

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування інституту, факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОГРАМНА РЕАЛІЗАЦІЯ ЧАТУ В РЕАЛЬНОМУ

ЧАСІ НА ОСНОВІ МІКРОСЕРВІСІВ

SOFTWARE IMPLEMENTATION OF A REAL-TIME CHAT

BASED ON MICROSERVICES

Виконав: студент(ка) 4 курсу, групи КНТ-218

Спеціальності 122 Комп'ютерні науки

(код і найменування спеціальності)

Освітня програма (спеціалізація)

Комп'ютерні науки

Лінець В.А.

(прізвище та ініціали)

Керівник Колпакова Т.О.

(прізвище та ініціали)

Рецензент Скрупський С.Ю.

(прізвище та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Інститут, факультет ФКНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 122 Комп'ютерні науки
(код і найменування)
 Освітня програма (спеціалізація) Комп'ютерні науки
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
С.О. Субботін
 “ ” 2022 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

Лінця Вадима Андрійовича
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) Програмна реалізація чату в реальному часі на основі мікросервісів. Software Implementation of a Real-Time Chat Based on Microservices

керівник проєкту (роботи) Колпакова Тетяна Олексіївна, к.т.н., доцент,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом закладу вищої освіти від “27” квітня 2022 року № 102

2. Строк подання студентом проєкту (роботи) 07 червня 2022 року
 3. Вихідні дані до проєкту (роботи) рекомендована література, технічне завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1. Аналіз проблеми та постановка завдань дослідження. 2. Матеріали і методи. 3. Опис програми. 4. Експлуатація, тестування та експериментальне дослідження програми.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	Колпакова Т.О., доцент		
Нормоконтроль	Липовець М.В., асистент		

7. Дата видачі завдання « 04 » квітня 2022 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 2
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис)

Лінець В.А.

(прізвище та ініціали)

Керівник проєкту (роботи)

(підпис)

Колпакова Т.О.

(прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 101 с., 35 рис., 7 табл., 3 додатки, 15 джерел.

ІНТЕРНЕТ-ЧАТИ, ВЕБЗАСТОСУНОК, GOLANG, GO, VSCODE, JAVASCRIPT.

Об'єкт дослідження – процес спілкування користувачів онлайн.

Предмет дослідження – вебзастосунки для спілкування користувачів у інтернет-чатах.

Мета роботи – створення програмного забезпечення для надання можливості спілкування в інтернеті за допомогою чатів або коментарів користувачам тих чи інших сервісів.

Матеріали, методи та технічні засоби: структурне та об'єктно-орієнтоване програмування, мови програмування Golang та JavaScript, інтегровані середовища розробки програм Microsoft Visual Studio Code, персональний IBM-сумісний комп'ютер з процесором Intel Core i5, оперативною пам'яттю 8 Гб, 14'' монітором, клавіатурою та маніпулятором типу «миша», що функціонує під управлінням операційної системи Microsoft Windows 10.

Результати. Створено програмний вебзастосунок, який надає можливість декільком людям спілкуватися між собою через вебсторінку у браузері.

Висновки. Розроблено вебзастосунок який надає можливість спілкуватися в інтернеті за допомогою чатів або коментарів користувачам.

Галузь використання – подібну програму можна використовувати як модуль, що підключається до існуючого сервісу для надання можливості спілкування або коментування користувачам.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 101 pages, 35 figures, 7 tables, 3 appendixes, 15 sources.

INTERNET CHAT, WEB-APPLICATION, GOLANG, GO, VSCODE, JAVASCRIPT.

The object of research is the process of online communication between users.

The subject of the research is web applications for communication of users in Internet chats.

The purpose of the work is to create a software system to enable communication on the Internet through chats or comments to users of certain services.

Materials, methods and technical means: structural and object-oriented programming, Golang and JavaScript programming languages, integrated Microsoft Visual Studio Code development environments, personal IBM-compatible computer with Intel Core i5 processor, 8 GB RAM, 14 " monitor, keyboard and mouse-type manipulator running Microsoft Windows 10.

Results. A software web application has been created that allows several people to communicate with each other through a web page in a browser.

Conclusions. Developed a web application that allows you to communicate online through chats or comments to users.

Field of use – such program can be used as a module that connects to the existing service to allow users to communicate or comment.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	8
Вступ.....	9
1 Аналіз предметної області.....	10
1.1 Аналіз проблеми.....	10
1.2 Порівняння аналогів	14
1.3 Огляд використовуваних технологій	15
1.4 Постановка завдань роботи.....	16
1.5 Висновки за розділом 1	17
2 Розробка архітектури програми.....	18
2.1 Особливості мови програмування Golang.....	18
2.2 Протокол gRPC як основний спосіб комунікації між мікросервісами...	19
2.3 Visual Studio Code як основна середовище розробки проєкту	21
2.4 RabbitMQ у якості основного брокера повідомлень	22
2.5 Розробка архітектури програми.....	22
2.6 Проєктування бази даних	25
2.7 Висновки за розділом 2	26
3 Основні рішення щодо реалізації компонентів системи.....	27
3.1 Функціонал блоку Api-gateway.....	27
3.2 Функціонал блоку Auth	34
3.3 Функціонал блоку Messages.....	35
3.4 Функціонал блоку Room	36
3.5 Функціонал блоку User.....	37
3.6 Інтерфейс програми	38

3.7 Загальний алгоритм функціонування програми	39
3.8 Висновки за розділом 3	40
4 Експлуатація, тестування та експериментальне дослідження програми	41
4.1 Опис застосування	41
4.2 Інструкція з експлуатації програми	41
4.3 Висновки за розділом 4	48
Висновки	49
Перелік джерел посилання	51
Додаток А Технічне завдання	53
Додаток Б Текст програми	57
Додаток В Слайди презентації.....	96

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

- AMQP – Advanced Message Queuing Protocol;
- API – Application Programming Interface;
- CLI – Command-line interface;
- CSS – Cascading Style Sheets;
- DDL – Data Definition Language;
- gRPC – Google Remote Procedure Call;
- HTTP – HyperText Transfer Protocol;
- RPC – Remote Procedure Call;
- БД – База даних.

ВСТУП

На поточний момент в інтернеті існує більш ніж 1,88 мільярдів різних сайтів. На багатьох з них в той чи іншій формі присутні блоки, де користувачі можуть або інтерактивно спілкуватися між собою, або висловлювати реакцію на представлену інформацію. Це можуть бути чати, коментарі, тощо.

Дуже часто для створення системи коментарів розробники сайтів витрачають багато часу, бо в інтернеті мало універсальних модулів або варіантів коду, які було б легко адаптувати під потреби поточного проєкту. Через це доводиться кожен раз розроблювати систему майже з нуля.

В роботі розглянуто можливість створення такого чату, який можна легко перероблювати, який водночас є дуже швидким та максимально адаптивним, і який дозволив би його використання на ранніх етапах роботи над будь яким проєктом без внесення суттєвих змін в його код.

Стати таким гнучким рішенням може чат, розроблений за допомогою мікросервісів. Використання мікросервісів дозволяє представити застосунок у вигляді набору окремих частин однієї програми, які працюють незалежно одна від одної. Також для підтримки миттєвого спілкування в реальному часі доцільним може виявитись використання вебсокетів.

Розробка цього вебзастосунку вирішить багато проблем, що виникають у програм зі схожим функціоналом. Ось деякі з цих проблем:

- безпека повідомлень користувачів. Завдяки цьому вебзастосунку розробники зможуть самі керувати протоколами безпеки не покладаючись на окремий сервіс;

- гнучкість програми. Сторонні сервіси потребують якогось свого формату роботи з повідомленнями, що може бути незручним для користувача.

Дані проблеми мають бути розв'язані в дипломній кваліфікаційній роботі за рахунок розроблення відповідного програмного рішення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз проблеми

Чат у буквальному перекладі з англійської означає «дружня бесіда». Цей жанр інтернет-комунікації є спілкуванням в режимі реального часу (on-line). Відвідувачі чату входять до комунікативного простору чату під ніками (мережевими іменами). Відвідувачі можуть побачити поле чату, а саме репліки тих, хто вже спілкується, та історію переписки за певний період. Якщо хід бесіди співрозмовники у чаті викликають у користувача інтерес, він може також опублікувати свою репліку і вступити у спілкування. Зазвичай запис реплік будується за моделлю: нік адресанта, нік адресата, текст висловлювання [1].

На даний момент в інтернеті існує багато не тільки сайтів з різноманітними видами чатів, але й окремих програм для організації групових та індивідуальних чатів (месенджерів). Месенджер – програма, мобільний додаток або вебсервіс для миттєвого обміну повідомленнями [2]. Окрім цього на різних сайтах, наприклад на YouTube ті інших платформах з медіа-контентом, є блоки коментарів. Їх також можна вважати окремим видом чату для користувачів.

Нижче наведені деякі з популярних месенджерів на прикладі яких розроблявся цей проєкт, а також таблиця 1.1 з порівнянням деяких з них.

1.1.1 Telegram

Однією з найголовніших переваг цього месенджера на даний час є одна з найкращих серед усіх подібних програм система захисту і шифрування даних. Окрім цього в ньому можна знайти такі корисні для користувачів функції як хмарне зберігання файлів розміром до 1,5 Гб, наявність Payment-платформи, що дозволяє користувачам робити оплату безпосередньо через Telegram, можливість додати у кімнату чату до 5000 осіб, систему

секретних чатів, в якій повідомлення видаляються автоматично після прочитання, та багато іншого [2].

Однак, не зважаючи на багато переваг цього месенджера у нього є і недоліки. Наприклад, такими недоліками можуть стати обов'язкова прив'язка акаунта користувача до номеру телефона, а отже навіть якщо користувач сидить у месенджері анонімно, його можна знайти за номером телефона. Також недоліком є обмеженість Telegram у десктопній та веб версії у порівнянні з мобільною. Окрім цього, Telegram не має можливості по вбудовуванню конкретного чату на сайт. Це можна зробити лише шляхом створення спеціального боту, як посередника між сервісом та вебзастосунком клієнта.

1.1.2 WhatsApp

WhatsApp – це безкоштовний месенджер, можливості якого включають до себе обмін повідомленнями, дзвінки через додаток, та доступний на будь-яких мобільних пристроях по всьому світу. На сьогоднішній день одною з важливих для користувача політик месенджера є заборона будь-яких рекламних оголошень та банерів. Окрім цього WhatsApp має певний функціонал, що робить його привабливим для тих, хто хоче спілкуватися саме через цей месенджер. Таким функціоналом є, наприклад, аудіо- і відеодзвінки, функціонал створення груп, можливість відправляти різні документи обсягом до 100 МБ, створення голосових повідомлень, тощо. Також WhatsApp можна вбудовувати на сайти завдяки віджетам, які можна легко знайти в інтернеті [2].

У якості недоліків можна зазначити дуже погану роботу з декількома пристроями, проблеми з безпекою як для даних з чатів, так і для самих пристроїв, незручність використання.

1.1.3 Facebook Messenger

Facebook Messenger – це продукт компанії Meta (раніше вона називалася Facebook), популярної соцмережі, яка відома по всьому світу. Загалом цей месенджер охопив понад мільярд користувачів завдяки тому, що після реєстрації у Facebook система пропонує скачати цей месенджер для більш зручного обміну повідомленнями між користувачами. Facebook Messenger зовсім не один з найзручніших месенджерів, але не дивлячись на це він може використовуватись, наприклад для корпоративних потреб, якщо компанія, або її робітники активно користується Facebook'ом, або якщо потрібна альтернатива контактній формі на сайті. Слід відзначити, що загалом в світі Facebook Messenger більш популярний, ніж в Україні, тому його можна використовувати також для зручного, для обох сторін, спілкування з іноземними клієнтами. Крім стандартного, для будь-якого месенджера, функціоналу в цьому додатку є ігри, опитування, нагадування, персональний помічник Facebook M, а також є можливість запускати рекламу як в самому месенджері, так і на сторінках Facebook. Окрім цього дозволяє прямо на своєму вебсайті згенерувати віджет для подальшої вставки на свою вебсторінку [2].

Серед недоліків сервісу можна виділити погану конфіденційність, можливість видаляти повідомлення лише на своєму пристрої, неможливість редагувати повідомлення.

1.1.4 Viber

Viber зараз є найпопулярнішим месенджером серед українців. За даними українського представництва компанії, месенджером користуються понад 20 мільйонів осіб, тобто половина населення країни. Функціонал Viber найбільш ефективний для eCommerce. Через цей сервіс можна відправляти проморозилки, розсилати транзакційні і тригерні повідомлення клієнтам і

передплатникам, надсилати повідомлення, що інформують про доставку замовлення, термін дії персональної знижки, кількість бонусів на рахунку. Відправляти такі повідомлення через додаток істотно дешевше, ніж за допомогою СМС. Також Viber дозволяє встановлювати час життя повідомлення – неактуальна інформація не буде турбувати одержувача після того, як термін дії акції закінчився [2].

1.1.5 Wechat

Wechat приваблює багатофункціональністю: через цей месенджер можна відправляти текстові повідомлення, здійснювати відеодзвінки, ділитися відео і фотографіями. Він розробляється переважно для китайської аудиторії та як правило враховує лише її потреби, тож свій повний функціонал він розкриває лише при прив'язці до китайського номеру телефону [2].

Сервіс має багато недоліків, таких як складний для користувача дизайн, поганий переклад функцій месенджера, відсутність адаптації під інші країни та світовий ринок.

1.1.6 Skype

Skype – безкоштовне програмне забезпечення, що розробляється компанією Microsoft та забезпечує текстовий, голосовий та відеозв'язок через Інтернет, а також платні послуги для дзвінків на мобільні і стаціонарні телефони. Це незамінний інструмент для ділового спілкування, що надає можливість обміну повідомленнями та файлами, а також здійснення дзвінків і відеоконференцій. Програмою користуються близько 300 мільйонів осіб у всьому світі. За допомогою Skype клієнти можуть здійснювати дзвінки прямо з сайту. Базові функції Skype включають текстові чати, конференції на кілька учасників і, звичайно, відеодзвінки. Останнє укупі з функцією «демонстрація екрану» залишається однією з головних причин, з якої цей месенджер

затребуваний в b2b-сегменті. Також у Skype є API (Application Programming Interface, прикладний програмний інтерфейс), що дозволяє розміщувати на сайті активну кнопку месенджера – вона дозволяє почати чат або зателефонувати в один клік [2].

1.2 Порівняння аналогів

Після аналізу існуючих аналогів створюваної програми можна побачити, що всі вони мають можливість спілкування користувачів у реальному часі, але далеко не всі з них є продуктами з відкритим кодом. Через це неможливо переробити їх під свої потреби або вимоги. Окрім цього у великої кількості розглянутих сервісів не має можливості інтегрування їх до вебсторінки, що робить неможливим їх використання для деяких цілей. Більш детальне порівняння деяких з цих аналогів і розроблюваної програми наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння програми з аналогами

	Telegram	Facebook Messenger	Розробка
Чат у реальному часі	+	+	+
Відкритість розробки	+	-	+
Можливість інтеграції у вебсторінку	+/-	+	+
Безпека даних	+	-	+
Можливість додати свої дані в повідомлення	-	-	+

1.3 Огляд використовуваних технологій

У цьому проєкті планується використовувати такі технології як мікросервіси та вебсокети. Розглянемо детальніше кожен з цих важливих для проєкту технологій.

1.3.1 Мікросервіси

Мікросервіси – це архітектурний стиль, натхненний сервісно-орієнтованими обчисленнями, який нещодавно почав набирати популярність. Мікросервісні архітектури були запропоновані для розбиття архітектури додатків на незалежні одна від одної частини, які можна розгортати не зважаючи на інші розроблені частини, та які можна доволі швидко розгорнути на будь-якому сервері за виникнення такої потреби. Зазвичай мікросервіси підтримуються платформою розгортання та оркестрування (наприклад, такою платформою може бути kubernetes), що дозволяє їм часто та незалежно розгортатися за довільним графіком та не дуже сильно залежати один від одного, навіть якщо один з мікросервісів перестане працювати [3].

Через свою незалежність один від одного мікросервіси є дуже стійкими до різноманітних проблем. Навіть якщо один з мікросервісів вийде з ладу, це ніяк не позначиться на інших мікросервісах. Це досягається шляхом повної ізоляції кожного окремого сервісу від інших. У кожного з сервісів повинна бути своя окрема БД (база даних), якщо вона потрібна, та інші зовнішні модулі, а в ідеальному варіанті ще й свій окремий сервер. Через все це мікросервіси займають набагато більше місця, та їх складніше тестувати. Однак все це окупається тим, що потрібні частини мікросервісів можна легко масштабувати при великому навантаженні одного з них, легко змінювати код та підключати до проєкту нових розробників, оскільки кожен з мікросервісів відповідає за свою задачу і не такий великий, як в монолітній архітектурі.

Також при використанні мікросервісів можна зв'язувати між собою різні сервіси, навіть написані на різних мовах програмування [4].

1.3.2 Вебсокети

Згідно зі стандартом інтернету RFC 6455, протокол WebSocket забезпечує двосторонній зв'язок між клієнтом, який виконує ненадійний код у контрольованому середовищі, та віддаленим хостом, який вибрав зв'язок із цим кодом. Модель безпеки, яка використовується для цього – це модель безпеки на основі оригінального захисту, який зазвичай використовують браузерери [5].

Протокол складається з початкового рукостискання з наступним базовим кадруванням повідомлень, пошаровим над TCP. Метою цієї технології є забезпечення механізму для програм на основі браузера, які потребують двостороннього зв'язку з серверами, і які не покладаються на відкриття кількох HTTP-з'єднань [6].

1.4 Постановка завдань роботи

Не дивлячись на те, що в інтернеті дуже багато різноманітних реалізацій чатів, дуже складно знайти реалізацію, яка підходила б до абсолютно будь-якої системи. Як правило, для кожної окремої системи створюється своя реалізація чату, яка ідеально підходить для вже написаної частини. Однак розробка власної версії чату потребує як часу, так і додаткових витрат. Саме через це було вирішено розробити програму, яку буде дуже легко інтегрувати до практично будь-якої системи.

Для реалізації цієї задачі було вирішено розбити програму на декілька частин та з'єднати їх за допомогою мікросервісів. Це дозволить легко відключати кожну з частин вже готової програми, та ставити на їх місце частини вже написані частини своєї програми. Окрім цього, зроблену таким

чином програму буде легше інтегрувати, оскільки взаємодіяти потрібно буде лише з вже написаною, а отже знайомою частиною коду.

Також для універсальності було вирішено взяти базу даних, що може зберігати будь-які дані. До таких БД відносяться так звані NOSQL бази даних. Найпопулярнішою серед них є MongoDB, а отже використовувати будемо саме її.

Було прийняте рішення розробляти вебзастосунок у вигляді чату, який буде розбитий на декілька мікросервісів для більшої простоти інтеграції у програму користувача. Для зберігання даних буде використовуватись БД MongoDB.

1.5 Висновки за розділом 1

В цьому розділі були розглянуті існуючі аналоги створюваної програми, такі як Telegram, Viber, Skype, тощо, зроблене порівняння деяких аналогів з розроблюваною програмою, розглянуто поняття вебсокету та мікросервісу, та поставлено завдання до роботи. Було прийняте рішення розробляти чат створений за допомогою мікросервісів та вебсокетів який буде використовувати MongoDB у якості бази даних.

2 РОЗРОБКА АРХІТЕКТУРИ ПРОГРАМИ

2.1 Особливості мови програмування Golang

Go (часто також Golang) — багатопоточна мова програмування, що компілюється, розроблена всередині компанії Google. Розробка Go почалася у вересні 2007 року, його безпосереднім проєктуванням займалися Роберт Грізмер, Роб Пайк і Кен Томпсон, що до цього займалися проєктом розробки операційної системи Inferno. Офіційно мову було представлено у листопаді 2009 року.

Go – це одна з найбільш цікавих мов для створення вебзастосунків на сьогодні. Станом на березень 2022 року він посідає 11 місце в індексі спільноти програмування TIOBE. Все більше компаній хочуть переписати свої програми з таких мов програмування як PHP, Java, Node.js та інших мов програмування на Go і використовують його для написання нових проєктів. Багато програм які на даний момент використовуються в дуже великій кількості компаній (такі як Kubernetes, Docker та Heroku CLI) були написані на мові програмування Golang. Один з творців цієї мови, Роб Пайк, зазначає, що одна з головних речей, яка зробила Go настільки популярним – це його простота. Окрім цього в Golang є дуже проста, але водночас ефективна система паралельних обчислень. Саме завдяки цій системі паралельних обчислень Go і отримав свою першу аудиторію, яка вже пізніше розповіла про цю мову і про її переваги іншим розробникам.

Однак не тільки простота зробила Golang таким популярним. Однією з дуже важливих переваг цієї мови є також те, що пізніші версії підтримують код з більш ранніх версій. Через це компаніям не треба хвилюватись, що при оновленні версії Golang треба буде переписувати частину, або навіть весь код програми [7]. Окрім цього, gRPC (Google Remote Procedure Call) був написаний у Google, а отже у Go є вбудована підтримка цього стандарту для мікросервісів.

Окрім всього вище сказаного однією з важливих переваг Go є те, що це мова програмування, що компілюється. Через це програми, написані на цій мові, є дуже швидкими. В деяких випадках вони є навіть швидшими, ніж програми написані на мові програмування C++. Також, не дивлячись на те, що Golang – це мова з статичною типізацією, в ній є й динамічний варіант типізації. Завдяки цьому ми можемо зробити більш універсальні програми, які будуть по-різному реагувати на дані в залежності від того, який тип у цих даних.

Підсумовуючи все вище сказане вважаю, що Golang – це один з найкращих варіантів для реалізації цього проєкту. Завдяки простоті цієї мови програмування програми, написані на ній, можуть бути легко перероблені достатньо професіональними програмістами під потреби свого проєкту. Окрім цього в Golang є вбудована підтримка протоколу gRPC, що дозволяє доволі швидко розробити програму на мікросервісах.

2.2 Протокол gRPC як основний спосіб комунікації між мікросервісами

Віддалений виклик процедур (RPC, Remote Procedure Call) дозволяє взаємодіяти системним модулям, що працюють у різних адресних просторах. Використовуючи RPC, один модуль може викликати процедури з іншого модуля, отримуючи подальше повернення даних результату. RPC можна досягти, вказавши процедури, які можна викликати віддалено, і окремі вхідні та вихідні параметри в кожній процедурі. Таким чином, коли різні частини програми повинні спілкуватися, вони роблять це шляхом серіалізації даних і надсилання їх по всій мережі. Однією спільною особливістю, яка використовується в реалізації RPC для забезпечення взаємодії, є використання мов опису даних (DDL), яка є мовою для визначення структур даних, щоб дозволити взаємодію між віддаленими модулями, незалежно від платформи та мови, що використовуються в реалізації процедури [8].

gRPC було розроблено в Google у 2015 році як наступне покоління інфраструктури RPC Stubby. Він використовує транспортний протокол HTTP/2 та Protobuf у якості мови опису інтерфейсу. gRPC надає наступні функції:

- аутентифікація;
- двоспрямована потокова передача та керування потоком;
- блокуючі та неблокуючі зв'язування;
- дедлайни та відміни.

Окрім цього, через дуже велику популярність gRPC для нього написано багато клієнтських та серверних бібліотек для багатьох мов програмування. Найчастіше gRPC використовують для комунікації сервісів в мікросервісній архітектурі або для підключення клієнтських мобільних додатків до серверних [9]-[10].

Не дивлячись на велику кількість плюсів gRPC, використання протоколу HTTP/2 робить неможливою пряму реалізацію клієнта gRPC у браузері, тому тим, хто буде використовувати цей протокол, доводиться замість цього реалізовувати комунікації через написаний окремо проксі-сервер.

Protocol Buffers, або ProtoBuf, як на нього часто посилаються, – це механізм серіалізації, який використовує мову опису, щоб визначити, як структуровані дані, які підлягають серіалізації. Спочатку він був розроблений всередині Google, щоб використовувати його як основне рішення компанії для зберігання та обміну різними видами даних. ProtoBuf з моменту його початкової розробки був з відкритим кодом і тепер доступний за ліцензією безкоштовного програмного забезпечення [11].

Структури даних заздалегідь визначаються як типи повідомлень у файлах .proto у форматі, зрозумілому людині, а потім компілюються для створення вихідного коду для легкого запису та читання даних будь-якою з підтримуваних мов програмування. Google розробив ProtoBuf з урахуванням

продуктивності та простоти з повідомленнями, закодованими у двійковій форматі [12].

Отже протокол gRPC є майже ідеальним варіантом для створення мікросервісної архітектури. Також завдяки популярності цього протоколу, та тому, що він підтримується багатьма мовами програмування, його можна легко підключити до модуля, що написаний на зовсім іншій мові програмування.

2.3 Visual Studio Code як основна середа розробки проєкту

Visual Studio Code (VS Code) — це безкоштовний редактор з відкритим вихідним кодом для розробки та налагодження сучасних хмарних і вебдодатків, який доступний безкоштовно в Linux, OS X і Windows. VS Code підтримує понад 30 різних мов програмування, розмітки та баз даних, серед яких є JavaScript, C#, C++, PHP, Java, HTML, R, CSS, SQL, Markdown, TypeScript, Less, Sass, JSON, XML та Python [13].

Завдяки підтримці багатьох мов програмування, плагінів та можливості безкоштовного використання, VSCode є одним з найбільш популярних редакторів коду на сьогоднішній час. Для розробки на мові програмування Golang в VSCode присутній офіційний плагін від Google, який надає можливості для тестування, налагодження коду та багато чого іншого для комфортної розробки на цій мові програмування. Окрім цього на Visual Studio Code можна поставити й багато інших плагінів для того, щоб зробити процес розробки ще більш зручним. Це можуть бути плагіни для роботи з контейнерами Docker, або плагіни для автодоповнення коду.

З огляду на все вищесказане VSCode є досить непоганим вибором для розробки на ньому програм на мові програмування Go. Завдяки своїй бібліотеці плагінів кожен може доробити його під себе і свої потреби.

2.4 RabbitMQ у якості основного брокера повідомлень

RabbitMQ – це брокер обміну повідомленнями з відкритим кодом, який реалізує протокол AMQP (Advanced Message Queuing Protocol). Останні кілька років його популярність невпинно зростає. Спочатку він використовувався найсміливішими компаніями, а зараз багато хто відкриває не тільки особливі переваги RabbitMQ, а й позитивний вплив використання обміну повідомленнями в розробці програмного забезпечення. Дійсно, з появою хмарних обчислень потреба в розробці та створенні систем для масштабування стала більш актуальною. Вибираючи слабко зв'язані архітектури, пов'язані між собою повідомленнями, що передаються через брокерів, таких як RabbitMQ, інженери програмного забезпечення можуть задовольнити потреби сучасної розробки додатків [14]-[15].

Завдяки тому, що RabbitMQ є дуже гнучким, ми можемо настроїти його так, як потрібно для реалізації функціоналу поточного проєкту. Через це RabbitMQ є досить непоганим вибором для цього проєкту.

2.5 Розробка архітектури програми

Проведений аналіз аналогів програми показав, що вона повинна відповідати наступним функціональним вимогам:

- реєстрація нових користувачів чату;
- вхід за логіном та паролем;
- перевірка дозволів користувача за токеном;
- отримання списку користувачів;
- отримання інформації про користувача за його ідентифікатором;
- оновлення інформації користувача;
- видалення користувача;
- створювання нових кімнат користувачем;
- додавання інших користувачів до кімнат;

- видалення користувачів з кімнати;
- отримання списку кімнат в яких присутній користувач;
- оновлення інформації о кімнаті;
- видалення кімнати;
- отримання повідомлень кімнати;
- створення нових повідомлень;
- редагування існуючих повідомлень;
- видалення повідомлень.

За даними вимогами була розроблена діаграма, що демонструє можливості користувачів по роботі з програмою. Ця діаграма представлена на рисунку 2.1.



Рисунок 2.1 – Діаграма прецедентів користувача

Загалом програма буде складатись з 5 частин. Кожна з них повинна відповідати за один окремий блок роботи з програмою та виконувати лише те, для чого вона призначена. Всі блоки та їх призначення представлені у таблиці 2.1.

Таблиця 2.1 – Складові програми

Назва блоку	Опис
Api-gateway	Цей блок є сполучним між усіма іншими блоками. Саме він відповідає за комунікацію між іншими блоками програми.
Auth	Цей блок відповідає за реєстрацію користувачів та перевірку валідності токену користувача.
Messages	Цей блок відповідає за роботу з повідомленнями користувачів.
Room	Цей блок відповідає за роботу з кімнатами чату, а також за створення зв'язків між кімнатою та користувачами, що мають до неї доступ
User	Цей блок відповідає за роботу з акаунтами користувачів.

Більшість з цих блоків працюють через протокол gRPC та передають через нього свою інформацію. Однак gRPC може використовуватися лише тоді коли нам потрібно отримувати запити в форматі запит-відповідь. Для сервісу Messages такий підхід не є вірним тому, що створене повідомлення треба відправити всім користувачам у кімнаті. Через це для коректної роботи сервісу нам потрібно використовувати брокер повідомлень. Для цих цілей й був узятий RabbitMQ. Загальна схема зв'язків між блоками представлена на рисунку 2.2.

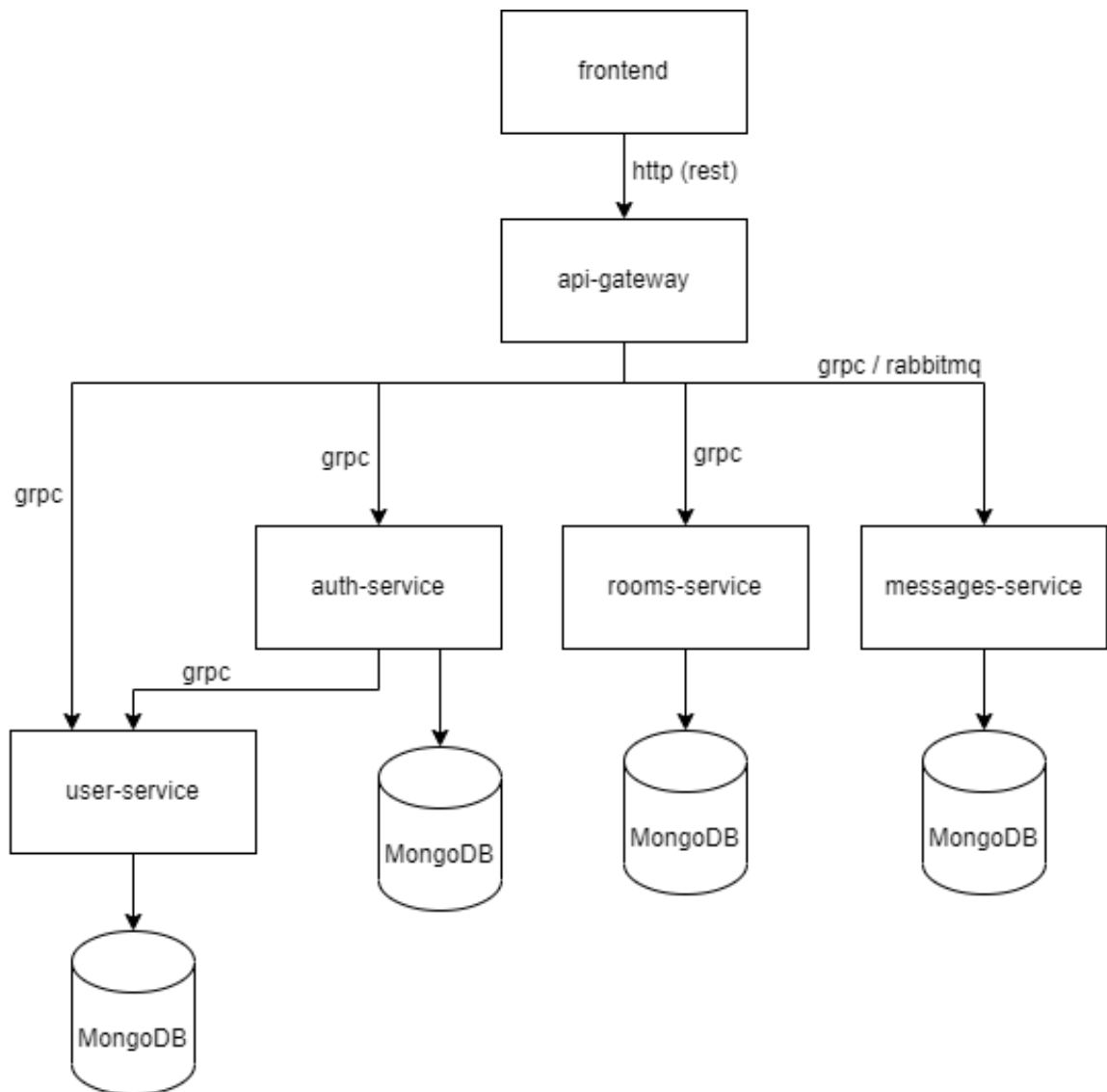


Рисунок 2.2 – Загальна схема зв'язків між блоками програми

2.6 Проєктування бази даних

У якості бази даних було вирішено обрати документоорієнтовану систему керування базами даних MongoDB. Вона була обрана через те, що ця БД може зберігати будь-які дані та є доволі популярною, а отже й багато розробників мають досвід у роботі з нею.

Всього у БД буде зберігатися 5 різних таблиць: User, Room_user, Room, Messages та Auth. Структура всіх таблиць показана на рисунку 2.3.

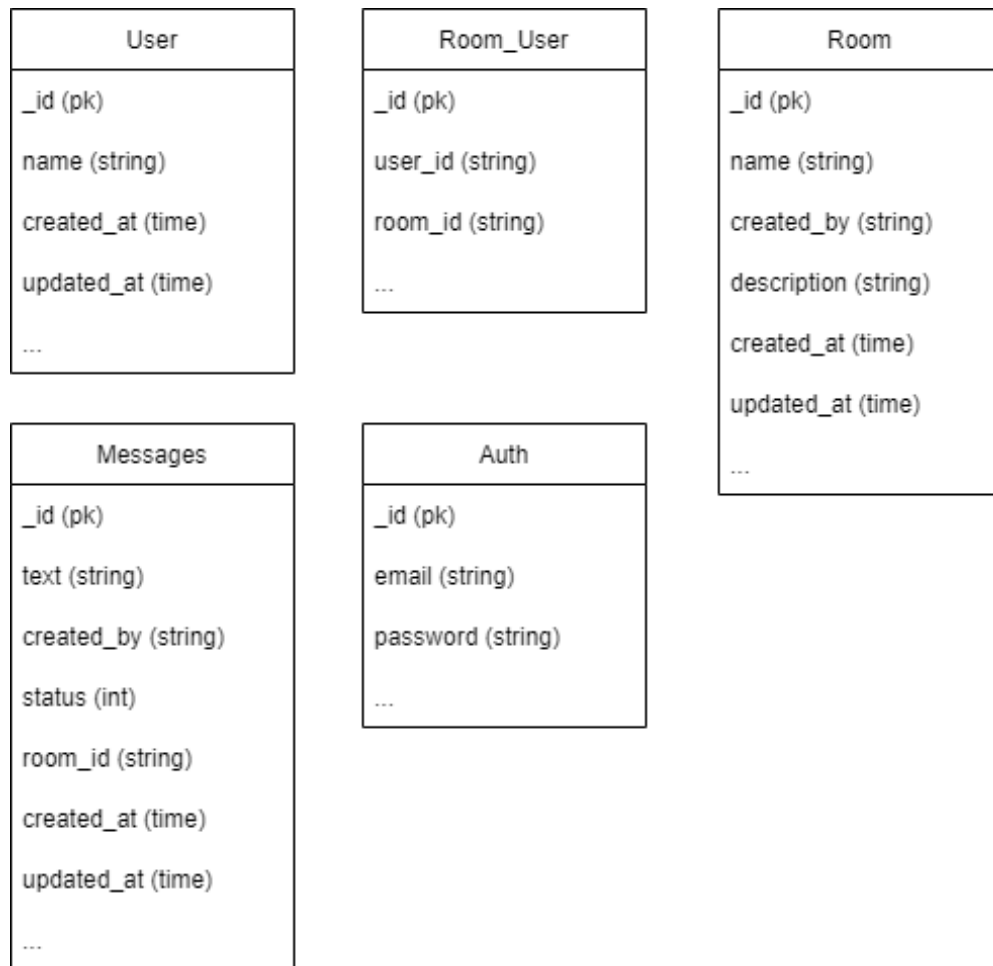


Рисунок 2.3 – Загальна структура всіх таблиць у БД

2.7 Висновки за розділом 2

Було розглянуто особливості мови програмування Golang, принципи роботи протокола gRPC, проаналізовано, як він використовується при написанні мікросервісної архітектури, оглянуто функціонал та можливості середі розробки Visual Studio Code. Також було проаналізовано можливості брокера RabbitMQ і зроблено висновки, як його простота в настройці може допомогти у створенні проєкту. Описано архітектуру програми, розроблено схему таблиць БД та розроблено загальну схему зв'язків між мікросервісами проєкту.

З ОСНОВНІ РІШЕННЯ ЩОДО РЕАЛІЗАЦІЇ КОМПОНЕНТІВ СИСТЕМИ

3.1 Функціонал блоку Api-gateway

Api-gateway складається здебільшого з серверу та клієнтів для інших блоків. Окрім цього, там знаходиться підмодуль «з'єднувач». Він реалізує функціонал обробки повідомлень, що приходять через вебсокет. Загалом схема роботи api-gateway представлена на наступних рисунках.

На рисунку 3.1 представлена схема роботи підмодулю з'єднувача, а точніше його частини, що відповідає за отримання повідомлень з фронтенду та їх подальшу обробку за типом події, що відбулася.

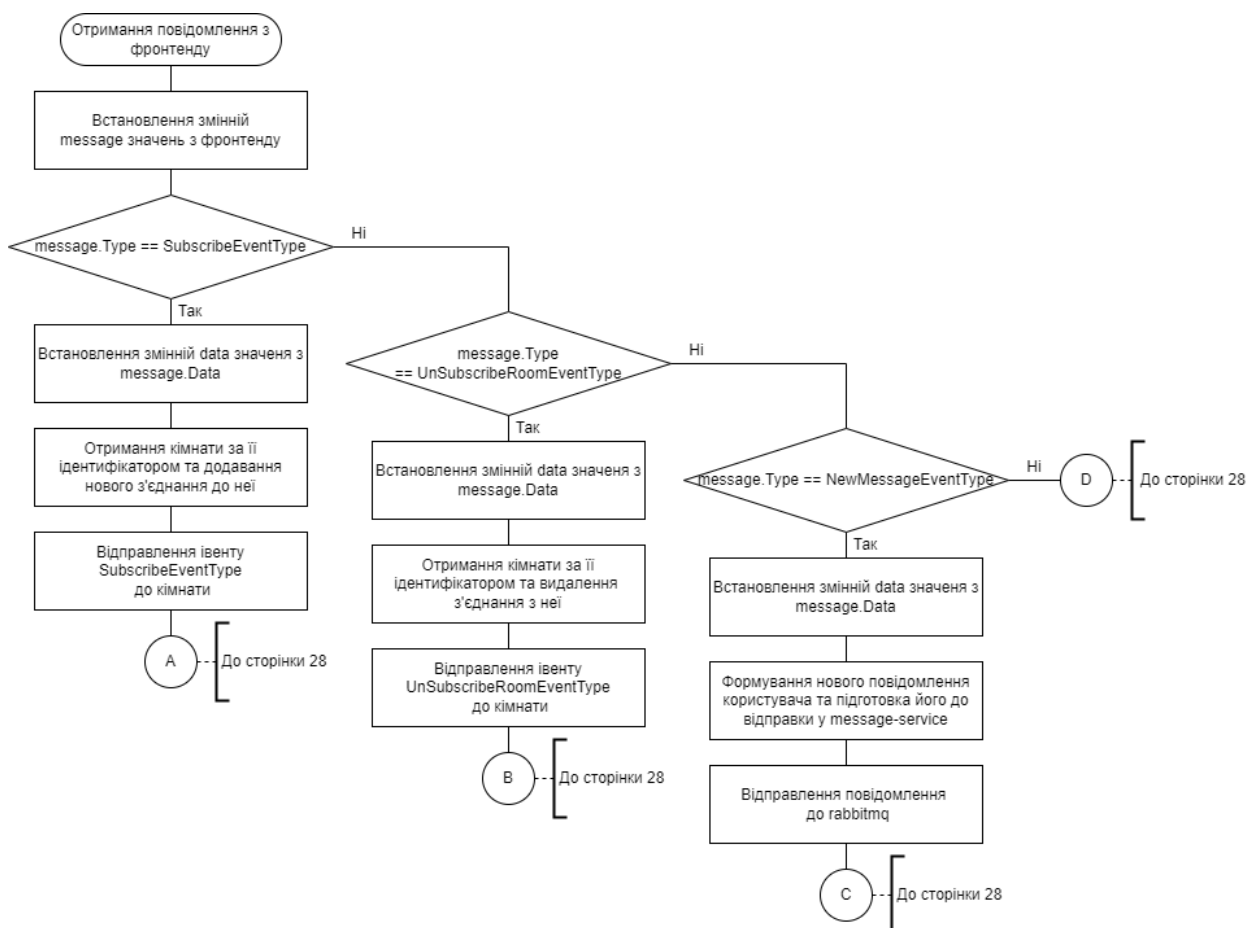


Рисунок 3.1 – Схема роботи «з'єднувача», аркуш 1

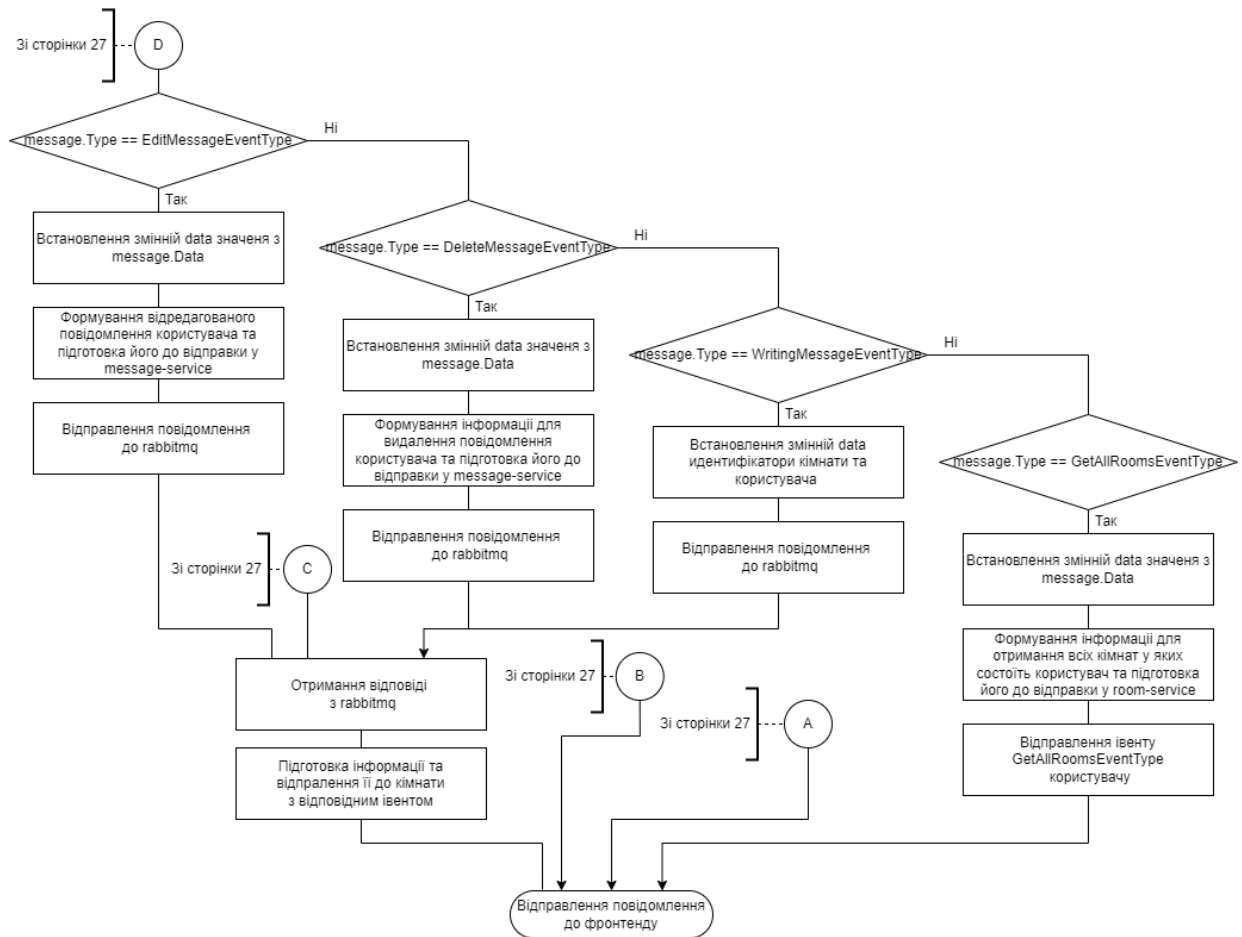


Рисунок 3.1, аркуш 2

На рисунку 3.2 представлена UML діаграма, яка показує шлях виконання запиту на додавання користувача до кімнати.

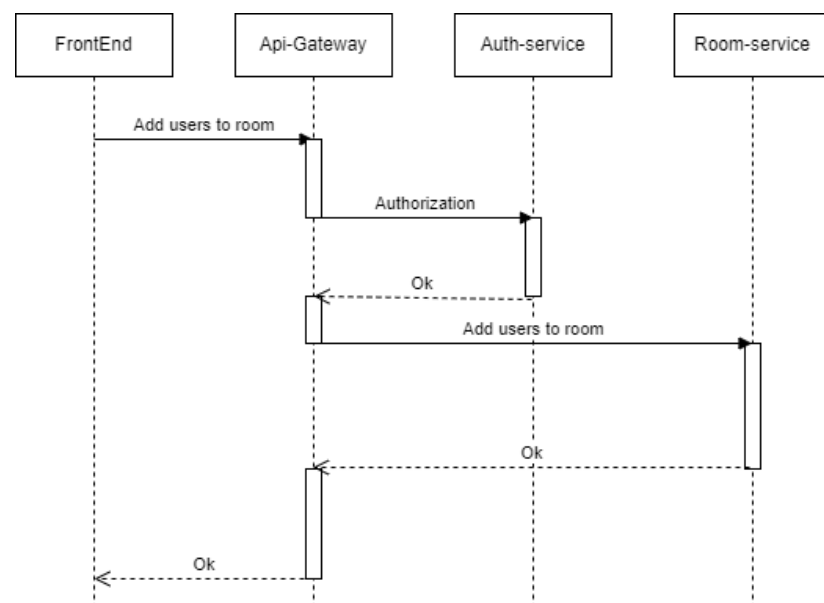


Рисунок 3.2 – Схема додавання користувачів до кімнати

На рисунку 3.3 представлена UML діаграма, яка показує шлях виконання запиту на створення нових кімнат.

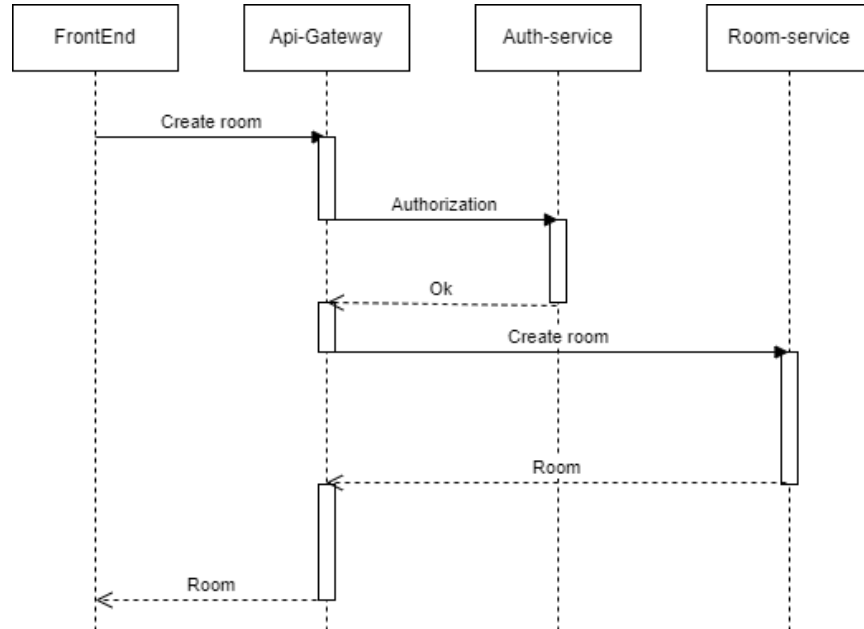


Рисунок 3.3 – Схема додавання нових кімнат

На рисунку 3.4 представлена UML діаграма, яка показує шлях виконання запиту на видалення кімнати.

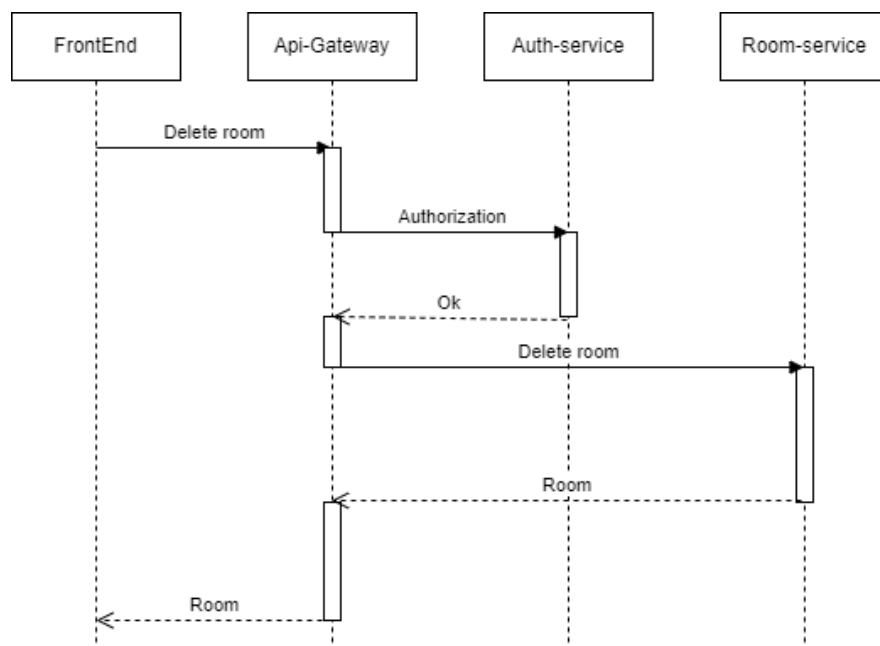


Рисунок 3.4 – Схема видалення кімнат

На рисунку 3.5 представлена UML діаграма, яка показує шлях виконання запиту на видалення користувачів.

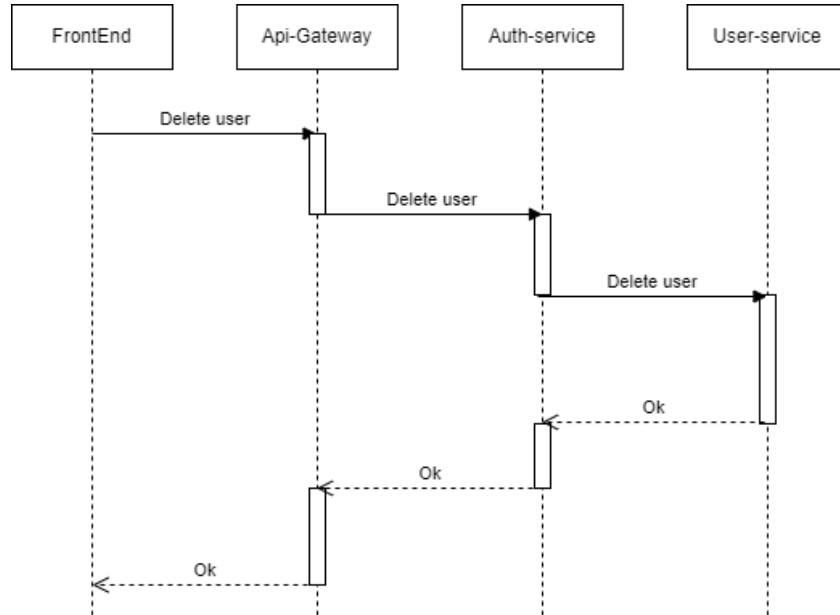


Рисунок 3.5 – Схема видалення користувачів

На рисунку 3.6 представлена UML діаграма, яка показує шлях виконання запиту на видалення користувачів з кімнат.

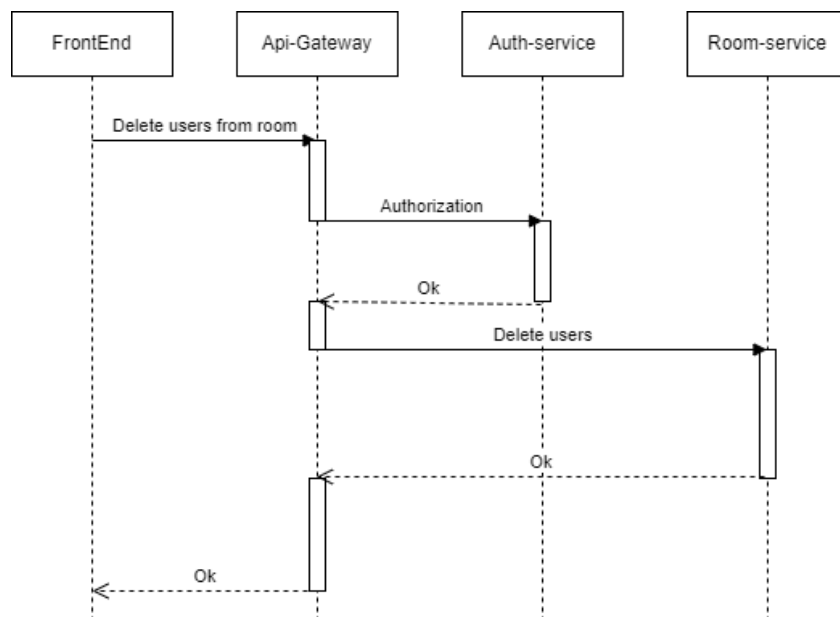


Рисунок 3.6 – Схема видалення користувачів з кімнат

На рисунку 3.7 представлена UML діаграма, яка показує шлях виконання запиту на отримання повідомлень кімнати.

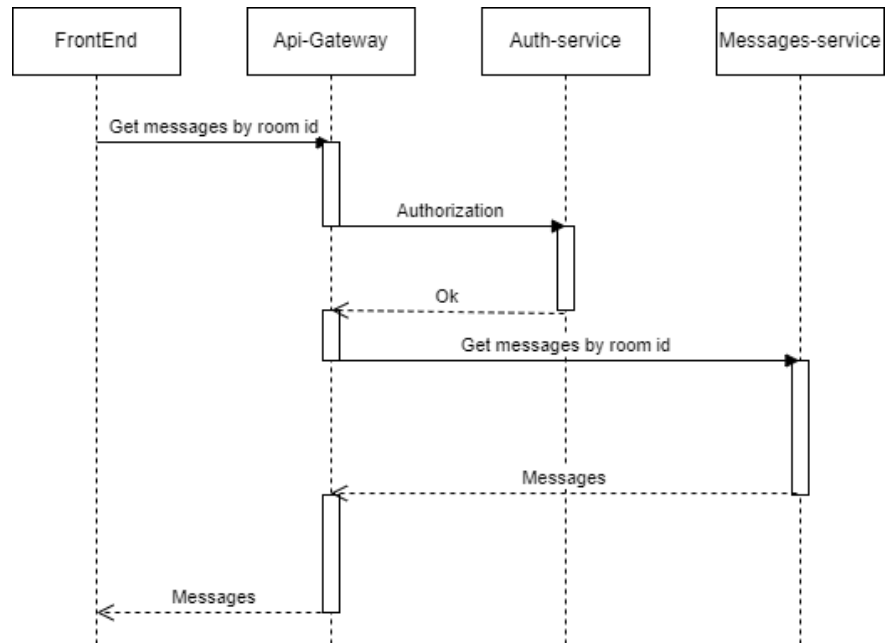


Рисунок 3.7 – Схема отримання повідомлень кімнати

На рисунку 3.8 представлена UML діаграма, яка показує шлях виконання запиту на отримання кімнат користувача.

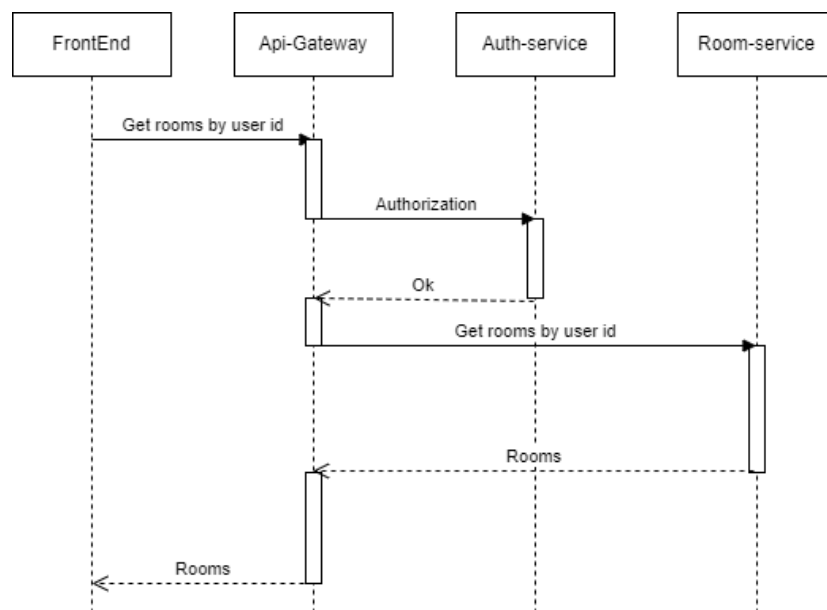


Рисунок 3.8 – Схема отримання кімнат користувача

На рисунку 3.9 представлена UML діаграма, яка показує шлях виконання запиту на отримання користувача за ідентифікатором.

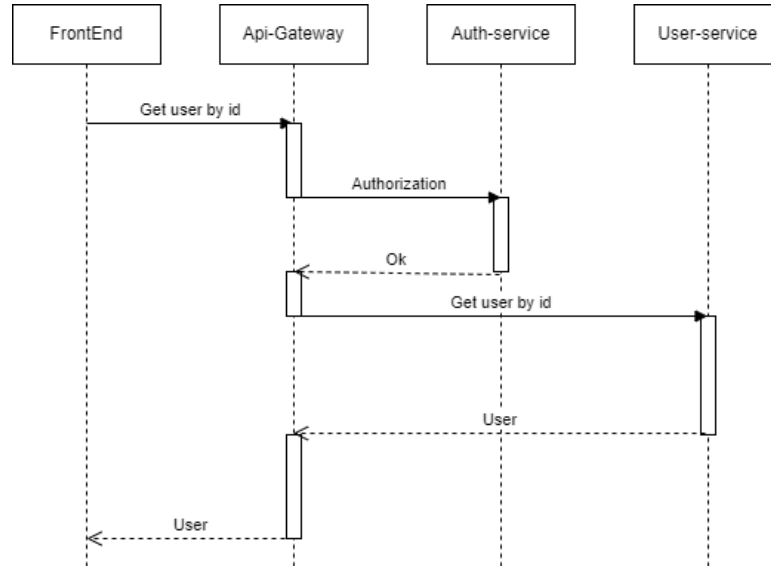


Рисунок 3.9 – Схема отримання користувача за ідентифікатором

На рисунку 3.10 представлена UML діаграма, яка показує шлях виконання запиту вхід користувача до сервісу.

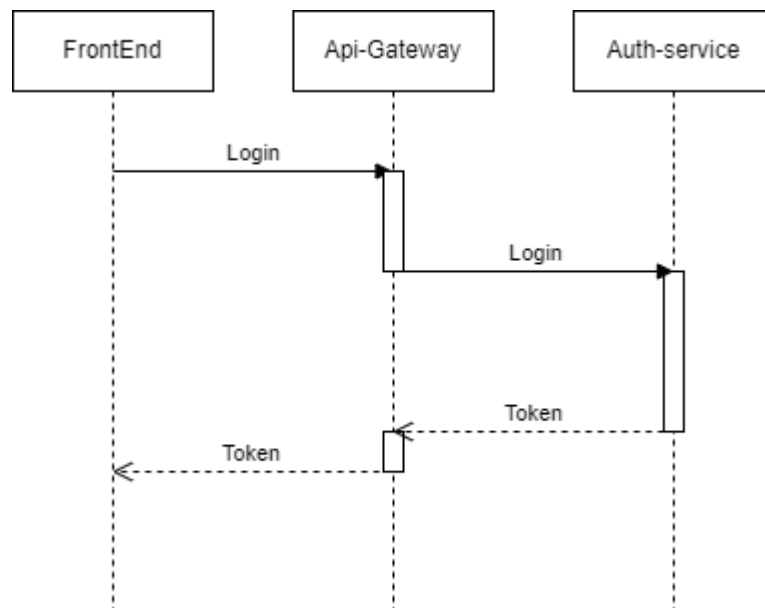


Рисунок 3.10 – Схема входу користувача до сервісу

На рисунку 3.11 представлена UML діаграма, яка показує шлях виконання запиту реєстрацію користувача.

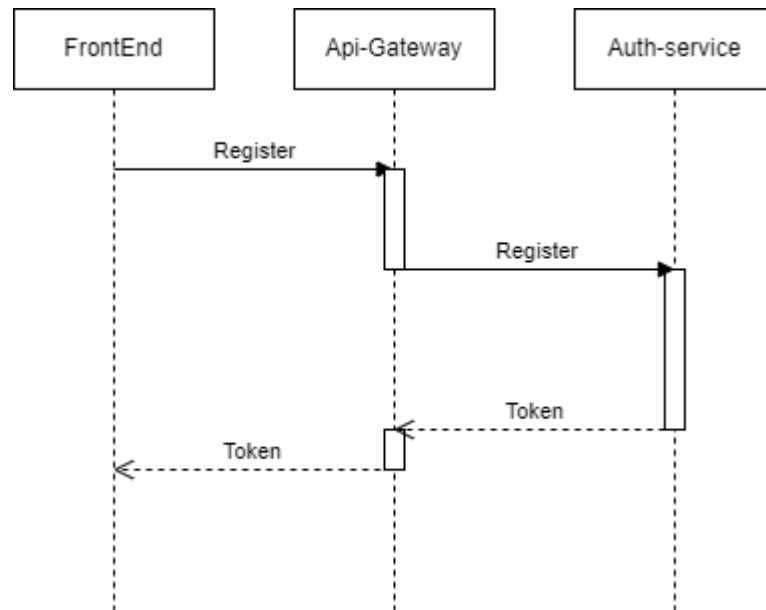


Рисунок 3.11 – Схема реєстрації користувача

На рисунку 3.12 представлена UML діаграма, яка показує шлях виконання запиту на оновлення інформації кімнати.

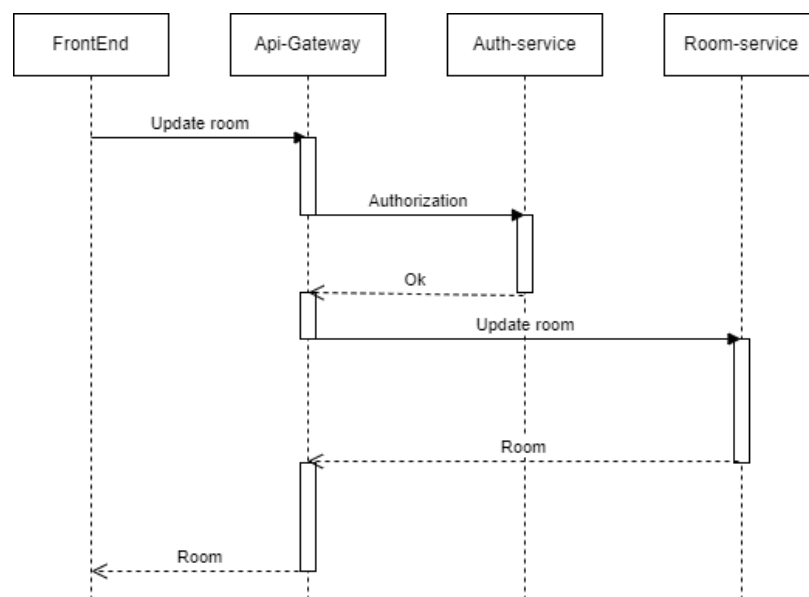


Рисунок 3.12 – Схема оновлення інформації кімнати

На рисунку 3.13 представлена UML діаграма, яка показує шлях виконання запиту на оновлення інформації користувача.

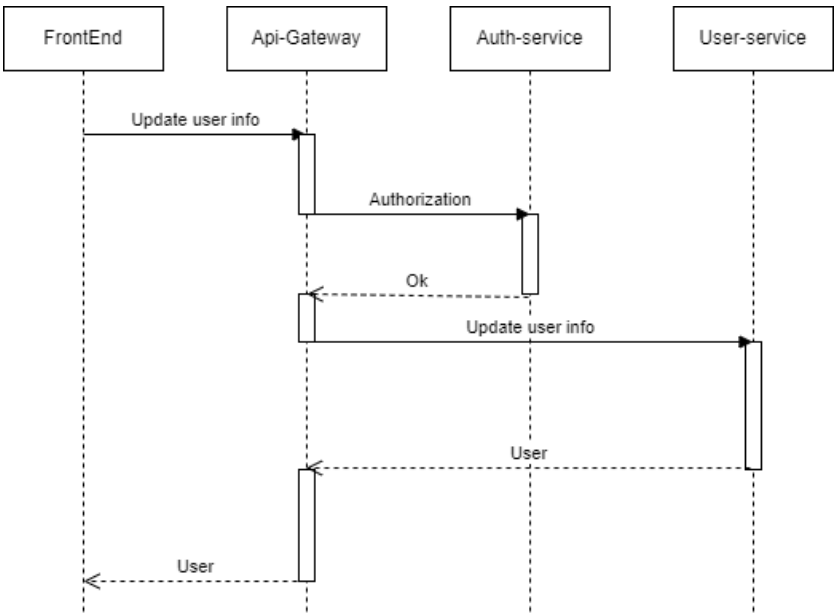


Рисунок 3.13 – Схема оновлення інформації користувача

3.2 Функціонал блоку Auth

Блок auth складається з чотирьох функцій, що виконують базовий функціонал для аутентифікації користувачів. Назви функцій та їх опис представлені у таблиці 3.1.

Таблиця 3.1 – Складові блоку Auth

Назва функції	Опис
1	2
Authentication	Відповідає за перевірку валідності токена. На вході ця функція приймає токен. На виході вона повертає ID користувача
Delete user	Відповідає за видалення даних користувача. На вході приймає ID користувача. На виході вона не віддає нічого.

Продовження таблиці 3.1

1	2
Login	Відповідає за створення токена для вже існуючого користувача. Окрім цього перевіряє коректність паролю та логіну. На вході приймає логін і пароль. На виході віддає токен.
Registration	Відповідає за створення нового користувача. Пароль нового користувача хешується за допомогою бібліотеки bcrypt. Приймає на вхід інформацію о користувачі. На виході віддає токен.

3.3 Функціонал блоку Messages

Блок messages складається з чотирьох функцій, що виконують базовий функціонал для роботи з повідомленнями користувачів. Більша частина функцій спілкується з другими блоками через RabbitMQ.

Назви функцій, їх опис та метод спілкування з іншими блоками представлені у таблиці 3.2.

Таблиця 3.2 – Складові блоку Messages

Назва функції	Метод спілкування	Опис
1	2	3
createMessage	RabbitMQ	Відповідає за збереження нових повідомлень у БД. На вхід приймає повідомлення і віддає його ж при успіху.

Продовження таблиці 3.2

1	2	3
updateMessage	RabbitMQ	Відповідає за оновлення повідомлення у БД. На вхід приймає повідомлення і віддає його ж при успіху.
deleteMessage	RabbitMQ	Відповідає за видалення повідомлення з БД. На вхід приймає ID повідомлення і віддає його ж при успіху.
GetMessagesByRoom	gRPC	Відповідає за отримання повідомлень кімнати. На вхід отримує ID кімнати, ліміт повідомлень та відступ. На виході віддає повідомлення.

3.4 Функціонал блоку Room

Блок room складається з семи функцій, що виконують базовий функціонал для роботи з кімнатами. Назви функцій та їх опис представлені у таблиці 3.3.

Таблиця 3.3 – Складові блоку Room

Назва функції	Опис
1	2
GetRoom	Відповідає за перевірку валідності токена. На вході ця функція приймає ID кімнати. На виході вона повертає кімнату.
CreateRoom	Відповідає за збереження нових кімнат у БД. На вхід приймає інформацію о кімнаті і віддає кімнату при успіху.

Продовження таблиці 3.3

1	2
DeleteRoom	Відповідає за видалення кімнати з БД. На вхід приймає ID повідомлення. На виході не видає нічого.
UpdateRoom	Відповідає за оновлення даних кімнати у БД. На вхід приймає кімнату і віддає її ж при успіху.
AddUsers	Відповідає за додання користувачів у кімнату. На вхід приймає ID користувачів. На виході не видає нічого.
DeleteUsers	Відповідає за видалення користувачів з кімнати. На вхід приймає ID користувачів. На виході не видає нічого.
GetAllRooms	Відповідає за отримання кімнат у яких наявний користувач. На вхід отримує ID користувача, ліміт кімнат та відступ. На виході віддає кімнати.

3.5 Функціонал блоку User

Блок user складається з п'яти функцій, що виконують базовий функціонал для роботи з інформацією про користувачів. Даний блок не зберігає у собі паролі користувачів. Назви функцій та їх опис представлені у таблиці 3.4.

Таблиця 3.4 – Складові блоку User

Назва функції	Опис
1	2
GetUserByID	Відповідає за перевірку валідності токenu. На вході ця функція ID користувача. На виході вона повертає користувача

Продовження таблиці 3.4

1	2
GetUsers	Відповідає за видалення даних користувача. На вході приймає нічого. На виході вона віддає всіх користувачів у БД.
CreateUser	Відповідає за створення нового користувача у БД. На вхід приймає користувача і віддає його ж при успіху.
UpdateUser	Відповідає за оновлення користувача у БД. На вхід приймає користувача і віддає його ж при успіху.
DeleteUser	Відповідає за видалення користувачів з кімнати. На вхід приймає ID користувача. На виході не видає нічого.

3.6 Інтерфейс програми

Інтерфейс програми являє собою html сторінку з JavaScript кодом. Ця сторінка автоматично відправляється користувачу при запиті до сервера. Вона складається з 2 сторінок: сторінки входу користувача та сторінки з чатами. Зовнішній вигляд інтерфейсу представлений на рисунках 3.14-3.15.

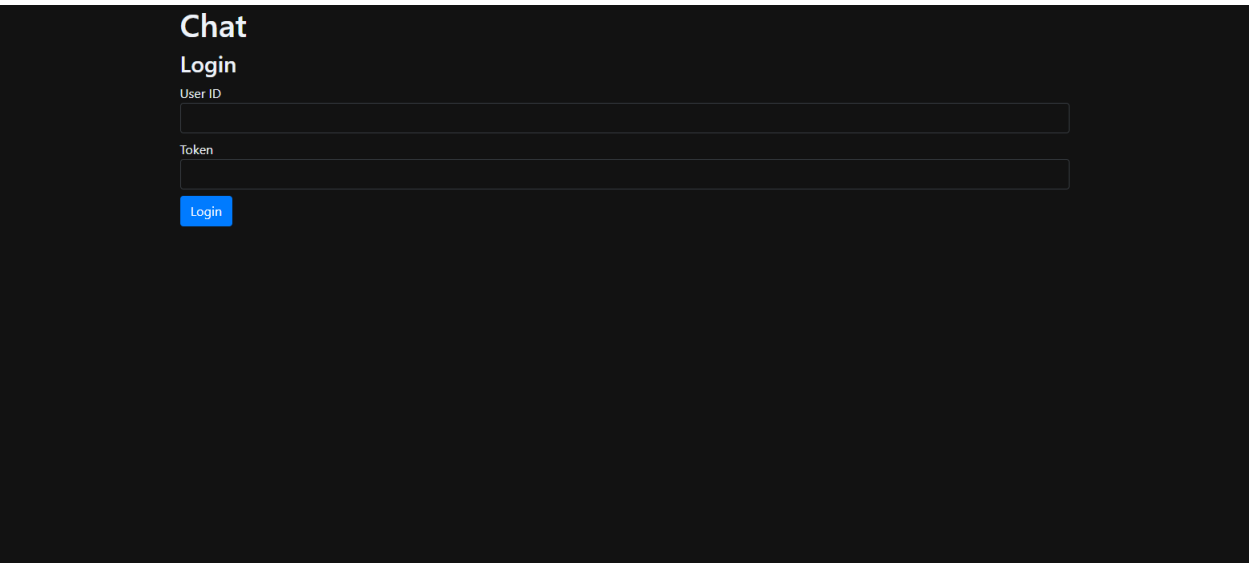


Рисунок 3.14 – Сторінка входу в аккаунт

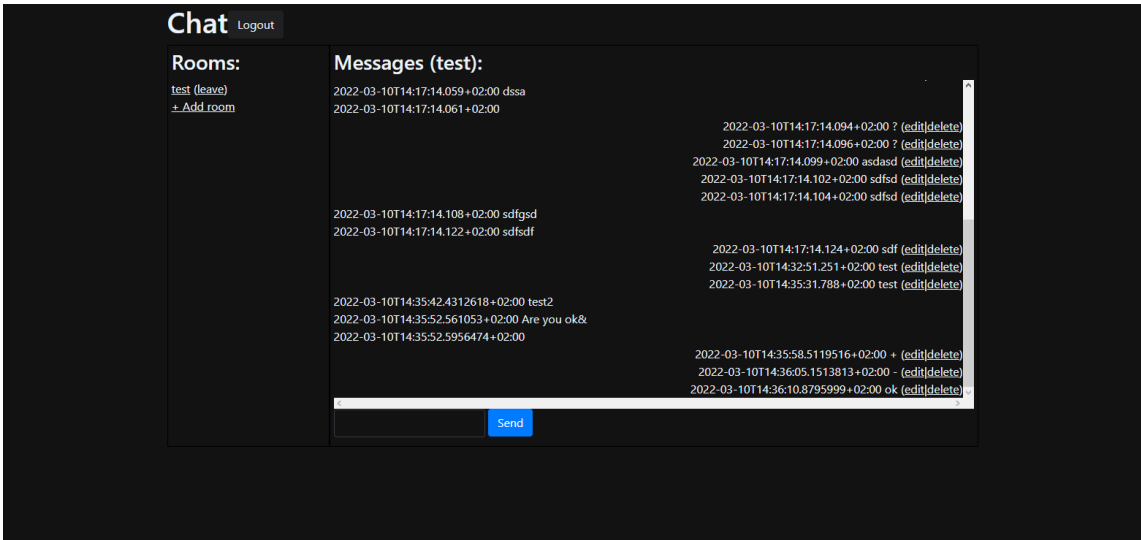


Рисунок 3.15 – Головна сторінка

3.7 Загальний алгоритм функціонування програми

Підсумовуючи алгоритми роботи кожного з окремих модулів було створено загальний алгоритм функціонування програми, який продемонстровано на рисунку 3.16.

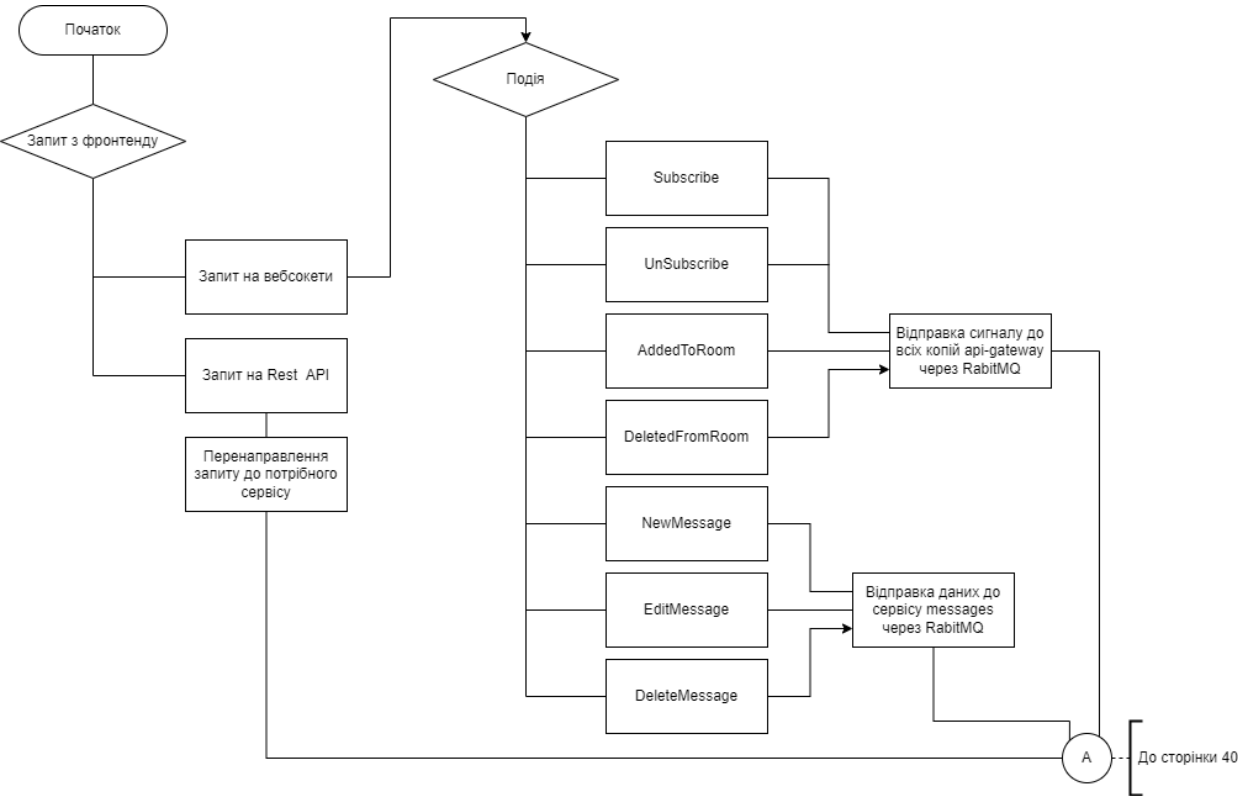


Рисунок 3.16 – Загальний алгоритм функціонування програми, аркуш 1



Рисунок 3.16, аркуш 2

3.8 Висновки за розділом 3

В розділі було розглянуто основні рішення щодо реалізації програми, наведено детальний список розроблених мікросервісів та їх базовий функціонал. Окрім цього, було запропоновано інтерфейс програми.

4 ЕКСПЛУАТАЦІЯ, ТЕСТУВАННЯ ТА ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ПРОГРАМИ

4.1 Опис застосування

Цей вебзастосунок призначений для реалізації чату в реальному часі за допомогою мікросервісів та вебсокетів на мові програмування Golang. Він може використовуватися у вже готових проєктах для додавання до них можливостей по написанню коментарів або чату в реальному часі. Однак, не зважаючи на це, окрім використання цього проєкту як додаткового модулю, також можливо використовувати його у якості незалежного застосунку для, наприклад, спілкування робітників у корпоративній мережі.

Крім того, завдяки тому, що цей проєкт розміщений у репозиторії GitHub, будь-який розробник може використовувати його у власних проєктах, або створити на основі цього проєкту свою модифікацію.

Для використання проєкту потрібен комп'ютер, що здатен запускати та підтримувати контейнери Docker. Окрім цього може знадобитися встановлена мова програмування Golang версії не меншої за 1.15. Для запуску проєкту використовувався персональний IBM-сумісний комп'ютер з процесором Intel Core i5, оперативною пам'яттю 8 Гб, 14'' монітором, клавіатурою та маніпулятором типу «миша», що функціонує під управлінням операційної системи Microsoft Windows 10.

4.2 Інструкція з експлуатації програми

4.2.1 Керівництво системного програміста

Перед спробою запуску програми потрібно перевірити коректність налаштувань та створити .env файл для кожного з блоків. Приклади файлів знаходяться у відповідних папках під назвою .env.example. Для створення .env файлу достатньо перейменувати файл .env.example у .env та змінити

налаштування у самому файлі відкривши його через будь-який текстовий редактор.

Для коректного запуску програми потрібно окремо компілювати та запускати кожен з блоків. Це також треба мати на увазі при спробі запустити проєкт через Docker.

Протестувати програму можна завдяки вбудованим засобам тестування Golang, через програму Postman або її аналоги, або через спрощений інтерфейс, що надається разом з програмою.

Для коректної роботи програми окрім запуску самої програми потрібно ще запустити Docker-контейнери з MongoDB та RabbitMQ.

4.2.2 Керівництво програміста

Для розробки проєкту використовувалося IDE Microsoft Visual Studio Code. Для подальшої розробки або переробки цієї програми рекомендовано використовувати саме її, оскільки програма вже містить всі необхідні налаштування для роботи з цією IDE.

Загалом проєкт складається з 8 папок: “.vscode”, “api-gateway”, “auth”, “deploy”, “messages”, “proto”, “room”, “user”. Окрім цього в корені є 3 файли: “.gitignore”, “go.mod”, “go.sum”. Вся структура представлена на рисунку 4.1.

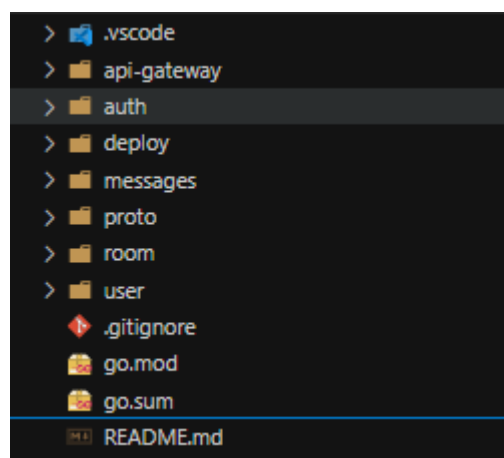


Рисунок 4.1 – Структура кореневої папки програми

В таблиці 4.1 представлений детальний опис елементів кореневої папки проєкту.

Таблиця 4.1 – Опис елементів кореневої папки проєкту

Назва елементу	Опис
.vscode	В цій папці знаходяться налаштування для запуску програми у режимі налагоджування
api-gateway	В цій папці знаходиться елементи блоку api-gateway
auth	В цій папці знаходиться елементи блоку auth
deploy	В цій папці знаходиться файли, які потрібні для запуску необхідних для роботи програми в цілому, контейнерів.
messages	В цій папці знаходиться елементи блоку messages
proto	В цій папці знаходяться proto-файли та їх скомпільовані версії. Вони потрібні програмі для спілкування через протокол gRPC
room	В цій папці знаходиться елементи блоку room
user	В цій папці знаходиться елементи блоку user
.gitignore	Цей файл зберігає в собі назви та шаблони файлів, що не повинні потрапити у GitHub
go.mod	В цьому файлі знаходяться залежності проєкту від інших бібліотек. Окрім цього там зберігається системна інформація щодо назви проєкту та версії Golang
go.sum	В цьому файлі знаходяться хеші залежностей проєкту від інших бібліотек

Розглянемо папку `api-gateway`. В цій папці містяться наступні папки та елементи:

- папка `clients` містить у собі функції для роботи з rest запитамі. Також саме функції з цієї папки взаємодіють з іншими блоками програми;
- папка `config` містить у собі структури для збереження налаштувань користувача з `.env` файлу. Окрім цього, там можна знайти функції для ініціалізації потрібних для роботи цієї частини програми бібліотек (наприклад, зв'язок з RabbitMQ або MongoDB);
- папка `connector` містить функції для роботи з вебсокетами. Вона слугує для спілкування користувача з сервером у реальному часі;
- папка `model` містить у собі структури, які можуть змінюватися кінцевими користувачами під свої потреби;
- папка `public` містить файл для демонстрації можливостей програми через інтерфейс;
- папка `server` включає функції, потрібні для роботи серверу та вебсокетів. Такими функціями є, наприклад, `middleware`;
- в папці `tools` знаходяться додаткові функції необхідні для роботи програми.

Розглянемо папку `auth`. В цій папці містяться наступні папки та елементи:

- папка `auth`. Ця папка містить у собі основні інструменти для роботи з базою даних для реалізації основного функціоналу блоку `auth`. Там знаходяться функції для отримування та обробки повідомлень з gRPC;
- папка `config`. Ця папка містить у собі структури для збереження налаштувань користувача з `.env` файлу. Окрім цього там можна знайти функції для ініціалізації потрібних для роботи цієї частини програми бібліотек (наприклад зв'язок з RabbitMQ або MongoDB);
- в папці `utils` знаходяться додаткові функції необхідні для роботи програми.

Розглянемо папку `messages`. В цій папці містяться наступні папки та елементи:

- папка `messages` містить у собі основні інструменти для роботи з базою даних для реалізації основного функціоналу блоку `messages`. Там знаходяться функції для отримувannya та обробки повідомлень з gRPC;
- папка `config` містить у собі структури для збереження налаштувань користувача з `.env` файлу. Окрім цього, там можна знайти функції для ініціалізації потрібних для роботи цієї частини програми бібліотек (наприклад, зв'язок з RabbitMQ або MongoDB).

Розглянемо папку `proto`. В цій папці містяться `.proto` файли, які необхідні для реалізації спілкування протоколом gRPC. Вони описують, якими мають бути повідомлення, що відправляються через gRPC.

Розглянемо папку `room`. В цій папці містяться наступні папки та елементи:

- папка `room` містить у собі основні інструменти для роботи з базою даних для реалізації основного функціоналу блоку `room`. Там знаходяться функції для отримувannya та обробки повідомлень з gRPC;
- папка `config` містить у собі структури для збереження налаштувань користувача з `.env` файлу. Окрім цього, там можна знайти функції для ініціалізації потрібних для роботи цієї частини програми бібліотек (наприклад зв'язок з RabbitMQ або MongoDB).

Розглянемо папку `user`. В цій папці містяться наступні папки та елементи:

- папка `user` містить основні інструменти для роботи з базою даних для реалізації основного функціоналу блоку `user`. Там знаходяться функції для отримувannya та обробки повідомлень з gRPC;
- папка `config` містить у собі структури для збереження налаштувань користувача з `.env` файлу. Окрім цього, там можна знайти функції для ініціалізації потрібних для роботи цієї частини програми бібліотек (наприклад зв'язок з RabbitMQ або MongoDB).

4.2.3 Керівництво оператора

Після запуску програми необхідно перейти за адресою, яка вказана у файлах конфігурації (за замовчуванням це 127.0.0.1:16568). Після цього користувач побачить вікно авторизації, наведене на рисунку 4.2.

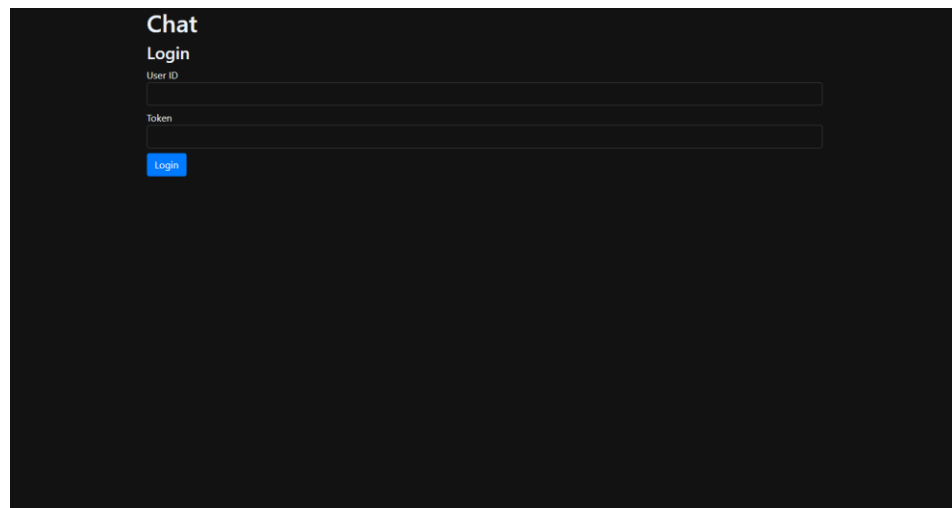


Рисунок 4.2 – Вікно авторизації програми

В цьому вікні необхідно ввести ідентифікатор користувача та його токен. Ці дані можливо отримати через запит створення або входу користувача, який необхідно надіслати до серверу. Після цього користувач побачить головний інтерфейс програми представлений на рисунку 4.3.

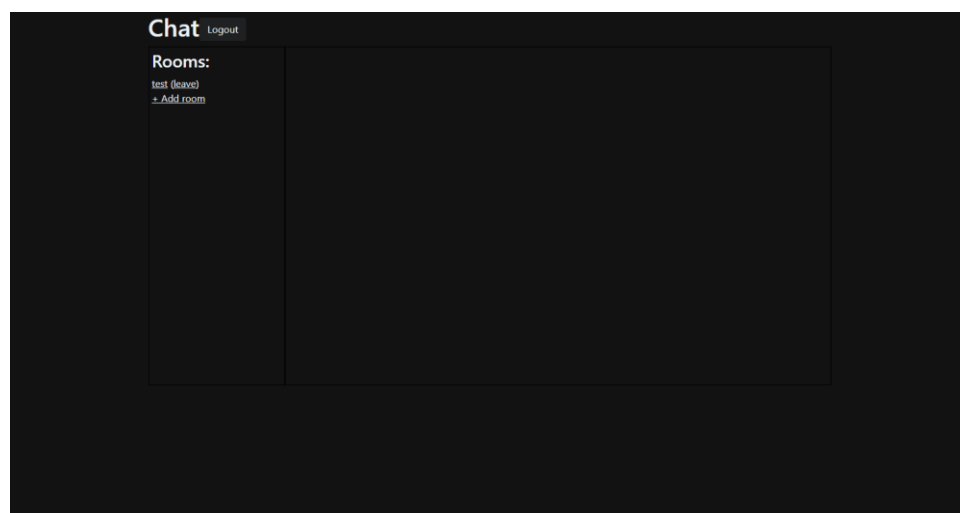


Рисунок 4.3 – Головний інтерфейс програми

Далі необхідно обрати одну з існуючих кімнат на натиснути на неї, або створити нову кімнату. Після цього у правій частині сторінки чату ми побачимо інтерфейс чату, наведений на рисунку 4.4.

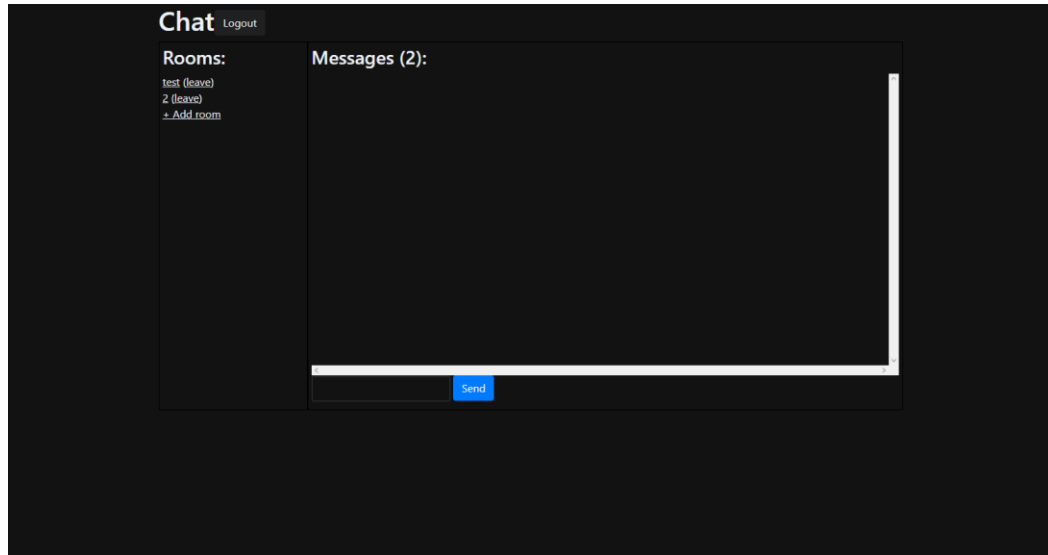


Рисунок 4.4 – Вигляд вікна чату

Приклад спілкування з іншими користувачами у цій кімнаті наведений на рисунку 4.5.

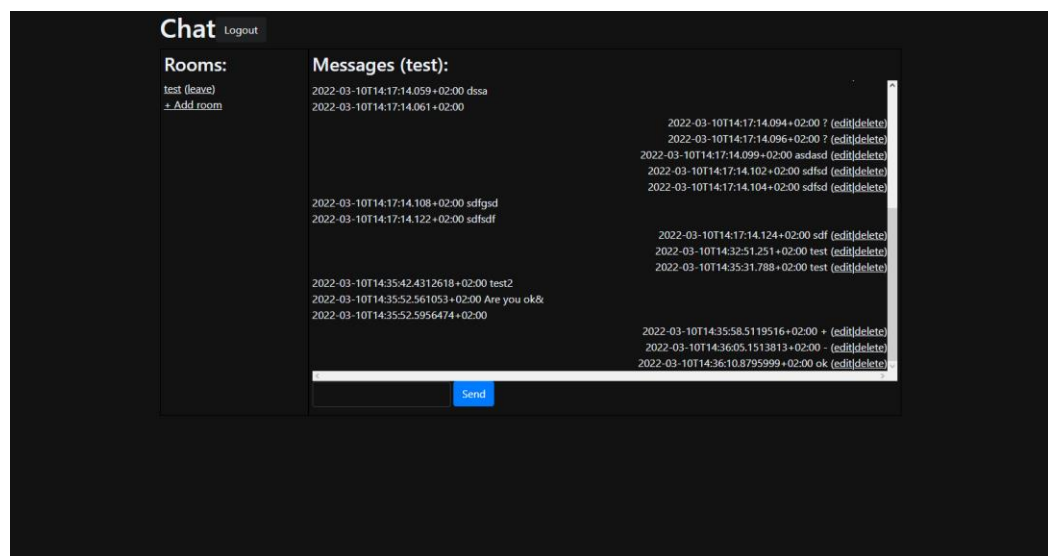


Рисунок 4.5 – Спілкування користувачів сервісу

4.3 Висновки за розділом 4

В розділі було розглянуто призначення та способи застосування розробленого вебзастосунку, описані системні вимоги, надані інструкції з експлуатації програми для системного програміста, програміста та користувача.

ВИСНОВКИ

В даній роботі були вирішені наступні завдання:

- проаналізовані аналоги створюваного вебзастосунку;
- досліджені технології, що планувалися для використання;
- вивчена взаємодія з мовою програмування Golang;
- досліджені методи роботи з вебсокетами;
- проаналізовані способи спілкування та створення мікросервісів;
- створена реалізація чату у реальному часі на мікросервісах;
- протестовано створений вебзастосунок.

В результаті проведеної роботи було створено програмну реалізацію чату у реальному часі на мікросервісах.

Мікросервіси є зручним способом зробити легко розширюване програмне забезпечення завдяки тому, що програма поділяється на декілька незалежних одна від одної частин, а також вони допомагають полегшити розуміння коду іншими програмістами.

Завдяки gRPC можна дуже легко створити простий та швидкий зв'язок між мікросервісами, який буде гарантувати однаковість даних у обох мікросервісів за будь-яких умов через особливості його роботи, та тому факту що він заснований на технології Protobuf.

RabbitMQ є сервісом, що легко налаштовується та є дуже швидким. Завдяки цьому брокеру можна легко розробити гнучку систему для обміну повідомленнями.

Для створення мікросервісів і вебзастосунків дуже зручно використовувати мову програмування Golang через її простоту, швидкість та легкість.

Розробленою програмною реалізацією може користуватися абсолютно будь-яка людина або компанія для впровадження чату у реальному часі у свої проєкти або в якості незалежної програми. Завдяки універсальності та створеній архітектурі дана програмна реалізація є легко масштабованою.

Проведені випробування показують, що створена програмна реалізація відповідає всім вимогам замовника та являється економічно доцільною для впровадження.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Колокольцева Т. Н. Диалогичность в жанрах интернет-коммуникации (чат, форум, блог) / Татьяна Николаевна Колокольцева // Жанры речи / Татьяна Николаевна Колокольцева., 2016. – С. 96–104.
2. Носенко Ю. М. Месенджери – Сучасний інструмент цифрового маркетингу / Носенко Ю. М., Сінельник Л. М. // Зрошуване землеробство. – 2020. – №74. – С. 164–168.
3. Microservices: Yesterday, Today, and Tomorrow / Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente та ін. // Present and Ulterior Software Engineering / Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente та ін., 2017. – С. 195–216.
4. Claus Pahl. Microservices: A Systematic Mapping Study / Claus Pahl, Pooyan Jamshidi, 2016. – С. 137–146.
5. I. Fette. The WebSocket Protocol [Electronic resource] / I. Fette, A. Melnikov // Internet Engineering Task Force (IETF). – 2011. – Access mode: [https://www.hjp.at/\(en\)/doc/rfc/rfc6455.html](https://www.hjp.at/(en)/doc/rfc/rfc6455.html).
6. Andrew Lombardi. WebSocket: Lightweight Client-Server Communications / Andrew Lombardi., 2015. – 144 с.
7. Інтерв'ю з Робом Пайком [Електрон. ресурс] – Режим доступа: https://evrone.ru/rob-pike-interview?utm_source=teletype&utm_medium=article&utm_campaign=golang.
8. Performance analysis of computational offloading on embedded platforms using the gRPC framework / Mateus Araújo, Marcio Maia, Paulo Rego, Jose de Souza. – 2020. – С. 1–8.
9. Kasun Indrasiri. gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes / Kasun Indrasiri, Danesh Kuruppu., 2020. – 204 с.

10. Пацей Н. В. Интеграция микросервисов на основе RPC / Н. В. Пацей, А. М. Шитько. // Эпоха науки. – 2021. – №27. – С. 32–37.

11. Malin Eriksson. Comparison between JSON and ProtoBuf for data serialization [Electronic resource] / Malin Eriksson, Victor Hallberg. – 2011. – Access mode: <https://www.csc.kth.se/utbildning/kth/kurser/DD143X/dkand11/Group2Mads/ProjectsSpecMailnErikssonViktorHallberg.pdf>.

12. ProIO: An Event-Based I/O Stream Format for Protobuf Messages / D. Blyth, J. Alcaraz, S. Binet, S.V. Chekanov. // Elsevier. – 2019. – С. 1–9.

13. T. Kahlert. Visual Studio Code Tips & Tricks Vol. 1 / T. Kahlert, K. Giza. – Deutschland: Microsoft Deutschland GmbH, 2016. – 26 с. – (Visual Studio Code Tips & Tricks; т. 1).

14. David Dossot. RabbitMQ Essentials / David Dossot., 2014. – 182 с.

15. Philippe Dobbelaere. Kafka versus RabbitMQ [Electronic resource] / Philippe Dobbelaere, Kyumars Sheykh Esmaili. – 2017. – Access mode: <https://arxiv.org/pdf/1709.00333.pdf>.

ДОДАТОК А
Технічне завдання

Вступ

Програмне забезпечення направлене на прискорення розробки вебзастосунків шляхом надання можливості підключити у якості зовнішнього модуля мікросервісний чат. Завдяки мікросервісам, цей чат можна підключати блоками, що спрощує процес інтеграції цієї програми.

A.1 Підстави для розробки

Підставою для розробки було завдання на дипломну кваліфікаційну роботу за темою «Програмна реалізація чату в реальному часі на основі мікросервісів» на основі наказу № 102 від 27 квітня 2022 р. за Національним університетом «Запорізька політехніка».

A.2 Призначення розробки

Розробка призначена для спрощення розробки вебзастосунку з чатом шляхом додавання цього чату у якості окремого модулю. Окрім того, програма дозволяє замінювати свої модулі іншими при виконанні всіх умов.

A.3 Основні вимоги до програми

A.3.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик складаються з наступних:

- реєстрація нових користувачів чату;
- вхід за логіном та паролем;
- перевірка дозволів користувача за токеном;
- отримання списку користувачів;
- отримання інформації про користувача за його ідентифікатором;
- оновлення інформації користувача;

- видалення користувача;
- створювання нових кімнат користувачем;
- додавання інших користувачів до кімнат;
- видалення користувачів з кімнати;
- отримання списку кімнат в яких присутній користувач;
- оновлення інформації о кімнаті;
- видалення кімнати;
- отримання повідомлень кімнати;
- створення нових повідомлень;
- редагування існуючих повідомлень;
- видалення повідомлень.

A.3.2 Вимоги до надійності

Надійність роботи системи має забезпечуватися надійністю організації технічної складової серверної сторони, надійністю вбудованого модулю auth, надійністю системи управління базами даних та обробленням даних, які вводить користувач для уникнення проблем з перетворенням типів.

A.3.3 Умови експлуатації

Для експлуатації програми необхідно забезпечити виділений сервер або хостинг, який буде відповідати необхідній кількості запитів на оброблення, що передбачається відповідним масштабом системи.

A.3.4 Вимоги до складу та параметрів технічних засобів

Апаратні вимоги до сервера:

- жорсткий диск з вільним простором – 0.5 Гб;
- оперативна пам'ять – 8 Гб;

– процесор: 4 ядра, 2,8 ГГц частота.

Клієнт представлений браузером користувача. Для виконання вимог до нього потрібний встановлений сучасний браузер з доступом до мережі інтернет.

A.4 Порядок контролю та приймання

Контроль за виконанням дипломної кваліфікаційної роботи покладається на керівника. Остаточне рішення приймається екзаменаційною комісією під час захисту роботи.

ДОДАТОК Б
Текст програми

Б.1 Текст файла auth_microservice.go

```

package clients

import (
    "context"
    "net/http"
    "github.com/Reoneks/microservice_chat/proto"
    "github.com/asim/go-micro/v3/client"
    "github.com/labstack/echo/v4"
)

type AuthMicroservice struct {
    auth      proto.AuthService
    authAddr string
}

func NewAuthMicroservice(auth proto.AuthService, authAddr string)
*AuthMicroservice {
    return &AuthMicroservice{
        auth:      auth,
        authAddr: authAddr,
    }
}

func (a *AuthMicroservice) Register(ctx echo.Context) error {
    var req proto.RegistrationRequest
    if err := ctx.Bind(&req); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    rsp, err := a.auth.Registration(context.Background(), &req,
client.WithAddress(a.authAddr))
    if err != nil {
        return ctx.String(http.StatusInternalServerError, err.Error())
    } else if rsp.Token == "" {
        return ctx.String(http.StatusUnauthorized, rsp.Status.Error)
    }

    return ctx.JSON(http.StatusOK, rsp.Token)
}

func (a *AuthMicroservice) Login(ctx echo.Context) error {
    var req proto.LoginRequest
    if err := ctx.Bind(&req); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    rsp, err := a.auth.LoginHandler(context.Background(), &req,
client.WithAddress(a.authAddr))
    if err != nil {

```

```

        return ctx.NoContent(http.StatusInternalServerError)
    } else if rsp.Token == "" {
        return ctx.NoContent(http.StatusUnauthorized)
    }

    return ctx.JSON(http.StatusOK, rsp.Token)
}

```

Б.2 Текст файла messages_microservice.go

```

package clients

import (
    "context"
    "net/http"

    "github.com/Reoneks/microservice_chat/proto"
    "github.com/asim/go-micro/v3/client"

    "github.com/labstack/echo/v4"
)

type AuthMicroservice struct {
    auth      proto.AuthService
    authAddr string
}

func NewAuthMicroservice(auth proto.AuthService, authAddr string)
*AuthMicroservice {
    return &AuthMicroservice{
        auth:      auth,
        authAddr: authAddr,
    }
}

func (a *AuthMicroservice) Register(ctx echo.Context) error {
    var req proto.RegistrationRequest
    if err := ctx.Bind(&req); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    rsp, err := a.auth.Registration(context.Background(), &req,
client.WithAddress(a.authAddr))
    if err != nil {
        return ctx.String(http.StatusInternalServerError, err.Error())
    } else if rsp.Token == "" {
        return ctx.String(http.StatusUnauthorized, rsp.Status.Error)
    }
}

```

```

    }

    return ctx.JSON(http.StatusOK, rsp.Token)
}

func (a *AuthMicroservice) Login(ctx echo.Context) error {
    var req proto.LoginRequest
    if err := ctx.Bind(&req); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    rsp, err := a.auth.LoginHandler(context.Background(), &req,
client.WithAddress(a.authAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if rsp.Token == "" {
        return ctx.NoContent(http.StatusUnauthorized)
    }

    return ctx.JSON(http.StatusOK, rsp.Token)
}

```

Б.3 Текст файла room_microservice.go

```

package clients

import (
    "context"
    "encoding/json"
    "net/http"
    "strconv"

    "github.com/Reoneks/microservice_chat/api-gateway/model"
    "github.com/Reoneks/microservice_chat/proto"
    "github.com/asim/go-micro/v3/client"

    "github.com/labstack/echo/v4"
)

type RoomMicroservice struct {
    room          proto.RoomsService
    roomAddr      string
    DefaultLimit  int64
    DefaultOffset int64
}

func (u *RoomMicroservice) GetRooms(ctx echo.Context) error {

```

```

    limit, err := strconv.ParseInt(ctx.QueryParam("limit"), 10, 64)
    if err != nil {
        limit = u.DefaultLimit
    }

    offset, err := strconv.ParseInt(ctx.QueryParam("offset"), 10, 64)
    if err != nil {
        offset = u.DefaultOffset
    }

    rsp, err := u.room.GetAllRooms(context.Background(),
&proto.GetAllRoomsRequest{
    Limit: limit,
    Offset: offset,
    UserID: ctx.Get("user_id").(string),
}, client.WithAddress(u.roomAddr))
    if err != nil {
        return ctx.String(http.StatusInternalServerError, err.Error())
    } else if !rsp.Status.Ok {
        return ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
    }

    var resp model.PaginationRoomsResponse
    err = json.Unmarshal(rsp.Room, &resp.Rooms)
    if err != nil {
        return ctx.String(http.StatusInternalServerError, err.Error())
    }

    resp.Limit = limit
    resp.Offset = offset

    return ctx.JSON(http.StatusOK, resp)
}

func (u *RoomMicroservice) CreateRoom(ctx echo.Context) error {
    var room map[string]interface{}
    if err := ctx.Bind(&room); err != nil {
        return ctx.String(http.StatusBadRequest, err.Error())
    }

    room["created_by"] = ctx.Get("user_id").(string)
    var (
        req proto.RoomStruct
        err error
    )

    req.RoomInfo, err = json.Marshal(&room)
    if err != nil {
        return ctx.String(http.StatusInternalServerError, err.Error())
    }

```

```

        rsp, err := u.room.CreateRoom(context.Background(), &req,
client.WithAddress(u.roomAddr))
        if err != nil {
            return ctx.String(http.StatusInternalServerError, err.Error())
        } else if !rsp.Status.Ok {
            return ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
        }

        var resp map[string]interface{}
        err = json.Unmarshal(rsp.Room, &resp)
        if err != nil {
            return ctx.String(http.StatusInternalServerError, err.Error())
        }

        return ctx.JSON(http.StatusOK, resp)
    }

func (u *RoomMicroservice) UpdateRoom(ctx echo.Context) error {
    var room map[string]interface{}
    if err := ctx.Bind(&room); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    roomInfo, err := json.Marshal(room)
    if err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    var data proto.UpdateRequest
    data.Room = roomInfo
    data.RoomID = ctx.Param("id")

    rsp, err := u.room.UpdateRoom(context.Background(), &data,
client.WithAddress(u.roomAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Status.Ok {
        return ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
    }

    var resp map[string]interface{}
    err = json.Unmarshal(rsp.Room, &resp)
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    }

    return ctx.JSON(http.StatusOK, resp)
}

```

```

func (u *RoomMicroservice) DeleteRoom(ctx echo.Context) error {
    var room proto.DeleteRequest
    room.RoomID = ctx.Param("id")

    rsp, err := u.room.DeleteRoom(context.Background(), &room,
client.WithAddress(u.roomAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Ok {
        return ctx.String(http.StatusInternalServerError, rsp.Error)
    }

    return ctx.NoContent(http.StatusNoContent)
}

func (u *RoomMicroservice) AddUsers(ctx echo.Context) error {
    var userIDs model.UserIDsRequest
    if err := ctx.Bind(&userIDs); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    var req proto.AddUsersRequest
    req.UserIDs = userIDs.UserIDs
    req.RoomID = ctx.Param("id")

    rsp, err := u.room.AddUsers(context.Background(), &req,
client.WithAddress(u.roomAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Ok {
        return ctx.String(http.StatusInternalServerError, rsp.Error)
    }

    return ctx.NoContent(http.StatusOK)
}

func (u *RoomMicroservice) DeleteUsers(ctx echo.Context) error {
    var userIDs model.UserIDsRequest
    if err := ctx.Bind(&userIDs); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    var req proto.AddUsersRequest
    req.UserIDs = userIDs.UserIDs
    req.RoomID = ctx.Param("id")

    rsp, err := u.room.DeleteUsers(context.Background(), &req,
client.WithAddress(u.roomAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    }

```

```

    } else if !rsp.Ok {
        return ctx.String(http.StatusInternalServerError, rsp.Error)
    }

    return ctx.NoContent(http.StatusNoContent)
}

func NewRoomMicroservice(room proto.RoomsService, DefaultLimit,
DefaultOffset int64, roomAddr string) *RoomMicroservice {
    return &RoomMicroservice{
        room: room,
        DefaultLimit: DefaultLimit,
        DefaultOffset: DefaultOffset,
        roomAddr: roomAddr,
    }
}

```

Б.4 Текст файла user_microservice.go

```

package clients

import (
    "context"
    "encoding/json"
    "net/http"

    "github.com/Reoneks/microservice_chat/proto"
    "github.com/asim/go-micro/v3/client"

    "github.com/labstack/echo/v4"
)

type UserMicroservice struct {
    user proto.UserService
    auth proto.AuthService

    userAddr string
    authAddr string
}

func (u *UserMicroservice) GetUsers(ctx echo.Context) error {
    rsp, err := u.user.GetUsers(context.Background(), &proto.Empty{}),
client.WithAddress(u.userAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Status.Ok {

```

```

        return    ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
    }

    return ctx.JSON(http.StatusOK, rsp)
}

func (u *UserMicroservice) GetUserByID(ctx echo.Context) error {
    id := ctx.Param("id")

    rsp, err := u.user.GetUserByID(context.Background(),
&proto.UserID{UserID: id}, client.WithAddress(u.userAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Status.Ok {
        return    ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
    }
    return ctx.JSON(http.StatusOK, rsp)
}

func (u *UserMicroservice) UpdateUser(ctx echo.Context) error {
    var req proto.UserStruct
    var user map[string]interface{}
    var err error

    if err = ctx.Bind(&user); err != nil {
        return ctx.NoContent(http.StatusBadRequest)
    }

    user["_id"] = ctx.Get("user_id").(string)
    req.UserInfo, err = json.Marshal(user)
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    }

    rsp, err := u.user.UpdateUser(context.Background(), &req,
client.WithAddress(u.userAddr))
    if err != nil {
        return ctx.NoContent(http.StatusInternalServerError)
    } else if !rsp.Status.Ok {
        return    ctx.String(http.StatusInternalServerError,
rsp.Status.Error)
    }
    return ctx.JSON(http.StatusOK, rsp)
}

func (u *UserMicroservice) DeleteUser(ctx echo.Context) error {
    ID := ctx.Get("user_id").(string)
    req := proto.UserID{UserID: ID}

```

```

        authRSP, err := u.auth.Delete(context.Background(), &req,
client.WithAddress(u.authAddr))
        if err != nil {
            return ctx.NoContent(http.StatusInternalServerError)
        } else if !authRSP.Status.Ok {
            return ctx.JSON(http.StatusInternalServerError,
authRSP.Status.Error)
        }

        userRSP, err := u.user.DeleteUser(context.Background(), &req,
client.WithAddress(u.userAddr))
        if err != nil {
            return ctx.NoContent(http.StatusInternalServerError)
        } else if !userRSP.Status.Ok {
            return ctx.JSON(http.StatusInternalServerError,
userRSP.Status.Error)
        }

        return ctx.JSON(http.StatusOK, userRSP)
    }

func NewUserMicroservice(user proto.UserService, auth
proto.AuthService, userAddr, authAddr string) *UserMicroservice {
    return &UserMicroservice{
        user:      user,
        auth:      auth,
        userAddr:  userAddr,
        authAddr:  authAddr,
    }
}

```

Б.5 Текст файла connection.go

```

package connector

import (
    "net/http"

    "github.com/gorilla/websocket"

type Connection interface {
    GetMessageChan() chan []byte
    GetDisconnectChan() chan struct{}
    Connect() error
    Disconnect() error
    SendMessage(data interface{}) error
    ReadMessage() (messageType int, p []byte, err error)
    GetUserID() string
}

```

```

    GetCurrentRoom() string
    SetCurrentRoom(roomID string)
}

type WSConnection struct {
    r          *http.Request
    conn       *websocket.Conn
    user       string
    currentRoomID string

    isConnected    bool
    messageChan    chan []byte
    disconnectChan chan struct{} // on disconnect
    closeConnChan  chan struct{} // on close
}

func (c *WSConnection) GetMessageChan() chan []byte {
    return c.messageChan
}

func (c *WSConnection) GetDisconnectChan() chan struct{} {
    return c.disconnectChan
}

func (c *WSConnection) Connect() error {
    if c.isConnected {
        return nil
    }
    c.isConnected = true
    go c.connect()
    return nil
}

func (c *WSConnection) connect() {
    for {
        select {
        case <-c.closeConnChan:
            c.isConnected = false
            return
        default:
            _, msgData, err := c.conn.ReadMessage()
            if err != nil {
                c.isConnected = false
                c.disconnectChan <- struct{}{}
                return
            }
            c.messageChan <- msgData
        }
    }
}

```

```

func (c *WSConnection) Disconnect() error {
    c.closeConnChan <- struct{}{}
    return c.conn.Close()
}

func (c *WSConnection) SendMessage(data interface{}) error {
    return c.conn.WriteJSON(data)
}

func (c *WSConnection) ReadMessage() (messageType int, p []byte, err
error) {
    return c.conn.ReadMessage()
}

func (c *WSConnection) GetUserID() string {
    return c.user
}

func (c *WSConnection) GetCurrentRoom() string {
    return c.currentRoomID
}

func (c *WSConnection) SetCurrentRoom(roomID string) {
    c.currentRoomID = roomID
}

func NewWSConnection(r *http.Request, conn *websocket.Conn, userID
string) Connection {
    return &WSConnection{
        r:          r,
        conn:       conn,
        user:       userID,
        currentRoomID: "",
        isConnected: false,
        messageChan: make(chan []byte),
        disconnectChan: make(chan struct{}),
        closeConnChan: make(chan struct{}),
    }
}

```

Б.6 Текст файлу connector.go

```

package connector

import (
    "encoding/json"
    "sync"

```

```

    "github.com/Reoneks/microservice_chat/api-gateway/config"
    "github.com/Reoneks/microservice_chat/proto"
    "github.com/sirupsen/logrus"
    "github.com/streadway/amqp"
)

type Connector interface {
    AddConnection(conn Connection)
    SendMessageByRoom(roomID string, data interface{})
}

type WSConnectorImpl struct {
    sync.RWMutex

    log *logrus.Logger
    rooms map[string]*RoomConnections
    msgs <-chan amqp.Delivery

    room proto.RoomsService
    defaultLimit int64
    defaultOffset int64

    ch *amqp.Channel

    //& publish
    qSendName string
    exchange string
    mandatory bool //^ false
    immediate bool //^ false
}

func (c *WSConnectorImpl) AddConnection(conn Connection) {
    c.connect(conn)
}

func (c *WSConnectorImpl) SendMessageByRoom(roomID string, data
interface{}) {
    if c.rooms[roomID] == nil {
        c.log.Error("connector: room is nil")
        return
    }
    room := *c.rooms[roomID]
    room.SendMessage(data)
}

func (c *WSConnectorImpl) connect(conn Connection) {
    err := conn.Connect()
    if err != nil {
        c.log.Error(err)
    }
    c.listen(conn)
}

```

```

}

func (c *WSConnectorImpl) disconnect(conn Connection) {
    if c.rooms[conn.GetCurrentRoom()] == nil {
        c.log.Error("connector: room is nil")
        return
    }
    room := *c.rooms[conn.GetCurrentRoom()]
    room.RemoveConnection(conn)
}

func (c *WSConnectorImpl) listen(conn Connection) {
loop:
    for {
        select {
        case msg := <-conn.GetMessageChan():
            c.onEventMessage(conn, msg)
        case <-conn.GetDisconnectChan():
            c.disconnect(conn)
            break loop
        case msg := <-c.msgs:
            var message proto.RabbitMessage

            err := json.Unmarshal(msg.Body, &message)
            if err != nil {
                c.log.Error(err)
                continue
            }

            var messageE EventMessage
            if message.EventType != int64(DeleteMessageEventType) {
                msgData := make(map[string]interface{})
                err = json.Unmarshal(message.Message, &msgData)
                if err != nil {
                    c.log.Error(err)
                    continue
                }

                messageE = EventMessage{
                    Type: EventType(message.EventType),
                    Data: msgData,
                }
            } else {
                messageE = EventMessage{
                    Type: EventType(message.EventType),
                    Data: map[string]interface{}{"id":
string(message.Message)},
                }
            }

            c.SendMessageByRoom(message.RoomID, messageE)
        }
    }
}

```

```

    }
}

func NewWSConnector(
    log *logrus.Logger,
    ch *amqp.Channel,
    cfg *config.Config,
    msgs <-chan amqp.Delivery,
    room proto.RoomsService,
) Connector {
    return &WSConnectorImpl{
        log:      log,
        rooms:    map[string]*RoomConnections{},
        room:     room,
        defaultLimit: cfg.DefaultLimit,
        defaultOffset: cfg.DefaultOffset,
        msgs:     msgs,
        ch:       ch,
        qSendName: cfg.SendName,
        exchange: cfg.Exchange,
        mandatory: cfg.Mandatory,
        immediate: cfg.Immediate,
    }
}

```

Б.7 Текст файлу event_message.go

```

package connector

import (
    "context"
    "encoding/json"

    "github.com/Reoneks/microservice_chat/api-gateway/tools"
    "github.com/Reoneks/microservice_chat/proto"
    "github.com/streadway/amqp"
)

type EventType int64

const (
    SubscribeEventType      EventType = iota + 1 // Connect to
    room Data: "1","2","3"
    UnSubscribeRoomEventType // Disconnect from
    room Data: ""
    NewMessageEventType      //
    Message Data: "Text Message"
    EditMessageEventType
)

```



```

data := struct {
    RoomID string `json:"room_id"`
}{}

err := tools.Bind(message.Data, &data)
if err != nil {
    c.log.Error(err)
    return
}

roomNumber := data.RoomID
pRoom := c.rooms[roomNumber]
if pRoom == nil {
    return
}

room := *pRoom
room.RemoveConnection(conn)
sendToFront := struct {
    RoomId string
    UserId string
}{
    RoomId: roomNumber,
    UserId: conn.GetUserID(),
}

msg := EventMessage{
    Type: SubscribeEventType,
    Data: sendToFront,
}

c.SendMessageByRoom(roomNumber, msg)
///! //////////////////////////////////////
case NewMessageEventType:
    data := struct {
        Text string `json:"text"`
        QuoteMessageId *string `json:"quote_message_id"`
    }{}

    err := tools.Bind(message.Data, &data)
    if err != nil {
        c.log.Error(err)
        return
    }

    msg := map[string]interface{}{
        "text": data.Text,
        "status": 1,
        "room_id": conn.GetCurrentRoom(),
        "created_by": conn.GetUserID(),
    }

```

```

    if data.QuoteMessageId != nil {
        msg["quote_message_id"] = *data.QuoteMessageId
    }

    bytes, err := json.Marshal(&msg)
    if err != nil {
        c.log.Error(err)
        return
    }

    message := proto.RabbitMessage{
        Message:      bytes,
        MessageType:  "create",
        EventType:    int64(NewMessageEventType),
        RoomID:       conn.GetCurrentRoom(),
    }

    bytes, err = json.Marshal(&message)
    if err != nil {
        c.log.Error(err)
        return
    }

    err = c.ch.Publish(
        "",
        c.qSendName,
        c.mandatory,
        c.immediate,
        amqp.Publishing{
            ContentType: "application/json",
            Body:        []byte(bytes),
        },
    )
    if err != nil {
        c.log.Error(err)
        return
    }
    ///! //////////////////////////////////////
case EditMessageEventType:
    data := struct {
        Text      string `json:"text"`
        MessageID string `json:"message_id"`
    }{}

    err := tools.Bind(message.Data, &data)
    if err != nil {
        c.log.Error(err)
        return
    }

```

```

msg := map[string]interface{}{
    "_id":    data.MessageID,
    "text":   data.Text,
    "status": 3,
}

bytes, err := json.Marshal(&msg)
if err != nil {
    c.log.Error(err)
    return
}

message := proto.RabbitMessage{
    Message:      bytes,
    MessageType:  "update",
    EventType:    int64(EditMessageEventType),
    RoomID:       conn.GetCurrentRoom(),
}

bytes, err = json.Marshal(&message)
if err != nil {
    c.log.Error(err)
    return
}

err = c.ch.Publish(
    "",
    c.qSendName,
    c.mandatory,
    c.immediate,
    amqp.Publishing{
        ContentType: "application/json",
        Body:        []byte(bytes),
    },
)
if err != nil {
    c.log.Error(err)
    return
}

///! //////////////////////////////////////
case DeleteMessageEventType:
    data := struct {
        MessageID string `json:"message_id"`
    }{}

    err := tools.Bind(message.Data, &data)
    if err != nil {
        c.log.Error(err)
        return
    }

```

```

message := proto.RabbitMessage{
    Message:    []byte(data.MessageID),
    MessageType: "delete",
    EventType:   int64(DeleteMessageEventType),
    RoomID:     conn.GetCurrentRoom(),
}

bytes, err := json.Marshal(&message)
if err != nil {
    c.log.Error(err)
    return
}

err = c.ch.Publish(
    "",
    c.qSendName,
    c.mandatory,
    c.immediate,
    amqp.Publishing{
        ContentType: "application/json",
        Body:        []byte(bytes),
    },
)
if err != nil {
    c.log.Error(err)
    return
}

///! //////////////////////////////////////
case WritingMessageEventType:
    sendToFront := struct {
        RoomId string `json:"room_id"`
        UserId string `json:"user"`
    }{
        RoomId: conn.GetCurrentRoom(),
        UserId: conn.GetUserID(),
    }

    b, err := json.Marshal(sendToFront)
    if err != nil {
        c.log.Error(err)
        return
    }

    msg := proto.RabbitMessage{
        Message:    b,
        EventType:   int64(WritingMessageEventType),
        RoomID:     sendToFront.RoomId,
    }

    b, err = json.Marshal(&msg)
    if err != nil {

```

```

        c.log.Error(err)
        return
    }

    err = c.ch.Publish(
        c.exchange,
        "",
        c.mandatory,
        c.immediate,
        amqp.Publishing{
            ContentType: "application/json",
            Body:        b,
        },
    )
    if err != nil {
        c.log.Error(err)
        return
    }
    ///! //////////////////////////////////////
    case GetAllRoomsEventType:
        data := struct {
            Limit  int64 `json:"limit"`
            Offset int64 `json:"offset"`
        }{}

        err := tools.Bind(message.Data, &data)
        if err != nil {
            c.log.Error(err)
            return
        }

        if data.Limit <= 0 {
            data.Limit = c.defaultLimit
        }

        if data.Offset < 0 {
            data.Offset = c.defaultOffset
        }

        req := &proto.GetAllRoomsRequest{
            Limit:  data.Limit,
            Offset: data.Offset,
            UserID: conn.GetUserID(),
        }

        resp, err := c.room.GetAllRooms(context.Background(), req)
        if err != nil {
            c.log.Error(err)
            return
        }

```

```

var sendToFront []map[string]interface{}
err = json.Unmarshal(resp.Room, &sendToFront)
if err != nil {
    c.log.Error(err)
    return
}

message := EventMessage{
    Type: WritingMessageEventType,
    Data: sendToFront,
}
conn.SendMessage(message)
}
}

```

Б.8 Текст файла room_connection.go

```

package connector

import "sync"

type RoomConnections interface {
    AddConnection(conn Connection)
    RemoveConnection(conn Connection)
    SendMessage(data interface{})
}

type RoomConnectionsImpl struct {
    sync.RWMutex

    roomID      string
    connections []Connection
    unsubscribeChan chan struct{}
}

func (rc *RoomConnectionsImpl) AddConnection(conn Connection) {
    rc.Lock()
    defer rc.Unlock()
    rc.connections = append(rc.connections, conn)
}

func (rc *RoomConnectionsImpl) RemoveConnection(conn Connection) {
    rc.Lock()
    defer rc.Unlock()
    var newConn []Connection
    for _, roomConn := range rc.connections {
        if roomConn != conn {
            newConn = append(newConn, roomConn)
        }
    }
}

```

```

    }
    rc.connections = newConn
}

func (rc *RoomConnectionsImpl) SendMessage(data interface{}) {
    rc.RLock()
    defer rc.RUnlock()
    for _, conn := range rc.connections {
        err := conn.SendMessage(data)
        if err != nil {
            rc.RemoveConnection(conn)
        }
    }
}

func NewRoomConnections(roomID string) RoomConnections {
    return &RoomConnectionsImpl{
        roomID:      roomID,
        connections: make([]Connection, 0),
        unsubscribeChan: make(chan struct{}),
    }
}

```

Б.9 Текст файла auth_handler.go

```

package auth

import (
    "context"
    "encoding/json"
    "errors"

    "github.com/Reoneks/microservice_chat/auth/utils"
    "github.com/Reoneks/microservice_chat/proto"
    "github.com/asim/go-micro/v3/client"

    "github.com/go-chi/jwtauth"
)

type LoginRequest struct {
    Email    string `json:"email" validate:"email"`
    Password string `json:"password" validate:"min=4,max=32,alphanum"`
}

type Auth struct {
    authService AuthService
    userService proto.UserService
    jwt          *jwtauth.JWTAuth
    userAddr     string
}

```

```

}

func NewAuth(authService AuthService, userService proto.UserService, jwt
*jwtauth.JWTAuth, userAddr string) *Auth {
    return &Auth{
        authService: authService,
        userService: userService,
        jwt:         jwt,
        userAddr:    userAddr,
    }
}

func (a *Auth) AuthHandler(ctx context.Context, req *proto.Token, resp
*proto.UserID) error {
    resp.Status = new(proto.Status)

    if req.Token == "" {
        resp.Status.Ok = false
        resp.Status.Error = "token is required"
        return errors.New("token is required")
    }

    payload, err := jwtauth.VerifyToken(a.jwt, req.Token)
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    }

    userID, ok := payload.Get("user_id")
    if !ok {
        resp.Status.Ok = false
        resp.Status.Error = "there is no user id"
        return errors.New("there is no user id")
    }

    resp.UserID = userID.(string)
    resp.Status.Ok = true
    return nil
}

func (a *Auth) Delete(ctx context.Context, req *proto.UserID, resp
*proto.DeleteUserResponse) (err error) {
    resp.Status = new(proto.Status)

    err = a.authService.Delete(req.UserID)
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    }
}

```

```

    resp.Status.Ok = true
    return nil
}

func (a *Auth) LoginHandler(ctx context.Context, req
*proto.LoginRequest, resp *proto.Token) (err error) {
    resp.Status = new(proto.Status)

    resp.Token, err = a.authService.Login(req.Email, req.Password)
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    } else if resp.Token == "" {
        resp.Status.Ok = false
        resp.Status.Error = "token is empty"
        return errors.New("token is empty")
    }

    resp.Status.Ok = true
    return nil
}

func (a *Auth) Registration(ctx context.Context, req
*proto.RegistrationRequest, resp *proto.Token) (err error) {
    resp.Status = new(proto.Status)

    userResp, err := a.userService.CreateUser(context.Background(),
    ToUser(req), client.WithAddress(a.userAddr))
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    }

    var userRespMap map[string]interface{}
    err = json.Unmarshal(userResp.UserInfo, &userRespMap)
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    }

    user := make(map[string]interface{})
    user["_id"] = userRespMap["id"].(string)
    user["email"] = req.Email
    user["password"], err = utils.Encrypt(req.Password)
    if err != nil {
        resp.Status.Ok = false
    }
}

```

```

        resp.Status.Error = err.Error()
        return err
    }

    resp.Token, err = a.authService.Register(user)
    if err != nil {
        resp.Status.Ok = false
        resp.Status.Error = err.Error()
        return err
    } else if resp.Token == "" {
        resp.Status.Ok = false
        resp.Status.Error = "token is empty"
        return errors.New("token is empty")
    }

    resp.Status.Ok = true
    return nil
}

```

Б.10 Текст файла auth_repository.go

```

package auth

import (
    "context"
    "fmt"

    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
)

type AuthRepository interface {
    GetUserByEmail(email string) (authUser map[string]interface{}, err error)
    AddUser(authUser map[string]interface{}) (map[string]interface{}, error)
    DeleteUser(id string) error
}

type AuthRepositoryImpl struct {
    authCollection *mongo.Collection
    ctx            context.Context
}

func NewAuthRepository(db *mongo.Client, dbName, collection string) AuthRepository {
    authCollection := db.Database(dbName).Collection(collection)
    return &AuthRepositoryImpl{authCollection, context.Background()}
}

```

```

func (a *AuthRepositoryImpl) GetUserByEmail(email string)
(map[string]interface{}, error) {
    filter := bson.M{"email": email}

    var auth map[string]interface{}
    err := a.authCollection.FindOne(a.ctx, filter).Decode(&auth)
    if err != nil {
        return nil, fmt.Errorf("FindOne error:\n\t%v", err)
    }

    return auth, nil
}

func (a *AuthRepositoryImpl) AddUser(authUser map[string]interface{})
(map[string]interface{}, error) {
    _, err := a.authCollection.InsertOne(a.ctx, authUser)
    if err != nil {
        return nil, fmt.Errorf("AddUser error:\n\t%v", err)
    }

    return authUser, nil
}

func (a *AuthRepositoryImpl) DeleteUser(id string) error {
    _, err := a.authCollection.DeleteOne(a.ctx, bson.M{"_id": id})
    return err
}

```

Б.11 Текст файла auth_service.go

```

package auth

import (
    "github.com/Reoneks/microservice_chat/auth/utils"
    "github.com/go-chi/jwtauth"
)

type AuthService interface {
    Login(email, password string) (string, error)
    Register(user map[string]interface{}) (string, error)
    Delete(id string) error
}

type AuthServiceImpl struct {
    jwt *jwtauth.JWTAuth
    authRepository AuthRepository
}

```

```

func (s *AuthServiceImpl) Login(email, password string) (string, error)
{
    receivedUser, err := s.authRepository.GetUserByEmail(email)
    if err != nil {
        return "", err
    }
    err = utils.Compare(receivedUser["password"].(string), password)
    if err != nil {
        return "", err
    }

    _, tokenString, err :=
s.jwt.Encode(map[string]interface{}{"user_id": receivedUser["_id"]})
    if err != nil {
        return "", err
    }
    return tokenString, nil
}

func (s *AuthServiceImpl) Register(user map[string]interface{}) (string,
error) {
    receivedUser, err := s.authRepository.AddUser(user)
    if err != nil {
        return "", err
    }

    _, tokenString, err :=
s.jwt.Encode(map[string]interface{}{"user_id": receivedUser["_id"]})
    if err != nil {
        return "", err
    }

    return tokenString, nil
}

func (s *AuthServiceImpl) Delete(id string) error {
    return s.authRepository.DeleteUser(id)
}

func NewAuthService(authRepository AuthRepository, jwt
*jwtauth.JWTAuth) AuthService {
    return &AuthServiceImpl{
        jwt,
        authRepository,
    }
}

```

Б.12 Текст файла password_utils.go

```
package utils

import (
    "golang.org/x/crypto/bcrypt"
)

func Encrypt(password string) (string, error) {
    hashedPassword, err := bcrypt.GenerateFromPassword([]byte(password),
bcrypt.DefaultCost)
    if err != nil {
        return "", err
    }
    return string(hashedPassword), nil
}

func Compare(userpass, password string) error {
    if err := bcrypt.CompareHashAndPassword([]byte(userpass),
[]byte(password)); err != nil {
        return err
    }
    return nil
}
```

Б.13 Текст файла messages_repository.go

```
package messages

import (
    "context"
    "fmt"
    "log"
    "time"

    "github.com/google/uuid"
    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"
)

type MessagesRepository interface {
    GetMessagesByRoom(roomID string, limit, offset int64)
([]map[string]interface{}, error)
    GetMessageByID(id string) (map[string]interface{}, error)
    CreateMessage(message map[string]interface{})
(map[string]interface{}, error)
}
```

```

        UpdateMessage(message      map[string]interface{})
    (map[string]interface{}, error)
        DeleteMessage(id string) error
    }

    type MessagesRepositoryImpl struct {
        messagesCollection *mongo.Collection
        ctx                context.Context
    }

    func (r *MessagesRepositoryImpl) GetMessagesByRoom(roomID string, limit,
    offset int64) ([]map[string]interface{}, error) {
        findOptions := options.Find()
        findOptions.SetLimit(limit).SetSkip(offset).SetSort(bson.M{"update
    d_at": -1})

        cur, err := r.messagesCollection.Find(r.ctx, bson.M{"room_id":
    roomID}, findOptions)
        if err != nil {
            return nil, fmt.Errorf("Find error:\n\t%v", err)
        }

        defer cur.Close(r.ctx)

        var results []map[string]interface{}
        for cur.Next(r.ctx) {
            var data map[string]interface{}

            err := cur.Decode(&data)
            if err != nil {
                log.Println(fmt.Errorf("Decode error:\n\t%v", err))
                continue
            }

            results = append(results, data)
        }

        if err := cur.Err(); err != nil {
            return nil, fmt.Errorf("Cursor error:\n\t%v", err)
        }

        return results, nil
    }

    func (r *MessagesRepositoryImpl) GetMessageByID(id string)
    (map[string]interface{}, error) {
        message := make(bson.M)
        err := r.messagesCollection.FindOne(r.ctx, bson.M{"_id":
    id}).Decode(&message)
        if err != nil {
            return nil, fmt.Errorf("CreateMessage error:\n\t%v", err)
        }
    }

```

```

    }

    return message, nil
}

func (r *MessagesRepositoryImpl) CreateMessage(message
map[string]interface{}) (map[string]interface{}, error) {
    message["_id"] = uuid.New().String()
    message["created_at"] = time.Now()
    message["updated_at"] = time.Now()
    _, err := r.messagesCollection.InsertOne(r.ctx, message)
    if err != nil {
        return nil, fmt.Errorf("CreateMessage error:\n\t%v", err)
    }

    return message, nil
}

func (r *MessagesRepositoryImpl) UpdateMessage(message
map[string]interface{}) (map[string]interface{}, error) {
    message["updated_at"] = time.Now()

    filter := bson.M{"_id": message["_id"]}
    update := bson.M{"$set": message}
    _, err := r.messagesCollection.UpdateOne(r.ctx, filter, update)
    if err != nil {
        return nil, fmt.Errorf("UpdateOne error:\n\t%v", err)
    }
    return message, nil
}

func (r *MessagesRepositoryImpl) DeleteMessage(id string) error {
    _, err := r.messagesCollection.DeleteOne(r.ctx, bson.M{"_id": id})
    return err
}

func NewMessagesRepository(db *mongo.Client, dbName, collection string)
MessagesRepository {
    messagesCollection := db.Database(dbName).Collection(collection)
    return &MessagesRepositoryImpl{messagesCollection,
context.Background()}
}

```

Б.14 Текст файла messages-handler.go

```

package messages

import (
    "context"

```

```

"encoding/json"
"log"

"github.com/Reoneks/microservice_chat/messages/config"
"github.com/Reoneks/microservice_chat/proto"

"github.com/streadway/amqp"
)

type messagesMicro struct {
    MessagesService MessagesService

    Ch *amqp.Channel
    Q  *amqp.Queue

    //& StartConsumer
    Consumer string    //^ ""
    AutoAsk   bool        //^ true
    Exclusive bool        //^ false
    NoLocal   bool        //^ false
    NoWait    bool        //^ false
    Arguments amqp.Table //^ nil

    //& publish
    Exchange string //^ ""
    Mandatory bool   //^ false
    Immediate bool   //^ false
}

type IMessagesMicro interface {
    StartConsumer() error
}

func NewMessagesMicro(
    cfg *config.Config,
    MessagesService MessagesService,
    Ch *amqp.Channel,
    Q *amqp.Queue,
    Arguments amqp.Table,
) IMessagesMicro {
    return &messagesMicro{
        MessagesService: MessagesService,

        Ch: Ch,
        Q:  Q,

        Consumer:  cfg.Consumer,
        AutoAsk:   cfg.AutoAsk,
        Exclusive: cfg.ConsumerExclusive,
        NoLocal:   cfg.NoLocal,
        NoWait:    cfg.ConsumerNoWait,
    }
}

```

```

        Arguments: Arguments,

        Exchange:  cfg.Exchange,
        Mandatory: cfg.Mandatory,
        Immediate: cfg.Immediate,
    }
}

func (u *messagesMicro) publish(msg *proto.RabbitMessage) error {
    bytes, err := json.Marshal(msg)
    if err != nil {
        return err
    }

    return u.Ch.Publish(
        u.Exchange,
        "",
        u.Mandatory,
        u.Immediate,
        amqp.Publishing{
            ContentType: "application/json",
            Body:        bytes,
        },
    )
}

func (u *messagesMicro) createMessage(data *proto.RabbitMessage) error {
    var msg map[string]interface{}
    err := json.Unmarshal(data.Message, &msg)
    if err != nil {
        log.Println("Unmarshal msg error: ", err)
    }

    message, err := u.MessagesService.CreateMessage(msg)
    if err != nil {
        return err
    }

    bytes, err := json.Marshal(message)
    if err != nil {
        return err
    }

    data.Message = bytes
    return u.publish(data)
}

func (u *messagesMicro) updateMessage(data *proto.RabbitMessage) error {
    var msg map[string]interface{}

```

```

err := json.Unmarshal(data.Message, &msg)
if err != nil {
    log.Println("Unmarshal msg error: ", err)
}

message, err := u.MessagesService.UpdateMessage(msg)
if err != nil {
    return err
}

bytes, err := json.Marshal(message)
if err != nil {
    return err
}

data.Message = bytes
return u.publish(data)
}

func (u *messagesMicro) deleteMessage(data *proto.RabbitMessage) error {
    err := u.MessagesService.DeleteMessage(string(data.Message))
    if err != nil {
        return err
    }

    return u.publish(data)
}

func (u *messagesMicro) StartConsumer() error {
    msgs, err := u.Ch.Consume(
        u.Q.Name,
        u.Consumer,
        u.AutoAck,
        u.Exclusive,
        u.NoLocal,
        u.NoWait,
        u.Arguments,
    )
    if err != nil {
        return err
    }

    for message := range msgs {
        var data proto.RabbitMessage

        err := json.Unmarshal(message.Body, &data)
        if err != nil {
            log.Printf("Unmarshal RabbitMessage error: %v", err)
        }
    }
}

```

```

switch data.MessageType {
case "create":
    var msg map[string]interface{}
    err = json.Unmarshal(data.Message, &msg)
    if err != nil {
        log.Printf("Unmarshal msg error: %v", err)
    }

    err := u.createMessage(&data)
    if err != nil {
        log.Printf("%v", err)
    }

case "update":
    err := u.updateMessage(&data)
    if err != nil {
        log.Printf("%v", err)
    }

case "delete":
    err := u.deleteMessage(&data)
    if err != nil {
        log.Printf("%v", err)
    }
}

return nil
}

type MessagesMicro struct {
    MessagesService MessagesService
}

func (u *MessagesMicro) GetMessagesByRoom(
    ctx context.Context,
    req *proto.MessageID,
    rsp *proto.GetMessagesByRoomResponse,
) error {
    rsp.Status = new(proto.Status)

    messages, err := u.MessagesService.GetMessagesByRoom(req.RoomID,
    req.Limit, req.Offset)
    if err != nil {
        rsp.Status.Ok = false
        rsp.Status.Error = err.Error()
        return err
    }

    bytes, err := json.Marshal(messages)
    if err != nil {

```

```

        rsp.Status.Ok = false
        rsp.Status.Error = err.Error()
        return err
    }

    rsp.Messages = bytes
    rsp.Status.Ok = true
    return nil
}

```

Б.15 Текст файла messages-service.go

```

package messages

type messagesService struct {
    messagesService MessagesRepository
}

type MessagesService interface {
    GetMessagesByRoom(roomID string, limit, offset int64)
    ([]map[string]interface{}, error)
    CreateMessage(message map[string]interface{})
    (map[string]interface{}, error)
    UpdateMessage(message map[string]interface{})
    (map[string]interface{}, error)
    DeleteMessage(id string) error
}

func (us *messagesService) GetMessagesByRoom(roomID string, limit,
offset int64) ([]map[string]interface{}, error) {
    messages, err := us.messagesService.GetMessagesByRoom(roomID, limit,
offset)
    if err != nil {
        return nil, err
    }

    for i := range messages {
        messages[i]["id"] = messages[i]["_id"]
        delete(messages[i], "_id")
    }

    return messages, nil
}

func (us *messagesService) CreateMessage(message
map[string]interface{}) (map[string]interface{}, error) {
    result, err := us.messagesService.CreateMessage(message)
    if err != nil {
        return nil, err
    }
}

```

```

    }

    result["id"] = result["_id"]
    delete(result, "_id")
    return result, nil
}

func (us *messagesService) UpdateMessage(message
map[string]interface{}) (map[string]interface{}, error) {
    _, err := us.messagesService.UpdateMessage(message)
    if err != nil {
        return nil, err
    }

    result, err :=
us.messagesService.GetMessageByID(message["_id"].(string))
    if err != nil {
        return nil, err
    }

    result["id"] = result["_id"]
    delete(result, "_id")
    return result, nil
}

func (us *messagesService) DeleteMessage(id string) error {
    return us.messagesService.DeleteMessage(id)
}

func NewMessagesService(messageService MessagesRepository)
MessagesService {
    return &messagesService{
        messagesService: messageService,
    }
}

```

Б.16 Текст файла room_users_repository.go

```

package room

import (
    "context"
    "fmt"
    "log"

    "go.mongodb.org/mongo-driver/bson"
    "go.mongodb.org/mongo-driver/mongo"
    "go.mongodb.org/mongo-driver/mongo/options"

```

```

)

type RoomUsersRepository interface {
    GetRoomsByUserId(id string, limit, offset int64)
    ([]map[string]interface{}, error)
    CreateUserRoomConnection(user map[string]interface{})
    (map[string]interface{}, error)
    DeleteUserRoomConnection(user map[string]interface{}) error
}

type RoomUsersRepositoryImpl struct {
    roomsUsersCollection *mongo.Collection
    roomsCollection      *mongo.Collection
    ctx                  context.Context
}

func (r *RoomUsersRepositoryImpl) GetRoomsByUserId(id string, limit,
offset int64) ([]map[string]interface{}, error) {
    var rooms []map[string]interface{}

    cur, err := r.roomsUsersCollection.Find(r.ctx, bson.M{"user_id":
id})
    if err != nil {
        return nil, fmt.Errorf("Find error:\n\t%v", err)
    }

    defer cur.Close(r.ctx)

    var roomUsers []map[string]interface{}
    for cur.Next(r.ctx) {
        var data map[string]interface{}

        err := cur.Decode(&data)
        if err != nil {
            log.Println(fmt.Errorf("Decode error:\n\t%v", err))
            continue
        }

        roomUsers = append(roomUsers, data)
    }

    if err := cur.Err(); err != nil {
        return nil, fmt.Errorf("Cursor error:\n\t%v", err)
    }

    var IDs []interface{}
    for _, user := range roomUsers {
        IDs = append(IDs, user["room_id"])
    }

    opts := options.Find()

```

```

    opts.SetLimit(limit)
    opts.SetSkip(offset)

    rcur, err := r.roomsCollection.Find(r.ctx, bson.M{"_id":
bson.M{"$in": IDs}}, opts)
    if err != nil {
        return nil, fmt.Errorf("Find error:\n\t%v; IDS: %v", err, IDs)
    }

    defer rcur.Close(r.ctx)

    for rcur.Next(r.ctx) {
        var data map[string]interface{}

        err := rcur.Decode(&data)
        if err != nil {
            log.Println(fmt.Errorf("Decode error:\n\t%v", err))
            continue
        }

        rooms = append(rooms, data)
    }

    if err := rcur.Err(); err != nil {
        return nil, fmt.Errorf("Cursor error:\n\t%v", err)
    }
    return rooms, nil
}

func (r *RoomUsersRepositoryImpl) CreateUserRoomConnection(user
map[string]interface{}) (map[string]interface{}, error) {
    _, err := r.roomsUsersCollection.InsertOne(r.ctx, user)
    if err != nil {
        return nil, fmt.Errorf("CreateUserRoomConnection error:\n\t%v",
err)
    }
    return user, nil
}

func (r *RoomUsersRepositoryImpl) DeleteUserRoomConnection(user
map[string]interface{}) error {
    _, err := r.roomsUsersCollection.DeleteOne(r.ctx, user)
    return err
}

func NewRoomUsersRepository(db *mongo.Client, dbName,
roomUserCollection, roomCollection string) RoomUsersRepository {
    roomsUsersCollection :=
db.Database(dbName).Collection(roomUserCollection)
    roomsCollection := db.Database(dbName).Collection(roomCollection)
    return &RoomUsersRepositoryImpl{roomsUsersCollection,
roomsCollection, context.Background()}
}

```

ДОДАТОК В
Слайди презентації

ПРОГРАМНА РЕАЛІЗАЦІЯ ЧАТУ В РЕАЛЬНОМУ ЧАСІ НА ОСНОВІ МІКРОСЕРВІСІВ SOFTWARE IMPLEMENTATION OF A REAL-TIME CHAT BASED ON MICROSERVICES

Виконав ст. гр. КНТ-218 Лінець В. А.
Керівник к.т.н., доцент — Колпакова Т. О.

2022

Рисунок В.1 – Слайд № 1

Об'єкт дослідження – процес спілкування користувачів онлайн.

Предмет дослідження – вебзастосунки для спілкування користувачів у інтернет-чатах.

Мета роботи – створення програмного забезпечення для надання можливості спілкування в інтернеті за допомогою чатів або коментарів користувачам тих чи інших сервісів.

Завдання дипломної кваліфікаційної роботи:

- аналіз предметної області;
- проектування програмного забезпечення;
- реалізація і тестування програмного забезпечення.

2

Рисунок В.2 – Слайд № 2

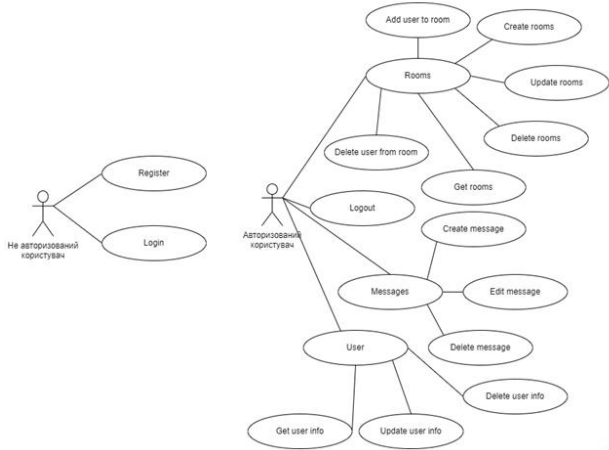
Порівняння аналогів

	Telegram	Facebook Messenger	Розробка
Чат у реальному часі	+	+	+
Відкритість розробки	+	-	+
Можливість інтеграції у веб-сторінку	+/-	+	+
Безпека даних	+	-	+
Можливість додати свої дані в повідомлення	-	-	+

3

Рисунок В.3 – Слайд № 3

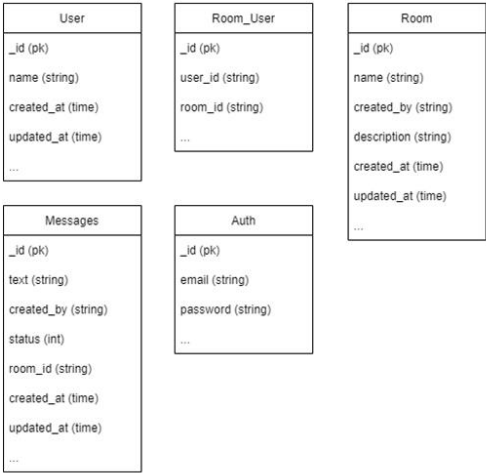
Діаграми прецедентів авторизованого і неавторизованого користувача



4

Рисунок В.4 – Слайд № 4

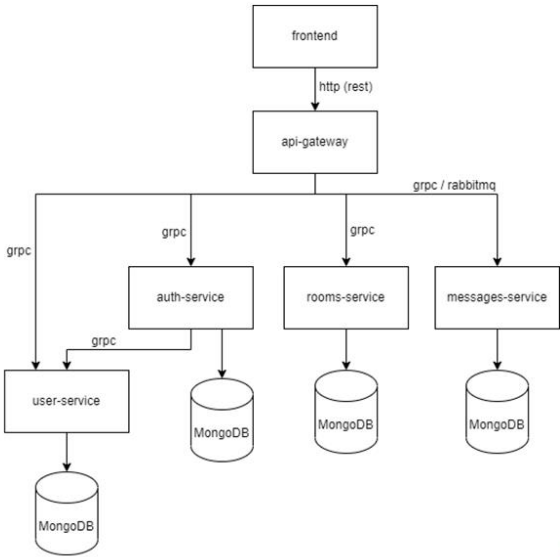
Схема бази даних



5

Рисунок В.5 – Слайд № 5

Архітектура програми



6

Рисунок В.6 – Слайд № 6

Схема функціонування програми

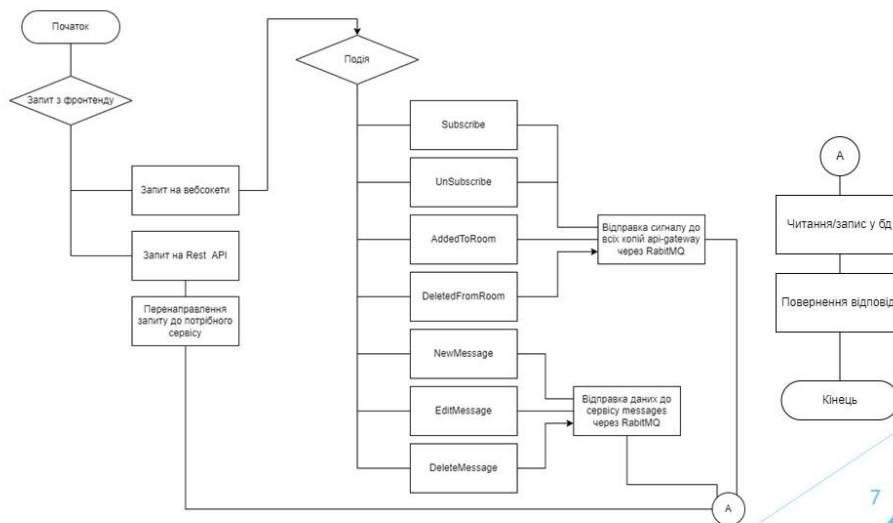


Рисунок В.7 – Слайд № 7

Інтерфейси користувача

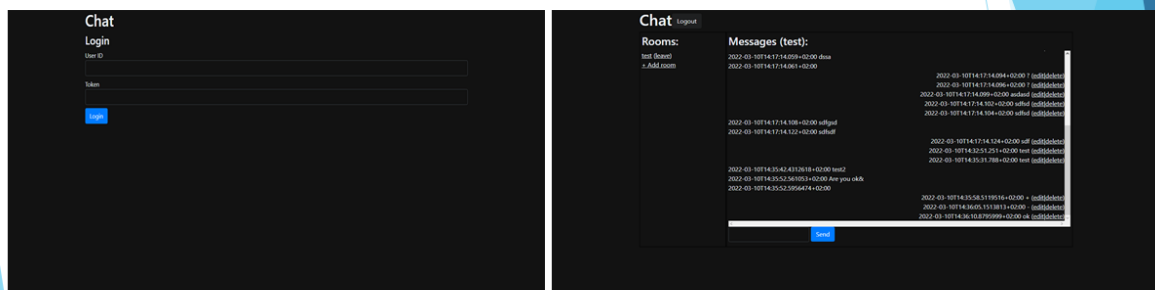


Рисунок В.8 – Слайд № 8

Висновки

В даній роботі були вирішені наступні завдання:

- проаналізовані аналоги створюваного веб-застосунку;
- досліджені технології, що планувалися для використання;
- вивчена взаємодія з мовою програмування Golang;
- досліджені методи роботи з вебсокетами;
- проаналізовані способи спілкування та створення мікросервісів;
- створена реалізація чату у реальному часі на мікросервісах;
- протестовано створений веб-застосунок.

В результаті проведеної роботи було створено програмну реалізацію чату у реальному часі на мікросервісах. Завдання дипломної кваліфікаційної роботи повністю виконано.

9

Рисунок В.9 – Слайд № 9