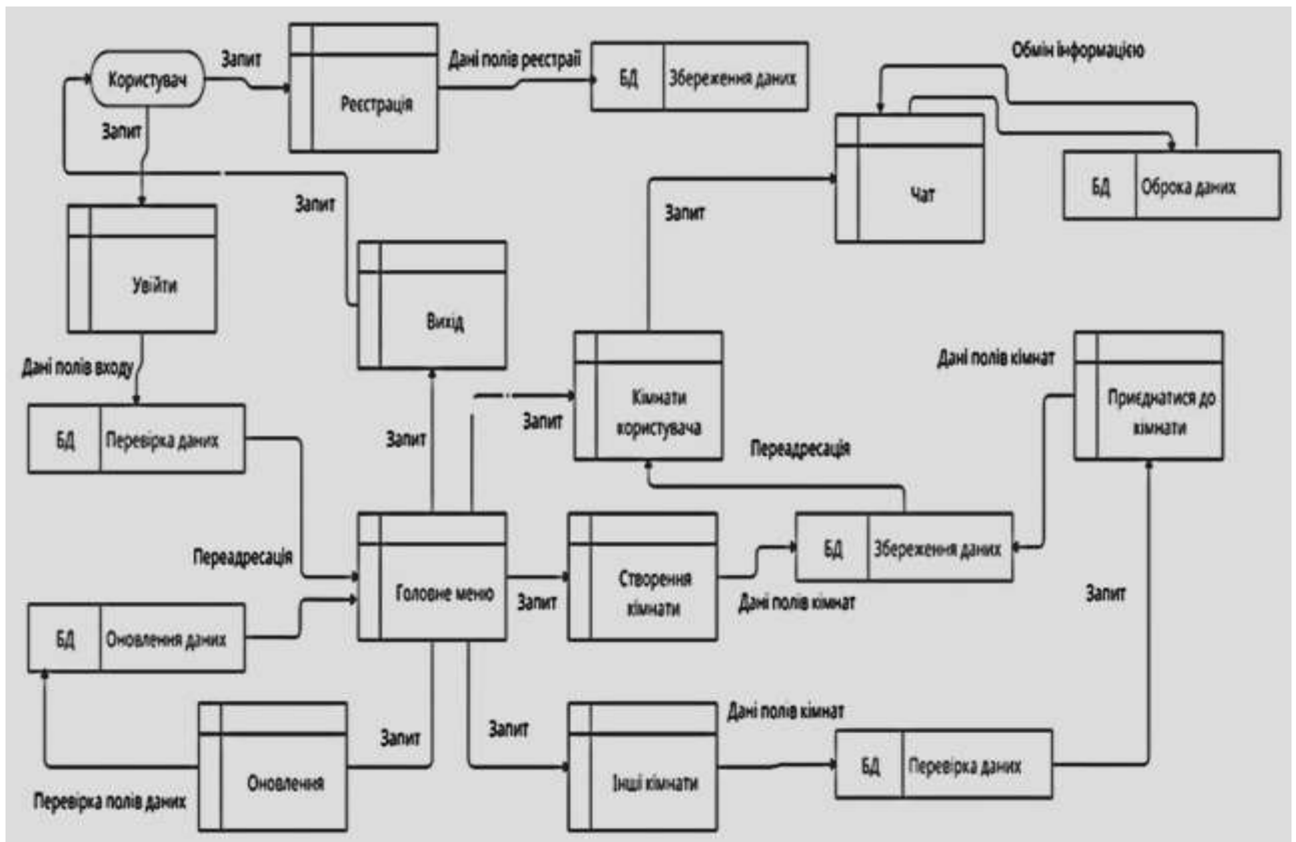

Рисунок 1.3 – Діаграма потоків даних.

Отже, план розвитку представляє візуальну ієрархічну структуру схеми потоку даних, що описує асинхронний процес від системи введення до інформації, що передається користувачеві [17].

1.5 Розробка моделей даних

Після завершення розробки та оцінки діаграми потоків даних можна розпочати етап створення системи бізнес-правил для предметної області месенджера. За даними DFD сформовано структуру передачі даних та ідеї виконання завдань майбутніх застосунків, і на основі цього вирішено представити таку систему бізнес-правил:

– кожен користувач містить особисті дані у вигляді унікального ідентифікаційного коду, імені, поштової адреси, паролю;

– відносно кімнат користувачі теж закріплюються на основі ключа id, здійснюючи створення або приєднання до конкретної кімнати, яка містить назву та силки на повідомлення і на учасників;

– можливість надсилання повідомлень здійснюється на основі посилання на власника та його кімнату, яка міститиме id з певним текстом;

– проміжним шаром між користувачем та кімнатою є створений зв'язок багато до багатьох на основі id кімнати та користувача.

Наступним кроком у розробці моделі даних є розробка схеми ER-моделі, яку ще називають «Фізичним зв'язком».

Як подано на рисунку 1.4, модель даних відображає таблиці та рядки бази даних, а також взаємозв'язки та залежності у графічній формі, з'єднаній рядками. Для великих баз даних побудова моделі ER дозволяє уникнути помилок проектування, забезпечуючи тим самим продуктивність під час тестування та експлуатації.

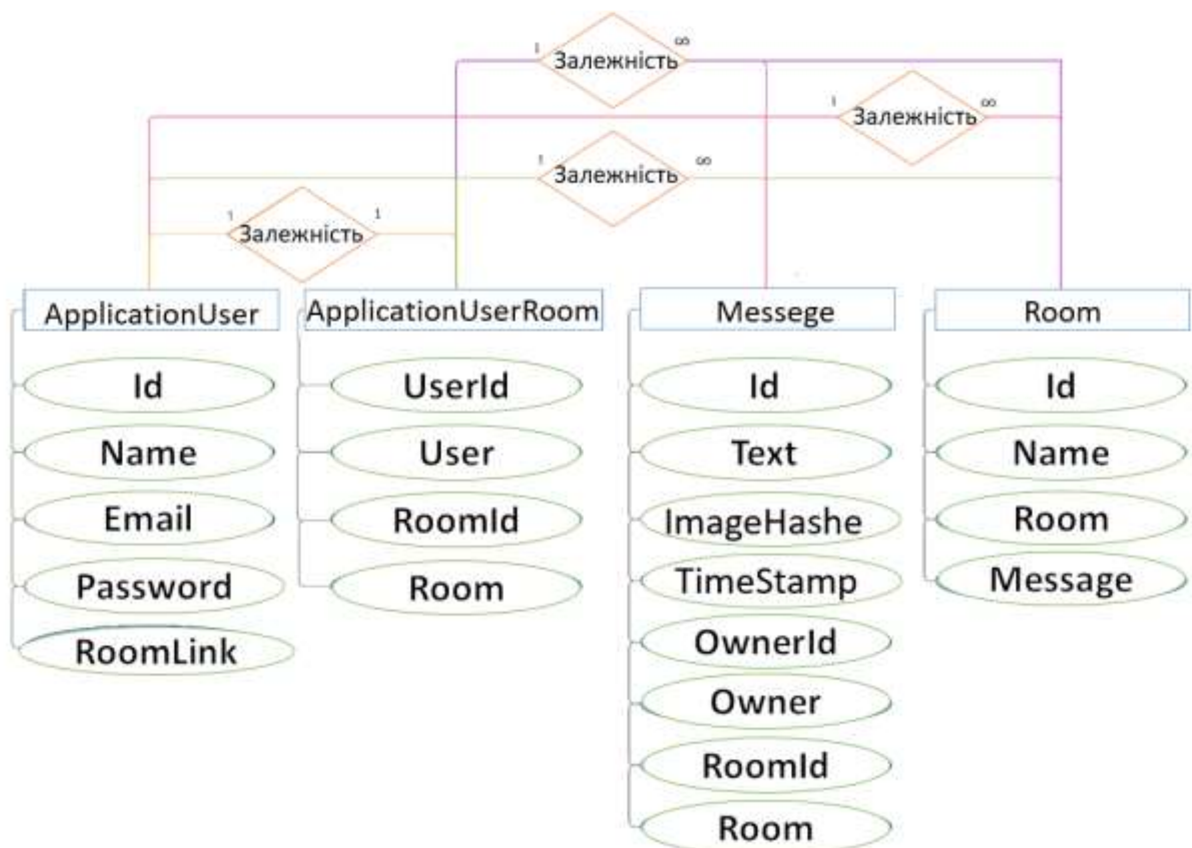

Рисунок 1.4 – Діаграма ER-моделі даних.

Основна мета цієї моделі – спростити процес проектування бази даних. Усі типи баз даних базуються на моделі ER: реляційні, ієрархічні, мережеві та об’єктні [18].

Після графічної реалізації структури бази даних відбувається етап створення логічної та фізичної моделей даних. Логічні моделі даних використовуються в ситуаціях, коли дані та взаємозв'язки повинні бути детально описані на дуже високому рівні [19]. Це не включає фізичне представлення даних у базі даних, але описано на дуже абстрактному рівні. В основному він включає сутності та взаємозв'язки між ними, а також атрибути кожної сутності.

Логічна модель даних включає первинний та зовнішній ключ кожної сутності, створюючи логічну модель даних. Ключі використовуються для ідентифікації першої сутності та її взаємозв’язків [20]. Потім визначаються атрибути кожної сутності. Логічна модель даних не залежить від системи управління базами даних, оскільки вона не описує фізичну структуру реальної бази даних. При розробці логічної моделі даних, яка представлена на рисунку 1.5, можна використовувати неформальні довгі імена для сутностей та атрибутів.

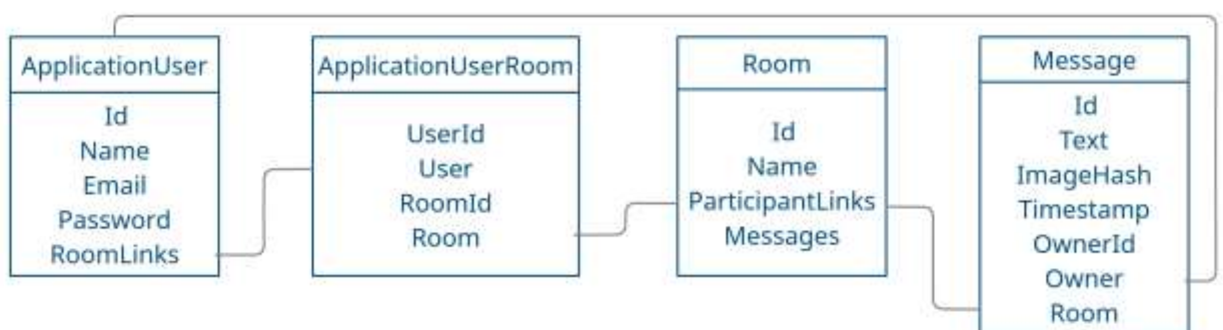

Рисунок 1.5 – Логічна модель.

Фізична модель даних описує фактичне зберігання інформації в базі даних. Вона включає специфікації для всіх таблиць і стовпців у ній. Фізична модель даних також містить первинний ключ кожної таблиці, а також

використовує зовнішні ключі, щоб показати взаємозв'язки між таблицями [21]. Крім того, фізична модель даних містить обмеження, що застосовуються до даних та компонентів, таких як тригери та збережені процедури, котрі подані на рисунку 1.6.

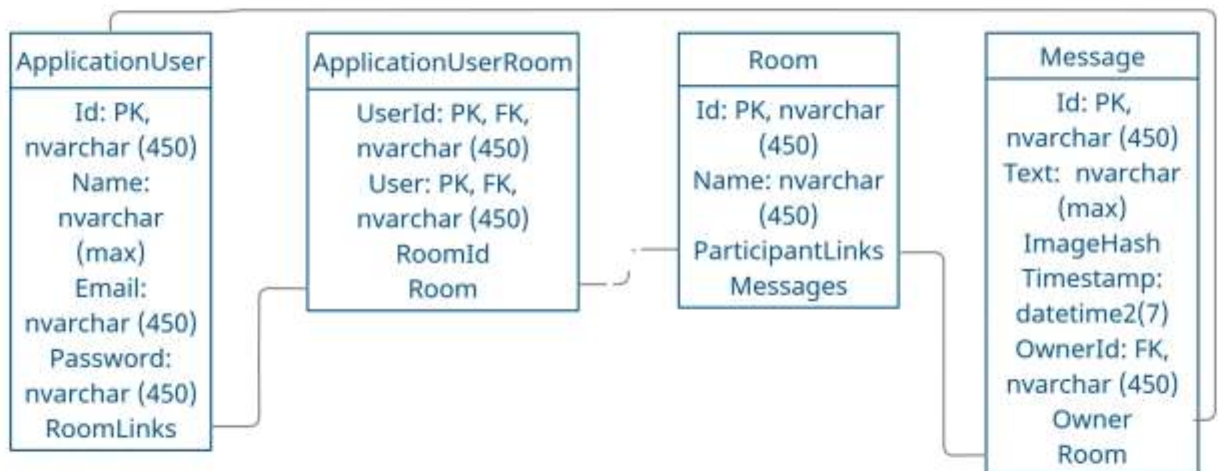

Рисунок 1.6 – Фізична модель

В кінцевому результаті було отримано необхідний набір моделей для програми обміну текстовими повідомленнями.

1.6 Висновок до першого розділу

В першому розділі кваліфікаційної роботи було поставлено ряд завдань, котрі будуть реалізовані шляхом використання розглянутих технологій, виділивши основні концепції розроблюваної програми. Одна з таких – клієнт-серверна архітектура та інші технології, на основі яких буде розроблена програма для обміну текстовими повідомленнями використовуючи платформу .Net.

Після того здійснюється моделювання архітектури програмних частин проекту. Тут не обійтись без розробки моделей даних, які надають можливість в подальшому здійснювати розробку програми для обміну текстовими повідомленнями на основі платформи .Net.

РОЗДІЛ 2. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО РІШЕННЯ

2.1 Проектування інтерфейсу програмного забезпечення

Розробка інтерфейсу програми для обміну текстовими повідомленнями на основі платформи .Net починається з побудови клієнт-серверної архітектури. Для початку опишемо роботу сервера:

– відбувається ініціалізація конфігурації Dependency Injection (DI) модуль;

– реєстрація handler для запитів;

– створення pipe для максимальної кількості підключень на сервер.

Оскільки це тестова бета версія на максимум два потоки, максимальна кількість буде рівна двом, подано в лістингу 2.1.

Лістинг 2.1 – Ініціалізація програми

```
public class Program
    {
        public static ApplicationStarter Application { get;
set; }

        public static void Main(string[] args)
        {
            Console.WriteLine("Initialization DI
container!");
            Console.WriteLine();

            IServiceCollection services =
ConfigureServices(new ServiceCollection());

            Console.WriteLine("DI container successfully
configured!");
```

Після з’єднання з базою даних, здійснюється підключення Listeners до кінцевої точки, присвоюється назва контролерів та pipe. Після цього відбувається реєстрація імен. Далі відбувається завантаження ядра, яке буде

чекати відповідний запит на сервері за попередньо визначеними кінцевими точками.

Запускаємо серверну частину програми – реалізовуємо host, який буде приймати запити для виклику ядра (Див. лістинг 2.2).

Лістинг 2.2 Підключення Listeners

```
     private static IIpcServiceHost
SetupListeners(IServiceCollection services)
             {
                 var host = new
IpcServiceHostBuilder(services.BuildServiceProvider())

.AddNamedPipeEndpoint<IMessageServiceHandler>(name:
"messageEndpoint", pipeName: "message")

.AddNamedPipeEndpoint<IRoomServiceHandler>(name: "roomEndpoint",
pipeName: "room")

.AddNamedPipeEndpoint<IApplicationUserServiceHandler>(name:
"applicationUserEndpoint", pipeName: "applicationUser")

.AddNamedPipeEndpoint<IAccountServiceHandler>(name:
"accountEndpoint", pipeName: "account")
                                 .Build();
                 return host;
             }
```

У якості ядра програми використовується клас ApplicationStarter, в середині якого:

- викликаються всі зв’язки бази даних;
- встановлюється Container для основи DI;
- викликається екземпляр класу БД;
- додається MessangerDbContext;
- перевіряється наявніть таблиць в базі даних;
- якщо таблиці в БД відсутні тоді накладається міграція.

Тобто перед запуском бази даних, програмі не обов’язково, щоб БД існувала. При необхідності, її буде створено автоматично, код подано в лістингу 2.3.

Лістинг 2.3 – Ядро програми

```
public class ApplicationStarter
    {
        private static ApplicationStarter Instance;

        private ApplicationStarter()
        {
            Start();
        }

        public static InjectionContainer InjectionContainer {
get; set; }

        private void Start()
        {
            Console.WriteLine("Initialize container!");

            InjectionContainer =
Activator.CreateInstance<InjectionContainer>();

            InjectionContainer.ConfigureServices();

            Console.WriteLine("Migrate database!");
```

Для створення об’єктів контролера використовується метод Stop для опрацювання запитів. Код програми подано в лістингу 2.4.

Лістинг коду 2.4 – Методи об’єкту контролера

```
        public void Stop()
        {
            InjectionContainer = null;
            Console.WriteLine("Handler succssesfully
stoped!");
        }

        public T ResolveController<T>() where T :
ControllerBase
        {
            var container =
InjectionContainer.GetContainer();
            using (var scope =
container.BeginLifetimeScope())
            {
                return scope.Resolve<T>();
            }
        }
```

Підключення між клієнтом та сервером здійснюється за допомогою SoketServer який призначений для швидкого обміну повідомленями. Після запуску встановлюється Listener та очікується вхідний запит (Див. лістинг 2.5).

Лістинг 2.5 – Реалізація SoketServer

```
public class Program
   {
        private static Listener _listener;

        private static List<Client> _sockets;

        public static void Main()
        {
            _sockets = new List<Client>();
            Console.WriteLine("Initialize listeners!");
            _listener = new Listener(8);
            _listener.SocketAccepted +=
_listener_SocketAccepted;

            _listener.Start();
            Console.WriteLine("Start listener!");

            Console.Read();
```

Вказується специфічний сокет, додавши список сокет-клієнтів. Після отримання списку клієнтів, повертається запит, що вже приєднано до такого то сокет-сервера і очікуєте на вхідний запит, подано в лістингу 2.6.

Лістинг 2.6 – Приєднання до сокет-сервера

```
              private  static  void  _listener_SocketAccepted(Socket
socket)
              {
                  Console.ForegroundColor = ConsoleColor.Green;
                  Console.WriteLine($"New                connection
{socket.RemoteEndPoint}{Environment.NewLine}{DateTime.Now}{Environ
ment.NewLine}===========");
                  Console.ForegroundColor = ConsoleColor.Gray;

                  var client = new Client(socket);
                  client.Received += Client_Received;
                  client.Disconnected += Client_Disconnected;
```

```
                _sockets.Add(client);
            }
```

На клієнтській чатині RoomForm вказуємо адресу та конфігураційні властивості. Підключення відбувається на локальному хості. Спочатку отримуємо доступ на вхідний запит, який спрацьовує після приєднання, подано в лістингу 2.7.

Лістинг 2.7 – Реалізація клієнтської частини RoomForm

```
namespace Messanger
{
    public partial class RoomForm : Form
    {
        private Room _room { get; set; }
        private Socket _socket;
        private static int _bufferSize = 8 * 1024;
        private byte[] _buffer = new byte[_bufferSize];
        private ApplicationUser _user;
        public RoomForm()
        {
            InitializeComponent();
            this.DesignViews();
            this.Load += RoomForm_Load;
            textBox1.KeyDown += TextBox1_KeyDown;
        }
        public void SetRoom(Room room)
        {
            _room = room;
            _user =
AccountApi.GetInstance().GetCurrentUser();
            this.Text = _room.Name;
            _socket = new Socket(AddressFamily.InterNetwork,
SocketType.Stream, ProtocolType.Tcp);

            _socket.BeginConnect("127.0.0.1", 8,
ConnectCallback, null);
        }
```

Далі отримуємо повідомлення і вказуємо, що готові приймати наступні. Отримуємо повідомлення з сокета конвертуючи його в текстовий тип даних JSON та додаємо в чат і знову вказуємо, що готові очікувати наступне повідомлення (Див. лістинг 2.8).

Лістинг 2.8 – Конвертування повідомлення

```
private void ConnectCallback(IAsyncResult AR)
        {
            try
            {
                _socket.EndConnect(AR);
                _socket.BeginReceive(_buffer, 0, _buffer.Length, 
SocketFlags.None, ReceiveCallback, null);
            }
            catch (SocketException ex)
            {
                MessageBox.Show(ex.Message);
            }
            catch (ObjectDisposedException ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private void ReceiveCallback(IAsyncResult AR)
        {
            try
            {
                int received = _socket.EndReceive(AR);

                if (received == 0)
                {
                    return;
                  }
```

Реалізація клієнтської частини подана в класі Client де виконується ряд наступних інструкцій:

– присвоєння сокету Id;

– виклик методу MessageController для збереження повідомлень;

– виконується дишифрування повідомлень;

– очікування наступних повідомлень;

– реєстрація на сокет;

– виклик методу Received для відправки повідомлення збереженим сокетам.

Кожен сокет має знати кімнату на якій працює за відповідною roomId. За допомогою Client_Received здійснюється цикл відправки на сокети з вказаною Id. Так працює сокет-клієнт (Див. лістинг 2.9).

Лістинг 2.9 – Клієнтської частини програми

```
    public class Client
    {
        public string Id { get; private set; }

        public string RoomId { get; private set; }
        private readonly MessageController
_messageController;

        public IPEndPoint EndPoint { get; private set; }

        private readonly Socket _socket;

        private static int _messageSize => 8 * 1024;

        private byte[] _dump = new byte[_messageSize];

        public Client(Socket socket)
        {
            _socket = socket;

            Id = Guid.NewGuid().ToString();
            _messageController =
ApplicationStarter.GetInstance().ResolveController<MessageControll
er>();
            EndPoint = (IPEndPoint)socket.RemoteEndPoint;

            _socket.BeginReceive(_dump, 0, _dump.Length,
SocketFlags.None, ReceiveCallback, null);
        }
```

Реалізація зв'язків в сервері розроблена наступним чином. Спочатку в сервері очікується запит, при отриманні опрацьовується. Якщо подати запит на аутентифікацію, викликається клас AccountServiceHandler далі викликається метод Login. Робиться об'єкт _accountController, виконується запит перевірки моделі до _accountService. Цей метод в якому перевіряється чи існує користувач, якщо так, тоді паролі збігаються далі повертаємося на клієнта користувача, подано в лістингу 2.10.

Лістинг 2.10 – Перевірка користувача

```
public class AccountServiceHandler : IAccountServiceHandler
    {
        private readonly AccountController
_accountController;
```

```
            public AccountServiceHandler()
            {
                _accountController =
Program.Application.ResolveController<AccountController>();
            }

            public string Login(LoginModel model) =>
                _accountController?.Login(model);

            public string Register(RegisterModel model) =>
                _accountController?.Register(model);
        }
    }
```

За допомогою метода ApplicationUserService надається доступ до основних методів цієї програми, таких як: Get, GetByEmail, Create, Update, Delete який окремо на пряму, працює з репозиторієм. Маємо абстракцію репозиторія який працює для всіх Entity де можна отримати всі запити з БД (Див. лістинг 2.11).

Лістинг 2.11 – Реалізація методів програми

```
        public class ApplicationUserService :
IApplicationUserService
        {
            private readonly IApplicationUserRepository
_applicationUserRepository;

            public
ApplicationUserService(IApplicationUserRepository
applicationUserRepository)
            {
                _applicationUserRepository =
applicationUserRepository;
            }

            public ApplicationUser Get(string id)
            {
                return _applicationUserRepository.FindOne(x =>
x.Id == id);
            }

            public ApplicationUser GetByEmail(string email)
            {
                return _applicationUserRepository.FindOne(x =>
x.Email == email);
            }
```

```
            public ApplicationUser Create(ApplicationUser entity)
            {
                return
_applicationUserRepository.InsertOne(entity);
            }
```

Розробка клієнтської частини програми аунтентифікації реалізована в класі LoginForm. За допомогою методів зчитуються дані які потрібні вказати, завдяки класу AccountApi який відповідає за з'єднання клієнта з сервером. Вказується хендлер за такою кінцевою точкою. Якщо звернути увагу на сервіс, там реєструються кінцеві точки завдяки яким вже відомо куди приєднатись. Це працює за допомогою TCP з'єднання тільки іншим шляхом, подано в лістингу 2.12.

Лістинг 2.12 – Реалізації аунтентифікації

```
    public partial class LoginForm : Form
        {
            public LoginForm()
            {
                InitializeComponent();
                this.DesignViews();
            }

            private async void btnLogin_Click(object sender,
EventArgs e)
            {
                var result = await
AccountApi.GetInstance().Login(txtEmail.Text, txtPassword.Text);
                if (result == null) return;

                var form = new MainForm();
                form.Show();
                this.Hide();
            }

            private void btnRegister_Click(object sender,
EventArgs e)
            {
                var form = new RegisterForm();
                form.Show();
                this.Hide();
            }
        }
    }
```

З'єднання клієнта з сервером реалізований шляхом використання класу AccountApi та Pattern Singleton де реалізована логіка, спочатку здійснюється звернення до _accountClient надсилаючи метод Login, в свою чергу отримуючи відповідний хендлер (Див. лістинг 2.13).

Лістинг 2.13 – З'єднання клієнта з сервером

```
    public class AccountApi
    {
        private static AccountApi _instance = null;
        private readonly
IpcServiceClient<IAccountServiceHandler> _accountClient;

        private readonly
IpcServiceClient<IApplicationUserServiceHandler> _userClient;

        private ApplicationUser _currentUser;

        private AccountApi()
        {
            _accountClient = new
IpcServiceClientBuilder<IAccountServiceHandler>()

.UseNamedPipe("account")
                                                                .Build();

            _userClient = new
IpcServiceClientBuilder<IApplicationUserServiceHandler>()

.UseNamedPipe("account")
                                                                .Build();
        }
```

Дані пересилаються всі за допомогою JSON оскільки це один із найпопулярніших способів передачі даних між Ip [22].

2.2 Реалізація інтерфейсу бази даних

Завдяки класу MessangerDbContext здійснюється під'єднання до БД вказавши адресу використовуючи SQL Server. Цей клас є контекстом бази даних який реалізованих за принципом згенерованих таблиць. Побудовані на

основі DbSet яка є обгорткою над таблицями типізованими Entity та маючи збережену конфігурацію яка подана в лістингу 2.14.

Лістинг 2.14 – Ініціалізація БД

```
 public class MessangerDbContext : DbContext

    {
        public DbSet<ApplicationUser> Users { get; set; }
        public DbSet<Room> Rooms { get; set; }
        public DbSet<Message> Messages { get; set; }

        protected override void
OnConfiguring(DbContextOptionsBuilder optionsBuilder)
        {
            optionsBuilder.UseSqlServer(@"Server=DESKTOP-
8G2ITVR;Database=messanger;Trusted_Connection=True;");
            optionsBuilder.EnableSensitiveDataLogging();
            optionsBuilder.EnableDetailedErrors();
        }

        protected override void OnModelCreating(ModelBuilder
modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            if (modelBuilder == null) return;

            modelBuilder.ApplyConfiguration(new
ApplicationUserEntityConfiguration());
            modelBuilder.ApplyConfiguration(new
RoomEntityConfiguration());
            modelBuilder.ApplyConfiguration(new
ApplicationUserRoomEntityConfiguration());
            modelBuilder.ApplyConfiguration(new
MessageEntityConfiguration());
        }
    }
}
```

На основі цього було створено поточну міграцію – згенерований автоматично саме в Entity framework код який при викликанні команди Migrate конвертується в SQL код, тоді виконується на рівні бази даних. Це відбувається у класі ApplicationStarter який подано в додатку А3.

Шляхом виконання вище згаданих маніпуляцій згенеровано таблицю Migrate, яка містить типи даних id міграції, де вказано інформацію про створення та версії яку містить (Див. рисунок 2.1).

DESKTOP-8G2ITVR....plicationUserRoom | DESKTOP-8G2ITVR.m...Migr

Имя столбца	Тип данных	Разрешить ...
MigrationId	nvarchar(150)	☐
ProductVersion	nvarchar(32)	☐

Рисунок 2.1 – Типи даних Migrate

Розглянемо метод конфігурації таблиць який містить об’єкт builder. Типізуючи подану таблицю до того типу даних який унаслідується від IEntity який перевіряє дані.

Таким чином за цей тип даних відповідає клас ApplicationUsers. Створюється інтерфейс IEntity з відповідними параметрами, які були згадані раніше та додається ключ який буде створюватись на рівні бази даних. Вказавши що користувач має кімнати, тобто йому надається доступ до них який подано в додатку Б1.

На основі поданих інструкцій для реалізації таблиці ApplicationUser подано на рисунку 2.2, яка згенерована відповідно вказаним особливостям виглядає наступним чином.

DESKTOP-8G2ITVR.m....application_users

Имя столбца	Тип данных	Разрешить ...
Id	nvarchar(450)	☐
name	nvarchar(MAX)	☐
email	nvarchar(MAX)	☐
password	nvarchar(MAX)	☐

Рисунок 2.2 – Згенеровано у БД

Тому під час авторизації або аунтентифікації користувач без проблем зможе виконати дані операції, вказавши необхідні параметри. Інтерфейс котрий відповідає за ці маніпуляції, поданий на рисунках 2.3 та 2.4.

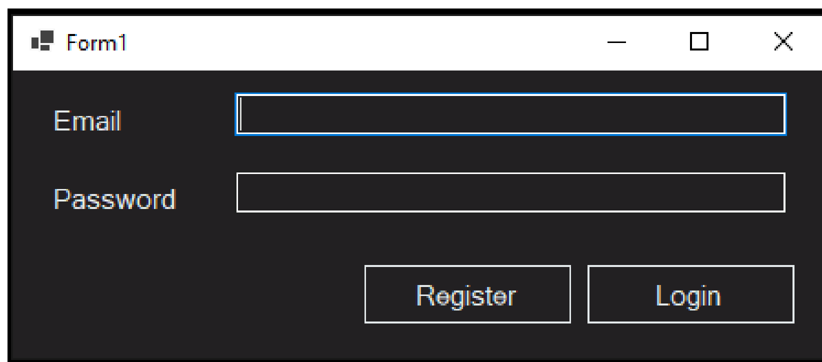

Рисунок 2.3 – Форма аунтентифікації

В результаті було спроектовано інтерфейс ApplicationUser для форм авторизації та аунтентифікації.

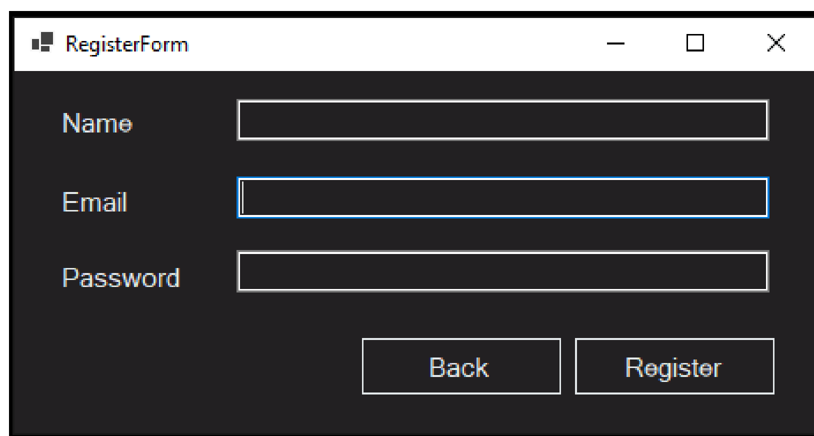

Рисунок 2.4 – Форма авторизації

Далі перейдемо до класу Room який теж містить згенерований id з відповідними параметрами які були раніше згадані, спочатку здійснюється виконання коду у середовищі Visual Studio.

Вказавши тип даних, переходимо до створення безпосередньо до самої таблиці Room. Надавши залежність, що ця кімната має багато повідомлень з

іншою кімнатою, тобто одне повідомлення має тільки одну кімнату, відповідно зв'язок який використовується є один до багатьох який подано в додатку Б2.

В результаті поданих інструкцій реалізується таблиця яка містить вказані параметри згенеровані за допомогою команди Migrate, подано на рисунку 2.5.

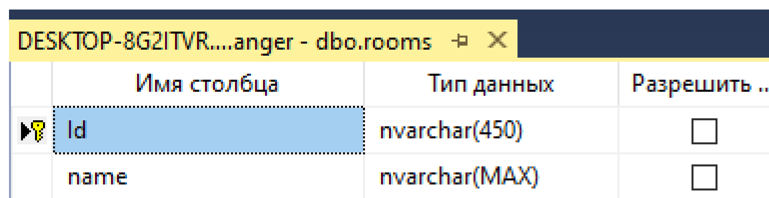

Рисунок 2.5 – Реалізовано у БД

Інтерфейс котрий відповідає даній області тобто кімнат подано на рисунку 2.6. З можливістю створення нової кімнати та відображення власних та інших кімнат.

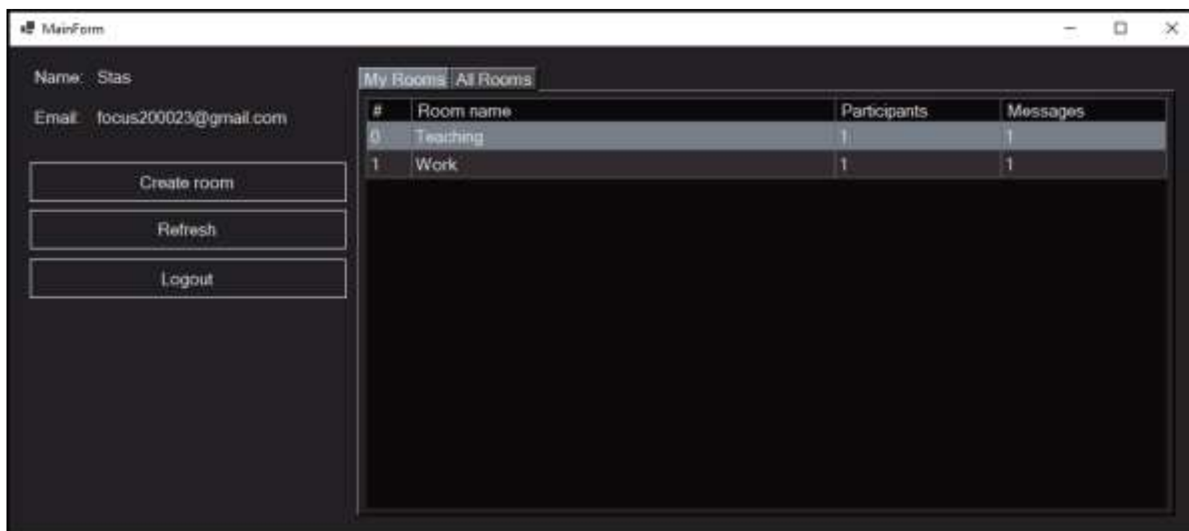

Рисунок 2.6 – Головне меню

На основі раніше побудованої таблиці, а саме Room надається можливість використання повідомлень для комунікації між користувачами. Реалізується дана можливість за допомогою класу Message. Надається посилання на власника повідомлення та на кімнату (Див. додаток Б3).

Отже в результаті вище згаданих інструкцій здійснюєтся реалізація таблиці з відповідними параметрами на рівні бази даних (див. рисунок 2.7).

DESKTOP-8G2ITVR....ger - dbo.message DESKTOP-8G2ITVR.m...ger -

Имя столбца	Тип данных	Разрешить ...
Id	nvarchar(450)	☐
text	nvarchar(MAX)	☑
ImageHash	nvarchar(MAX)	☑
Timestamp	datetime2(7)	☐
OwnerId	nvarchar(450)	☑
RoomId	nvarchar(450)	☑
RoomId1	nvarchar(450)	☑
		☐

Рисунок 2.7 – Message згенерований на рівні БД

За допомогою класу RoomEntityConfiguration котрий є проміжним шаром між користувачами та кімнатами, який має одного користувача, а користувач може мати багато інших кімнат, яка може містити багато користувачів. В результаті використовується зв'язок багато до багатьох подано в лістингу 2.15.

Лістинг 2.15 – Створеня кімнати

```
          public class RoomEntityConfiguration :
IEntityTypeConfiguration<Room>
          {
              public void Configure(EntityTypeBuilder<Room>
builder)
              {
                  if (builder == null) return;
                  builder
                      .ToTable("rooms");
                  builder
                      .HasKey(o => o.Id);
                  builder
                      .Property<string>("Name")
                          .HasColumnName("name")
                          .IsRequired();
                  builder
                      .HasMany<Message>()
                      .WithOne(x => x.Room)
                      .HasForeignKey(x => x.RoomId)
```

```
                    .OnDelete(DeleteBehavior.Cascade);
            }
        }
    }
```

В результаті отримуємо згенеровану таблицю з відповідними типами даних, які дозволяють реалізувати вище згадані можливості взаємодії користувача та кімнати, яке створене в середовищі SQL Server подано на рисунку 2.8.

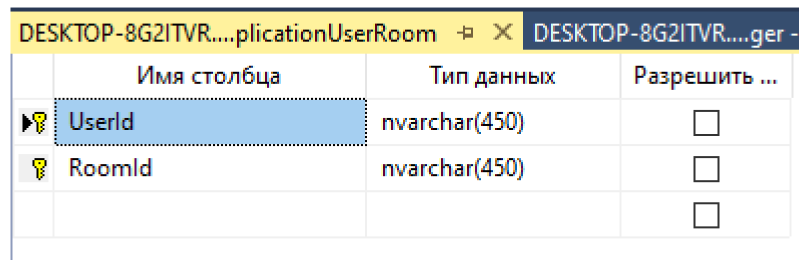

Рисунок 2.8 – Реалізована таблиця у БД

Важливим елементом є інтерфейс котрий надає користувачеві можливість взаємодії з іншими користувачами, а саме комунікація за допомогою чату. Надсилаючи повідомлення в конкретну кімнату з іншими учасниками подано на рисунку 2.9.

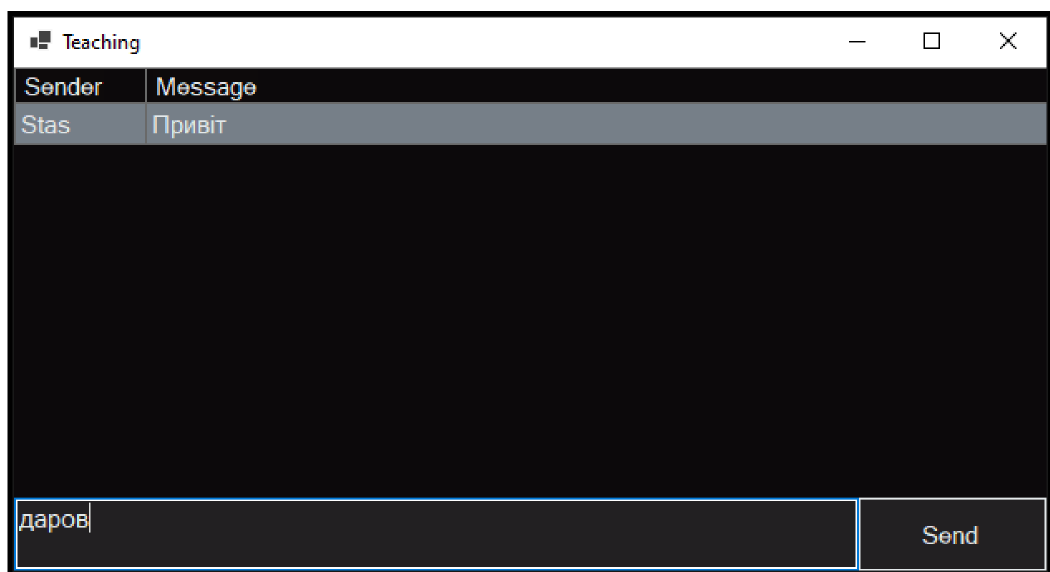

Рисунок 2.9 – Вікно чату

На даний момент було реалізовано можливість обміну текстовими повідомленнями між користувачами системи.

2.3 Проектування методів для операцій з базою даних

Наступним етапом є створення класу DomainException в якому описані поточні методи для операції з базами даних, використовуючи певну функцію для своїх цілей подано в лістингу 2.16.

Лістинг 2.16 – Виклик методів для операції БД

```
 public class DomainException : Exception
     {
          public string Error { get; set; }
          public DomainException(string error)
          {
              this.Error = error;
          }

          public DomainException()
          {
          }

          public DomainException(string message, Exception
innerException) : base(message, innerException)
     }
```

У якості обгортки використовується клас Repository над цим джерелом це є dbContext який являється контекстом бази даних. Repository містить узагальнений тип, який може працювати з різними типами даних, котрі належать IEntity. Потім через MessangerDbContext отримується доступ до функцій які подано в додатку В1.

Для повної функціональності бази даних додано можливість обчислення значень на основі поданих даних за допомогою функції COUNT.

Щоб здійснити видалення певних об’єктів вказані інструкції які повинна дотримуватись база даних. Виконання цих інструкцій відбувається через функцію DELETE, яку подано в додатку В1.

Не варто забувати про таку можливість як пошук конкретних об'єктів у розмірі більше ніж одного, шляхом виконання поданої умови функцією FIND_MANY для БД (Див. додаток В2).

Вставлення певних даних здійснюється при використанні функції INSERT відповідно до вказаних параметрів подано в додатку В2.

Якщо вносяться певні зміни у базі даних використовується функція UPDATE для оновлення даних у зв'язку з конкретними змінами котрі відбулись Отже опираючись на вище згадане, база даних надає доступ до функцій котрі розширюють її функціональні можливості з даними які отримує, подано в додатку В3.

2.4 Тестування та встановлення програмного забезпечення

Під час початкового завантаження застосунку здійснюється підготовка основних елементів програми для обміну текстовими повідомленнями в єдину цілісну систему на основі котрої буде відбуватися виконання поставлених задач користувачем (див. рисунок 2.10).

Рисунок 2.10– Ініціалізація компонентів

Початковим кроком для використання цієї програми є здійснення авторизації вказавши необхідні параметри. Як бачимо на наступному рисунку ініціалізацію того, що було створено профіль який можна використовувати в по дальшому, подано на рисунку 2.11.

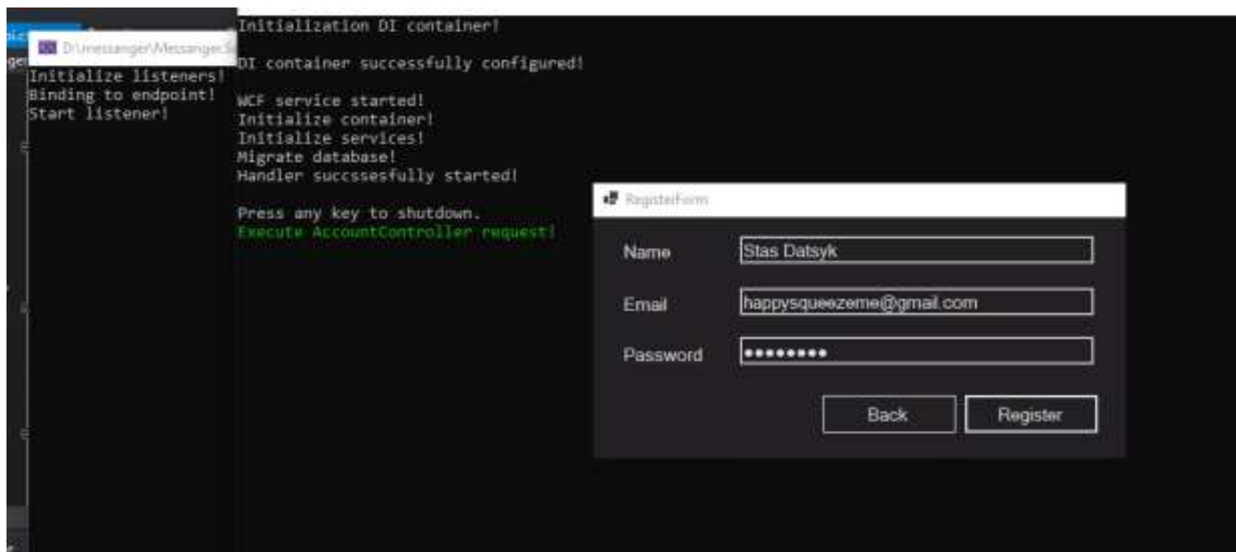

Рисунок 2.11 – Авторизація акаунту

Після виконання попереднього пункту вказуємо у формі аунтентифікації раніше вказаної електроної адреси та паролю, підтвердивши кнопкою «Login» (див. риунок. 2.12).

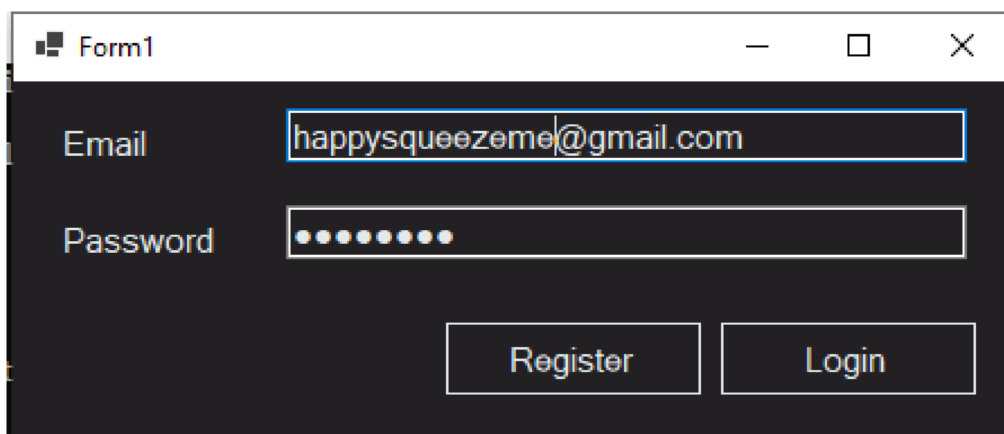

Рисунок 2.12– Авторизація профілю

Далі нас зустрічає головне меню в котрому користувач отримує повний функціонал який надається цим застосунком. Форма яка відображає дії контролера, відповідає за доступ до кімнати отримавши дозвіл на вхід, подано на рисунку 2.13.

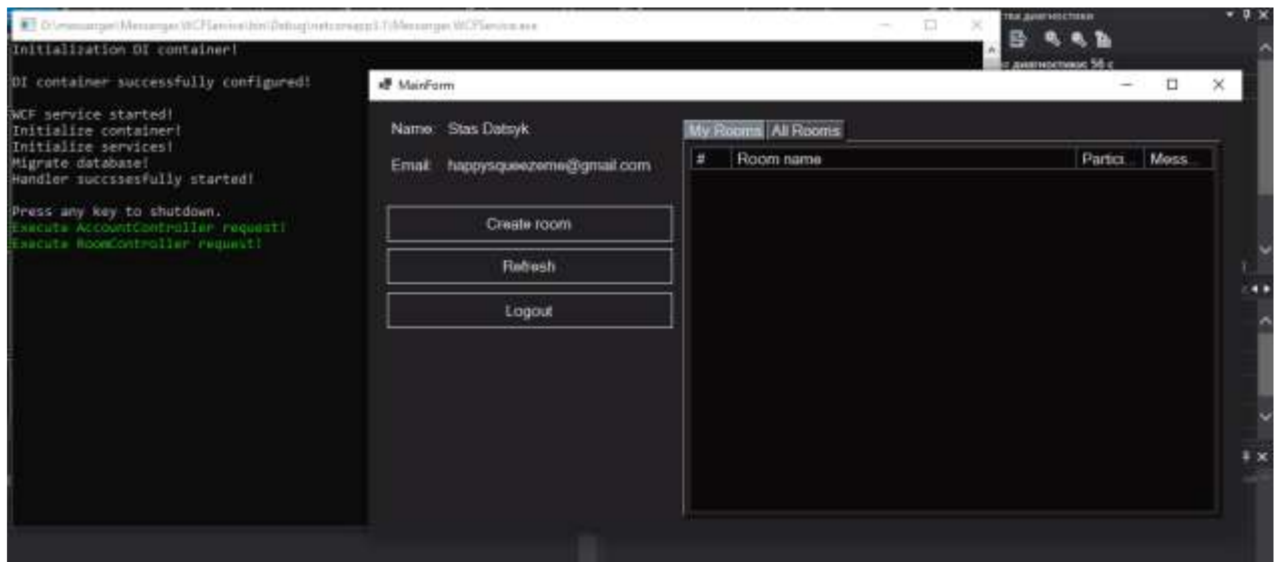

Рисунок 2.13 – Головне меню

Створення кімнат здійснюється через вікно в якому необхідно вказати назву та додати кнопкою «Add» (див. рисунок 2.14).

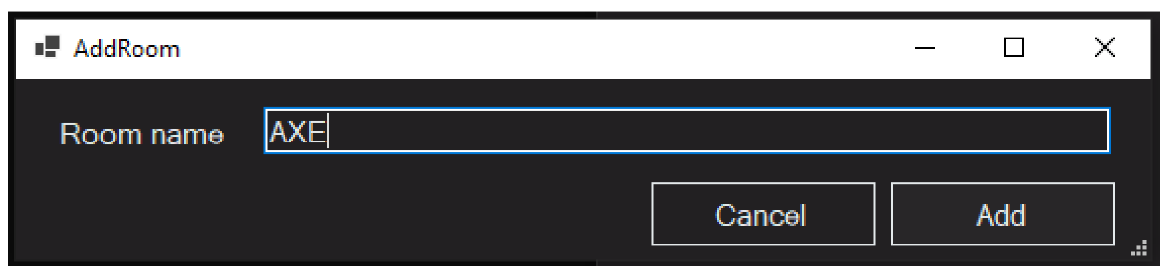

Рисунок 2.14 – Вікно створення кімнати

Отже, раніше створена кімната відображається у поточному списку кімнат. Підтвердженням того є виконання контролера котрий відповідає за дану маніпуляцію. Створення кімнати пройшло успішно, тому продовжуємо рухатись далі, подано на рисунку 2.15.

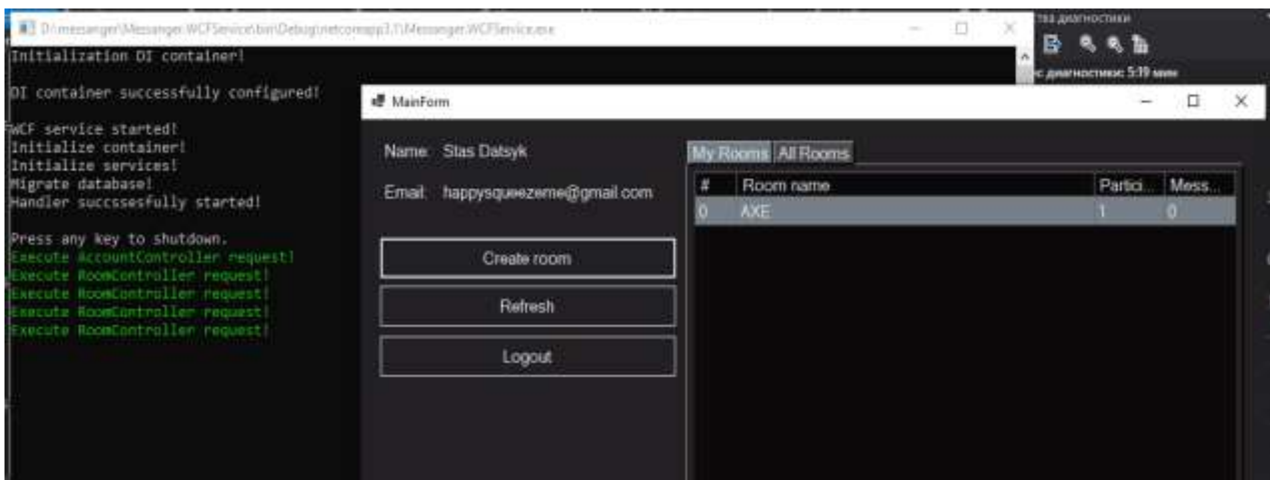

Рисунок 2.15– Власні кімнати

Іншим способом поповненням власного списку кімнат є пошуку інших кімнат користувачів, щоб приєднатись до них потрібно натиснути на кнопку «Join» (див. рисунок 2.16).

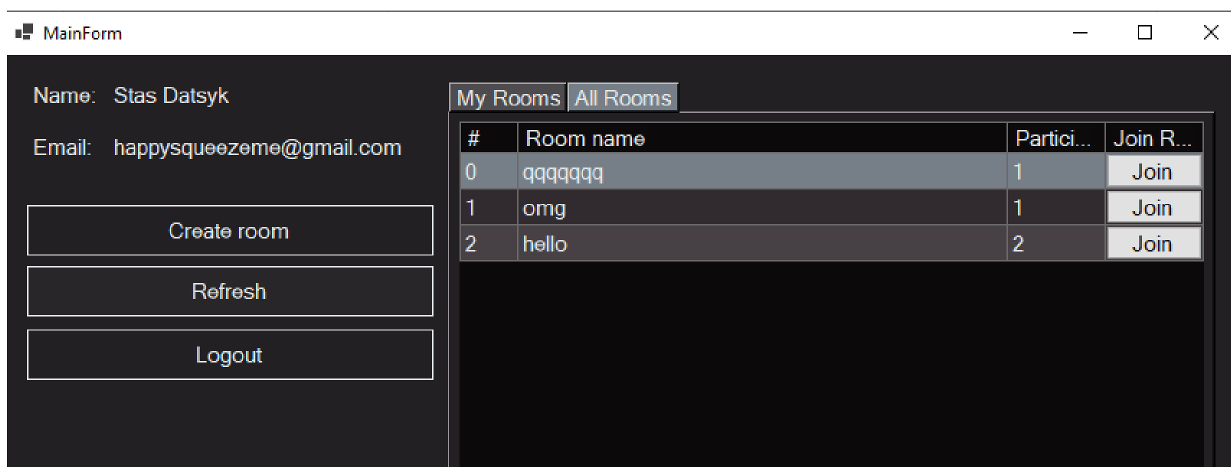

Рисунок 2.16 – Кімнати інших користувачів

В результаті отримуємо, що вибрали виконавши попередні інструкції, подано на рисунку 2.17.

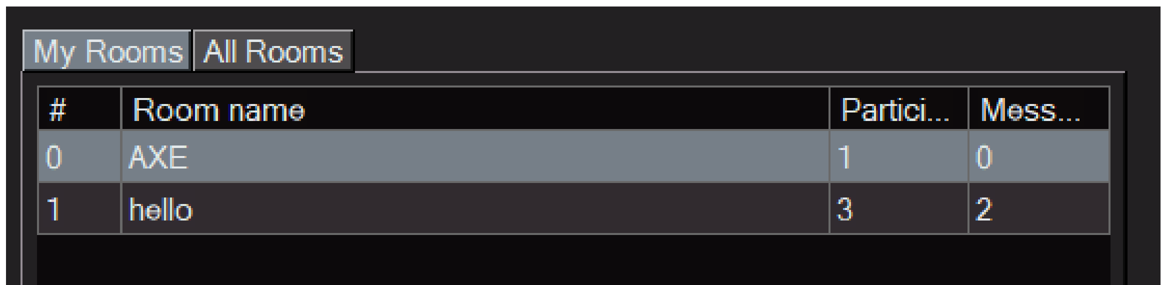

Рисунок 2.17 – Додані кімнати

І так, роль цієї програми це можливіть передачі текстових повідомлень між користувачами. Як бачимо повідомлення успішно надсилаються між учасниками кімнати та ініціалізацію локального серверу через котрий все це відбуватиметься (див. рисунок 2.18).

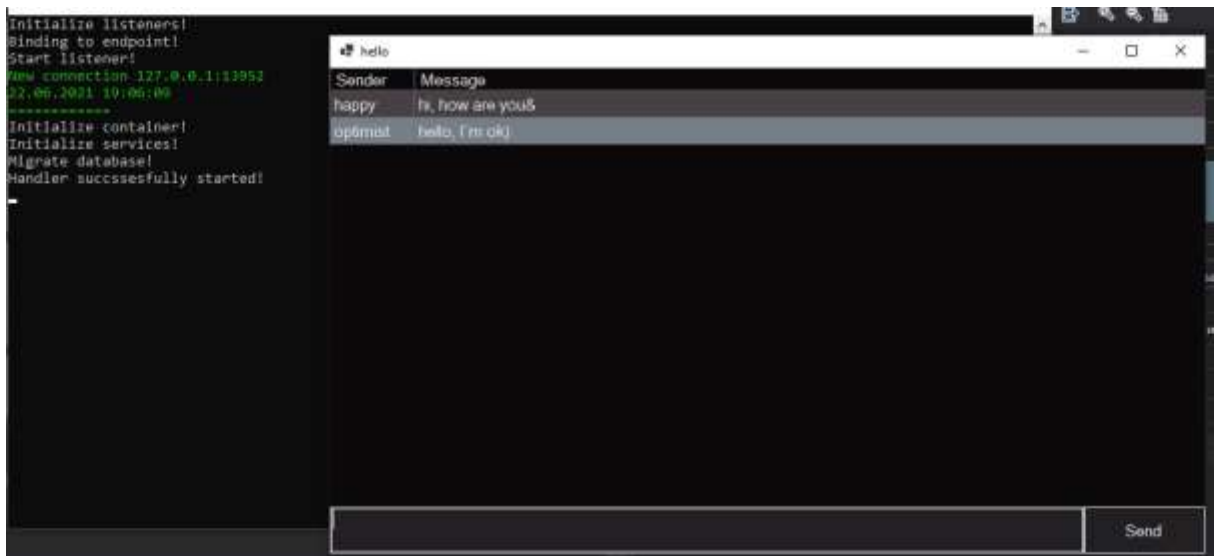

Рисунок 2.18 – Чат

Продемонструємо процес надіслання повідомлення яке пересилається у форматі JSON – це абсолютно незалежний від мови реалізації формат файлу який використовується для обміну даними [23], подано на рисунку 2.19.

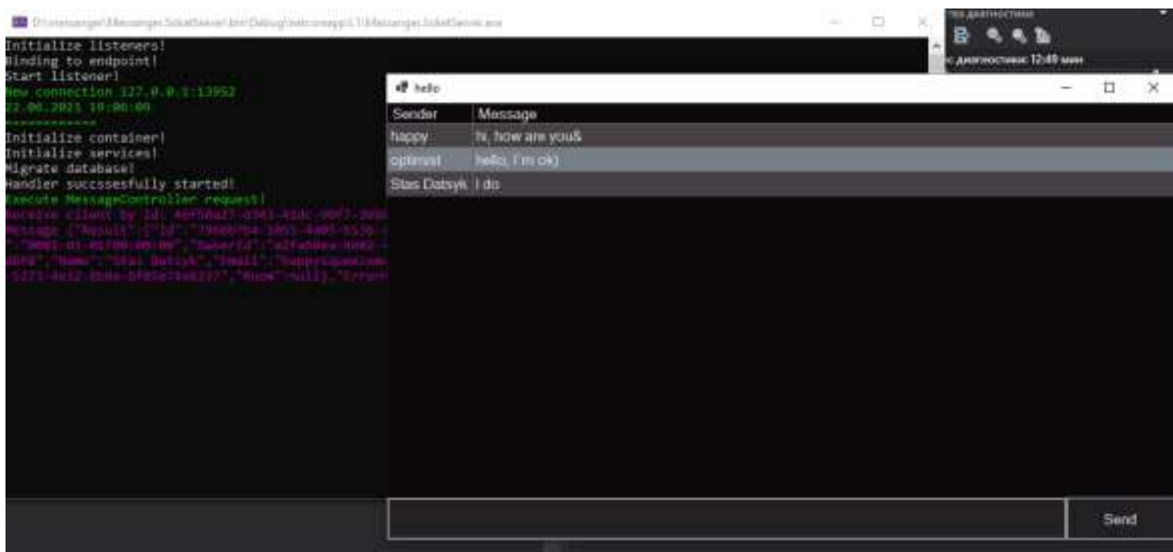

Рисунок 2.19 – Відправлення повідомлення

Після тестування робимо висновок, що дана програма для обміну текстовими повідомленнями на основі платформи .NET відповідає заданим вимогам та функціонує у повній мірі.

Встановлювати програму для обміну текстовими повідомленнями не потрібно, тому що вона являється Portable версією – версія програми для комп'ютера, якій не потрібна установка. Їх ще називають портативними або переносними програмами. Зберігати їх можна на портативному жорсткому диску, USB-флешці, карті пам'яті, і з них же вони і запускаються

2.5 Висновок до другого розділу

В другому розділі кваліфікаційної роботи здійснено проектування та реалізацію проекту на основі клієнт-серверної архітектури. Реалізувавши інтерфейс програми клієнтської та серверної частини призначеної для обміну текстовими повідомленнями на основі платформи Net.

Використовуючи роглянуті в першому розділі технології, які застосовувались для розробки поставленого завдання. В результаті створено програму, яку перевірено на коректність виконання функціональних можливостей.

РОЗДІЛ 3. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

3.1 Соціальні та психологічні фактори ризику

Соціальними та психологічними називаються небезпеки, що широко розповсюджуються в суспільстві і загрожують життю і здоров'ю людей. Носіями соціальних небезпек є люди, що створюють певні соціальні групи. Особливість соціальних небезпек полягає в тому, що вони загрожують великій кількості людей. Розповсюдження соціальних небезпек зумовлено особливостями поведінки людей і окремих соціальних груп. Соціальні небезпеки досить численні. До соціальних належать всі протиправні (незаконні) форми насилля, вживання речовин, що порушують психологічну і фізіологічну рівновагу людини (алкоголь, наркотики, паління), шахрайство, самогубство, та інші дії, що здатні принести шкоду здоров'ю людей.

Соціальні та психологічні небезпеки можуть бути класифіковані за певними ознаками.

За походженням можуть бути виділені такі групи небезпек:

– небезпеки, пов'язані з психічним впливом на людину (шантаж, шахрайство, крадіжки);

– небезпеки, пов'язані з фізичним насильством (розбій, бандитизм, терор, ґвалтування, утримання заручників);

– небезпеки, пов'язані з вживанням речовин, що руйнують організм людини (наркоманія, алкоголізм, паління);

– небезпеки, пов'язані з хворобами (СНІД, венеричні захворювання);

– небезпеки самогубства.

За масштабами подій соціальні небезпеки можна розділити на:

– локальні;

– регіональні;

– глобальні.

Причини соціальних небезпек.

В основі своїй соціальні небезпеки породжуються соціально-економічними процесами, що відбуваються в суспільстві. В той же час треба відзначити суперечливий характер причин, наслідком яких є соціальні небезпеки.

Недосконалість людської природи головна передумова появи соціальних небезпек. Наявність адекватної правової системи може бути основною умовою попередження і захисту від соціальних небезпек. Залучення громадськості та краща інформованість населення про можливість появи соціальних небезпек, а також про можливі наслідки від їх прояву, також є елементом захисту від соціальних небезпек.

Адаптація (пристосування) – це динамічний процес, завдяки якому в організмі підтримується сталість внутрішнього середовища в мінливому зовнішньому середовищі.

Адаптація характеризується:

– розширенням фізіологічних можливостей;
– підвищенням фізіологічної опірності організму зовнішнім впливам;
– збільшенням працездатності;
– зміною порогів чутливості аналізаторів;
– підвищенням стабільності фізіологічних систем;
– переходу фізіологічних систем на більш високі рівні функціонування;
– розширенням діапазону фізіологічних резервів;
– мобілізацією енергетичних ресурсів та захисних сил.

Психічна адаптація – процес встановлення оптимальної відповідності між особистістю і навколишнім середовищем, яка сприяє задоволенню актуальних потреб і реалізації значущих цілей за умови збереження фізичного і психічного здоров'я [31].

Для правильного розуміння й тлумачення конфліктів, їхньої сутності, особливостей, функцій і наслідків важливе значення має типологізація, основних типів конфліктів на основі виявлення подібності та розходження.

3.2 Загальні вимоги безпеки з охорони праці для користувачів ПК

До основними вимогами безпеки з охорони праці під час використання ПК належать.

Вимоги безпеки перед початком роботи:

– оглянути робоче місце і навести на ньому лад; впевнитись, що на ньому немає сторонніх предмети, все обладнання і блоки ПК з’єднані з системним блоком з’єднувальними шнурами;

– перевірити надійність встановлення апаратури на робочому столі. Монітор не має стояти на краю стола. Повернути монітор так, щоб було зручно дивитися на екран — під прямим кутом (а не збоку) і трохи зверху вниз; при цьому екран має бути трохи нахиленим — нижній край ближче до користувача;

– перевірити загальний стан апаратури, справність електропроводки, з’єднувальних шнурів, штепсельних вилок, розеток, заземлення захисного екрана;

– вставити вилку в розетку і впевнитися, що вона міцно тримається. Заборонено вставляти і виймати вилку мокрими руками;

– відрегулювати та зафіксувати висоту крісла та зручний для користувача нахил спинки;

– за потреби приєднати до комп’ютера необхідну апаратуру (принтер, сканер). Усі кабелі, що з’єднують системний блок із іншими пристроями, вмикати та вимикати лише при вимкненому комп’ютері;

– відрегулювати яскравість свічення, контрастність монітора.

– про всі виявлені несправності інформувати керівника робіт і не братися до роботи, доки їх не буде усунено.

Вимоги безпеки під час виконання роботи під час роботи на ПК:

– стійко встановити клавіатуру на робочому столі, не допускаючи її хитання, водночас передбачити можливість її поворотів та переміщень;

– якщо в конструкції клавіатури не передбачено простору для упору долонь, клавіатуру розміщують на відстані не менше 100 мм від краю столу в оптимальній зоні моніторного поля;

– під час роботи на клавіатурі сидіти рівно, не напружуватися;

– щоб зменшити несприятливе навантаження на користувача при роботі з комп'ютерною мишею (вимушена поза, необхідність постійно контролювати якість дій), забезпечити велику вільну поверхню столу для переміщення комп'ютерної миші та зручного упору ліктьового суглоба;

– періодично при вимкненому комп'ютері прибирати пил із поверхонь апаратури спеціальними серветками.

При роботі з ПК заборонено:

– самостійно розбирати та ремонтувати системний блок (корпус ноутбука), монітор, клавіатуру, комп'ютерну мишу;

– встромляти сторонні предмети до вентиляційних отворів ПК, ноутбука або монітора;

– ставити на системний блок ПК та периферійні пристрої металеві предмети, ємкості з водою (вази, горщики для квітів, склянки), оскільки через потрапляння води у середину апарата може статися пожежа або ураження електрострумом;

– тривалість безперервної роботи за ПК не має перевищувати 2 год. Після цього необхідно зробити 15-хвилинну перериву;

– якщо виник зоровий дискомфорт або інші неприємні відчуття, необхідно зробити нетривалу перерву;

– для зниження нервово-емоційного напруження, стомлення зорового аналізатора, поліпшення мозкового кровообігу, подолання несприятливих наслідків гіподинамії, запобігання втомі доцільно під час декількох перерв виконувати комплекс вправ.

Вимоги безпеки після закінчення роботи:

– зберегти інформацію;

– вимкнути ПК, монітор чи ноутбук;

– вимкнути стабілізатор, якщо комп’ютер під’єднаний до мережі через нього;

– прибрати робоче місце [32].

3.3 Соціальне значення охорони праці

Соціальне значення охорони праці полягає в сприянні росту ефективності суспільного виробництва шляхом безперервного вдосконалення і поліпшення умов праці, підвищення їх безпеки, зниження виробничого травматизму і профзахворювань.

Соціальне значення охорони праці проявляється в зростанні продуктивності праці, збереженні трудових ресурсів і збільшенні сукупного національного продукту.

Зростання продуктивності праці відбувається в результаті збільшення фонду робочого часу завдяки скороченню внутрішньо-змінних простоїв шляхом ліквідації мікротравм або зниження їх кількості, а також завдяки запобіганню передчасного стомлення шляхом раціоналізації і покра-щення умов праці та введенню оптимальних режимів праці і відпочинку та інших заходів, які сприяють підвищенню ефективності використання робочого часу.

Збереження трудових ресурсів і підвищення професійної активності працюючих відбувається зав-дяки покращенню стану здоров'я і подовженню середньої тривалості життя шляхом покращення умов праці, що супроводжується високою трудовою активністю і підвищенням виробничого стажу. Підвищується професійний рівень також завдяки зростанню кваліфікації і майстерності.

Збільшення сукупного національного продукту відбувається завдяки покращенню вище перелічених показників та їх складових компонентів [33]. Плинність робочої сили завдає серйозних збитків підприємствам, оскільки тих, що звільняють, певний період працюють із зниженою продуктивністю.

3.4 Висновок до третього розділу

В третьому розділі кваліфікаційної роботи розглянуто питання з безпеки життєдіяльності яке полягає у тому, як впливають соціальні та психологічні фактори ризику на суспільство. В цілому небезпека від даних факторів ризику є досить таки великою. Особливість соціальних небезпек полягає в тому, що вони загрожують великій кількості людей. Розповсюдження соціальних небезпек зумовлено особливостями поведінки людей і окремих соціальних груп.

Щодо охорони праці розглянуто два питання одне полягає у дотримані правил користування ПК перед початком роботи, при роботі з ПК, при закінченні роботи, і соціальне значення охорони праці яке проявляється в зростанні продуктивності праці, збереженні трудових ресурсів та збільшенні сукупного національного продукту.

ВИСНОВКИ

Основною метою здійсненої в ході виконання кваліфікаційної роботи розробки програмного рішення є забезпечення користувачів можливістю обміну текстовими повідомленнями між собою.

У першому розділі роботи було проведено аналіз методів та засобів розробки з метою отримання можливості обміну текстовими повідомленнями на основі платформи .Net. На основі цього виокремлено вимоги до майбутньої програми та обрано актуальні технології для реалізації проекту. Після цього сформовано концепцію моделі даних та архітектури програми.

У другому розділі наведено результати оформлення та розробки програми, отримані з використанням відповідних моделей та діаграм. Зокрема, використано діаграми потоків даних та логічну і фізичну моделі даних. Опис кожного методу супроводжується відповідними графічними об’єктами програми та бази даних. В кінцевому результаті, на основі створеної програми, здійснюється перевірка працездатності функціональних можливостей, яким вона повинна відповідати. Також описано процес розгортання (інсталяції) програми.

У третьому розділі розглянуто соціальні та психологічні фактори ризику на суспільство в цілому, визначивши чинники з різними наслідками. Розповсюдження даної поведінки залежить від соціальних груп, при цьому наслідки загрожують великій кількості людей. Щодо дотримання правил користування ПК, було розглянуто основні правила користування. Розглянуто соціальне значення охорони праці, яке сприяє росту ефективності суспільного вдосконалення і поліпшенню умов праці, підвищення їх безпеки.

Завдяки написанню кваліфікаційної роботи отримано хороший практичний досвід щодо використання сучасних методів та засобів проектування програмного забезпечення для обміну текстовими повідомленнями на основі платформи .Net.

ПЕРЕЛІК ДЖЕРЕЛ

1. Вступ в C# .Net [Електронний ресурс] – 2020. – Режим доступу до ресурсу: https://cutt.ly/nJaAIZI.
2. Шифрування та захист баз даних [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/NJvFhSG.
3. Архітектура та проектування програмного забезпечення [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/6JRclCV.
4. Клієнт-серверна архітектура та ролі серверів [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://cutt.ly/SJitIEm.
5. Що таке архітектура програмного забезпечення [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://inlnk.ru/XODYgK.
6. В чому різниця між socket і web-socket [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://cutt.ly/YJid9DC.
7. WebSocket [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/RJRvIjE.
8. Що таке .NET [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/8JaIBpB.
9. Загальні відомості про платформу NET [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/cJaOemX.
10. Порівняльні характеристики sql СУБД [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://cutt.ly/EJaOlBp.
11. Система управління базами даних [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/tJRQ7Tv.
12. Принципи SOLID [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://cutt.ly/TJaLc3s.
13. Асинхронний веб [Електронний ресурс]. – 2017. – Режим доступу до ресурсу: https://cutt.ly/7JRRtRX.

14. Ведення в Entity Framework Core [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://cutt.ly/tJaPXSY.

15. Нові можливості .NET для класичних програм [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/iJaZcS2.

16. Діаграми потоків даних DFD [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://cutt.ly/sJaSoDc.

17. Моделювання потоків даних [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://cutt.ly/MJaSnmB.

18. Поняття ER-моделі та сутності entity [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://cutt.ly/8JaSZCV.

19. Різниця між логічною та фізичною моделлю даних [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/SJRUtqD.

20. Місце та роль БД та баз знань у сучасних комп'ютерних інформаційних технологіях [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/eJaDxwN.

21. Методологія інформаційних систем та баз даних [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://cutt.ly/JJRIbz9.

22. Призначення JSON [Електронний ресурс]. – 2018. – Режим доступу до ресурсу: https://cutt.ly/OJaFWpt.

23. Що таке JSON [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://cutt.ly/iJaG4qc.

24. Сервіси та методи ConfigureServices [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://cutt.ly/7JaD16d.

25. Виявлення адреси клієнта в soketserver [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://cutt.ly/AJaFiOR.

26. Основи роботи з TCP/IP [Електронний ресурс]. – 2019. – Режим доступу до ресурсу: https://cutt.ly/KJaFxIn.

27. Автоматичний Code First Migrations [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://cutt.ly/hJaFKvh.

28. Голуб Б.М. C#. Концепція та синтаксис: навчальний посібник / Голуб Б.М.. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2019. – 136 с.

29. Ендрю Т. Мова програмування C# 7 і платформи .NET і .NET Core: Пер. з англ. / Ендрю Т., Філіп Д., – СПб.: ООО "Діалектика", 2018. – 1328 с.

30. Організація баз даних: навчальний посібник. 2-ге вид. виправ. і доповн. / Трофименко О. Г., Прокоп Ю. В., Логінова Н. І., Копитчук І. М.. – Одеса: Фенікс, 2019. – 246 с.

31. Запорожець О. І. Безпека життєдіяльності, соціальні небезпеки [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://subj.ukr-lit.com/bezpeka-zhittyediyalnosti-zaporozhec-o-i/.

32. Інструкція з охорони праці при роботі на персональному комп'ютері [Електронний ресурс]. – 2021. – Режим доступу до ресурсу: https://www.sop.com.ua/article/485-nstruktsya-z-ohoroni-prats-pri-robot-na-personalnomu-kompyuter.

33. Правові основи цивільної безпеки, працеохоронної політики та охорони праці [Електронний ресурс]. – 2020. – Режим доступу до ресурсу: https://msn.khnu.km.ua/mod/page/view.php?id=110814.

ДОДАТКИ

Додаток А

Лістинги програмного коду інтерфейсу програмного забезпечення

Лістинг коду створення ініціалізації БД

```
namespace Messanger.WCFService
{
    public class Program
    {
        public static ApplicationStarter Application { get; set; }

        public static void Main(string[] args)
        {
            Console.WriteLine("Initialization DI container!");
            Console.WriteLine();

            IServiceCollection services = ConfigureServices(new ServiceCollection());

            Console.WriteLine("DI container successfully configured!");
            Console.WriteLine();

            IIpcServiceHost host = SetupListeners(services);

            var source = new CancellationTokenSource();

            Task.WaitAll(host.RunAsync(source.Token), Task.Run(() =>
            {
                Console.WriteLine("WCF service started!");

                Application = ApplicationStarter.GetInstance();

                Console.WriteLine();

                Console.WriteLine("Press any key to shutdown.");

                Console.ReadKey();

                Application.Stop();
                source.Cancel();
            }));

            Console.WriteLine("Server stopped.");
        }
```

```
            private static IIpcServiceHost
SetupListeners(IServiceCollection services)
            {
                var host = new
IpcServiceHostBuilder(services.BuildServiceProvider())

.AddNamedPipeEndpoint<IMessageServiceHandler>(name:
"messageEndpoint", pipeName: "message")

.AddNamedPipeEndpoint<IRoomServiceHandler>(name: "roomEndpoint",
pipeName: "room")

.AddNamedPipeEndpoint<IApplicationUserServiceHandler>(name:
"applicationUserEndpoint", pipeName: "applicationUser")

.AddNamedPipeEndpoint<IAccountServiceHandler>(name:
"accountEndpoint", pipeName: "account")
                                .Build();
                return host;
            }

            private static IServiceCollection
ConfigureServices(IServiceCollection services)
            {
                return services
                    .AddLogging(builder =>
                    {
                        builder.SetMinimumLevel(LogLevel.Trace);
                    })
                    .AddIpc(builder =>
                    {
                        builder
                            .AddNamedPipe(options =>
                            {
                                options.ThreadCount = 2;
                            })
                            .AddService<IAccountServiceHandler,
AccountServiceHandler>()

.AddService<IApplicationUserServiceHandler,
ApplicationUserServiceHandler>()
                            .AddService<IRoomServiceHandler,
RoomServiceHandler>()
                            .AddService<IMessageServiceHandler,
MessageServiceHandler>();
                    });
            }
        }
    }
```

Лістинг коду ядра програми

```
namespace Messanger.Server
{
    public class ApplicationStarter
    {
        private static ApplicationStarter Instance;

        private ApplicationStarter()
        {
            Start();
        }

        public static InjectionContainer InjectionContainer {
get; set; }

        private void Start()
        {
            Console.WriteLine("Initialize container!");

            InjectionContainer =
Activator.CreateInstance<InjectionContainer>();

            InjectionContainer.ConfigureServices();

            Console.WriteLine("Migrate database!");

            using(var scope =
InjectionContainer.GetContainer().BeginLifetimeScope())
            {
               var context =
scope.Resolve<MessangerDbContext>();
                if(context.Database.EnsureCreated())
                    context.Database.Migrate();
            }

            Console.WriteLine("Handler succssesfully
started!");
        }

        public void Stop()
        {
            InjectionContainer = null;
            Console.WriteLine("Handler succssesfully
stoped!");
        }

        public T ResolveController<T>() where T :
ControllerBase
        {
            var container =
InjectionContainer.GetContainer();
```

```
                  using (var scope =
container.BeginLifetimeScope())
                  {
                      return scope.Resolve<T>();
                  }
              }


              public static ApplicationStarter GetInstance()
              {
                  if (Instance == null)
                      Instance = new ApplicationStarter();
                  return Instance;
              }
          }
      }
```

Лістинг коду реалізації SoketServer

```
      namespace Messanger.SoketServer
      {
         public class Program
         {
              private static Listener _listener;

              private static List<Client> _sockets;

              public static void Main()
              {
                  _sockets = new List<Client>();
                  Console.WriteLine("Initialize listeners!");
                  _listener = new Listener(8);
                  _listener.SocketAccepted +=
_listener_SocketAccepted;

                  _listener.Start();
                  Console.WriteLine("Start listener!");

                  Console.Read();

              }

              private static void _listener_SocketAccepted(Socket
socket)
              {
                  Console.ForegroundColor = ConsoleColor.Green;
                  Console.WriteLine($"New connection
{socket.RemoteEndPoint}{Environment.NewLine}{DateTime.Now}{Environ
ment.NewLine}===========");
                  Console.ForegroundColor = ConsoleColor.Gray;

                  var client = new Client(socket);
```

```
                client.Received += Client_Received;
                client.Disconnected += Client_Disconnected;


                _sockets.Add(client);
            }

            private static void Client_Disconnected(Client
sender)
            {
                Console.ForegroundColor = ConsoleColor.DarkRed;
                Console.WriteLine($"Disconnected
{sender.EndPoint}{Environment.NewLine}{DateTime.Now}{Environment.N
ewLine}===========");
                Console.ForegroundColor = ConsoleColor.Gray;

            }

            private static void Client_Received(Client sender,
string json)
            {
                Console.ForegroundColor = ConsoleColor.Magenta;
                Console.WriteLine($"Receive client by id:
{sender.Id}");
                Console.WriteLine($"Message {json}");
                _sockets.ForEach(x => x.Send(sender.RoomId,
json));
                Console.WriteLine();
                Console.ForegroundColor = ConsoleColor.Gray;

            }
        }
    }
```

Лістинг коду реалізації обміну повідомлень

```
    namespace Messanger.SoketServer
    {
       public class Program
       {
            private static Listener _listener;

            private static List<Client> _sockets;

            public static void Main()
            {
                _sockets = new List<Client>();
                Console.WriteLine("Initialize listeners!");
                _listener = new Listener(8);
                _listener.SocketAccepted +=
_listener_SocketAccepted;
```

```
                    _listener.Start();
                    Console.WriteLine("Start listener!");

                    Console.Read();

                }

                private static void _listener_SocketAccepted(Socket
socket)
                {
                    Console.ForegroundColor = ConsoleColor.Green;
                    Console.WriteLine($"New connection
{socket.RemoteEndPoint}{Environment.NewLine}{DateTime.Now}{Environ
ment.NewLine}============");
                    Console.ForegroundColor = ConsoleColor.Gray;

                    var client = new Client(socket);
                    client.Received += Client_Received;
                    client.Disconnected += Client_Disconnected;


                    _sockets.Add(client);
                }

                private static void Client_Disconnected(Client
sender)
                {
                    Console.ForegroundColor = ConsoleColor.DarkRed;
                    Console.WriteLine($"Disconnected
{sender.EndPoint}{Environment.NewLine}{DateTime.Now}{Environment.N
ewLine}============");
                    Console.ForegroundColor = ConsoleColor.Gray;

                }

                private static void Client_Received(Client sender,
string json)
                {
                    Console.ForegroundColor = ConsoleColor.Magenta;
                    Console.WriteLine($"Receive client by id:
{sender.Id}");
                    Console.WriteLine($"Message {json}");
                    _sockets.ForEach(x => x.Send(sender.RoomId,
json));
                    Console.WriteLine();
                    Console.ForegroundColor = ConsoleColor.Gray;

                }
            }
        }
```

Лістинг коду клієнтської частини програми

```
namespace Messanger.SoketServer
{
    public class Client
    {
        public string Id { get; private set; }

        public string RoomId { get; private set; }
        private readonly MessageController
_messageController;

        public IPEndPoint EndPoint { get; private set; }

        private readonly Socket _socket;

        private static int _messageSize => 8 * 1024;

        private byte[] _dump = new byte[_messageSize];

        public Client(Socket socket)
        {
            _socket = socket;

            Id = Guid.NewGuid().ToString();
            _messageController =
ApplicationStarter.GetInstance().ResolveController<MessageControll
er>();
            EndPoint = (IPEndPoint)socket.RemoteEndPoint;

            _socket.BeginReceive(_dump, 0, _dump.Length,
SocketFlags.None, ReceiveCallback, null);
        }

        private void ReceiveCallback(IAsyncResult
asyncResult)
        {
            try
            {
                var received =
_socket.EndReceive(asyncResult);
                var buffer = new byte[received];
                Array.Copy(_dump, buffer, received);

                var text = Encoding.UTF8.GetString(buffer);
                if (text.StartsWith("roomId",
StringComparison.InvariantCultureIgnoreCase))
                {
                    var id = text.Split('|')[1];
                    RoomId = id;
                }
                else if(text.StartsWith('{'))
                {
```

```
                        var message =
JsonConvert.DeserializeObject<Message>(text);
                        var resultMes =
_messageController.Create(message);
                        Received?.Invoke(this, resultMes);
                    }
                    _socket.BeginReceive(_dump, 0, _dump.Length,
SocketFlags.None, ReceiveCallback, null);
                }
                catch (Exception ex)
                {
                    Console.WriteLine(ex.Message);
                    Close();
                    Disconnected?.Invoke(this);
                }
            }

            public void Send(string roomId, string text)
            {
                if (RoomId == roomId)
                {
                    var data = Encoding.UTF8.GetBytes(text);
                    _socket.BeginSend(data, 0, data.Length,
SocketFlags.None, new AsyncCallback(SendCallback), null);
                }
            }

            private void SendCallback(IAsyncResult asyncResult)
            {
                _socket.EndSend(asyncResult);
            }

            public void Close()
            {
                _socket.Close();
                _socket.Dispose();
            }

            public delegate void ClientReceivedHandler(Client
sender, string json);
            public delegate void ClientDisconnectedHandler(Client
sender);

            public event ClientReceivedHandler Received;
            public event ClientDisconnectedHandler Disconnected;
        }
    }
```

Лістинг коду методу AccountServiceHandler

```
    namespace Messanger.WCFService.Services
    {
        public class AccountServiceHandler :
IAccountServiceHandler
```

```
        {
            private readonly AccountController
_accountController;

            public AccountServiceHandler()
            {
                _accountController =
Program.Application.ResolveController<AccountController>();
            }

            public string Login(LoginModel model) =>
                _accountController?.Login(model);

            public string Register(RegisterModel model) =>
                _accountController?.Register(model);
        }
    }
```

Лістинг коду методу AccountServiceHandler

```
    namespace Messanger.Services
    {
        public class ApplicationUserService :
IApplicationUserService
        {
            private readonly IApplicationUserRepository
_applicationUserRepository;

            public
ApplicationUserService(IApplicationUserRepository
applicationUserRepository)
            {
                _applicationUserRepository =
applicationUserRepository;
            }

            public ApplicationUser Get(string id)
            {
                return _applicationUserRepository.FindOne(x =>
x.Id == id);
            }

            public ApplicationUser GetByEmail(string email)
            {
                return _applicationUserRepository.FindOne(x =>
x.Email == email);
            }

            public ApplicationUser Create(ApplicationUser entity)
            {
                return
_applicationUserRepository.InsertOne(entity);
            }
```

```
            public ApplicationUser Update(ApplicationUser entity)
            {
                return
_applicationUserRepository.UpdateOne(entity);
            }

            public ApplicationUser Delete(string id)
            {
                return _applicationUserRepository.DeleteOne(x =>
x.Id == id);
            }
        }
    }
```

Лістинг коду реалізації аунтентифікації

```
    namespace Messanger
    {
        public partial class LoginForm : Form
        {
            public LoginForm()
            {
                InitializeComponent();
                this.DesignViews();
            }

            private async void btnLogin_Click(object sender,
EventArgs e)
            {
                var result = await
AccountApi.GetInstance().Login(txtEmail.Text, txtPassword.Text);
                if (result == null) return;

                var form = new MainForm();
                form.Show();
                this.Hide();
            }

            private void btnRegister_Click(object sender,
EventArgs e)
            {
                var form = new RegisterForm();
                form.Show();
                this.Hide();
            }
        }
    }
```

Лістинг коду з’єднання клієнта з сервером

```
    namespace Messanger.Client.Core
    {
        public class AccountApi
        {
```

```
            private static AccountApi _instance = null;
            private readonly
IpcServiceClient<IAccountServiceHandler> _accountClient;
            private readonly
IpcServiceClient<IApplicationUserServiceHandler> _userClient;

            private ApplicationUser _currentUser;

            private AccountApi()
            {
                _accountClient = new
IpcServiceClientBuilder<IAccountServiceHandler>()

.UseNamedPipe("account")
                                                            .Build();

                _userClient = new
IpcServiceClientBuilder<IApplicationUserServiceHandler>()

.UseNamedPipe("account")
                                                            .Build();
            }

            public static AccountApi GetInstance()
            {
                if (_instance == null)
                    _instance = new AccountApi();
                return _instance;
            }

            public ApplicationUser GetCurrentUser() =>
                _currentUser;

            public async Task<ApplicationUser> Login(string
email, string password)
            {
                var loginModel = new LoginModel { Email = email,
Password = password };
                var responce = await _accountClient.InvokeAsync(x
=> x.Login(loginModel));

                if (string.IsNullOrWhiteSpace(responce))
                    return null;

                var result =
JsonConvert.DeserializeObject<ApiResult<ApplicationUser>>(responce
);
                if (result.Result == null)
                {
                    MessageBox.Show(result.ErrorMessage,
"Error!");
                    return null;
                }
```

```
                    _currentUser = result.Result;

                    return result.Result;
                }

                public async Task<bool> Register(string name, string
email, string password)
                {
                    var registerModel = new RegisterModel { Name =
name, Email = email, Password = password };
                    var responce = await _accountClient.InvokeAsync(x
=> x.Register(registerModel));

                    if (string.IsNullOrWhiteSpace(responce))
                        return false;

                    var result =
JsonConvert.DeserializeObject<ApiResult<bool>>(responce);
                    if (result.Result == false)
                    {
                        MessageBox.Show(result.ErrorMessage,
"Error!");
                    }
                    return result.Result;
                }

                public async Task<ApplicationUser> JoinToRooms(Room
room)
                {
                    var user = GetCurrentUser();

                    if (user.RoomLinks == null)
                        user.RoomLinks = new
List<ApplicationUserRoom>();

                    user.RoomLinks.Add(new ApplicationUserRoom
                    {
                        RoomId = room.Id,
                        UserId = user.Id
                    });

                    var responce = await _userClient.InvokeAsync(x =>
x.Update(user));
                    if (string.IsNullOrWhiteSpace(responce))
                        return null;

                    var result =
JsonConvert.DeserializeObject<ApiResult<ApplicationUser>>(responce
);
                    if (result.Result == null)
                    {
                        MessageBox.Show(result.ErrorMessage,
"Error!");
```

Додаток Б

Лістинги програного коду інтерфейсу програмного рішення

Лістинг коду реалізації класу ApplicationUserEntityConfiguration

```
Messanger.Persistance.Infrastructure.EntityConfigurations
{
    public class ApplicationUserRoomEntityConfiguration : IEntityTypeConfiguration<ApplicationUserRoom>
    {
        public void Configure(EntityTypeBuilder<ApplicationUserRoom> builder)
        {
            if (builder == null) return;

            builder
                .HasKey(x => new { x.UserId, x.RoomId });

            //If you name your foreign keys correctly, then you don't need this.
            builder
                .HasOne(pt => pt.User)
                .WithMany(p => p.RoomLinks)
                .HasForeignKey(pt => pt.UserId);

            builder
                .HasOne(pt => pt.Room)
                .WithMany(t => t.ParticipantLinks)
                .HasForeignKey(pt => pt.RoomId);
        }
    }
}
```

Лістинг коду типу даних користувача

```
namespace Messanger.Persistance.Entities
{
    public class ApplicationUser : IEntity
    {
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public string Id { get; set; }
        public string Name { get; set; }
        public string Email { get; set; }
        public string Password { get; set; }
        public List<ApplicationUserRoom> RoomLinks { get; set; }

        public bool Validate()
        {
```

```
                return !string.IsNullOrWhiteSpace(Name) &&
                        !string.IsNullOrWhiteSpace(Email) &&
                        !string.IsNullOrWhiteSpace(Password);
```

Лістинг коду типу даних кімнати

```
namespace Messanger.Persistance.Entities.Chat
{
    public class Room : IEntity
    {
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public string Id { get; set; }
        public string Name { get; set; }
        public List<ApplicationUserRoom> ParticipantLinks {
get; set; }
        public List<Message> Messages { get; set; }

        public bool Validate()
        {
            return !string.IsNullOrWhiteSpace(Name);
        }
    }
}
```

Лістинг коду створеня таблиці кімнати

```
namespace
Messanger.Persistance.Infrastructure.EntityConfigurations
{
    public class RoomEntityConfiguration :
IEntityTypeConfiguration<Room>
    {
        public void Configure(EntityTypeBuilder<Room>
builder)
        {
            if (builder == null) return;

            builder
                .ToTable("rooms");

            builder
                .HasKey(o => o.Id);

            builder
                .Property<string>("Name")
                    .HasColumnName("name")
                    .IsRequired();

            builder
                .HasMany<Message>()
                .WithOne(x => x.Room)
                .HasForeignKey(x => x.RoomId)
```

Лістинг коду типу даних повідомлень

```
namespace Messanger.Persistance.Entities.Chat
{
    public class Message : IEntity
    {
        [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
        public string Id { get; set; }
        public string Text { get; set; }
        public string ImageHash { get; set; }
        public DateTime Timestamp { get; set; }
        public string OwnerId { get; set; }
        public ApplicationUser Owner { get; set; }
        public string RoomId { get; set; }
        public Room Room { get; set; }

        public bool Validate()
        {
            return !string.IsNullOrWhiteSpace(Text) &&
                   !string.IsNullOrWhiteSpace(OwnerId) &&
                   !string.IsNullOrWhiteSpace(RoomId);
        }
    }
}
```

Лістинг коду типу даних повідомлень

```
namespace
Messanger.Persistance.Infrastructure.EntityConfigurations
{
    public class ApplicationUserRoomEntityConfiguration :
IEntityTypeConfiguration<ApplicationUserRoom>
    {
        public void
Configure(EntityTypeBuilder<ApplicationUserRoom> builder)
        {
            if (builder == null) return;
            builder
                .HasKey(x => new { x.UserId, x.RoomId });
            //If you name your foreign keys correctly, then
you don't need this.
            builder
                .HasOne(pt => pt.User)
                .WithMany(p => p.RoomLinks)
                .HasForeignKey(pt => pt.UserId);
            builder
                .HasOne(pt => pt.Room)
                .WithMany(t => t.ParticipantLinks)
                .HasForeignKey(pt => pt.RoomId);
        }
```

Додаток В

Лістинги програного коду методів для операції з базою даних

Лістинг коду обгортки dbContext

```
namespace Messanger.Persistance.Infrastructure
{
    public abstract class Repository<TEntity> :
IRepository<TEntity> where TEntity : class, IEntity
    {
        protected readonly MessangerDbContext _dbContext;
        protected Repository(MessangerDbContext dbContext)
        {
           _dbContext = dbContext;
```

Лістинг коду функції обрахунку

```
public int Count(Expression<Func<TEntity, bool>> predicate)
        {
            try
            {
                var dbSet = _dbContext.Set<TEntity>();
                return dbSet.Count(predicate);
            }
            catch (Exception ex)
            {
                throw new DomainException("DbException Count
" + typeof(TEntity).Name + ": " + ex.Message, ex);
            }
        }
```

Лістинг коду функція видалення

```
public TEntity DeleteOne(Expression<Func<TEntity, bool>>
predicate)
        {
            try
            {
                var dbSet = _dbContext.Set<TEntity>();
                var entity = dbSet.FirstOrDefault(predicate);
               if (entity == null)
                {
                    throw new DomainException("Exception in
DeleteOne " + typeof(TEntity).Name);
                }
                _dbContext.Set<TEntity>().Remove(entity);
                _dbContext.SaveChanges();
                return entity;
            }
```

```
                  catch (Exception ex)
                  {
                      throw new DomainException("DbException
DeleteOne " + typeof(TEntity).Name + ": " + ex.Message, ex)
```

Лістинг коду функції пошуку об’єктів

```
        public IEnumerable<TEntity>
FindMany(Expression<Func<TEntity, bool>> predicate)
              {
                  try
                  {
                      var dbSet = _dbContext.Set<TEntity>();
                      return dbSet.Where(predicate).ToList();
                  }
                  catch (Exception ex)
                  {
                      throw new DomainException("DbException
FindMany " + typeof(TEntity).Name + ": " + ex.Message, ex);
                  }
              }
```

Літинг коду функції встановлення

```
    public IEnumerable<TEntity> InsertMany(IEnumerable<TEntity>
entities)
        {
            try
            {
                _dbContext.Set<TEntity>().AddRange(entities);
                _dbContext.SaveChanges();
                return entities;
            }
            catch (Exception ex)
            {
                throw new DomainException("DbException InsertMany
" + typeof(TEntity).Name + ": " + ex.Message, ex);
            }
        }
        public TEntity InsertOne(TEntity entity)
        {
            try
            {
                var result =
_dbContext.Set<TEntity>().Add(entity);
                _dbContext.SaveChanges();
                return result.Entity;
            }
            catch (Exception ex)
            {
```

Лістинг коду функції оновлення вікна програми

```
    #region UPDATE
        public IEnumerable<TEntity>
UpdateMany(IEnumerable<TEntity> entities)
        {
            try
            {
                _dbContext.Set<TEntity>().UpdateRange(entities);
                _dbContext.SaveChanges();
                return entities;
            }
            catch (Exception ex)
            {
                throw new DomainException("DbException UpdateMany
" + typeof(TEntity).Name + ": " + ex.Message, ex);
            }
        }
        public IEnumerable<TEntity>
UpdateMany(Expression<Func<TEntity, bool>> predicate,
Action<TEntity> action)
        {
            try
            {
                var dbSet = _dbContext.Set<TEntity>();

dbSet.ForEachAsync(action).GetAwaiter().GetResult();
                _dbContext.SaveChanges();
                return dbSet.Where(predicate).ToList();
            }
            catch (Exception ex)
            {
                throw new DomainException("DbException UpdateMany
" + typeof(TEntity).Name + ": " + ex.Message, ex);
            }
        }
        public TEntity UpdateOne(TEntity entity)
        {
            try
            {
                if (entity == null) throw new
DomainException("Exception in UpdateOne " + typeof(TEntity).Name);
                _dbContext.Set<TEntity>().Update(entity);
                _dbContext.SaveChanges();
                return entity;
            }
            catch (DomainException)
            {
                throw;
            }
            catch (Exception ex)
            {
```