МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ОНЛАЙН ЧАТУ НА БАЗІ ПРОТОКОЛУ WEBSOCKET»**

Виконав: студент 2 курсу, групи 8.1212-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
М.В. Бобровський
(ініціали та прізвище)

Керівник завідувач кафедри програмної інженерії, доцент, к.ф.-м.н. Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри комп'ютерних наук, доцент, д.т.н. Шило Г.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2023 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Бобровському Михайлу Віталійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка онлайн чату на базі протоколу WebSocket

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » травня 2023 року № 642-с

2. Строк подання студентом роботи 27.11.2023 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка онлайн чату.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 03.05.2023

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	17.05.2023	
2.	Збір вихідних даних.	31.05.2023	
3.	Обробка методичних та теоретичних	28.06.2023	
	джерел.		
4.	Розробка першого та другого розділу.	30.08.2023	
5.	Розробка третього розділу.	01.11.2023	
6.	Оформлення та нормоконтроль	20.11.2023	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	14.12.2023	

Студент _____ М.В. Бобровський
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка онлайн чату на базі протоколу WebSocket»: 68 с., 29 рис., 1 табл., 16 джерел, 2 додатки.

BOUNDED CONTEXT, DDD, DOCKER, DOMAIN, GOLANG, PINIA, POSTGRESQL, QUASAR, TYPESCRIPT, UBIQUITOUS LANGUAGE, VUEJS, WEBSOCKET.

Об’єкт дослідження роботи – обмін інформації між вебсервером та браузером в режимі реального часу.

Предмет дослідження – вебдодаток для обміну повідомленнями між користувачами (чат).

Мета роботи: провести дослідження сучасних платформ для розробки онлайн чатів, проаналізувати та обґрунтувати вибір технологій для розробки вебдодатків та на їх основі створити онлайн чат.

Метод дослідження – системний аналіз, методи програмної інженерії, компонентне проєктування, мікросервісна архітектура, клієнт-серверна архітектура, архітектура DDD, UML-проєктування.

SUMMARY

Master’s qualifying paper «Development of The Online Chat Based on the WebSocket Protocol»: 68 pages, 29 figures, 1 table, 16 references, 2 supplements.

BOUNDED CONTEXT, DDD, DOCKER, DOMAIN, GOLANG, PINIA, POSTGRESQL, QUASAR, TYPESCRIPT, UBIQUITOUS LANGUAGE, VUEJS, WEBSOCKET.

The object of the study is the exchange of information between the web server and the browser in real time.

Subject Research is a web application for exchanging messages between users (chat).

The aim of the study is to conduct a study of modern platforms for the development of online chats, to analyze and justify the choice of technologies for the development of web applications and to create an online chat based on them.

The methods of research are system analysis, software engineering methods, component design, microservice architecture, client-server architecture, DDD architecture, UML design.

ЗМІСТ

ВСТУП

Дослідження розробки онлайн чату на базі протоколу WebSocket має високу актуальність в даний час з кількох причин.

По-перше, зростання популярності вебдодатків, особливо на базі JavaScript, призвело до збільшення попиту на реальний час взаємодії між клієнтом та сервером. WebSocket надає двосторонній зв'язок між клієнтом та сервером, що дозволяє миттєво обмінюватися інформацією. Це особливо важливо для онлайн чатів, де користувачі очікують швидких та актуальних оновлень.

По-друге, використання протоколу WebSocket дозволяє уникнути недоліків традиційного підходу до реалізації чату з використанням HTTP-запитів. З’являється можливість передачі в реальному часі без використання зайвих запитів до серверу, що підвищує продуктивність і знижує навантаження на сервер.

Крім того, дослідження з розробки онлайн чату на базі протоколу WebSocket може мати практичну цінність для розробників, які зацікавлені у створенні ефективних та масштабованих додатків. WebSocket може бути використаний не тільки для реалізації чату, але й для інших програм, таких як онлайн ігри або потокове передавання мультимедіа.

Таким чином, дослідження з розробки онлайн чату на базі протоколу WebSocket є актуальним у світлі все більшої потреби в реальному часі взаємодії вебдодатків, переваг використання WebSocket перед традиційними методами та потенційної практичної користі для розробників.

Об’єкт дослідження роботи – обмін інформацією між вебсервером та браузером в режимі реального часу.

Предмет дослідження – створений вебдодаток для обміну повідомленнями між користувачами (чат).

Мета роботи: провести дослідження сучасних платформ для розробки

онлайн чатів, проаналізувати та обґрунтувати вибір технологій для розробки вебдодатків та на їх основі створити онлайн чат.

Завдання роботи:

– зробити огляд технологій WebSocket та визначити його ключові особливості;
– сформувати технічні та функціональні вимоги до системи;
– виконати проєктування архітектури та моделей даних;
– реалізувати клієнтську та серверну частини додатку;
– описати схему розгортання проєкту;
– протестувати додаток.

1 ОГЛЯД ТЕХНОЛОГІЙ

Онлайн чат може означати будь-який тип спілкування через Інтернет, який пропонує передачу текстових повідомлень у реальному часі від відправника до одержувача.Повідомлення чату зазвичай короткі, щоб інші учасники могли швидко відповісти. Це створює відчуття розмови, яке відрізняє чат від інших текстових форм спілкування в Інтернеті, таких як Інтернет-форуми та електронна пошта.

Перша система онлайн чату називалася Talkomatic і була створена Дугом Брауном і Девідом Р. Вуллі в 1973 році на системі PLATO в Університеті Іллінойсу. Він пропонував кілька каналів, кожен з яких міг вмістити до п’яти осіб, а повідомлення з’являлися на екранах усіх користувачів символ за символом у міру введення. Talkomatic був дуже популярний серед користувачів PLATO до середини 1980-х років. У 2014 році Браун і Вуллі випустили вебверсію Talkomatic.

Першу онлайн систему, яка використовує фактичну команду «чат», створили для The Source у 1979 році Том Вокер і Фріц Тейн з Dialcom, Inc.

Інші чат платформи процвітали в 1980-х роках. Серед перших із графічним інтерфейсом користувача був BroadCast, розширення Macintosh, яке стало особливо популярним в університетських містечках Америки та Німеччини.

Перший трансатлантичний Інтернет-чат відбувся між Оулу, Фінляндія, та Корваллісом, Орегон, у лютому 1989 року.

Першим спеціальним онлайн сервісом чату, який був широко доступний для громадськості, був CompuServe CB Simulator у 1980 році, створений виконавчим директором CompuServe Олександром «Сенді» Тревором у Колумбусі, штат Огайо.

Чат реалізовано в багатьох інструментах для відеоконференцій. Дослідження використання чату під час відеоконференцій, пов’язаних із

роботою, виявило, що чат під час зустрічей дозволяє учасникам спілкуватися, не перериваючи зустрічі, планувати дії щодо спільних ресурсів і забезпечує більшу взаємодію [1].

1.1 Періодичне опитування

Найпростіший спосіб отримати нову інформацію з сервера – періодичне опитування. Тобто регулярні запити до сервера: “Привіт, я тут, у вас є якась інформація для мене?”. Наприклад, раз на 10 секунд.

У відповідь сервер спочатку помічає собі, що клієнт онлайн, а по-друге – надсилає пакет повідомлень, які він отримав до цього моменту.

Це працює, але є і мінуси:

– повідомлення передаються із затримкою до 10 секунд (між запитами);
– навіть якщо повідомлень немає, сервер отримує запити кожні 10 секунд, навіть якщо користувач перейшов кудись в інше місце або сайт не використовується; з точки зору продуктивності, це може створювати велике навантаження.

Отже, якщо ми говоримо про невеликий сервіс, підхід може бути життєздатним, але загалом він потребує вдосконалення [2].

1.2 Тривале опитування

Так зване «тривале опитування» (Long polling) – це набагато кращий спосіб опитування сервера.

Також він дуже простий у реалізації та дозволяє отримувати повідомлення без затримок (див. рис. 1.1).

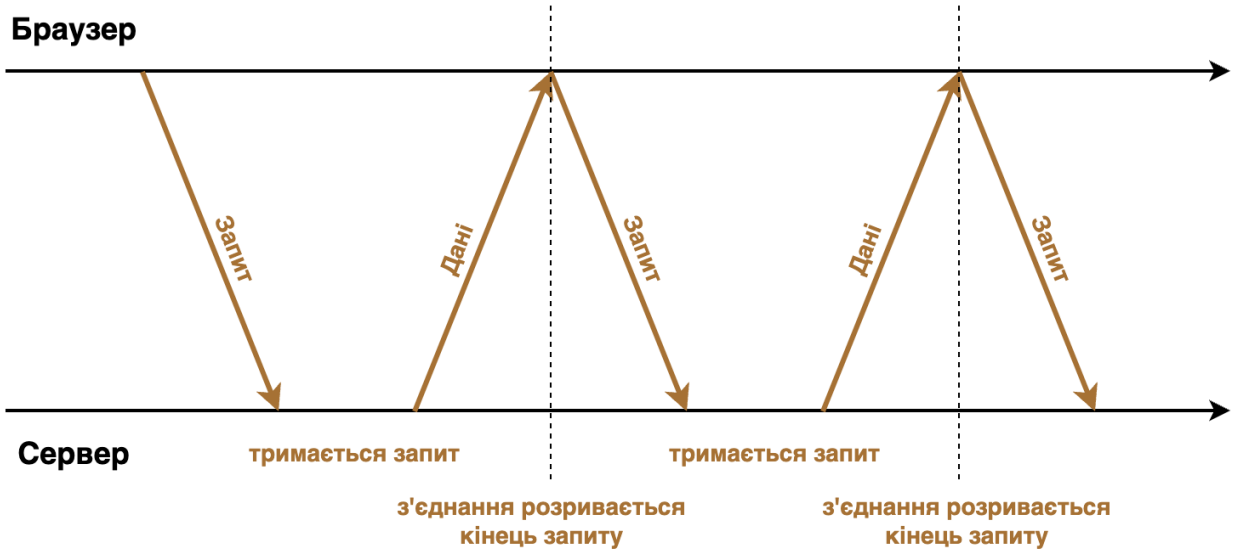

Рисунок 1.1 – Тривале опитування

Як це працює:

– на сервер надсилається запит;
– сервер не закриває з’єднання, поки не з’явиться повідомлення для відправки;
– коли з’являється повідомлення – сервер відповідає ним на запит;
– браузер негайно робить новий запит;
– ситуація, коли браузер відправив запит і має відкрите з’єднання з сервером, є стандартною для цього методу; лише після доставки повідомлення з’єднання із сервером закривається та відкривається нове [2].

1.3 Протокол WebSocket

WebSocket – це одночасно двонаправлений протокол, що призначений для обміну інформацією між браузером і вебсервером в режимі реального часу.

Вебсокети дозволяють клієнтському браузеру та серверу обмінюватись інформацією в реальному часі. На відміну від HTTP, клієнт отримує дані не лише після запиту на сервер, а й сервер сам може надсилати дані клієнту [3].

1.3.1 Ключові особливості WebSockets

WebSocket надає кілька ключових функцій, які роблять їх потужним інструментом для обміну даними між клієнтами і серверами в режимі реального часу. Розглянемо ключові особливості.

Двонаправлений, повнодуплексний зв'язок. WebSocket забезпечує двонаправлений, повнодуплексний зв'язок між клієнтом і сервером, що означає, що дані можуть бути відправлені та отримані в обох напрямках одночасно. Це забезпечує зв'язок в режимі реального часу між клієнтом і сервером без необхідності для клієнта постійно опитувати сервер на предмет оновлень.

Низька затримка. WebSocket здійснює зв’язок між клієнтом і сервером з низькою затримкою, що означає, що оновлення можуть бути відправлені й отримані швидко. Це особливо важливо для додатків, що вимагають спілкування в режимі реального часу, таких як онлайн ігри, кімнати чатів та фінансові торгові платформи.

Постійне з'єднання. WebSocket створює постійне з'єднання між клієнтом і сервером, що означає, що з'єднання залишається відкритим, навіть коли дані не відправляються або не приймаються. Це дозволяє серверу надсилати оновлення клієнта в режимі реального часу, без необхідності для клієнта робити запит.

Підтримка двійкових даних. WebSocket дозволяє передавати двійкові дані в додаток до текстових даних. Це корисно для додатків, яким потрібна передача великих обсягів даних, таких як потокове відео або аудіо.

Масштабованість. WebSocket можна використовувати для створення масштабованих додатків, оскільки вони гарантують ефективну комунікацію між клієнтами і серверами. Оскільки з'єднання залишається відкритим, сервер може обробляти декілька запитів одночасно без додаткових витрат на відкриття і закриття з'єднань для кожного запиту клієнта.

Міждоменна підтримка. WebSocket можна використовувати для обміну даними між різними доменами, що означає, що програми можуть бути створені для обміну даними між різними серверами або службами. Це особливо корисно

при створенні розподілених систем або мікросервісів.

Безпека. WebSocket надає вбудовані функції безпеки, включаючи шифрування і аутентифікації, які допомагають запобігти несанкціонований доступ до каналу зв'язку.

В цілому, ключові функції WebSocket забезпечують ефективний зв'язок між клієнтами і серверами в режимі реального часу з низькою затримкою. Вони широко використовуються в сучасних вебдодатках для різних варіантів використання і стали важливим інструментом для побудови масштабованих розподілених систем [3].

1.3.2 Як WebSocket працює

WebSocket працюють шляхом встановлення постійного двонаправленого каналу зв'язку між клієнтом і сервером, який забезпечує зв'язок у режимі реального часу без необхідності для клієнта постійно опитувати сервер на предмет оновлень. Протокол WebSocket заснований на протоколі управління передачею (TCP), який забезпечує надійну, упорядковану і перевірену на помилки доставку даних.

Процес встановлення з'єднання з WebSocket включає в себе кілька кроків (див. рис. 1.2).

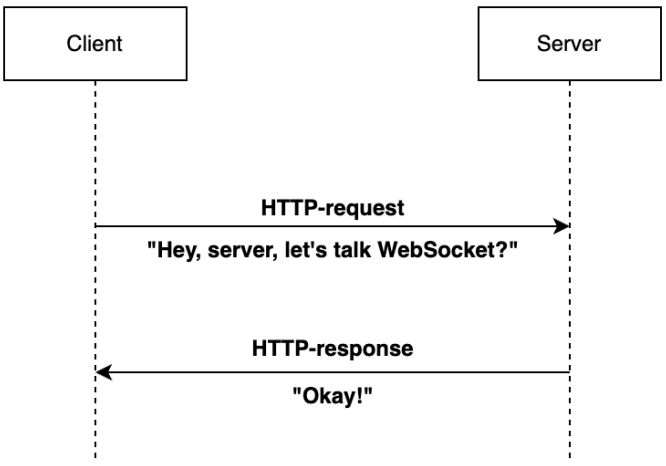

Рисунок 1.2 – Відкриття вебсокета

Клієнт відправляє handshake запит на сервер для ініціювання підключення до WebSocket (див. рис. 1.3). Цей запит виконується з використанням протоколу HTTP і включає в себе спеціальний заголовок “Upgrade”, який вказує, що клієнт хоче оновити з'єднання WebSocket.

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-Websocket-Key: Iv8io/9i+lYFgZWcXczP8Q
Sec-Websocket-Version: 13
```

Рисунок 1.3 – Запит «рукостискання»

Сервер відповідає на запит клієнта HTTP відповіддю, який включає спеціальний заголовок “Upgrade”, який вказує, що сервер готовий поновити з'єднання до з'єднання WebSocket (див. рис. 1.4).

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-Websocket-Accept: hsBlbuDTkk24srzE0TBUlZAlC2g
```

Рисунок 1.4 – Відповідь запиту «рукостискання»

Якщо сервер приймає запит клієнта, виконується квитування WebSocket. Це включає в себе обмін набором заголовків між клієнтом і сервером для встановлення з'єднання з WebSocket. Квитування включає заголовок “Sec-WebSocket-Key”, який являє собою випадково згенерований ключ, використовуваний для забезпечення безпеки з'єднання.

Як тільки з'єднання з WebSocket встановлено, що дані можуть бути

відправлені та отримані в обох напрямках між клієнтом і сервером. Це досягається за рахунок використання фреймів WebSocket, які являють собою пакети даних, що надсилаються між клієнтом і сервером. Фрейми WebSocket мають заголовок і корисне навантаження, і їх можна використовувати як для надсилання текстових, так і двійкових даних.

Підключення до WebSocket залишається відкритим до того часу, поки клієнт або сервер не закриють з'єднання чи не виникне мережева помилка.

Якщо з'єднання закрито, клієнт і сервер можуть ініціювати нове підключення до WebSocket, повторивши процес встановлення зв'язку.

Протокол WebSocket також включає в себе кілька вбудованих функцій, які забезпечують ефективний зв'язок між клієнтами і серверами. Наприклад, фрейми WebSocket можуть бути стиснуті, щоб зменшити обсяг даних, переданих по мережі. Крім того, повідомлення WebSocket можуть бути фрагментовані, щоб забезпечити передачу великих повідомлень меншими порціями.

В цілому, протокол WebSocket надає потужний інструмент для створення комунікаційних додатків реального часу, яким потрібно двонаправлений зв'язок між клієнтами і серверами з низькою затримкою. Встановлюючи постійне з'єднання між клієнтом і сервером, WebSocket забезпечують ефективну комунікацію, яка добре підходить для різних варіантів використання, включаючи кімнати чатів, онлайн ігри платформи для спільного редагування [3, 4].

1.3.3 Проблеми безпеки та їх усунення

Хоча WebSocket надають потужний інструмент для створення комунікаційних додатків реального часу, вони також створюють певні ризики для безпеки, які необхідно враховувати. Ось деякі з поширених проблем безпеки, пов'язаних з WebSocket, поряд з деякими стратегіями виправлення.

Міжсайтове перехоплення WebSocket (CSWSH) відбувається, коли

зловмисник використовує вразливість міжсайтового скриптинга (XSS), щоб вкрасти з'єднання WebSocket і використовувати його для зв'язку з сервером. Щоб запобігти цьому, важливо переконатися, що з'єднання з WebSocket встановлюється тільки між довіреними сторонами. Впровадження автентифікації на стороні сервера може допомогти запобігти такому типу атаки.

Підробка міжсайтових запитів (CSRF) відбувається, коли зловмисник відправляє запит WebSocket на сервер від імені користувача-жертви. Щоб запобігти цьому, важливо використовувати токени захисту від CSRF, щоб гарантувати, що запит WebSocket приймається лише з надійних джерел.

Ін'єкційні атаки відбуваються, коли зловмисник вводить шкідливий код або дані запиту, або відповідь WebSocket. Щоб запобігти атаці з використанням ін'єкцій, важливо правильно перевіряти і очищати всі дані, що відправляються по з'єднанню WebSocket. Реалізація перевірки вхідних даних і кодування вихідних даних може допомогти запобігти ін'єкційній атаці.

Атаки типу «відмова в обслуговуванні» (DoS) відбуваються, коли зловмисник завантажує сервер великою кількістю запитів WebSocket, що може перевантажити сервер і призвести до його аварійного завершення. Щоб запобігти DoS-атакам, важливо впроваджувати заходи по обмеженню швидкості і дроселювання. Це допоможе обмежити кількість запитів, які можуть бути оброблені сервером.

Атаки типу «Людина посередині» (MitM) відбуваються, коли зловмисник перехоплює трафік WebSocket і зчитує або змінює дані, що передаються між клієнтом і сервером. Щоб запобігти атаці MitM, важливо впровадити заходи шифрування і аутентифікації, такі як SSL / TLS, це буде гарантувати, що трафік WebSocket безпечний і не може бути перехоплений або змінений.

У цілому, важливо належним чином захистити з'єднання WebSocket, запровадивши такі заходи, як перевірка вхідних даних, кодування вихідних даних, аутентифікація, шифрування, обмеження швидкості і токени захисту від CSRF. Дотримуючись рекомендацій щодо забезпечення безпеки WebSocket, розробники можуть допомогти запобігти поширені проблеми безпеки і

гарантувати, що їх програми для обміну даними в реальному часі безпечні і надійні [3].

1.4 Функціональні вимоги до системи

У контексті роботи над чатом на базі протоколу WebSocket нам необхідно продумати і створити сторінку авторизації користувача, де він може ввести своє ім’я і, задля безпеки користувача, пароль, котрий буде складатись із букв і символів. Наступним кроком у роботі ми створимо сам чат, у якому користувач зможе бачити відображення своїх повідомлень та повідомлень інших учасників комунікації, а також список існуючих чатів. Також нам необхідно створити сторінку, де користувач зможе створювати колективний чат з декількома учасникам, додавати учасників та видаляти їх. Далі ми розробимо сам інпут створення повідомлень та їх редагування. Також нам необхідна сторінка налаштувань, де користувач зможе змінювати власне ім’я та пароль. Зважаючи на те, що чат має бути сучасним і зручним, ми розробимо дизайн адаптивний до мобільних девайсів. Варто підкреслити, що продукт буде працювати у форматі ріалтайму і повідомлення користувачу будуть приходити за допомогою WebSocket.

1.5 Висновки до розділу

Виходячи з технічного завдання та аналізу існуючих підходів до побудови вебдодатків для миттєвого обміну повідомленнями було обрано протокол WebSocket. Цей протокол має кращу продуктивність порівняно з опитуваннями завдяки встановленню лише одного з’єднання та ефективну браузерну підтримку.

2 АРХІТЕКТУРА

2.1 Предметно-орієнтоване проєктування

Розробка програмного забезпечення, що відповідає сучасним потребам і очікуванням бізнесу та користувачів – завдання непросте. Компанії-розробники програмного забезпечення поступово потребують дієвого способу для підвищення прозорості зв’язку між бізнесом і командою розробників. Предметно-орієнтоване проєктування (DDD) допомагає розв’язати цю проблему, сприяючи розумінню предмета й постійній співпраці між розробниками та фахівцями з бізнесу. Водночас зацікавлені сторони краще розуміють технічні можливості та обмеження.

Предметно-орієнтоване проєктування базується на кількох ключових концепціях, що дозволяють створювати предметно-орієнтоване програмне забезпечення.

Перший принцип – визначення пріоритетів предметної моделі. Вона представляє основні бізнес-суб’єкти, поведінку, відносини та правила. Реалізація коду безпосередньо відображає предметну модель, а не навпаки.

Другий ключовий принцип – розробка універсальної мови. Спільна лексика розробників і бізнес-експертів стандартизує термінологію та знання предметної галузі, усуваючи двозначність і неузгодженість між групами.

DDD включає стратегічний і тактичний етапи проєктування. Стратегічне проєктування – високорівнева організація предметної галузі у вигляді обмежених контекстів і підобластей. Тактичне проєктування охоплює шаблони та компоненти реалізації нижчого рівня – сутності, сервіси та репозиторії (див. рис. 2.1).

Поєднуючи методи моделювання, мови та контексту, DDD дозволяє створювати системи, що орієнтуються не лише на технічні вимоги, але й на основні концепції предметної галузі.

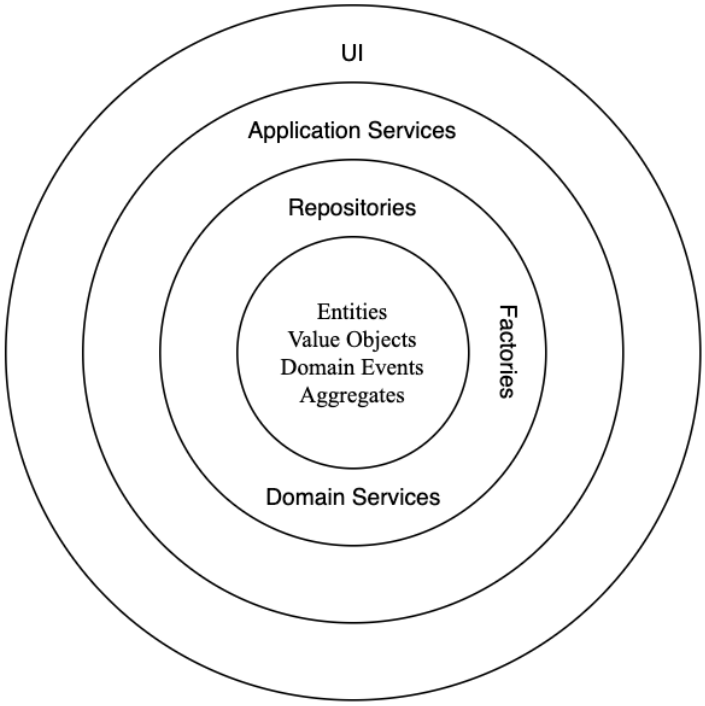

Рисунок 2.1 – Шари в предметно-орієнтованому програмуванні

У зв’язку з цим на думку одразу спадають гексагональна архітектура та чиста архітектура, що мають спільну мету – відокремлення завдань. Ви можете ізолювати основну бізнес-логіку від зовнішніх проблем, розділивши застосунки на вільно пов’язані компоненти.

У контексті DDD стратегічне проєктування є невід’ємною частиною розробки програмного забезпечення. Наведемо основні аспекти.

Огляд. Стратегічне проєктування починається з огляду проблемного предмета та цінності бізнесу. На цьому етапі досліджуються ключові концепції та процеси, визначаються основні бізнес-потреби та цілі.

Простір проблем і простір рішень. Структура стратегічного проєктування визначає ці два основні концептуальні простори. Простір проблем фокусується на дослідженні та аналізі предмета бізнесу, визначенні сутностей, агрегатів, послуг та взаємозв’язків між ними. Простір рішень стосується створення моделі, яка ефективно розв’язує проблеми, визначені в проблемному просторі.

Обмежені контексти (Bounded contexts). Обмежені контексти – це обмежені підрозділи предмета, що відповідають зонам відповідальності

конкретних груп розробників. Кожен контекст визначає свої сутності, агрегати, сервіси та правила. Керування межами контекстів має важливе значення для ізоляції та розуміння різних частин предмета.

Основний предмет. Основний предмет – це ядро бізнесу, його найважливіша і найцінніша частина. У межах стратегічного проєктування основний предмет критично важливий, оскільки є фокусом розробки та містить фундаментальні абстракції та бізнес-правила, що визначають функціональність програмного забезпечення.

Далі ми зосередимося на основі застосунку, також відомому як основний агрегат. Основний агрегат – це базовий елемент взаємодії, що містить ключову логіку предмета та цілісність даних. Він визначає основні операції та правила бізнесу.

Переходимо до інструментарію тактичного проєктування, що дає нам набір правил і шаблонів для побудови ефективної архітектури застосунків. Він містить такі поняття, як об’єкти-цінності, сутності, сервіси та агрегати. Цей інструментарій допомагає розробникам створювати гнучку архітектуру.

Тактичне проєктування також розрізняє сервіси застосунків і сервіси домену. Служби застосунків координують дії та взаємодію між різними сутностями та агрегатами в них. Що стосується доменних сервісів, то вони зберігають бізнес-логіку і виконують операції, пов’язані лише з предметною моделлю [4].

Так, тактичне проєктування допомагає створювати ефективні архітектури, які відображають предмет бізнесу та гарантують цілісність даних. Використання інструментів тактичного проєктування спрощує розробку та підтримку застосунків, полегшуючи розуміння та масштабування складних предметів.

Обмежений контекст у DDD – це локалізований набір моделей і правил, що застосовуються в межах певного предмета бізнесу. Це допомагає розмежувати та обмежити різні аспекти системи в межах певного контексту [4].

2.2 Стратегічний дизайн

Виходячи з вимог до проєкту нам потрібно лише 2 обмежені контексти (див. рис. 2.2).

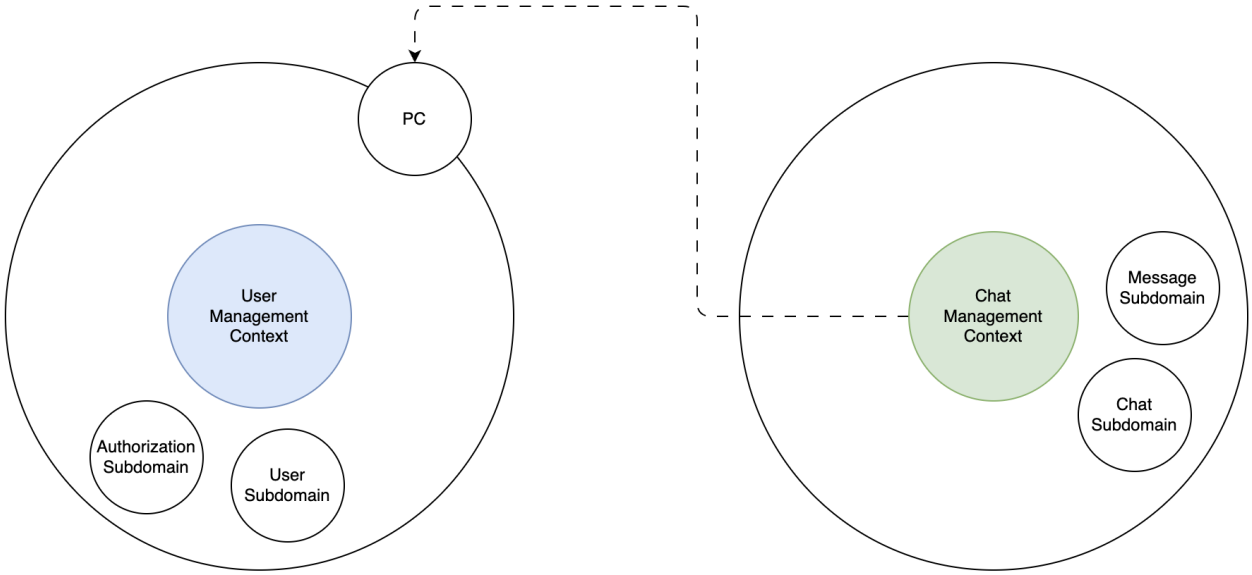

Рисунок 2.2 – Карта обмежених контекстів

Перший обмежений контекст по менеджменту користувачів. В цьому контексті буде проводитись авторизація, реєстрація, блокування, видалення, пошук та інші операції з користувачами. Він буде складатись з 2 піддоменів: Authorization та User.

Другий обмежений контекст по менеджменту чатів. В цьому контексті буде проводитись створення чатів, створення повідомлень, видалення чатів, редагування чатів і т.д. Тобто все що стосується обробки чатів та повідомлень. Він буде складатись з 2 піддоменів: Message та Chat.

Для більш зручної розробки, на початкових стадіях, ми не будемо виносити юзер менеджмент контекст як окремий мікросервіс, а лише будемо використовувати публічний контракт для комунікації. Але при написанні коду вони будуть максимально розділені і не взаємопов'язані. Таким чином ми зможемо в подальшому легко винести User Management Context як окремий мікросервіс, змінивши реалізацію публічного контракту на запит по http або

gRPC.

Ядром данного проєкту буде являтись Chat Management Context, так як в ньому буде зосереджена вся основна логіка. User Management Context тут виступає допоміжним і виконує роль тільки для авторизації.

Таким чином ми можемо використовувати цей чат в будь-якому з проєктів і підключати його як мікросервіс, не фокусуючись на авторизації та менеджменті користувачів, нам лише потрібен буде ендпоінт, за яким ми зможемо обрати користувачів. В якості авторизації будемо нами буде використовуватись той самий токен.

2.3 Тактичний дизайн

User Management Context розділяється на 2 субдомени: Authorization та User (див. рис. 2.3).

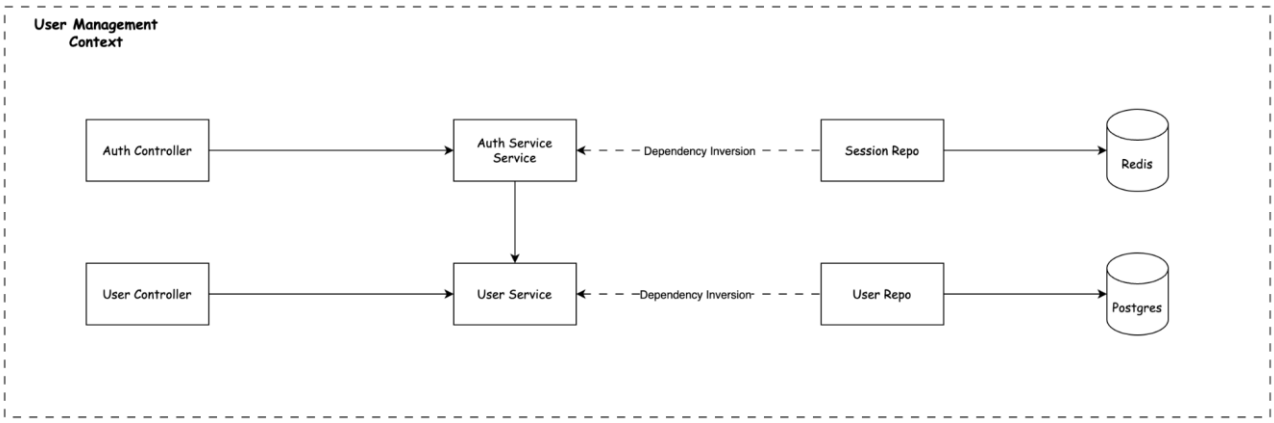

Рисунок 2.3 – Логічна топологія User management контексту

Авторизація буде виконуватись за допомогою емейлу або юзернейму та паролю. Після успішної валідації система поверне 32 значний токен, який ми збережемо в локал стореджі та будемо використовувати для запиту на всі ендпоінти. Для збереження юзер сесії ми будемо використовувати Redis, для більш швидкої вибірки користувача по токену. В якості entity буде анемічна

Session модель.

Для повного менеджменту користувача буде використовуватись User subdomain, а саме вибірка, пошук, фільтрація, створення, редагування, видалення, а також блокування (бан). Всі дані по юзеру будуть зберігатись в базі даних (Postgres). В якості entity буде анемічна User модель.

Всі запити будуть виконуватись через REST API запити.

По структурі бази даних в нас буде 2 таблиці: users та user_credentials (див. рис. 2.4). Зв’язок між таблицями буде один до одного. Таблиця users буде зберігати користувацькі дані. А в таблиці user_credentials зберігаються авторизаційні дані. Пароль будемо зберігати як hash, закодований за допомогою bcrypt.

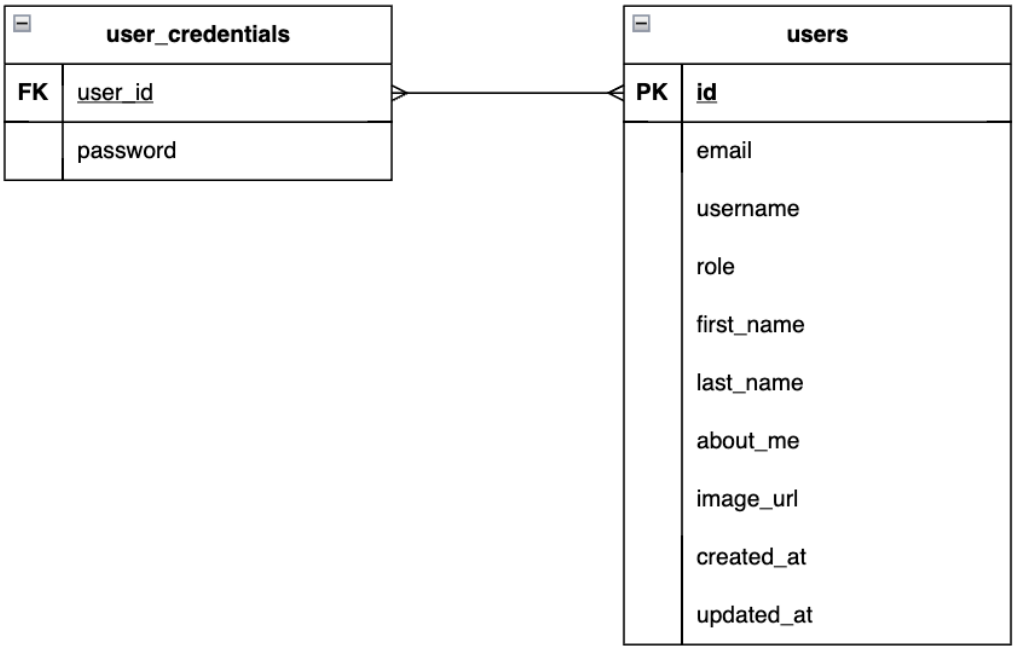

Рисунок 2.4 – Структура даних User management контексту

Chat Management Context розділяється на 2 домени: Chat та Message (див. рис. 2.5).

Chat subdomain буде виконувати роль менеджменту чатів. Всі чати будуть зберігатись в базі даних (Postgres). Message subdomain буде виконувати роль менеджменту повідомлень в чаті, а саме створення, редагування, видалення і т.д.

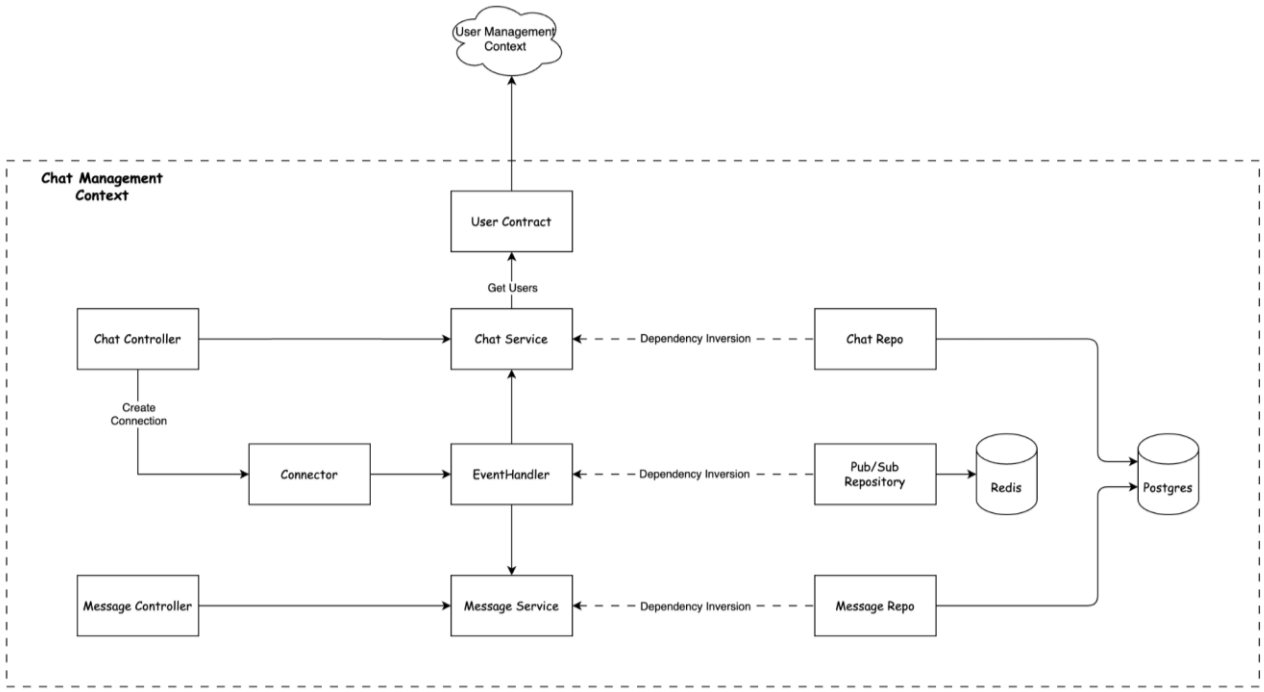

Рисунок 2.5 – Логічна топологія Chat management контексту

Для створення, редагування, прочитання а також видалення повідомлень буде використовуватись Websocket через EventHandler.

Для обох субдоменів вибірка всіх повідомлень для чату буде виконуватись через HTTP REST API запити.

По структурі бази даних в нас буде 3 таблиці: chats, messages та user_chats (див. рис. 2.6).

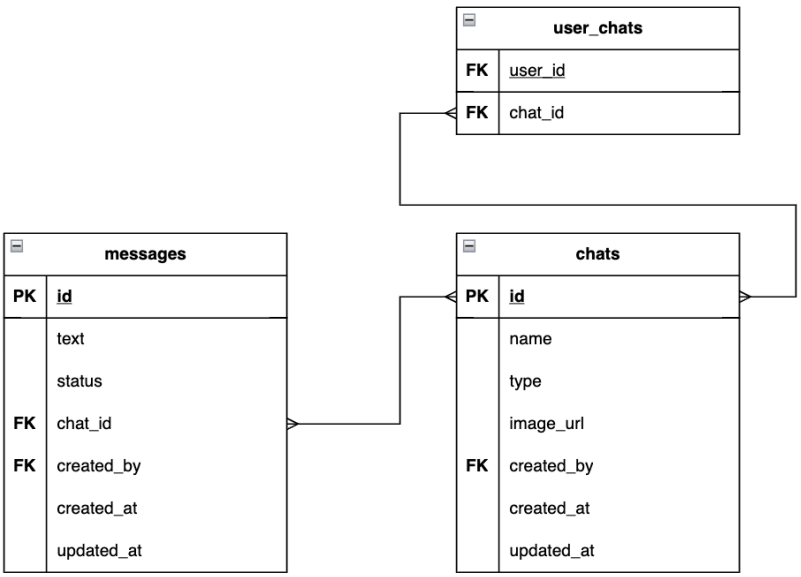

Рисунок 2.6 – Структура даних Chat management контексту

Зв’язок між chats та messages таблицями буде один до багатьох, а chats та user буде багато до багатьох через таблицю user_chats. Таблиця chats буде зберігати всі користувацькі чати. А в таблиці messages зберігаються повідомлення до чатів. Таблиця user_chats буде зберігати зв’язок між чатом та користувачем. Пароль будемо зберігати як hash, закодований за допомогою bcrypt.

Так як користувач буде приєднаний до однієї репліки чату, а інші користувачі будуть приєднані до іншої репліки, треба синхронізувати результати івентів за допомогою технології Pub/Sub (див. рис. 2.7). Для даної реалізації будемо використовувати Redis.

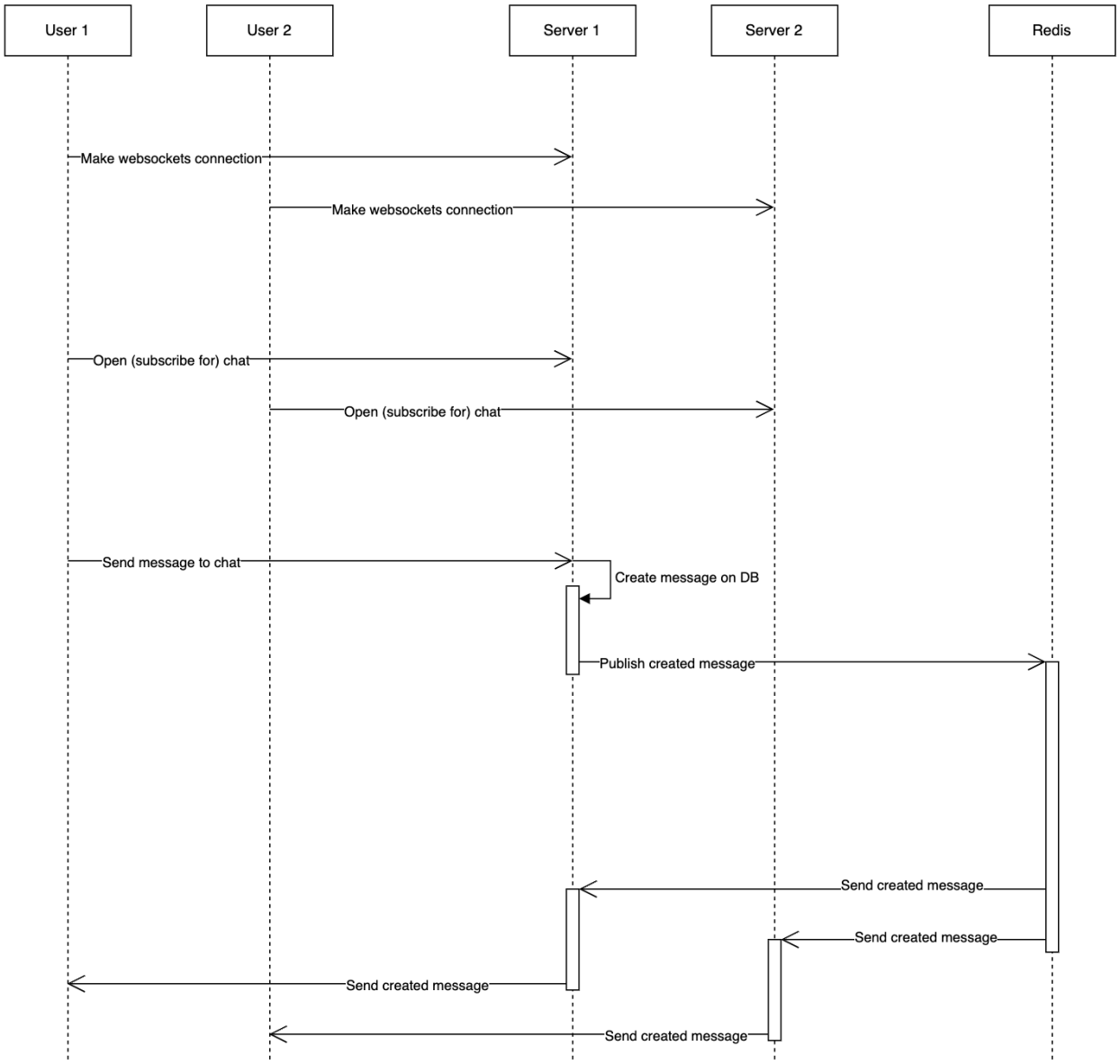

Рисунок 2.7 – Діаграма послідовності

2.4 UI/UX Дизайн

Для розробки UI/UX дизайну використаємо веб додаток Figma. Створимо 3 сторінки – логін, реєстрацію та сторінку чату (див. рис. 2.8 – 2.10).

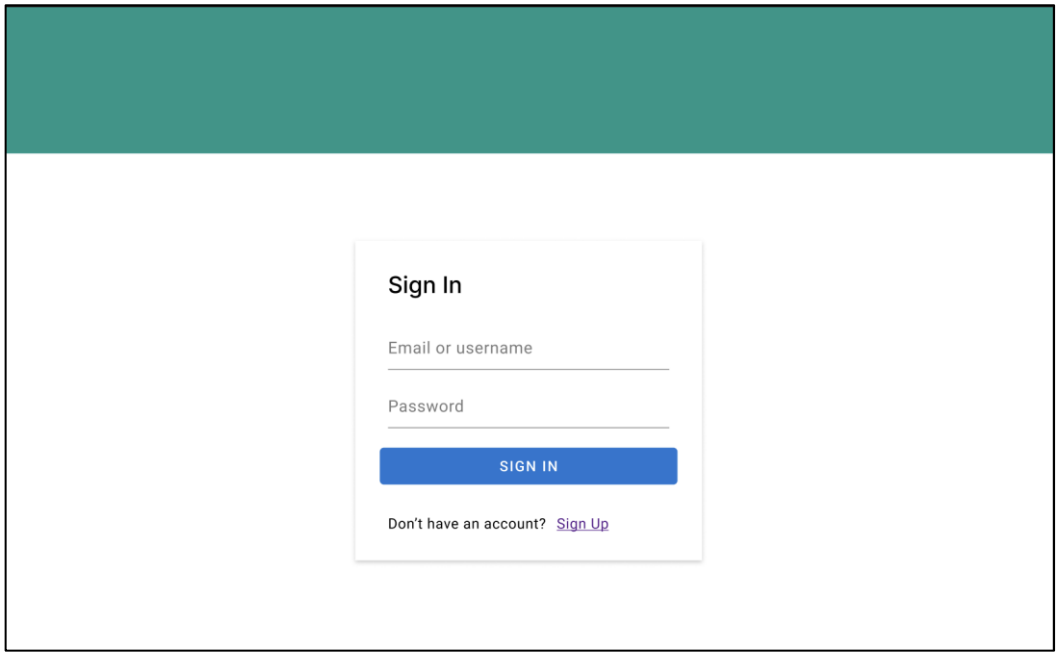

Рисунок 2.8 – Сторінка Sign In

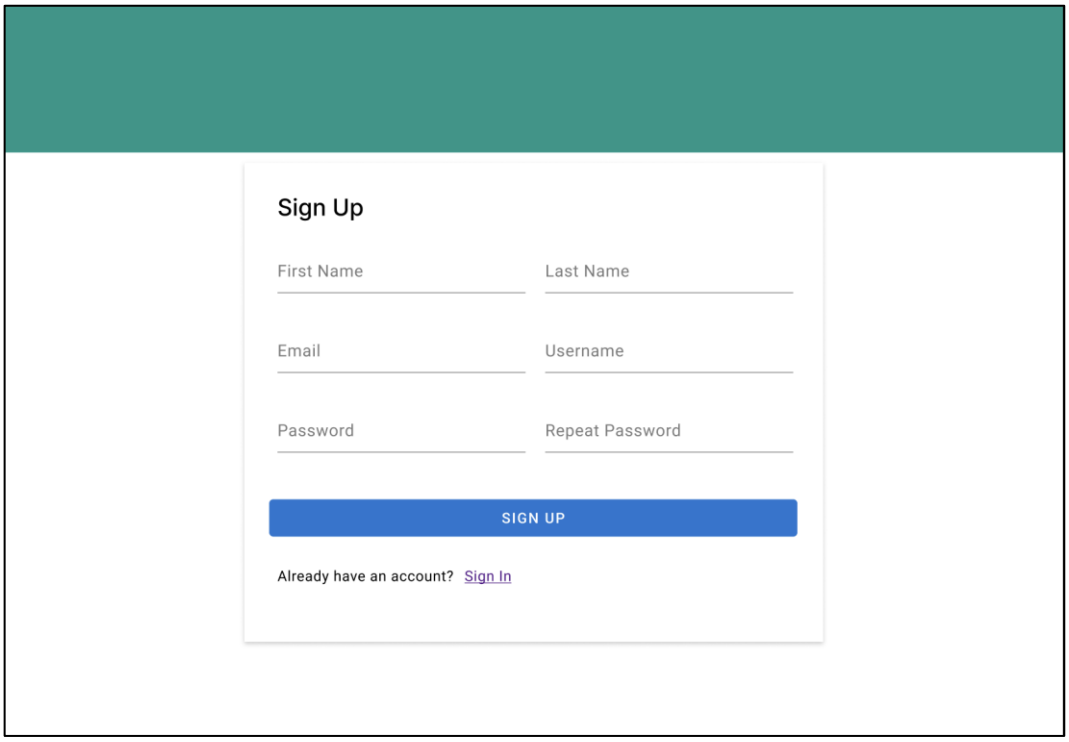

Рисунок 2.9 – Сторінка Sign Up

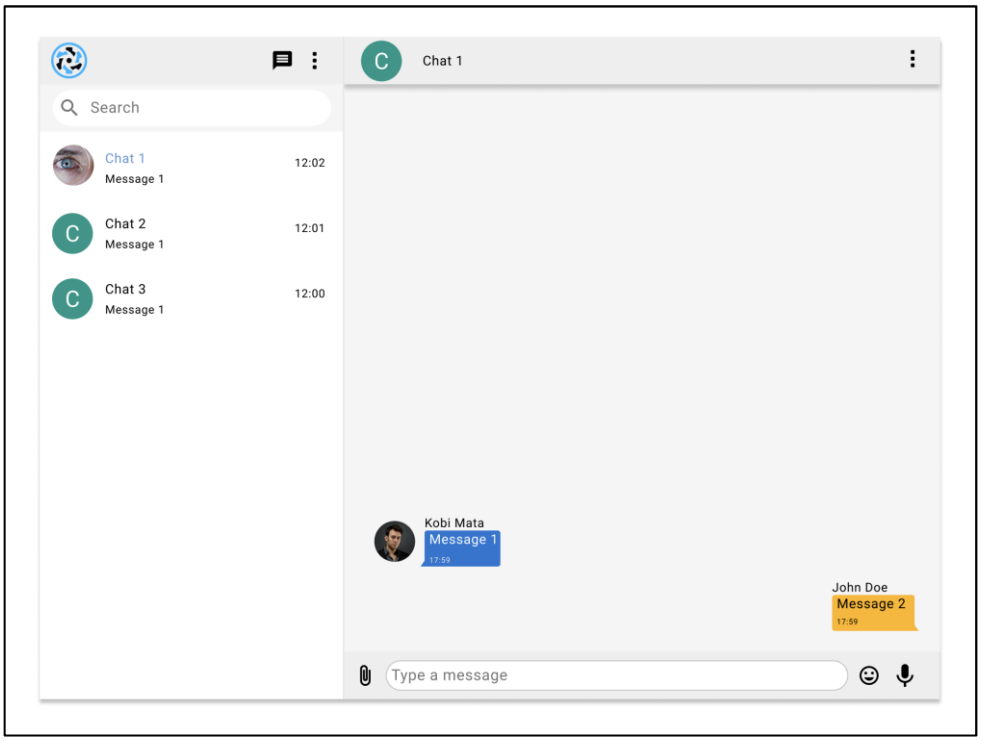

Рисунок 2.10 – Сторінка чату

Створимо 2 сторінки – логін, реєстрація для мобільного додатку (див. рис. 2.11).

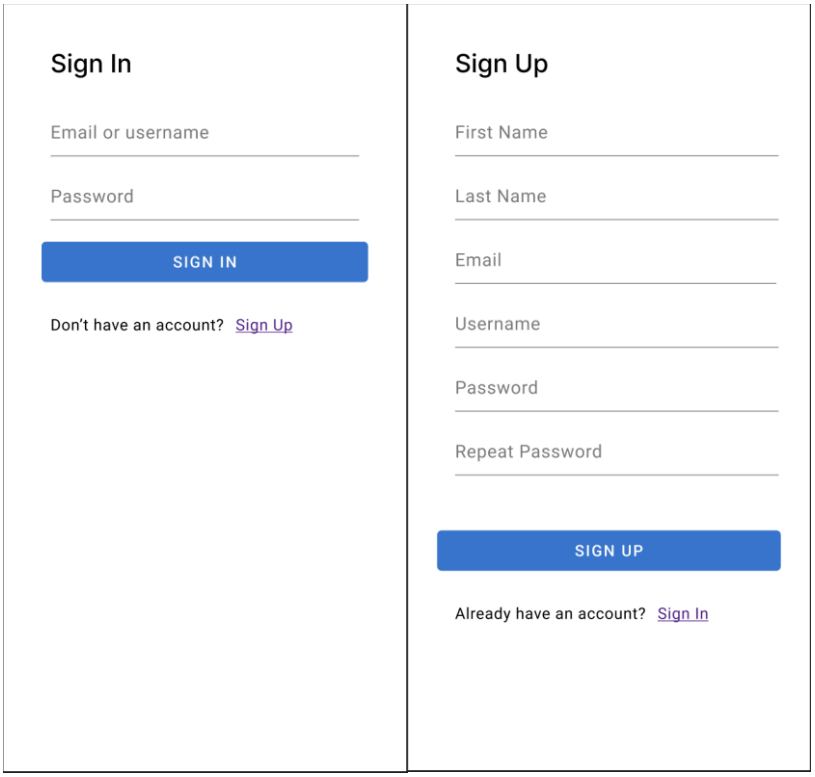

Рисунок 2.11 – Сторінка Sign In та Sign Up

Створимо сторінку чату для мобільного додатку (див. рис. 2.12).

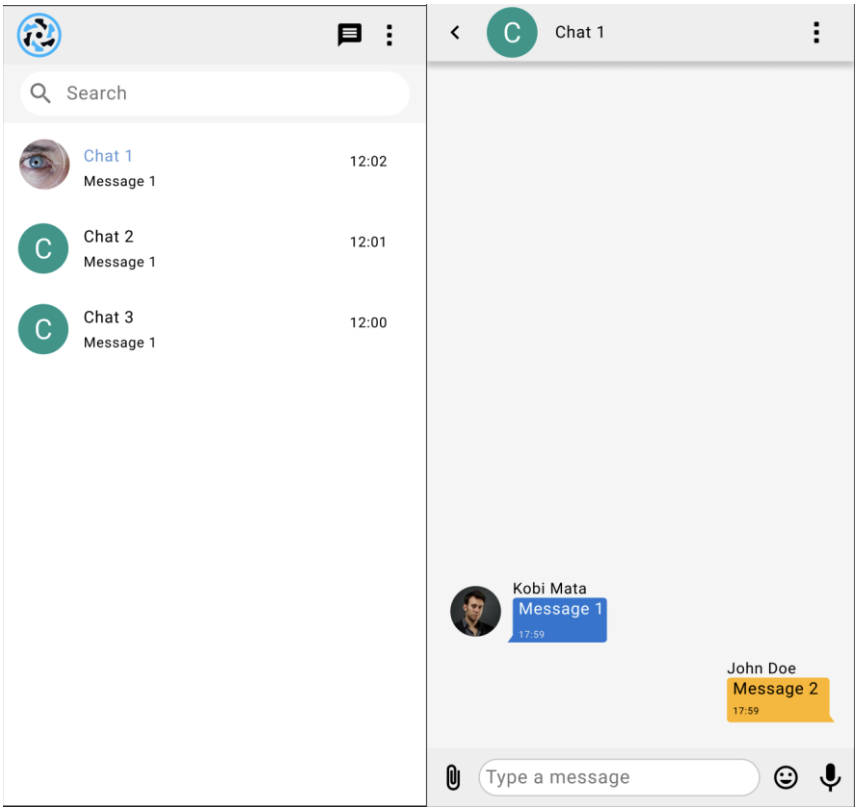

Рисунок 2.12 – Сторінка списку чатів та чату

2.5 Висновки до розділу

Виходячи з технічного завдання була спроєктована архітектура чату за всіма стандартами предметно орієнтованого програмування. Було проведено стратегічне та тактичне планування, побудовано карту обмежених контекстів, логічна топологія та структура даних User management та Chat management контексту, а також розроблено дизайн сторінок чату.

3 РЕАЛІЗАЦІЯ

3.1 Реалізація бекенду

Для розробки серверної частини була обрана мова Golang, так як вона проста в розробці, на ній легко писати модулі і мікросервіси, а також є дуже багато бібліотек [6].

Враховуючи те, що чат повинен відповідати дуже швидко, а також те, що буде дуже багато запитів на вибірку та пошук повідомлень, списку чатів, та користувацьких даних, для даної реалізації було обрано Fiber.

Fiber – це вебфреймворк, створений на основі Fasthttp, найшвидшого механізму HTTP для Go [7].

Для роботи з вебсокетами оберемо бібліотеку gorilla/websocket [8].

Для розробки чату нам потрібна надійна та стабільна реляційна база даних, так як буде суттєва взаємодія між таблицями під час пошуку та вибору даних. Тому було обрано PostgreSQL базу даних. Постгрес одна з найкращих безкоштовних баз даних на даний час. Її дуже добре використовувати для обробки великої кількості даних. А також набагато простіше та ефективніше масштабувати в порівнянні з іншими базами даних [9].

Для збереження сесії користувачів, а також для синхронізації повідомлень між репліками чату буде використано Redis [10].

Список ендпоінтів опишемо в таблиці (див. табл. 3.1).

Таблиця 3.1 – Ендпоінти

Ендпоінт	Опис
POST /auth/sign-in	Sign In ендпоінт використовується для авторизації в системі.
POST /auth/sign-up	За допомогою Sign Up ендпоінту користувач може зареєструватись в системі.

Продовження табл. 3.1

Ендпоінт	Опис
POST /auth/sign-out	Щоб вилогінитись та видалити користувацьку сесію, використовується Sign Out ендпоінт.
GET /users/current	Якщо необхідно отримати інформацію про користувача який залогований, можна використати ендпоінт Get Current User.
GET /users	Для пошуку користувачів використовується ендпоінт Get Users.
GET /chats	Get Chats ендпоінт використовується для вибірки списку чатів.
GET /chats/4	Щоб отримати інформацію по конкретному чату, використовується Get Chat ендпоінт.
GET /chats/4/messages	За допопомогою ендпоінту Get Chat Messages ми можемо вибрати весь список повідомлень по чату.

Для тестування, а також більш детальної документації по API використаємо Postman [11]. Збережемо Postman схему, а також змінні в нашому проєкті в папці docs (див. рис. 3.1).

```
docs
├── Chat.postman_collection.json
└── Local.postman_environment.json
```

Рисунок 3.1 – Документація по бекенд частині

Для створення, а в подальшому зміні, якщо буде необхідно, структури бази даних використовуються міграції. Всі міграції мають префікс з номером по порядку, що дозволяє нам керувати версійністю нашої бази даних (див. рис. 3.2). Кожна версія має два файли *.up.sql та *.down.sql для підвищення та зменшення версії. В up файлі, ми описуємо SQL запит, який синхронізує нашу базу даних з

конкретною версією (див. рис. 3.3). В down файлі описуємо SQL запит, який повертає зміни до попередньої версії (див. рис. 3.4).

Використаємо бібліотеку для міграції migrate/migrate [12].

```
migrations
├── 000001_create_users_table.down.sql
├── 000001_create_users_table.up.sql
├── 000002_create_user_credentials_table.down.sql
├── 000002_create_user_credentials_table.up.sql
├── 000003_create_chats_table.down.sql
├── 000003_create_chats_table.up.sql
├── 000004_create_user_chats_table.down.sql
├── 000004_create_user_chats_table.up.sql
├── 000005_create_messages_table.down.sql
└── 000005_create_messages_table.up.sql
```

Рисунок 3.2 – Список міграцій

```
CREATE TABLE IF NOT EXISTS users
(
    id         BIGSERIAL PRIMARY KEY,
    email      VARCHAR   NOT NULL,
    username   VARCHAR   NOT NULL,
    role       SMALLINT  NOT NULL DEFAULT 1,
    first_name VARCHAR   NOT NULL,
    last_name  VARCHAR   NOT NULL,
    about_me   VARCHAR   NOT NULL DEFAULT '',
    image_url  VARCHAR   NOT NULL DEFAULT '',
    created_at TIMESTAMP NOT NULL DEFAULT NOW(),
    updated_at TIMESTAMP NOT NULL DEFAULT NOW()
);
```

Рисунок 3.3 – Up міграція для створення таблиці users

```
DROP TABLE IF EXISTS users;
```

Рисунок 3.4 – Down міграція для створення таблиці users

3.2 Реалізація фронтенду

Для розробки клієнтської частини було обрано Vue.js фреймворк. Однією з перевега Vue.js є легкість і простота в розробці. В той же самий час вона має достатній функціонал в розробці великих проєктів [13].

В якості веб фреймворку було обрано Quasar, так як він оснований на Vue.js, дуже велика база UI компонентів, легко кастомізуються CSS стилі, та зручно розширюються JS компоненти [14].

В якості бібліотеки для управління станом використаємо Pinia, так як вона дуже проста в використанні, а також на відмінно від інших бібліотек є офіційною [15].

Для роботи з вебсокетами було обрано бібліотеку ReconnectingWebSocket, в якій вбудований реконнект, якщо було втрачено з'єднання з сервером [15].

Структура проєкту буде складатись з 2 основних частин: AuthLayout та MainLayout (див. рис. 3.5).

В першій частині, а саме авторизації, буде проводитись логін та реєстрація. На неї будуть потрапляти всі користувачі, які були незалоговані або не зареєстровані в системі. За допомогою навігаційних хуків будемо перевіряти чи може користувач потрапити на сторінку чату чи ні (див. рис. 3.6).

В другій частині буде виводитись список всіх доступних чатів, а також список повідомлень для обраного чату. За допомогою вебсокет конектора буде проводитись підключення до сервера, а також ми зможемо відправляти повідомлення на бекенд та створювати підписку на нові повідомлення чату (див. рис. 3.7).

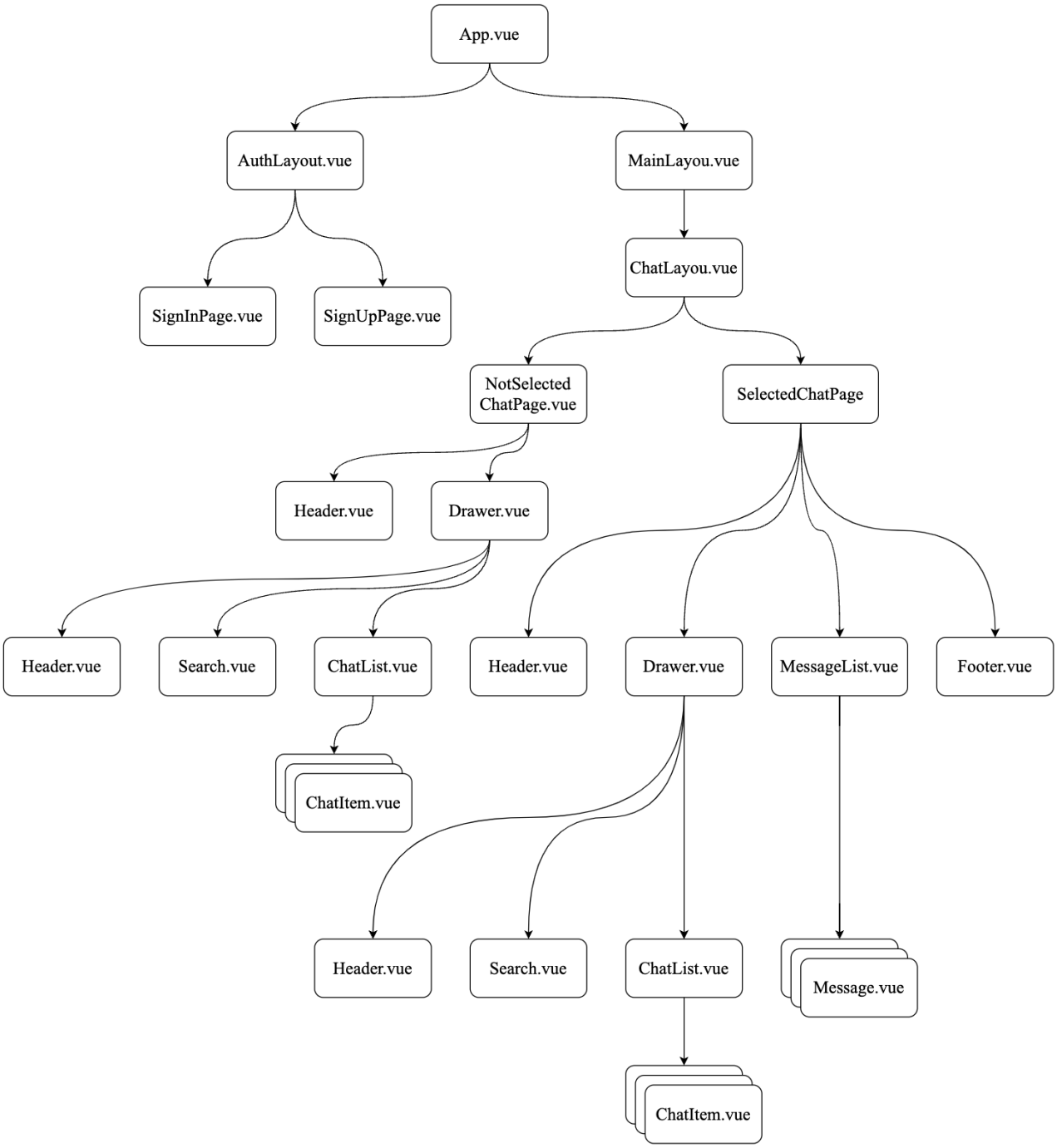

Рисунок 3.5 – Діаграма компонентів

```
router.beforeEach(async (to, from, next) => {
  const authStore = useAuthStore();
  if (!authStore.user) {
    await authStore.getCurrentUser();
  }
  next();
});
```

Рисунок 3.6 – Навігаційний хук beforeEach

```
export interface WebSocketConnector {
  connect(): void;
  disconnect(): void;
  sendMessage(type: number, data?: any): void;
  subscribeReconnect(func: ReconnectSubscriptionFunc): number;
  subscribe(eventType: number, func: SubscriptionFunc): number;
  unsubscribe(id: number): void;
}
```

Рисунок 3.7 – Вебсокет коннектор

3.3 Приклади роботи додатку

Коли користувач потрапить на головну сторінку його переадресовує на сторінку авторизації де він може внести свої авторизаційні дані та залогуватись в системі (див. рис. 3.8). Якщо користувач не має акаунту він може натиснути на посилання Sign Up та перейти на сторінку реєстрації (див. рис. 3.9), де він може прописати користувацькі дані та зареєструватись в системі.

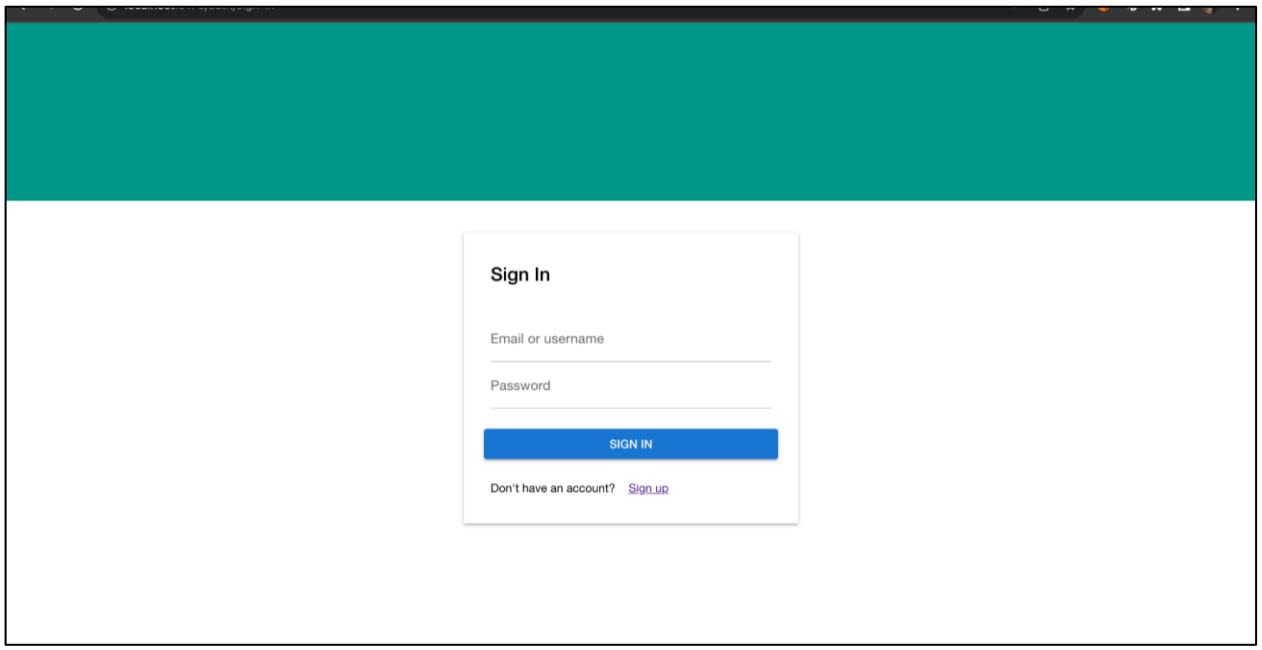

Рисунок 3.8 – Сторінка Sign In

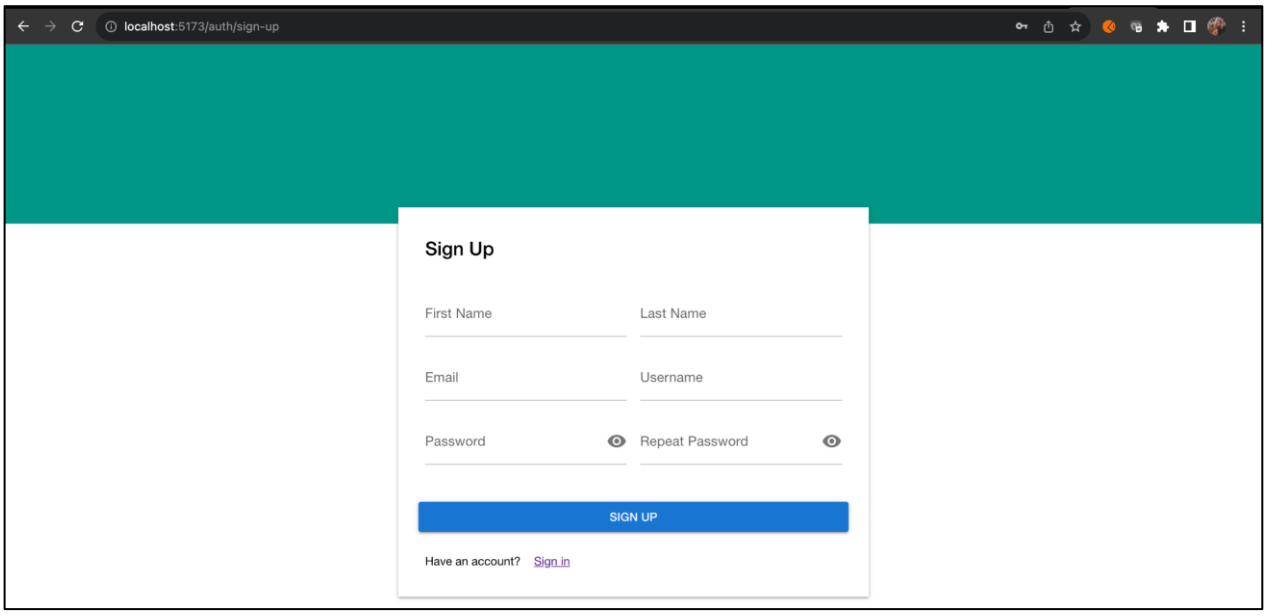

Рисунок 3.9 – Сторінка Sign Up

На наступному кроці, після авторизації, користувач потрапляє на сторінку чату (див. рис. 3.10).

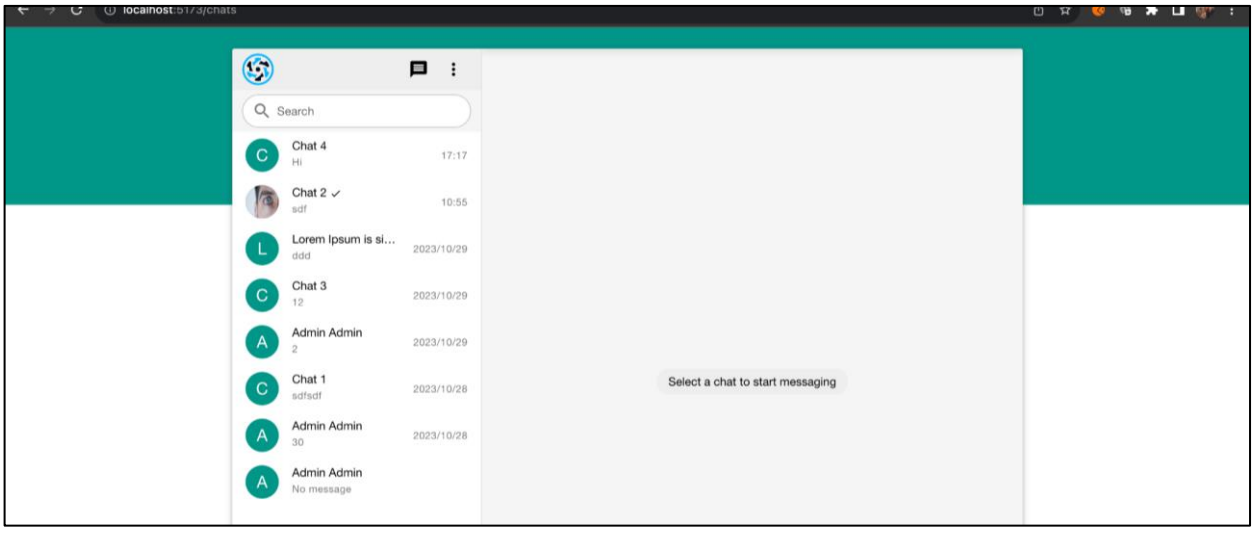

Рисунок 3.10 – Не обрано жодного чату

Так як він не обрав жодного чату, він бачить повідомлення «Select a chat to start messaging», що повідомляє користувачу необхідність обрати чат.

Коли користувач обрав чат (див. рис. 3.11), він може побачити всі повідомлення цього чату, а також може відправити повідомлення. Всі нові повідомлення автоматично будуть додані в цьому вікні без необхідності

оновлення сторінки. Наша система була адаптована і під мобільний девайс. Тому користувач може відкрити його через мобільний браузер (див. рис. 3.12).

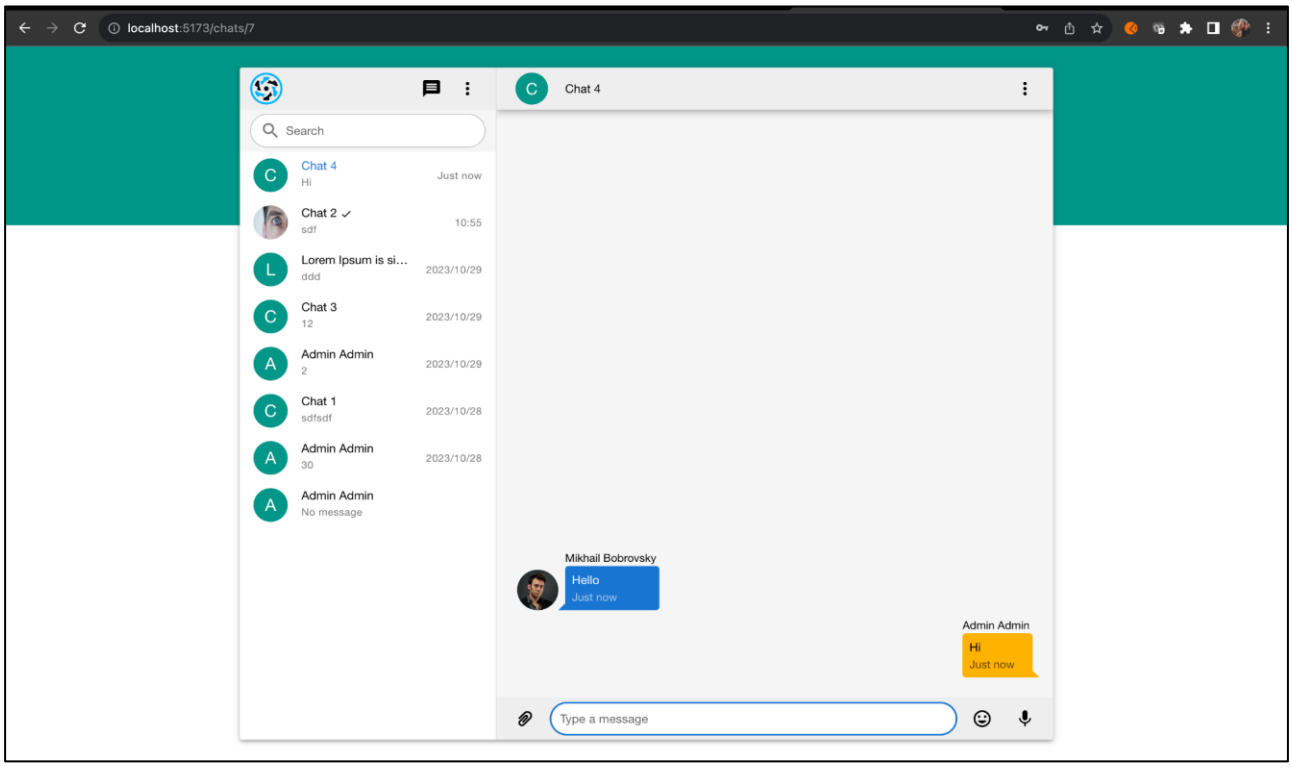

Рисунок 3.11 – Сторінка чату

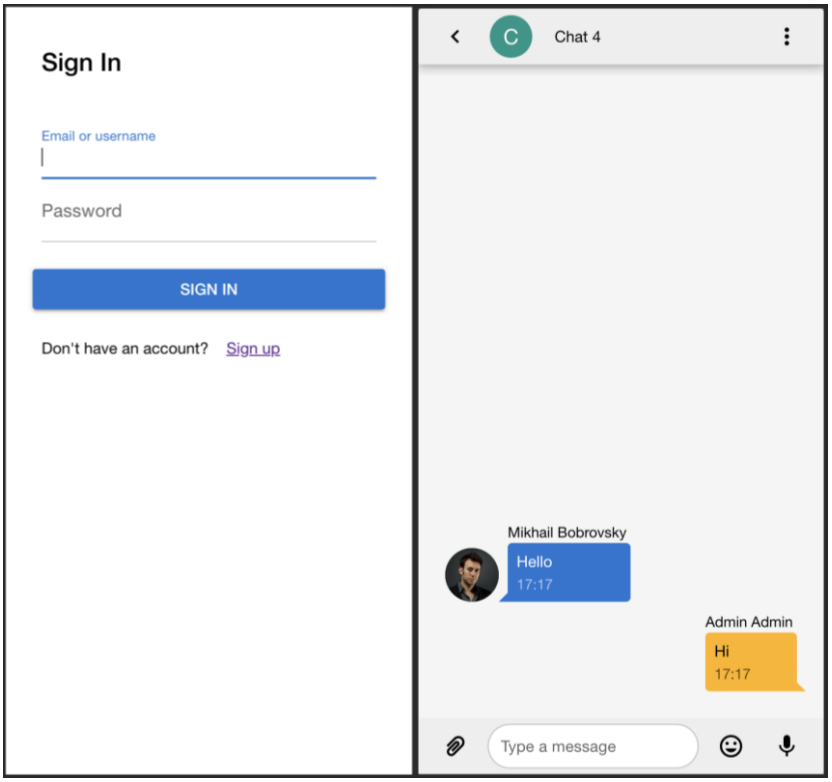

Рисунок 3.12 – Мобільний варіант чату

3.4 Розгортання

Для розгортання проєкту оберемо сервіси AWS, так як AWS зручний, гнучкий, безпечний, економічно ефективний та високопродуктивний.

AWS EKS (Elastic Kubernetes Service) для розгортання бекенду, CloudFront для розгортання фронтенду.

Для збереження медіафайлів будемо використовувати AWS S3 Bucket, яка буде окрема для кожного сервіса (user та chat) для більшої ізольованості.

В якості бази даних використаємо RDS а для Redis використаємо Amazon ElastiCache. Для налаштування DNS використаємо AWS Route 53, а для налаштування сертифікатів використаємо AWS Certificate Manager.

Для більш зручної роботи з AWS та розгортанням нашого продукту було розроблено фізичну топологію (див. рис. 3.13), на якій ми можемо побачити взаємодію компонентів.

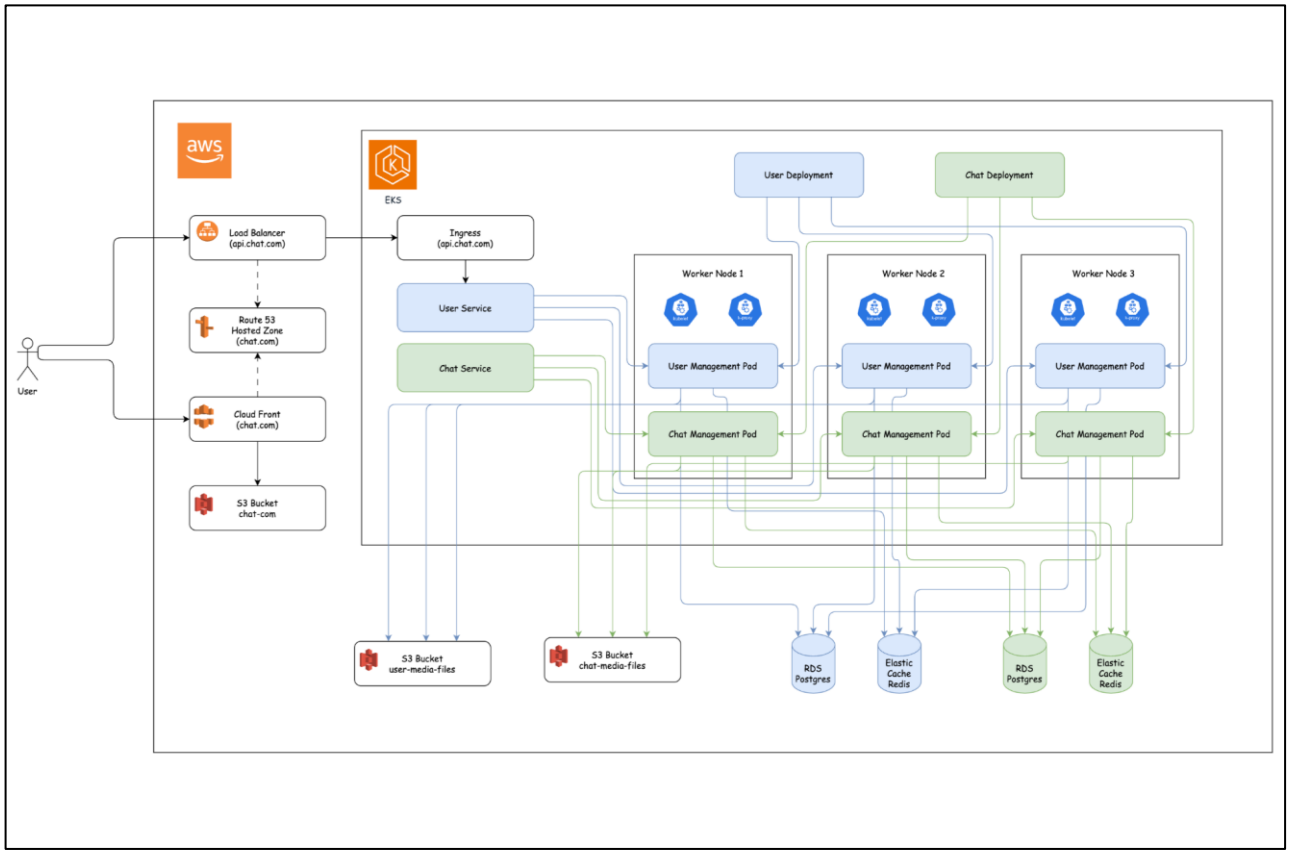

Рисунок 3.13 – Фізична топологія

3.5 Висновки до розділу

Було розроблено за архітектурними вимогами бекенд частину з урахуванням найновітніших технологій розробки. Фронтенд частина також відповідає високому рівню ефективності. Для збереження сесії користувачів, а також для синхронізації повідомлень між репліками чату було використано Redis. Продукт готовий, розгорнутий і протестований.

ВИСНОВКИ

У межах роботи було зроблено огляд технологій WebSocket та визначено його головні особливості. Так, зокрема було встановлено, що WebSocket надає кілька ключових функцій, які роблять їх потужним інструментом для обміну даними між клієнтами і серверами в режимі реального часу. До таких особливостей було віднесено: двонаправлений, повнодуплексний зв'язок; постійне з'єднання між сервером та клієнтом; низька затримка передачі даних; підтримка двійкових даних; масштабованість; міждоменна підтримка; безпека для клієнтів та серверу. Також було проаналізовано та відображено структуру роботи продуктів, що створені на базі протоколу WebSocket. Вдалось проаналізувати вимоги до проєктування продукту на базі протоколу WebSocket.

У ході роботи було досліджено і описано основні принципи предметно-орієнтованого програмування. Пильну увагу було зосереджено на основних принципах предметно-орієнтованого проєктування, стратегічному проєктуванні, тактичному проєктуванні, а також на ролі обмеженого контексту та ролі універсальної мови в DDD. Для реалізації практичної частини дослідницької роботи було створено стратегічний дизайн продукту, тактичний дизайн та UI/UX дизайн.

Для розробки серверної частини була обрана мова Golang, так як вона проста в розробці, на ній легко писати модулі і мікросервіси, а також є дуже багато бібліотек для розробки під Golang. Для розробки фронтенд частини було обрано Vue.js так як вона дуже просто, в той же час дуже функціональна і в повному обсязі забезпечує весь необхідний функціонал. Для розгортання проєкту оберемо сервіси AWS, так як AWS зручний, гнучкий, безпечний, економічно ефективний та високопродуктивний.

ПЕРЕЛІК ПОСИЛАНЬ

1. Contributors to Wikimedia projects. Online chat – Wikipedia. Wikipedia, the free encyclopedia. URL: https://en.wikipedia.org/wiki/Online_chat (дата звернення: 07.07.2023).
2. Тривале опитування. Сучасний підручник з JavaScript. URL: https://uk.javascript.info/long-polling (дата звернення: 10.07.2023).
3. 3. Real-time Communication Made Easy: An Introduction to WebSocket Protocol | CQR. CQR. URL: https://cqr.company/ua/wiki/protocols/real-time-communication-made-easy-an-introduction-to-websocket-protocol/ (дата звернення: 13.07.2023).
4. WebSocket. Сучасний підручник з JavaScript. URL: https://uk.javascript.info/websocket (дата звернення: 17.07.2023).
5. Предметно-орієнтоване проєктування (DDD): у чому користь підходу і хто його використовує. URL: https://dou.ua/forums/topic/45909/ (дата звернення: 20.07.2023).
6. The Go Programming Language. The Go Programming Language. URL: https://go.dev/ (дата звернення: 13.09.2023).
7. Welcome | Fiber. Welcome | Fiber. URL: https://docs.gofiber.io/ (дата звернення: 13.09.2023).
8. GitHub – gorilla/websocket: Package gorilla/websocket is a fast, well-tested and widely used WebSocket implementation for Go. GitHub. URL: https://github.com/gorilla/websocket (дата звернення: 13.09.2023).
9. PostgreSQL. PostgreSQL. URL: https://www.postgresql.org/ (дата звернення: 20.09.2023).
10. Redis as an in-memory data structure store quick start guide. Redis. URL: https://redis.io/docs/get-started/data-store/ (дата звернення: 27.09.2023).
11. API Documentation Tool | Postman. Postman API Platform. URL: https://www.postman.com/api-documentation-tool/ (дата звернення:

04.10.2023).

12. GitHub – golang-migrate/migrate: Database migrations. CLI and Golang library. GitHub. URL: https://github.com/golang-migrate/migrate (дата звернення: 10.10.2023).
13. Vue.js – The Progressive JavaScript Framework | Vue.js. Vue.js – The Progressive JavaScript Framework | Vue.js. URL: https://vuejs.org/ (дата звернення: 16.10.2023).
14. Quasar Framework – Build high-performance VueJS user interfaces in record time. Quasar Framework. URL: https://quasar.dev/ (дата звернення: 16.10.2023).
15. Pinia | The intuitive store for Vue.js. Pinia | The intuitive store for Vue.js. URL: https://pinia.vuejs.org/ (дата звернення: 16.10.2023).
16. GitHub – joewalnes/reconnecting-websocket: A small decorator for the JavaScript WebSocket API that automatically reconnects. GitHub. URL: https://github.com/joewalnes/reconnecting-websocket (дата звернення: 26.10.2023).

ДОДАТОК А

Приклади фронтенд коду

А.1 Макет авторизації

```
<script setup lang="ts"></script>

<template>
  <div class="auth-layout">
    <router-view />
  </div>
</template>

<style lang="scss">
.auth-layout {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100vh;
  width: 100%;
}
</style>
```

А.2 Основний макет

```
<script setup lang="ts">
import {useAuthStore} from "@/common/domain/authStore.js";
import AppSpinner from "@/ui-kit/AppSpinner/AppSpinner.vue";
```

```
import {onUnmounted} from "vue";

const authStore = useAuthStore();

onUnmounted(() => {
  authStore.resetStore();
});
</script>

<template>
  <div class="main-layout">
    <AppSpinner v-if="authStore.isFirstLoading" size="5em" />
    <router-view v-else-if="authStore.user" />
  </div>
</template>

<style lang="scss">
.main-layout {
  display: flex;
  align-items: center;
  justify-content: center;
  height: 100vh;
  width: 100%;
}
</style>
```

А.3 Макет чату

```
<script setup lang="ts">
import {onMounted, onUnmounted} from 'vue';
```

```
import {chatContext} from "@/chat/infrastructure/context.ts";
import {ChatService} from "@/chat/domain/ChatService.ts";
import {useChatsStore} from "@/chat/domain/chatsStore.ts";

const chatService = chatContext.get<ChatService>("ChatService");
const chatsStore = useChatsStore();

onMounted(async () => {
  chatService.connect();
  chatsStore.subscribeCreateMessage();
  chatsStore.subscribeUpdateMessagesStatus();
  await chatsStore.getChats();
});

onUnmounted(() => {
  chatsStore.clear();
  chatService.disconnect();
});
</script>

<template>
  <router-view />
</template>

<style lang="scss" scoped></style>
```

А.4 Список чатів

```
<script lang="ts" setup>
import {useRouter} from "vue-router";
```

```
import {useChatsStore} from "@/chat/domain/chatsStore.ts";
import {useChatStore} from "@/chat/domain/chatStore.ts";

import ChatItem from "./ChatItem/ChatItem.vue";

const router = useRouter();
const chatStore = useChatStore();
const chatsStore = useChatsStore();

function selectChat(id: number) {
  router.push({name: "chat", params: {id}});
  chatStore.getChat(id);
}
</script>

<template>
  <div class="chat-list">
    <q-list>
      <ChatItem
        v-for="chat in chatsStore.chats.items"
        :key="chat.id"
        clickable
        :chat="chat"
        :active="chatStore.chat?.id === chat.id"
        @click="selectChat(chat.id)"
      />
    </q-list>
  </div>
</template>
```

```
<style lang="scss" scoped>
@import "@/assets/styles/abstracts/scrollbar";

.chat-list {
  @include scrollbar;

  background-color: #FFFFFF;
  display: flex;
  height: 100%;
  width: 100%;
  overflow-x: hidden;
  overflow-y: auto;
}
</style>
```

A.5 WebSocket конектор

```
import ReconnectingWebSocket from "reconnecting-websocket";
import {TOKEN_LOCAL_STORAGE_KEY} from
"@/infrastructure/constants/localStorage.ts";
import type {Token} from "@/common/domain/Token.ts";

type Event = { type: number; data?: any };

export type ReconnectSubscriptionFunc = () => void;
export type SubscriptionFunc = (data?: any) => void;

export class WebSocketCoreImpl {
  protected lastSubscriptionId: number = 0;
```

```
  protected reconnectSubscriptions: { [key: string]: ReconnectSubscriptionFunc } =
{};
  protected eventTypeSubscriptions: { [key: string]: { [key: string]: SubscriptionFunc
} } = {};

  protected rws?: ReconnectingWebSocket;
  protected wsURL?: string;

  protected init(wsURL: string): void {
    this.wsURL = wsURL;
  }

  public connect(): void {
    const tokenJSON = localStorage.getItem(TOKEN_LOCAL_STORAGE_KEY);
    const token = tokenJSON ? JSON.parse(tokenJSON) as Token : null;

    if (!token) {
      return;
    }

    this.rws = new
ReconnectingWebSocket(`${this.wsURL}?token=${token.token}`);
    this.rws.onopen = () => {
      console.log("opened ws connection");
    }

    this.rws.onmessage = (e) => {
      console.log("got event: ", e.data);
```

```
    const event = JSON.parse(e.data) as Event;

    const subscriptions = this.eventTypeSubscriptions[event.type];
    if (!subscriptions) {
      return;
    }

    Object.values(subscriptions)?.forEach((func) => {
      func(event.data);
    });
  }

  this.rws.onclose = () => {
    console.log("closed ws connection");
  }
}

public disconnect(): void {
  this.rws?.close();
  this.reconnectSubscriptions = {};
  this.eventTypeSubscriptions = {};
  this.rws = undefined;
}

public sendMessage(type: number, data?: any): void {
  this.rws?.send(JSON.stringify({ type, data }));
}

public subscribeReconnect(func: ReconnectSubscriptionFunc): number {
  this.reconnectSubscriptions[++this.lastSubscriptionId] = func;
```

```
    return this.lastSubscriptionId;
  }

  public subscribe(eventType: number, func: SubscriptionFunc): number {
    if (!this.eventTypeSubscriptions[eventType]) {
      this.eventTypeSubscriptions[eventType] = {};
    }
    this.eventTypeSubscriptions[eventType][++this.lastSubscriptionId] = func;
    return this.lastSubscriptionId;
  }

  public unsubscribe(id: number): void {
    if (id <= 0) {
      return;
    }

    delete this.reconnectSubscriptions[id];

    Object.values(this.eventTypeSubscriptions).forEach((subscriptions) => {
      delete subscriptions[id];
    });
  }
}
```

ДОДАТОК Б

Приклади бекенд коду

Б.1 WebSocketConnector

```
package connector

import (
	"chat/internal/common/domain"
	"encoding/json"
	"github.com/fasthttp/websocket"
	"github.com/google/uuid"
)

type WebsocketConnection struct {
	connectionID string

	conn      *websocket.Conn
	connector Connector
	session   *domain.Session

	messageChan chan []byte
	closeChan   chan struct{}

	isConnected bool
	isClosed    bool
}

func (c *WebsocketConnection) IsClosed() bool {
```

```
	return c.isClosed
}

func (c *WebsocketConnection) GetConnectionID() string {
	return c.connectionID
}

func (c *WebsocketConnection) GetConnector() Connector {
	return c.connector
}

func (c *WebsocketConnection) SetConnector(connector Connector) {
	c.connector = connector
}

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

func (c *WebsocketConnection) GetCloseChan() chan struct{} {
	return c.closeChan
}

func (c *WebsocketConnection) SendEvent(eventType uint64, data any) error {
	var err error

	event := Event{
		Type: eventType,
	}
```

```
		event.Data, err = json.Marshal(data)
		if err != nil {
			return err
		}

		c.conn.WriteJSON(event)

		return nil
}

func (c *WebsocketConnection) Connect() {
		go c.connect()
}

func (c *WebsocketConnection) connect() {
		if c.isConnected || c.isClosed {
			return
		}

		c.isConnected = true

		defer func() {
			c.isConnected = false
			c.isClosed = true
		}()

		for {
			select {
			case <-c.closeChan:
				return
```

```
                default:
                        _, msgData, err := c.conn.ReadMessage()
                        if err != nil {
                                c.isConnected = false
                                return
                        }
                        c.messageChan <- msgData
                }
        }
}

func (c *WebsocketConnection) Close() {
        if !c.isConnected {
                return
        }
        close(c.closeChan)
        c.conn.Close()
}

func (c *WebsocketConnection) GetSession() *domain.Session {
        return c.session
}

func NewWebSocketConnection(conn *websocket.Conn, session *domain.Session)
*WebsocketConnection {
        return &WebsocketConnection{
                connectionID: uuid.NewString(),
                conn:         conn,
                session:      session,
                messageChan:  make(chan []byte),
```

```
            closeChan:    make(chan struct{}),
        }
}
```

Б.2 WebSocketConnection

```
package connector

import (
        "chat/internal/common/domain"
        "encoding/json"
        "github.com/fasthttp/websocket"
        "github.com/google/uuid"
)

type WebsocketConnection struct {
        connectionID string

        conn      *websocket.Conn
        connector Connector
        session   *domain.Session

        messageChan chan []byte
        closeChan   chan struct{}

        isConnected bool
        isClosed    bool
}
```

```
func (c *WebsocketConnection) IsClosed() bool {
	return c.isClosed
}

func (c *WebsocketConnection) GetConnectionID() string {
	return c.connectionID
}

func (c *WebsocketConnection) GetConnector() Connector {
	return c.connector
}

func (c *WebsocketConnection) SetConnector(connector Connector) {
	c.connector = connector
}

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

func (c *WebsocketConnection) GetCloseChan() chan struct{} {
	return c.closeChan
}

func (c *WebsocketConnection) SendEvent(eventType uint64, data any) error {
	var err error

	event := Event{
		Type: eventType,
	}
```

```
	event.Data, err = json.Marshal(data)
	if err != nil {
		return err
	}

	c.conn.WriteJSON(event)

	return nil
}

func (c *WebsocketConnection) Connect() {
	go c.connect()
}

func (c *WebsocketConnection) connect() {
	if c.isConnected || c.isClosed {
		return
	}

	c.isConnected = true

	defer func() {
		c.isConnected = false
		c.isClosed = true
	}()

	for {
		select {
		case <-c.closeChan:
			return
```

```
                default:
                        _, msgData, err := c.conn.ReadMessage()
                        if err != nil {
                                c.isConnected = false
                                return
                        }
                        c.messageChan <- msgData
                }
        }
}

func (c *WebsocketConnection) Close() {
        if !c.isConnected {
                return
        }
        close(c.closeChan)
        c.conn.Close()
}

func (c *WebsocketConnection) GetSession() *domain.Session {
        return c.session
}

func NewWebSocketConnection(conn *websocket.Conn, session *domain.Session)
*WebsocketConnection {
        return &WebsocketConnection{
                connectionID: uuid.NewString(),
                conn:         conn,
                session:      session,
                messageChan:  make(chan []byte),
```

```
		closeChan:    make(chan struct{}),
	}
}
```

Б.3 UserRepository

```
package repository

import (
	"chat/internal/common/domain"
	"chat/internal/common/errors"
	"chat/internal/common/repository"
	"chat/internal/user/common"
	"context"
	"database/sql"
	"fmt"
	"strings"
)

type UserRepoImpl struct {
	db *sql.DB
}

var (
	userFieldsMapping = map[string]string{
		"id":       "id",
		"email":    "email",
		"username": "username",
		"role":     "role",
```

```
            "firstName": "first_name",
            "lastName":  "last_name",
            "createdAt": "created_at",
            "updatedAt": "updated_at",
        }
)

func (r *UserRepoImpl) scan(rows *sql.Rows) ([]domain.User, error) {
        users := make([]domain.User, 0)

        for rows.Next() {
                var user domain.User

                fields := []any{
                        &user.ID,
                        &user.Email,
                        &user.Username,
                        &user.Role,
                        &user.FirstName,
                        &user.LastName,
                        &user.AboutMe,
                        &user.Image.Url,
                        &user.CreatedAt,
                        &user.UpdatedAt,
                }

                err := rows.Scan(fields...)
                if err != nil {
                        return nil, errors.NewDatabaseError(common.UserDomain, err)
                }
```

```
            users = append(users, user)
        }

        return users, nil
}

func (r *UserRepoImpl) buildFilter(filter domain.UserFilter) ([]any, []string) {
        values := make([]any, 0)
        where := make([]string, 0)

        if len(filter.IDs) > 0 {
                var params []string
                for _, id := range filter.IDs {
                        values = append(values, id)
                        params = append(params, fmt.Sprintf("$%d", len(values)))
                }
                where = append(where, fmt.Sprintf(
                        "u.id IN (%s) ", strings.Join(params, ",")))
        }

        if len(filter.Emails) > 0 {
                var params []string
                for _, email := range filter.Emails {
                        values = append(values, email)
                        params = append(params, fmt.Sprintf("$%d", len(values)))
                }
                where = append(where, fmt.Sprintf(
                        "u.email IN (%s) ", strings.Join(params, ",")))
        }
```

```
	if len(filter.UserNames) > 0 {
		var params []string
		for _, userName := range filter.UserNames {
			values = append(values, userName)
			params = append(params, fmt.Sprintf("$%d", len(values)))
		}
		where = append(where, fmt.Sprintf(
			"u.username IN (%s) ", strings.Join(params, ",")))
	}

	if len(filter.Roles) > 0 {
		var params []string
		for _, role := range filter.Roles {
			values = append(values, role)
			params = append(params, fmt.Sprintf("$%d", len(values)))
		}
		where = append(where, fmt.Sprintf(
			"role IN (%s) ", strings.Join(params, ",")))
	}

	if len(filter.Search) > 0 {
		values = append(values, fmt.Sprintf("%%%s%%", filter.Search))
		where = append(where, fmt.Sprintf(
			"CONCAT(u.id, u.email, u.username, u.first_name, u.last_name)
ILIKE $%d ", len(values)))
	}

	return values, where
}
```

```
func (r *UserRepoImpl) GetUser(ctx context.Context, id uint64) (*domain.User,
error) {
	sql := fmt.Sprintf("SELECT %s FROM %s AS u WHERE id = $1", userFields,
usersTableName)

	rows, err := r.db.QueryContext(ctx, sql, id)
	if err != nil {
		return nil, errors.NewDatabaseError(common.UserDomain, err, "error
on query user")
	}

	defer rows.Close()

	users, err := r.scan(rows)
	if err != nil {
		return nil, err
	}

	if len(users) > 0 {
		return &users[0], nil
	}

	return nil, nil
}

func (r *UserRepoImpl) GetUsers(ctx context.Context, filter *domain.UserFilter)
([]domain.User, error) {
	if filter == nil {
		filter = &domain.UserFilter{}
	}
```

```
	values, where := r.buildFilter(*filter)

	query := fmt.Sprintf("SELECT %s FROM %s AS u", userFields,
usersTableName)

	if len(where) > 0 {
		query = fmt.Sprintf("%s WHERE %s", query, strings.Join(where, "
AND "))
	}

	if filter.Sort != nil {
		query = fmt.Sprintf(`%s ORDER BY %s %s`,
			query,
			userFieldsMapping[filter.Sort.SortBy],
			filter.Sort.SortDir,
		)
	}

	if filter.Limit != nil {
		query = fmt.Sprintf(`%s LIMIT %d`, query, *filter.Limit)
	}

	if filter.Offset != nil {
		query = fmt.Sprintf(`%s OFFSET %d`, query, *filter.Offset)
	}

	rows, err := r.db.QueryContext(ctx, query, values...)
	if err != nil {
		return nil, errors.NewDatabaseError(common.UserDomain, err, "error
on query users")
```

```
	}

	defer rows.Close()

	users, err := r.scan(rows)
	if err != nil {
		return nil, err
	}

	return users, nil
}

func (r *UserRepoImpl) GetUsersCount(ctx context.Context, filter
*domain.UserFilter) (uint64, error) {
	if filter == nil {
		filter = &domain.UserFilter{}
	}

	values, where := r.buildFilter(*filter)

	query := fmt.Sprintf("SELECT COUNT(*) AS count FROM %s AS u",
usersTableName)

	if len(where) > 0 {
		query = fmt.Sprintf("%s WHERE %s", query, strings.Join(where, "
AND "))
	}

	rows, err := r.db.QueryContext(ctx, query, values...)
	if err != nil {
```

```
		return 0, errors.NewDatabaseError(common.UserDomain, err, "error on query users count")
	}

	defer rows.Close()

	var count uint64

	if rows.Next() {
		err := rows.Scan(&count)
		if err != nil {
			return 0, errors.NewDatabaseError(common.UserDomain, err, "error on scan users count")
		}
	}

	return count, nil
}

func (r *UserRepoImpl) CreateUser(ctx context.Context, user domain.User, tx repository.Tx) (*domain.User, error) {
	values := []any{
		user.Email,
		user.Username,
		user.Role,
		user.FirstName,
		user.LastName,
	}

	query := fmt.Sprintf(`
```

```
        WITH u AS (
            INSERT INTO %[1]s
                    (email, username, role, first_name, last_name)
                VALUES
                    ($1, $2, $3, $4, $5)
                RETURNING *
        )
        SELECT %[2]s FROM u
`, usersTableName, userFields)

var (
        err  error
        rows *sql.Rows
)

if tx != nil {
        rows, err = tx.QueryContext(ctx, query, values...)
} else {
        rows, err = r.db.QueryContext(ctx, query, values...)
}

if err != nil {
        return nil, errors.NewDatabaseError(common.UserDomain, err)
}

defer rows.Close()

users, err := r.scan(rows)
if err != nil {
        return nil, err
```

```
	}

	if len(users) == 0 {
		return nil, nil
	}

	return &users[0], nil
}

func NewUserRepoImpl(db *sql.DB) *UserRepoImpl {
	return &UserRepoImpl{
		db: db,
	}
}
```