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 8.1211-з
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
О.В. Мангеровський
(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 доцент кафедри комп'ютерних наук, доцент, к.т.н. Матвіїшина Н.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2022

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)

« ____ » __________ 2022 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Мангеровському Олександру Вікторовичу
(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 « 10 » травня 2022 року № 514-с

2. Строк подання студентом роботи ____________

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 ______________________________

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	27.05.2022	
2.	Збір вихідних даних.	10.06.2022	
3.	Обробка методичних та теоретичних	15.07.2022	
	джерел.		
4.	Розробка першого та другого розділу.	29.09.2022	
5.	Розробка третього розділу.	15.11.2022	
6.	Оформлення та нормоконтроль	25.11.2022	
	кваліфікаційної роботи.		
7.	Захист кваліфікаційної роботи.	16.12.2022	

Студент ____________________ О.В. Мангеровський
(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., 22 рис., 2 табл., 26 джерел, 1 додаток.

ВЕБДОДАТОК, ВЕБСОКЕТ, КОРИСТУВАЧ, ПОВІДОМЛЕННЯ, РОЗГОРТАННЯ, ТЕСТУВАННЯ, ЧАТ.

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

Мета роботи – спроєктувати та реалізувати онлайн чат.

Методи дослідження – аналіз браузерних технологій та існуючого програмного забезпечення, вивчення документації та найкращих практик.

В роботі були проаналізовані загальні підходи побудови додатків для миттєвого обміну інформацією між користувачами. Окремо розглянуто протокол WebSocket, що забезпечує двонаправлений зв’язок, є найзручнішим та найрозповсюдженим. Разом з тим, приділено увагу архітектурі клієнт-серверного додатку, його тестуванню та розгортанню (CI/CD) за допомогою сучасних технік хмарних сервісів.

Механізми, розглянуті в роботі можуть бути застосовані для розробки чатів або схожих додатків у якості самостійних продуктів, а також модулів(live chats) у складі більш складних систем.

SUMMURY

Master's qualifying work «Development of the Online Chat using WebSocket Protocol»: 68 pages, 22 figures, 2 tables, 26 references, 1 supplement.

WEB APPLICATION, WEB SOCKET, USER, MESSAGE, DEPLOYMENT, TESTING, CHAT.

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

The aim of the study is to design and implement an online chat.

The methods of research are analysis of browser technologies and existing software, study of documentation and best practices.

The work analyzes general approaches to building applications for instant exchange of information between users. The WebSocket protocol, which provides bidirectional communication, is the most convenient and the most widespread. At the same time, attention is paid to the architecture of the client-server application, its testing and deployment (CI/CD) using modern cloud services techniques.

The mechanisms considered in the work can be applied to the development of chats or similar applications as independent products, as well as modules (live chats) in more complex systems.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Огляд технологій 9
1.1 Основні принципи функціонування онлайн чату 9
1.1.1 Коротке опитування 12
1.1.2 Довге опитування 13
1.2 Вебсокети 15
1.2.1 Огляд принципу роботи протоколу WebSocket 15
1.2.2 Огляд бібліотек, що реалізують протокол WebSocket 18
1.3 Постановка технічного завдання 20
1.4 Висновки до розділу 21
2 Проєктування 22
2.1 Побудова діаграми прецедентів (варіантів використання) 22
2.2 Побудова діаграми послідовності дій 27
2.3 Архітектура серверної частини 29
2.3.1 Основні компоненти 29
2.3.2 Додаткові компоненти 34
2.4 Структури даних 34
2.5 Діаграма розгортання 37
2.6 Висновки до розділу 38
3 Реалізація та тестування 39
3.1 Приклади коду 39
3.2 Тестування 42
3.3 Впровадження 45
Висновки 48

Перелік посилань 49
Додаток А Приклади коду 52

ВСТУП

За останні роки онлайн чати суттєво наростили свою присутність у різних сферах: дозвілля, навчання, бізнес. Відповідне програмне забезпечення застосовується на різних платформах, зокрема в вебі. Дивлячись на динаміку зростання, можна прогнозовано очікувати, що актуальність пов'язаних технологій впродовж найближчого майбутнього буде тільки збільшуватись.

Основна причина, чому користувачі віддають перевагу чату в реальному часі, полягає в тому, що він дозволяє їм негайно отримати відповіді на свої запитання. За допомогою живого чату клієнтам надається можливість зв'язку саме тоді, коли у них виникнуть запитання чи проблеми, які вони не можуть вирішити. Це набагато краще, ніж надсилати електронний лист до служби підтримки; з електронною поштою важко знати, коли ви отримаєте відповідь. Негайність отримання допомоги є причиною того, що рейтинги задоволеності клієнтів у чаті зазвичай вищі, ніж у інших типах підтримки.

Об'єктом дослідження даної роботи є обмін інформації між браузером та вебсервером в режимі реального часу. Предмет дослідження – вебдодаток для обміну повідомленнями між користувачами(чат). Метою цієї роботи є проєктування та створення онлайн чату на базі протоколу зв'язку WebSocket, який наразі є стандартом, за усіма показниками випереджаючим альтернативні підходи.

Задачами проєкту є:

- огляд технологій WebSocket;
- аналіз вимог та проєктування;
- реалізація та тестування розробленого онлайн чату.

1 ОГЛЯД ТЕХНОЛОГІЙ

1.1 Основні принципи функціонування онлайн чату

Онлайн-чат може означати будь-який вид 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. Таким чином створюється відчуття, схоже на розмову, що відрізняє спілкування в чаті від інших текстових форм спілкування в Інтернеті, таких як Інтернет-форуми та електронна пошта. Онлайн-чат може стосуватися зв’язку «точка-точка», а також багатоадресного зв’язку від одного відправника до багатьох отримувачів, а також голосового та відеочату або може бути функцією служби веб-конференцій.

Онлайн-чат у менш суворому визначенні може бути насамперед будь-яким прямим текстовим або відео-чатом (веб-камери), чатом один на один або груповим чатом один-до-багатьох (офіційно також відомий як синхронна конференція), використовуючи такі інструменти, як миттєві месенджери, Інтернет-чат (IRC), розмовники та, можливо, MUD або інші онлайн-ігри. Вираз онлайн-чат походить від слова chat, що означає «неформальна розмова». Онлайн-чат містить веб-програми, які дозволяють спілкуватися між користувачами в багатокористувацькому середовищі, часто безпосередньо, але анонімно. Веб-конференції – це більш специфічна онлайн-служба, яка часто продається як послуга, розміщена на веб-сервері, контрольованому постачальником.

Перша система онлайн-чату називалася Talkomatic, 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, з повідомленнями, які з’являлися на екранах усіх користувачів символ

за символом, коли вони були набрані. Talkomatic був дуже популярний серед користувачів PLATO до середини 1980-х років. У 2014 році Браун і Вуллі випустили веб-версію Talkomatic [1].

Першу онлайн-систему, яка використовує фактичну команду «чат», створили для The Source у 1979 році Том Вокер і Фріц Тейн з Dialcom, Inc. [2].

Інші чат-платформи процвітали в 1980-х роках. Серед перших з GUI був BroadCast, розширення для Macintosh, яке стало особливо популярним в університетських містечках Америки та Німеччини [3].

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. Сервіс включав програмне забезпечення для мережевого чату, таке як «розмова» UNIX, яке використовувалося в 1970-х роках.

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. Дослідження також виявило, що чат може відволікати увагу та асиметрію інформації між учасниками.

Для участі у чаті необхідно зареєструватись. Ймовірні варіанти реєстрації – через контактний телефон або адресу електронної (вимоги можуть відрізнятися залежно від сайту), вказати ім'я користувача, під яким ви плануєте заходити до чату, і яке бачитимуть інші співрозмовники. Також потрібно вигадати пароль і двічі ввести його для входу в чат. Якщо було вказано адресу електронної пошти, на нього буде надіслано листа з посиланням для активації облікового запису. Якщо було вказано номер

телефону, на нього прийде SMS-повідомлення з паролем або кодом, який необхідно ввести у відповідне поле для верифікації облікового запису.

Надалі при кожному вході в чат вам необхідно буде вводити свій пароль та логін (логін) у відповідні поля для того, щоб розпочати спілкування.

Для користувача будь-якої соціальної мережі немає необхідності додаткової реєстрації в чатах такої мережі. Участь у публічних бесідах спільних чатів можна під своїм ім'ям цієї мережі. Адміністратори додають співрозмовників у приватні чати.

Відомі сучасні чати:

– Discord;
– Google Talk;
– Apple Messages;
– Teams;
– Skype;
– Slack;
– Telegram;
– WeChat;
– WhatsApp;
– Windows Live Messenger.

Вебсайти з браузерними службами чату (вебчат):

– Chat-Avenue;
– eBuddy;
– Facebook;
– Gmail;
– Talkomatic.

Вебдодатки спочатку були розроблені як проста клієнт-серверна модель, де вебклієнт ініціює HTTP-запит на отримання деяких даних із сервера. Наприклад, базова вебпрограма з потоком моделі клієнт-сервер буде

виглядати наступним чином:

– клієнт робить HTTP-запит із запитом на вебсторінку з сервера;
– сервер обчислює відповідь;
– сервер надсилає відповідь клієнту.

Оскільки розробники почали досліджувати способи впровадження програм у режимі реального часу, звичайна схема не задовольняла вимогам миттєвості і, як наслідок, почали з’являтися креативні способи маніпулювання моделлю HTTP-запит-відповідь.

1.1.1 Коротке опитування

Коротке опитування (Short Polling – запит → відповідь) – під час опитування клієнт робить запити XHR (XMLHttpRequest)/Ajax до сервера неодноразово через певний регулярний проміжок часу, щоб перевірити наявність нових даних [4]. Послідовність для короткого опитування буде наступною (див. рис. 1.1):

– клієнт ініціює запити через невеликі регулярні проміжки часу (наприклад, 0,5 секунди);
– сервер готує відповідь і надсилає її назад клієнту, як і звичайні HTTP-запити.

Здійснення повторних запитів до сервера витрачає ресурси, оскільки - потрібно встановити кожне нове вхідне з’єднання, передати заголовки HTTP, виконати запит на нові дані та створити та надіслати відповідь (зазвичай без нових даних).

Підключення має бути закрите, а всі ресурси очищені. Для того, щоб реалізувати задумку з оновленням у реальному часі, нам доведеться надсилати запити на оновлення документу, щонайменше кожну секунду, а це в свою чергу приведе нас до створення величезного навантаження на сервер, адже йому доведеться опрацьовувати величезну кількість запитів, навіть тоді,

коли не відбувається жодних змін в коді (див. табл. 1.1).

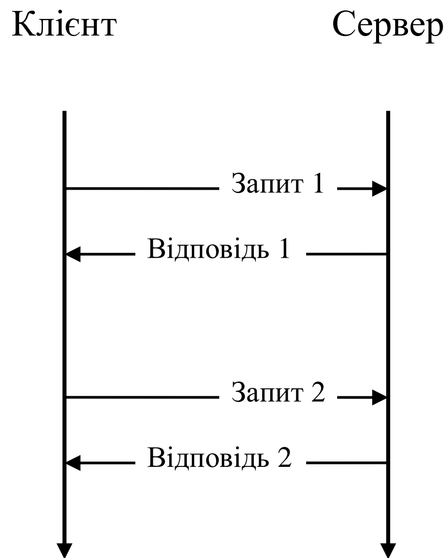

Рисунок 1.1 – Принципова схема роботи короткого опитування

Таблиця 1.1 – Характеристики короткого опитування

Переваги	**Недоліки**
Простота реалізації	Дуже багато зайвих запитів
	Події завжди приходять з запізненням
	Серверу потрібно зберігати події, поки клієнт їх не забере або вони не застаріють

1.1.2 Довге опитування

Довге опитування (Long Polling – запит → очікування → відповідь) – як і у звичайному опитуванні, замість повторення цього процесу кілька разів для кожного клієнта, доки не стануть доступними нові дані для даного клієнта, тривале опитування – це техніка, за якої сервер вирішує утримувати

з’єднання клієнта відкритим якомога довше, надаючи відповідь лише після того, як дані стануть доступними або досягнуто порогового значення часу очікування. Після отримання відповіді клієнт негайно надсилає наступний запит [5].

На стороні клієнта потрібно керувати лише одним запитом до сервера. Коли відповідь отримана, клієнт може ініціювати новий запит, повторюючи цей процес стільки разів, скільки необхідно.

Процес довгого опитування буде виглядати наступним чином (див. рис. 1.2):

– клієнт ініціює запит XHR/AJAX, запитуючи деякі дані з сервера;
– сервер не відповідає негайно з інформацією про запит, а чекає, доки не буде доступна нова інформація;
– коли з’являється нова інформація, сервер відповідає новою інформацією;
– клієнт отримує нову інформацію та негайно надсилає інший запит на сервер, перезапускаючи процес.

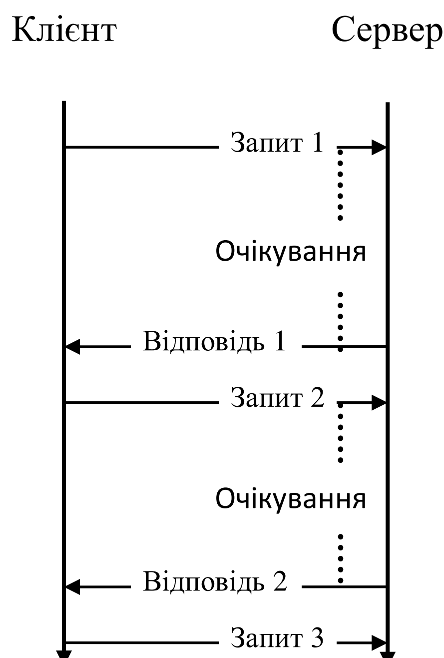

Рисунок 1.2 – Принципова схема роботи довгого опитування

Довгі запити відмінно працюють в тих випадках, коли повідомлення приходять рідко. При більшій кількості частих повідомлень графік прийому-відправки, приведений вище, перетворюється на «пилу». Кожне повідомлення – це новий запит, додатковий трафік заголовків. Переваги та недоліки схожі на попередній підхід з деякими відмінностями (див. табл. 1.2).

Незважаючи на деякі переваги технологій короткого та довгого опитування наразі вони суттєво поступаються в продуктивності та зручності сучасним аналогам і зустрічаються лише на старих проєктах. В той самий час переважна більшість нових чатів будується за допомогою технології, яка добре зарекомендувала себе, враховуючи вимоги оптимального навантаження на сервер та мінімальних затримок при обміні даними – WebSocket.

Таблиця 1.2 – Характеристики довгого опитування порівняно з коротким

Переваги	**Недоліки**
Мінімальна кількість запитів	Більш складна схема
Висока точність подій за часом	
Сервер зберігає події тільки під час повторного підключення	

1.2 Вебсокети

1.2.1 Огляд принципу роботи протоколу WebSocket

Вебсокет (WebSocket – клієнт ↔ сервер) – це бінарний повнодуплексний протокол, що дозволяє клієнту та серверу спілкуватися на

рівних. Цей протокол можна застосовувати для ігор, чатів і всіх тих додатків де вам потрібні максимально точні події близькі до реального часу [6].

Особливості протоколу:

– він відрізняється від HTTP, але сумісний з HTTP;
– працює через порт 80 і 443 (у разі шифрування TLS) і підтримує HTTP-проксі та посередників;
– для досягнення сумісності рукостискання WebSocket використовує заголовок Upgrade для оновлення протоколу до протоколу WebSocket.

Протокол WebSocket забезпечує взаємодію між клієнтом і вебсервером з меншими накладними витратами, забезпечуючи передачу даних у реальному часі від сервера та до нього. WebSocket підтримує з’єднання відкритим, дозволяючи передавати повідомлення між клієнтом і сервером. Таким чином між клієнтом і сервером може відбуватися двостороння безперервна розмова [7].

Процес підключення WebSocket буде виглядати наступним чином (див. рис. 1.3):

а) клієнт ініціює процес рукостискання WebSocket, надсилаючи запит, який також містить заголовок Upgrade для переходу на протокол WebSocket разом з іншою інформацією;
б) сервер отримує запит рукостискання WebSocket і обробляє його:
   – якщо сервер може встановити з’єднання та погоджується з умовами клієнта, він надсилає відповідь клієнту, підтверджуючи запит рукостискання WebSocket з іншою інформацією;
   – якщо сервер не може встановити з’єднання, він надсилає відповідь із підтвердженням того, що не може встановити з’єднання WebSocket;
в) щойно клієнт отримає успішний запит на підключення WebSocket, буде відкрито з’єднання WebSocket, після чого клієнт і сервери

можуть почати надсилати дані в обох напрямках, що забезпечує зв’язок у реальному часі;

г) з’єднання буде закрито, як тільки сервер або клієнт вирішить закрити з’єднання.

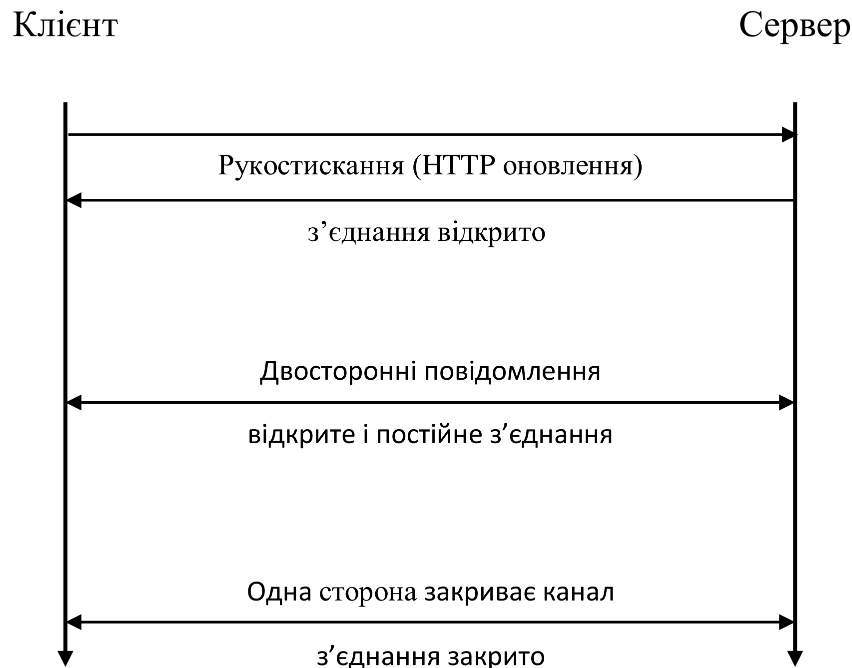

Рисунок 1.3 – Принципова схема роботи WebSocket

Основна перевага серверної частини WebSocket полягає в тому, що це не HTTP-запит (після рукостискання), а правильний протокол зв'язку на основі повідомлень [8]. Це дозволяє досягти величезних переваг у продуктивності та архітектурі. Наприклад, у Node.js ви можете використовувати один і ту ж пам’ять для різних з’єднань сокетів, щоб кожен із них міг отримати доступ до загальних змінних. Отже, вам не потрібно використовувати базу даних у якості точки обміну посередині (наприклад, з AJAX або довгим запитом з такою мовою, як PHP). Ви можете зберігати дані в ОЗУ або навіть відразу ж повторно опублікувати їх між сокетами.

Мінуси в порівнянні з довгим опитуванням:

– HTTP не сумісний протокол, потрібен сервер, ускладнюється налагодження;

– неповна підтримка браузерами (див. рис. 1.4) [9].

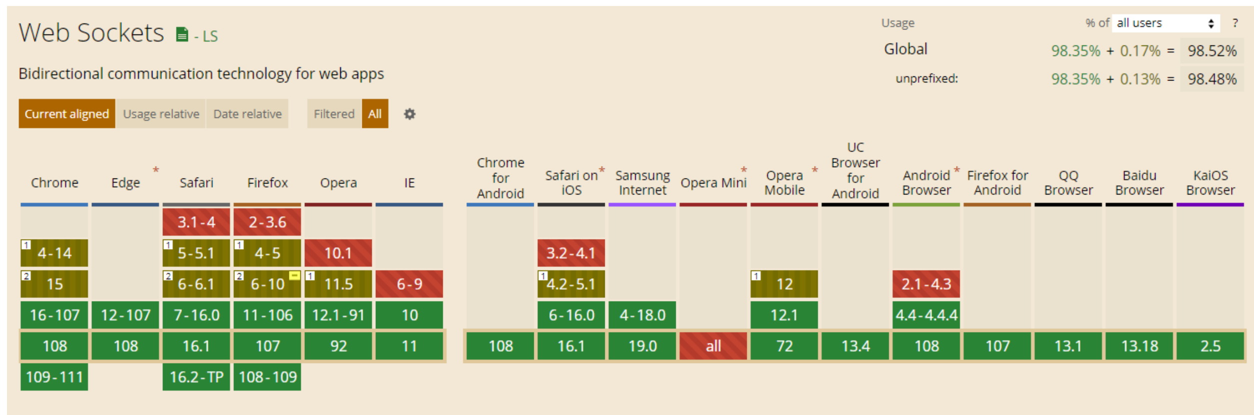

Рисунок 1.4 – Підтримка WebSocket у браузерах

1.2.2 Огляд бібліотек, що реалізують протокол WebSocket

Той факт, що бібліотеки WebSocket забезпечують двонаправлений канал зв’язку між браузером і сервером, одразу відкриває деякі дуже цікаві можливості для веб-додатків керування пристроями. Оскільки з’єднання є постійним, вбудований вебсервер тепер може ініціювати зв’язок із браузером. Вбудований вебсервер може надсилати сповіщення, оновлення, сповіщення тощо. Це додає абсолютно новий вимір до типів вебдодатків для керування пристроями, які можна створити [10].

WS – маючи понад 19,2 тисяч зірок на GitHub і близько 64 мільйона завантажень на тиждень на npm, ws є однією з найпопулярніших бібліотек Node.js WebSocket. Це швидка, проста у використанні, добре задокументована та ретельно перевірена реалізація клієнта та сервера WebSocket, яка підтримує майже всі браузери. Як наслідок, ws є улюбленою бібліотекою для багатьох розробників, що означає, що її спільнота є активною та надійною. WS – це, по суті, тип вебсокету, який надзвичайно швидкий, простий у використанні та ретельно перевірений. Разом з модулем WS є 2 додаткові модулі, які можна встановити. Ці додаткові модулі відомі

як бінарні розширення, які допомагають покращити певні операції. Існують попередньо зібрані бінарні файли, які доступні для найпопулярніших платформ, тому більше не потрібно встановлювати компілятор С++ на вашому комп’ютері.

SockJS – це бібліотека JavaScript, яка забезпечує зв’язок між клієнтом і сервером подібно до рідного API WebSockets. SockJS функціонує з серверним аналогом у формі sockjs-node і SockJS-client, клієнтської бібліотеки JavaScript. Маючи понад 8,1 тисяч зірок на GitHub і майже 12 мільйонів завантажень на npm на момент написання статті, SockJS є, мабуть, однією з найкращих бібліотек WebSocket для Node.js. SockJS дотримується правил HTML5 WebSocket API і надає об’єкти, подібні до WebSocket.

У SockJS є певні цілі, яких вони хочуть досягти. Деякі з них згадуються нижче:

- зручний для користувача і API на стороні веб-сервера, закрийте, доки це можливо;
- добре задокументовані методи масштабування та балансування навантаження;
- транспорт повинен підтримувати зв’язок між різними доменами;
- транспорт повинен плавно відступати у випадку обмежувальних проксі;
- встановлення поточного з’єднання бути достатньо швидким.

Socket.IO – це керована подіями бібліотека для вебдодатків у реальному часі [11]. Це забезпечує двонаправлений зв’язок у реальному часі між вебклієнтами та серверами. Він складається з двох частин: бібліотеки на стороні клієнта, яка працює в браузері, і бібліотеки на стороні сервера для Node.js. Обидва компоненти мають майже ідентичний API. Маючи майже 57,3 тисяч зірок на GitHub і майже 5,2 мільйонів завантажень на npm.

Socket.IO використовує протокол WebSocket з довгим опитуванням як запасним варіантом, забезпечуючи той самий інтерфейс. Хоча його можна використовувати просто як оболонку для WebSocket, він надає набагато

більше можливостей, включаючи трансляцію в кілька сокетів, зберігання даних, пов’язаних з кожним клієнтом, і асинхронний ввід-вивід.

Огляд основних можливостей:

– надає можливість реалізувати аналітику в реальному часі, двійкову потокову передачу, миттєвий обмін повідомленнями та співпрацю над документами (відомими користувачами є Microsoft Office, Yammer і Zendesk);
– Socket.IO обробляє з’єднання прозоро та автоматично оновить WebSocket, якщо це можливо, тобто розробнику не потрібно знати, як використовувати протокол WebSocket, щоб використовувати Socket.IO;
– Socket.IO не є бібліотекою WebSocket із резервними параметрами для інших протоколів реального часу, а спеціальною реалізацією транспортного протоколу реального часу на додаток до інших протоколів реального часу, що не дає можливості серверу із впровадженням Socket.IO підключитися до клієнта WebSocket, який не є реалізацією Socket.IO, бо клієнт із впровадженням Socket.IO не може спілкуватися з сервером WebSocket або Long Polling, що не є Socket.IO, вимагаючи використання бібліотек Socket.IO як на стороні клієнта, так і на стороні сервера.

1.3 Постановка технічного завдання

Необхідно реалізувати простий вебчат з можливістю авторизації, відправки та отримання повідомлень в режимі реального часу.

Базовий функціонал:

а) при завантаженні чату з’являється спливаюче вікно з формою для збереження імені (name) та ніку (nickname);
б) сам інтерфейс чату містить список повідомлень, поле для вводу

повідомлення та кнопку надіслати;

в) збоку від списку повідомлень розміщується список учасників чату:
   - елемент списку містить ім'я користувача та нік;
   - додатково елемент списку містить статус («label» певного кольору та надпис) online, offline, just appeared (якщо користувач у чаті менше 1 хв включно) та just left (якщо користувач покинув чат менше 1 хв включно);

г) чат містить історію з 100 повідомлень (очищення за принципом FIFO), яка доступна кожному користувачу, незважаючи коли він підключився;

д) коли користувач набирає текст, то в чаті під текстовим полем це можна бачити у форматі: "@userName is typing …";

е) якщо ввести повідомлення у форматі "@nickName", то у користувача котрому воно адресоване воно повинно бути підсвічене (задній фон іншого кольору наприклад);

ж) якщо користувач відключився, то іншим учасникам повідомляється, що він покинув чат і змінюється статус на offline.

1.4 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 хорошу браузерну підтримку.

Для спрощення розробки прийнято рішення застосувати бібліотеку Socket.io, яка реалізує роботу з WebSocket у більш зручний спосіб, гуртує навколо себе велике співтовариство розробників і є продуктом, перевіреним роками.

2 ПРОЄКТУВАННЯ

2.1 Побудова діаграми прецедентів (варіантів використання)

Діаграма прецедентів або діаграма варіантів використання (use case diagram) в UML – діаграма, що відображає відносини між акторами та прецедентами і є складовою моделі прецедентів, що дозволяє описати систему на концептуальному рівні [12].

Прецедент – можливість моделюється системи (частина її функціональності), завдяки якій користувач може отримати конкретний, вимірний і потрібний йому результат. Прецедент відповідає окремому сервісу системи, визначає один із варіантів її використання та визначає типовий спосіб взаємодії користувача з системою. Варіанти використання зазвичай використовуються для специфікації зовнішніх вимог до системи [13].

Основне призначення діаграми – опис функціональності та поведінки, що дозволяє замовнику, кінцевому користувачеві та розробнику спільно обговорювати проєктовану або існуючу систему.

При моделюванні системи за допомогою діаграми прецедентів системний аналітик прагне:

- чітко відокремити систему від її оточення;
- визначити дійових осіб (actor), їх взаємодію із системою та очікувану функціональність системи;
- визначити у глосарії предметної області поняття, які стосуються детального опису функціональності системи (тобто прецедентів).

Business Use Case (варіант використання з погляду бізнес-процесів) визначається як опис послідовності дій (потоку подій) у межах деякого бізнес-процесу, що приносить відчутний результат конкретної дійової особи.

Кожен Business Use Case відображає мету або потребу певної дійової

особи.

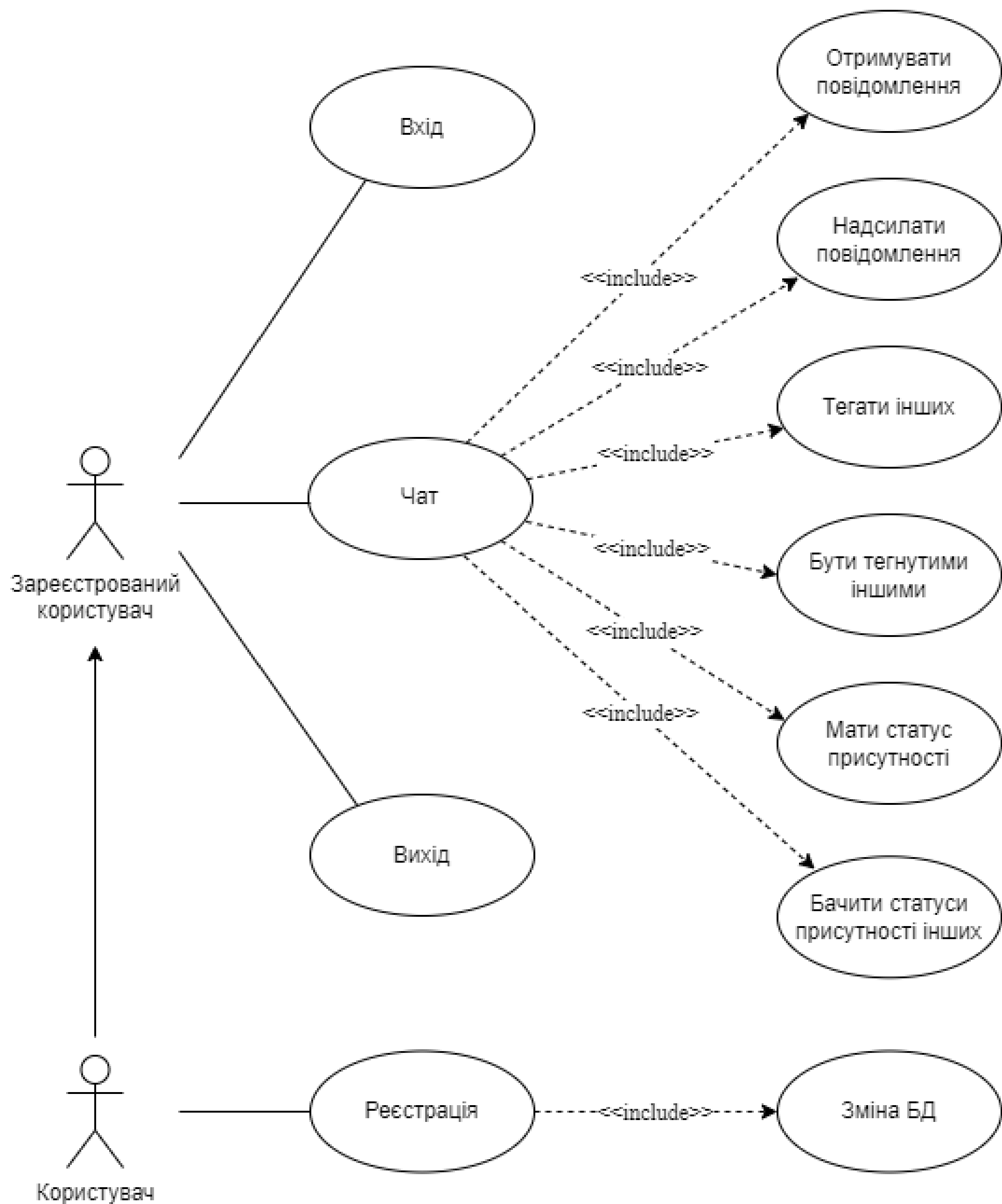

Рисунок 2.1 – Діаграма прецедентів онлайн чату

Опис Business Use Case є специфікацією, яка, подібно до звичайного варіанту використання, складається з наступних пунктів:

- найменування;
- короткий опис;
- цілі та результати (з точки зору дійової особи);

– опис сценаріїв (основного та альтернативних);
– спеціальні вимоги (обмеження за часом виконання чи іншими ресурсами);
– розширення (виключні ситуації);
– зв'язки з іншими Business Use Case;
– діаграми діяльності (для наочного опису сценаріїв за потреби).

Далі будуть розглянуті можливі сценарії онлайн чату.

Пройти реєстрацію.

Найменування – пройти реєстрацію.

Короткий опис – даний Use Case реалізує процес реєстрації користувача в онлайн чаті.

Цілі – ввести особисті дані на сторінці реєстраціє та отримати можливість увійти в чат і побачити головну сторінку.

Основний сценарій:

1) користувач заходить на сторінку реєстрації;
2) користувач вводить дані у форматі, що вимагає система;
3) користувач отримує повідомлення про успішну реєстрацію;
4) користувач перенаправляється на головну сторінку.

Альтернативні сценарії (для п. 3):

а) дані введені у невідповідному форматі – користувач отримує повідомлення про невідповідність формату та необхідність повторного вводу;

б) введений логін вже існує в чаті – користувач отримує повідомлення про існування логіну та необхідність обрати інший.

Вхід у чат.

Найменування – увійти в чат.

Короткий опис – даний Use Case реалізує процес входу користувача в онлайн чат.

Цілі – ввести особисті дані на сторінці входу та отримати можливість увійти в чат і побачити головну сторінку.

Основний сценарій:

1) користувач заходить на сторінку входу;
2) користувач вводить особисті дані;
3) користувач потрапляє на головну сторінку.

Альтернативні сценарії (для п. 3):

а) дані не знайдені в базі даних – користувач отримує повідомлення про невірно введені дані, залишаючись на сторінці логіну.

Вихід з чату.

Найменування – вийти з чату.

Короткий опис – даний Use Case реалізує процес виходу користувача з онлайн чату.

Цілі – натиснути кнопку виходу та потрапити на сторінку входу.

Основний сценарій:

- користувач заходить на головну сторінку;
- користувач натискає кнопку виходу;
- користувач потрапляє на сторінку входу.

Надсилання повідомлення.

Найменування – надіслати повідомлення.

Короткий опис – даний Use Case реалізує процес надсилання користувачем повідомлення в онлайн чат.

Цілі – надіслати повідомлення та побачити його в списку повідомлення на головній сторінці.

Основний сценарій:

- користувач заходить на головну сторінку;
- користувач набирає повідомлення та натискає кнопку відправки;
- користувач бачить щойно надіслане повідомлення у списку повідомлень.

Отримання повідомлення.

Найменування – отримати повідомлення.

Короткий опис – даний Use Case реалізує процес отримання

користувачем повідомлення в онлайн чаті.

Цілі – побачити в списку повідомлення, що було надіслане в чат.

Основний сценарій:

– користувач заходить на головну сторінку;
– в чат надсилається повідомлення;
– користувач бачить щойно надіслане повідомлення у списку повідомлень.

Відображення статусу користувача.

Найменування – відображення статусу користувача.

Короткий опис – даний Use Case реалізує процес відображення статусу у користувача в онлайн чаті.

Цілі – побачити в списку користувачів відповідні статуси.

Основний сценарій:

– користувач заходить на головну сторінку після авторизації;
– у списку користувачів він бачить себе зі статусом *just appeared*;
– через хвилину його статус змінюється на *online*.

Відображення статусу іншого користувача.

Найменування – відображення статусу іншого користувача.

Короткий опис – даний Use Case реалізує процес відображення статусу іншого користувача в онлайн чаті.

Цілі – побачити в списку користувачів відповідні статуси.

Основний сценарій:

– користувач заходить на головну сторінку після авторизації;
– у списку користувачів він бачить щойно приєднавшегося користувача зі статусом *just appeared*;
– через хвилину його статус змінюється на *online*.

Можливість бачити згадування про себе.

Найменування – можливість бачити згадування про себе.

Короткий опис – даний Use Case реалізує процес згадування свого

нікнейму іншим користувачем .

Цілі – побачити в списку повідомлень виділене повідомлення зі згадуванням твого нікнейму.

Основний сценарій:

- користувач заходить на головну сторінку після авторизації;
- у списку повідомлень він бачить повідомлення зі згадуванням свого нікнейму, виділене іншим кольором.

Можливість надсилати повідомлення зі згадування інших.

Найменування – можливість надсилати повідомлення зі згадування інших.

Короткий опис – даний Use Case реалізує процес згадування нікнейму іншого користувача в повідомленні.

Цілі – побачити в списку повідомлень виділене повідомлення зі згадуванням твого нікнейму.

Основний сценарій:

- користувач заходить на головну сторінку після авторизації;
- він надсилає повідомлення зі згадуванням нікнейму іншого користувача;
- у списку повідомлень він бачить повідомлення зі згадуванням нікнейму іншого користувача.

2.2 Побудова діаграми послідовності дій

Діаграма послідовності (*sequence diagram*) або діаграма послідовності систем (*system sequence diagram*) показує взаємодію процесів, організовану в часовій послідовності в галузі розробки програмного забезпечення. Вона описує задіяні процеси та послідовність повідомлень, якими обмінюються процеси, необхідні для виконання функцій. Діаграми послідовності іноді називають діаграмами подій або сценаріями подій.

Для конкретного сценарію використання діаграми показують події, які генерують зовнішні суб'єкти, їх порядок і можливі міжсистемні події [13]. Усі системи розглядаються як чорний ящик; діаграма робить наголос на подіях, які перетинають межу системи від акторів до систем. Для основного сценарію успіху варіанту використання та частих або складних альтернативних сценаріїв слід створити діаграму системної послідовності.

Створюємо діаграму послідовності для основного сценарію відправки та отримання повідомлення (див. рис. 2.2).

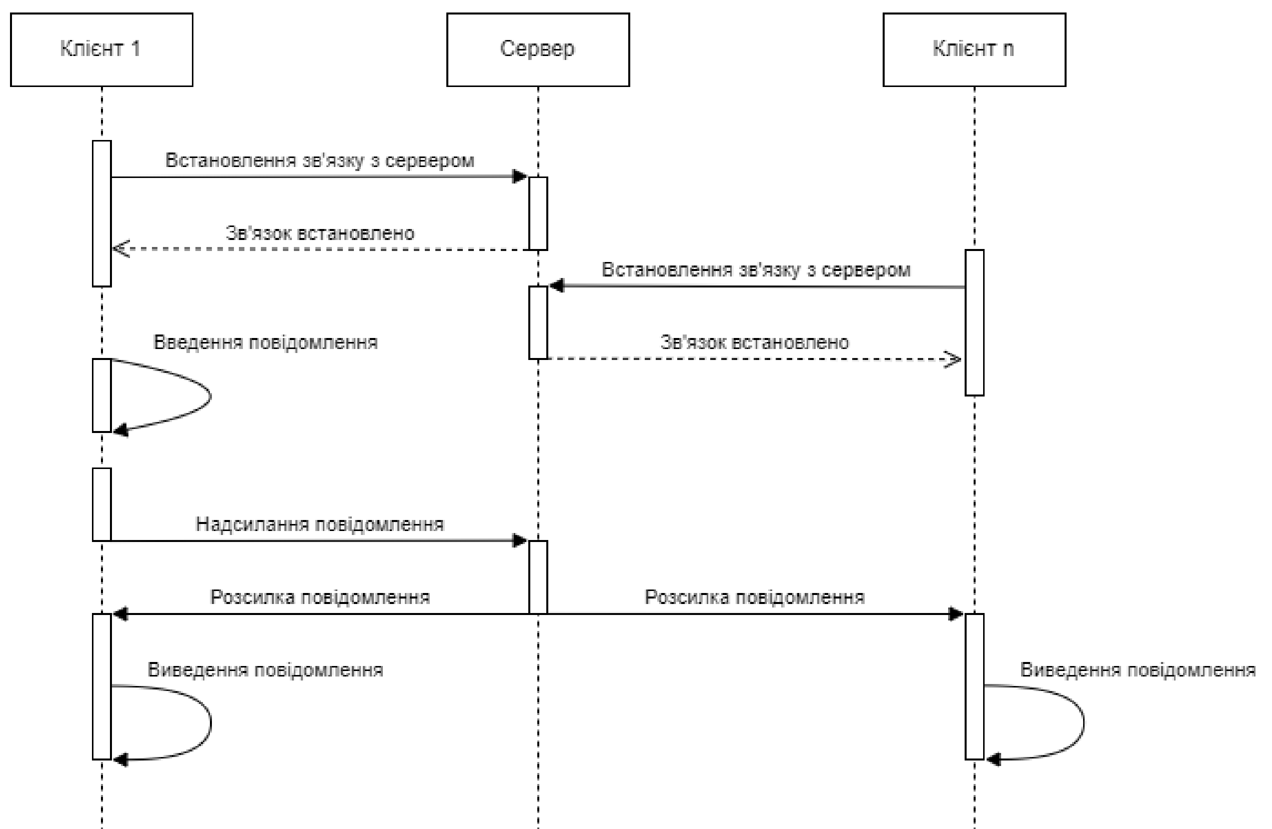

Рисунок 2.2 – Діаграма послідовності дій надсилання та отримання повідомлення

Діаграма послідовності показує у вигляді паралельних вертикальних ліній (ліній життя) різні процеси або об'єкти, які живуть одночасно, а у вигляді горизонтальних стрілок – повідомлення, якими вони обмінюються, у порядку їх виникнення. Це дозволяє специфікувати прості сценарії виконання в графічній формі.

Діаграма системної послідовності повинна вказувати та показувати наступне:

– зовнішні актори;
– повідомлення (методи), викликані цими акторами;
– значення, що повертаються (за наявності), пов’язані з попередніми повідомленнями;
– вказівка будь-яких циклів або області ітерації.

2.3 Архітектура серверної частини

Сучасна розробка серверу пов’язана з використанням багаторівневої архітектури. За подібним принципом працюють більшість сучасних фреймворків.

На стадії проєктування сервера спираються на набір класів компонентів із заздалегідь визначеними обов’язками [14].

Кожен фрагмент коду повинен міститись у компоненті (функції класу).

Існує величезний перелік цих компонентів із набором вказівок, яких слід дотримуватися під час їх використання, щоб процес розробки був гладкий.

Компоненти забезпечують узгодженість і спрощують обслуговування коду, оскільки ви вже знаєте, де можна знайти кожен фрагмент коду.

Кожен контейнер складається з кількох компонентів, які поділяються на два типи: основні компоненти; додаткові компоненти.

2.3.1 Основні компоненти

Розглянемо діаграму взаємодії основних компонентів (див. рис. 2.3). Слід використовувати саме ці компоненти, оскільки вони необхідні майже

для всіх типів вебпрограм:

1) Routes – Controllers – Requests – Actions – Tasks – Models – Views – Transformers;
2) Views – слід використовувати, якщо програма обслуговує сторінки HTML;
3) Transformers – слід використовувати, якщо програма обслуговує дані JSON або XML.

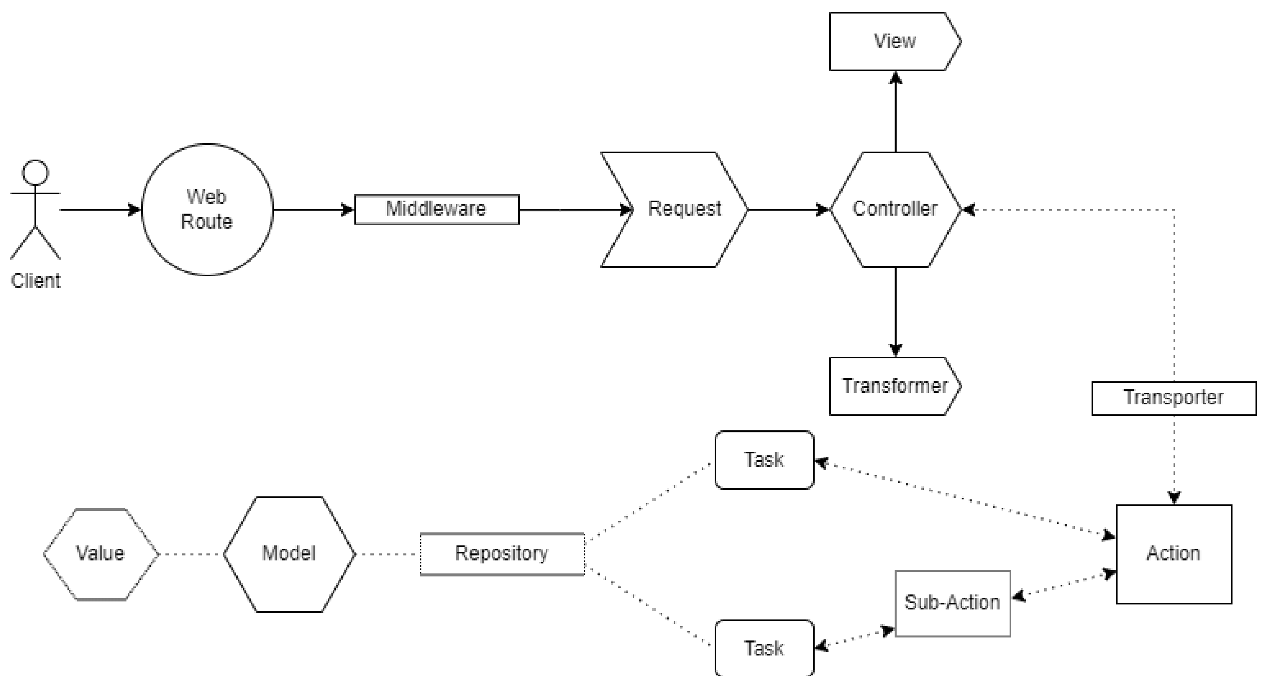

Рисунок 2.3 – Діаграма взаємодії основних компонентів

Базовий сценарій виклику API, навігація по основних компонентах:

– користувач викликає кінцеву точку у файлі маршруту;
– кінцева точка викликає проміжне програмне забезпечення для обробки автентифікації;
– кінцева точка викликає свою функцію контролера;
– запит, введений у контролер, автоматично застосовує правила перевірки та авторизації запиту;
– контролер викликає дію та передає їй дані кожного запиту;
– дія виконує бізнес-логіку, або може викликати стільки завдань,

скільки потрібно для виконання багаторазово використовуваних підмножин бізнес-логіки;

- завдання виконують повторно використовувані підмножини бізнес-логіки (завдання може виконувати одну частину основної дії);
- дія готує дані для повернення до контролера, деякі дані можна зібрати з завдань;
- контролер створює відповідь за допомогою View (або Transformer) і надсилає її назад користувачу.

Розглянемо далі визначення та принципи роботи основних компонентів.

Маршрути (Routes) є першими отримувачами HTTP-запитів. Маршрути відповідають за відображення всіх вхідних запитів HTTP на функції свого контролера.

Файли маршрутів містять кінцеві точки (шаблони URL-адрес, які ідентифікують вхідний запит).

Коли HTTP-запит надходить у вашу програму, кінцеві точки збігаються з шаблоном URL-адреси та викликають відповідну функцію контролера.

Контролери (Controllers) відповідають за підтвердження запиту, надання даних запиту та формування відповіді. Перевірка та відповідь відбуваються в окремих класах, але ініціюються контролером.

Принципи:

а) контролери не повинні знати нічого про бізнес-логіку чи будь-який бізнес-об’єкт;

б) контролер повинен виконувати лише такі роботи:
   - читання даних запиту (введення користувача);
   - виклик дії (і передача їй даних запиту);
   - створення відповіді (зазвичай створюється відповідь на основі даних, зібраних із виклику дії):

в) контролери не повинні мати жодну форму бізнес-логіки (слід викликати дію для виконання бізнес-логіки);

г) контролери не повинні викликати завдання контейнера, а можуть викликати лише дії (тоді дії можуть викликати завдання контейнера);

д) контролери можна викликати лише через кінцеві точки маршрутів.

Запити (Requests) в основному обслуговують введення користувача в програму. І вони дуже корисні для автоматичного застосування правил перевірки та авторизації.

Запити є найкращим місцем для застосування перевірок, оскільки правила перевірки стосуються кожного запиту. Запити також можуть перевіряти авторизацію, напр. перевірте, чи має цей користувач доступ до цієї функції контролера.

Приклад: перевірте, чи є певний користувач продуктом перед видаленням, або перевірте, чи є цей користувач адміністратором, щоб щось редагувати.

Принципи:

- запит може містити правила перевірки/авторизації;
- запити слід вводити лише в контролери, які автоматично перевіряють, чи дані запиту відповідають правилам перевірки, і якщо введення запиту недійсне, буде створено виняток;
- запити також можуть використовуватися для авторизації, вони можуть перевірити, чи користувач має право робити запит.

Дії (Actions) представляють випадки використання програми (дії, які може виконувати користувач або програмне забезпечення в програмі).

Дії можуть містити бізнес-логіку або вони оркеструють Завдання для виконання бізнес-логіки.

Дії приймають структури даних як вхідні дані, маніпулюють ними відповідно до бізнес-правил внутрішньо або через деякі завдання, а потім виводять нові структури даних.

Дії не повинні піклуватися, як дані збираються або як вони будуть представлені.

Просто переглянувши папку «Actions» контейнера, ви можете визначити, які випадки використання надає ваш контейнер. Переглянувши всі Дії, ви зможете зрозуміти, що може робити програма.

Завдання (Tasks) – це класи, які зберігають спільну бізнес-логіку між кількома діями в різних контейнерах. Кожне завдання відповідає за невелику частину логіки.

Приклад: якщо у вас є дія 1, яка потребує пошуку запису за його ідентифікатором у БД, а потім запускає подію. І у вас є дія 2, якій потрібно знайти той самий запис за його ідентифікатором, а потім зробити виклик зовнішнього API. Оскільки обидві дії виконують логіку «знайти запис за ідентифікатором», ми можемо взяти цю бізнес-логіку та помістити її в її власний клас, цим класом є Tasks. Тепер це завдання можна повторно використовувати як діями, так і будь-якими іншими діями, які ви можете створити в майбутньому.

Моделі (Models) забезпечують абстракцію даних, вони представляють дані в базі даних. Моделі відповідають за те, як слід обробляти дані. Вони переконуються, що дані належним чином надходять у серверне сховище (наприклад, базу даних).

Принципи:

– модель не повинна містити бізнес-логіку, вона може містити лише код і дані, які представляють себе (це зв’язки з іншими моделями, приховані поля, назва таблиці, заповнювані атрибути...);
– один контейнер може містити декілька моделей;
– модель може визначати зв’язки між собою та будь-якими іншими моделями (якщо зв’язок існує).

Представлення (Views) містять HTML, який обслуговує ваша програма. Їх головна мета – відокремити логіку програми від логіки презентації.

Принципи:

– перегляди можна використовувати лише з веб-контролерів;

– перегляди повинні бути розділені на кілька файлів і папок на основі того, що вони відображають;
– один контейнер може містити декілька файлів Views.

Трансформери (скорочена назва Responses Transformers) еквівалентні Views, але для відповідей JSON. Поки Views отримує дані та представляє їх у HTML, Transformers приймає дані та представляє їх у JSON.

Transformers – це класи, відповідальні за перетворення моделей у масиви. Transformers бере модель або групу моделей «Collection» та перетворює її на форматований серіалізований масив.

2.3.2 Додаткові компоненти

Ви можете додати ці компоненти, коли вони вам потрібні, залежно від потреб вашої програми, однак деякі з них настійно рекомендовані: Tests – Events – Listeners – Commands – Migrations – Seeders – Factories – Middlewares – Repositories – Criteria – Policies – Service Providers – Contracts – Traits – Jobs – Values – Transporters – Mails – Notifications...

Приклад контейнера з основними та додатковими компонентами представлено на рисунку 2.4.

2.4 Структури даних

Структура даних – це формат організації, керування та зберігання даних, який зазвичай вибирається для ефективного доступу до даних. Точніше кажучи, структура даних – це сукупність значень даних, зв’язків між ними та функцій або операцій, які можна застосувати до даних [15].

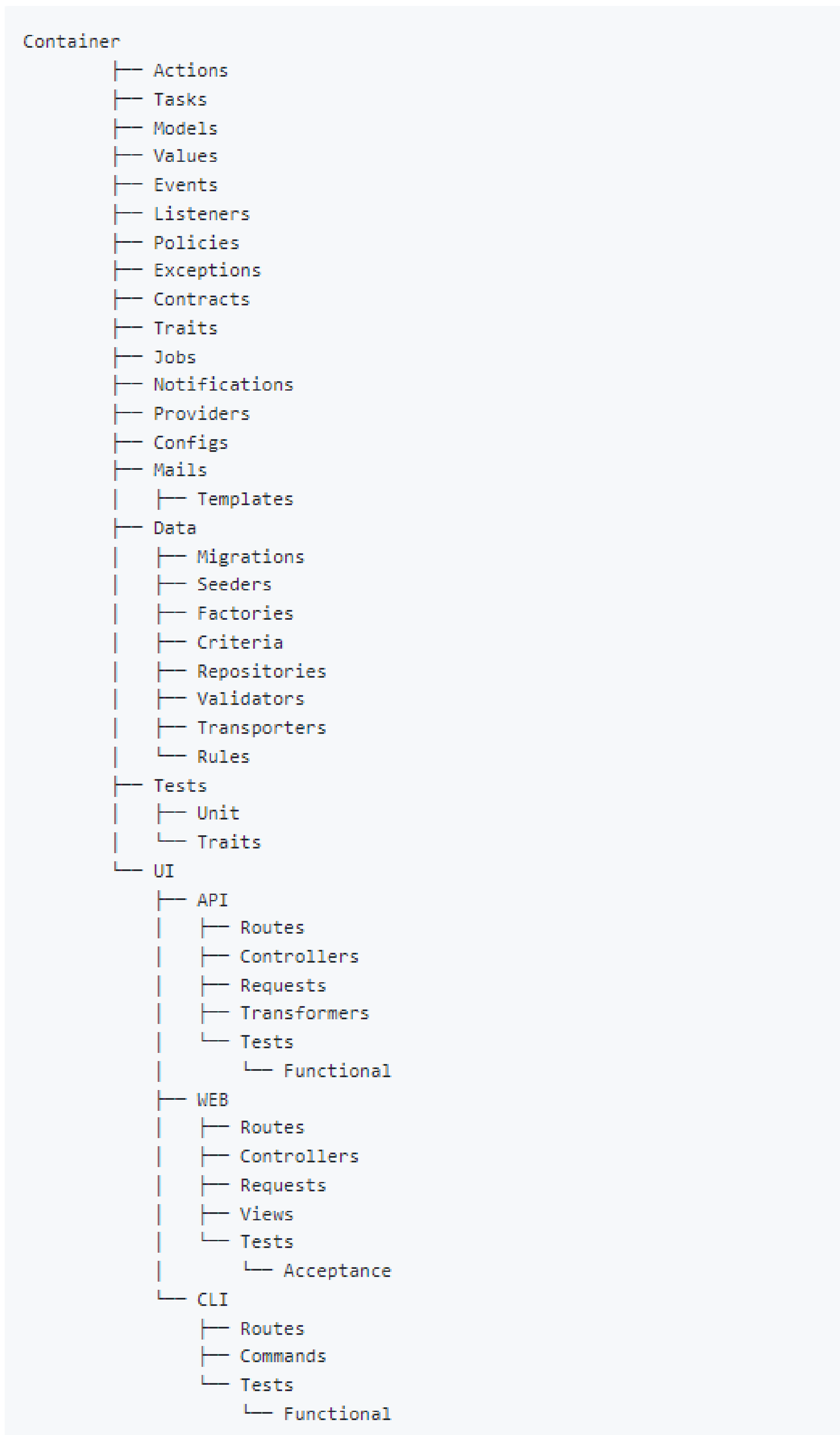

Рисунок 2.4 – Структура контейнера з основними та додатковими компонентами

Структури даних надають засоби для ефективного керування великими обсягами даних для таких цілей, як великі бази даних і служби індексування в Інтернеті.

Зазвичай ефективні структури даних є ключовими для розробки ефективних алгоритмів. Деякі формальні методи проєктування та мови програмування підкреслюють структури даних, а не алгоритми, як ключовий організуючий фактор у проєктуванні програмного забезпечення.

Структури даних можна використовувати для організації зберігання та пошуку інформації, що зберігається як в основній, так і в вторинній пам’яті.

На основі загальних підходів було розроблено структуру даних для сутностей, що формують онлайн чат (див. рис. 2.5).

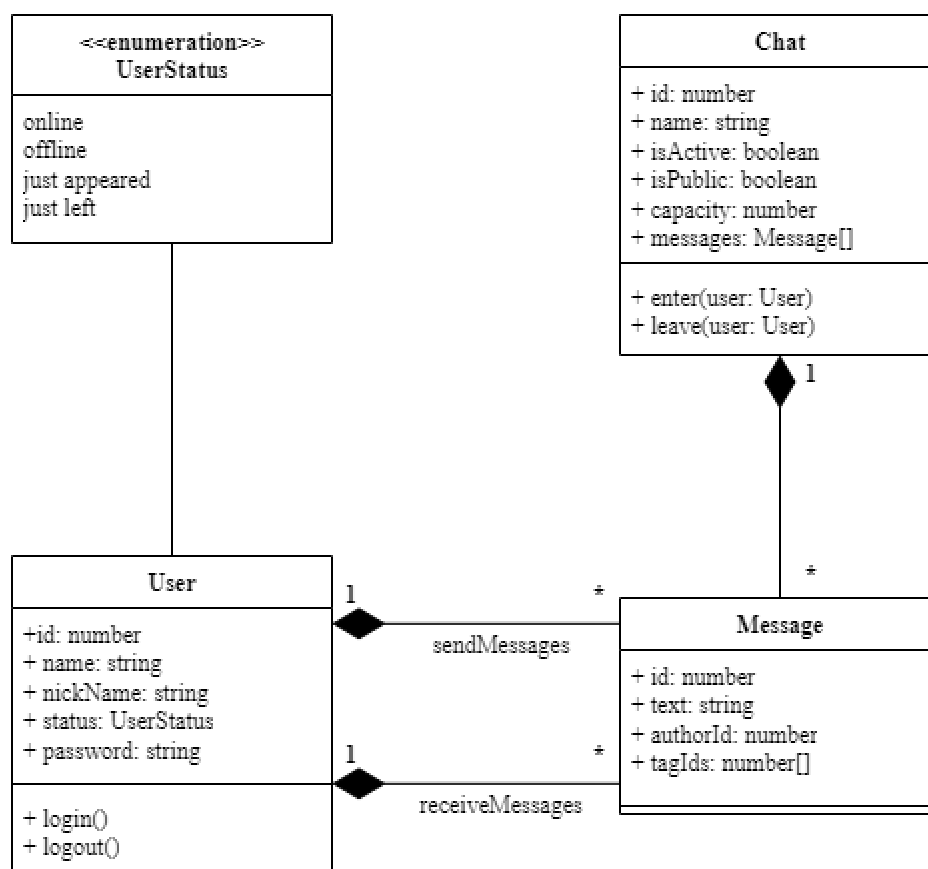

Рисунок 2.5 – Діаграма класів

2.5 Діаграма розгортання

Діаграма розгортання в уніфікованій мові моделювання (UML) моделює фізичне розгортання артефактів на вузлах.

Щоб описати веб-сайт, наприклад, діаграма розгортання покаже, які апаратні компоненти («вузли») існують (наприклад, веб-сервер, сервер додатків і сервер бази даних), які компоненти програмного забезпечення («артефакти») працюють на кожен вузол (наприклад, веб-додаток, база даних) і спосіб з'єднання різних частин (наприклад, JDBC, REST, RMI) (див. рис. 2.6).

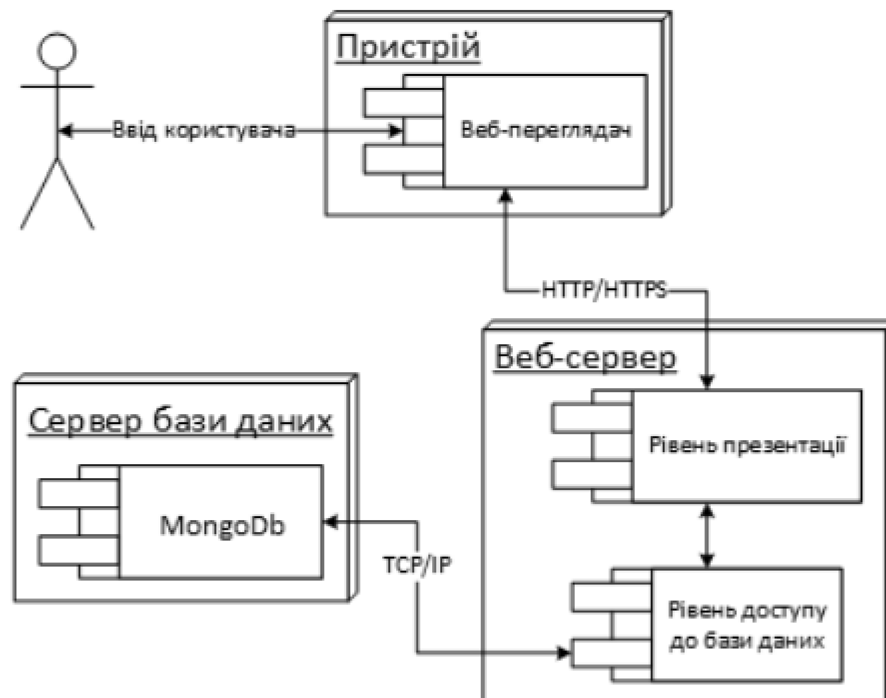

Рисунок 2.6 – Діаграма розгортання

Вузли відображаються у вигляді прямокутників, а артефакти, призначені для кожного вузла, відображаються у вигляді прямокутників усередині вікон. Вузли можуть мати підвузли, які відображаються як вкладені блоки. Один вузол на діаграмі розгортання може концептуально представляти кілька фізичних вузлів, наприклад кластер серверів бази даних.

Існує два типи вузлів:

- вузол пристрою;
- вузол середовища виконання.

2.6 Висновки до розділу

Виходячи з технічного завдання було побудовано діаграму прецедентів, яка враховую можливі варіанти використання онлайн чату.

Для основних варіантів використання, як то надсилання та отримання повідомлення було спроєктовано діаграму послідовності дій.

Прийнято використовувати на сервері монолітну багаторівневу архітектуру на базі патерну MVC з подальшим можливим винесенням функціоналу в окремі мікросервіси за рахунок побудови структури секцій з пов’язаними контейнерами [16].

Для представлення структури даних сутностей онлайн чату було розроблено діаграму класів.

Основні види взаємодій між частинами системи представлено діаграмою розгортання.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

Виходячи зі вимог до архітектури веб-додатку для розробки серверної частини обрано фреймворк Nest.js [17]. В якості бази даних буде використовуватися PostgresSQL [18]. Для зручності роботи розробка клієнтської сторони буде вестись за допомогою фреймворку React.js [19], бібліотек Sass [20], Material-ui [21] та інших. Для реалізації протоколу WebSocket і на клієнті, і на сервері задіяна бібліотека Socket.io.

3.1 Приклади коду

На рисунках 3.1 – 3.6 наведені основні фрагменти коду (більш докладно – див. дод. А).

```
export class UserController {
  constructor(private readonly service: UserService) {}

  @ApiBearerAuth()
  @ApiOperation({ description: 'Get list of users' })
  @ApiForbiddenResponse({ type: ForbiddenError })
  @ApiUnauthorizedResponse({ type: UnauthorizedError })
  @ApiInternalServerErrorResponse({ type: InternalServerError })
  @RolesDecorator(ListRolesEnum.ADMIN)
  @UseGuards(AuthGuard, RolesGuard)
  @Get()
  findAll(): Promise<UsersResponseDTO> {
    return this.service.findAll()
  }
}
```

Рисунок 3.1 – Контролер користувача

```
async findById(id: string): Promise<UserResponseDTO> {
  const user: UserResponseDTO = await this.userRepo.findOne(id)

  if (!user) {
    throw new HttpException('Not found User', HttpStatus.NOT_FOUND)
  }

  return user
}
```

Рисунок 3.2 – Сервіс користувача

```
import { BeforeInsert, Column, Entity } from 'typeorm'
import { ListTablesEnum } from '../../config/enums'
import { HashUserPass } from '../utils/hash-data'
import { BaseEntity } from './base-entity'

@Entity({ name: ListTablesEnum.USERS })
export class UserEntity extends BaseEntity {
  @Column('varchar', {
    length: 40,
    nullable: false,
  })
  name: string

  @Column('varchar', {
    length: 40,
    nullable: false,
  })
  nickName: string

  @Column('varchar', {
    length: 150,
    nullable: false,
    unique: true,
  })
  email: string

  @Column({
    nullable: false,
    select: false,
  })
  password: string

  @BeforeInsert()
  async encodePassword(): Promise<void> {
    this.password = await HashUserPass(this.password)
  }
}
```

Рисунок 3.3 – Модель користувача

```
socket.on('new message', msg => {
  messages.push(msg);
  io.emit('new message', getMessageWithAuthor(msg));
});

socket.on('i am typing', name => {
  socket.broadcast.emit('someone is typing', name);
});
```

Рисунок 3.4 – Робота WebSocket на боці сервера

```
useEffect(() => {
    socket.on('connect', () => {
      setIsConnected(true);
    });

    socket.on('disconnect', () => {
      setIsConnected(false);
    });

    socket.on('users list', publicUsers);

    socket.on('new user', publicUser);

    socket.on('my user id', userId => {
        user.id = userId;
    });

    socket.on('message history', publicPosts);

    socket.on('new message', publicPost);

    return () => {
      socket.off('connect');
      socket.off('disconnect');
      socket.off('users list');
      socket.off('new user');
      socket.off('my user id');
      socket.off('message history');
      socket.off('new message');
    };
  }, []);
```

Рисунок 3.5 – Робота WebSocket на боці клієнта

```
import React from 'react';
import { Col } from 'elmo-elements';
import { StyledRow } from './styles';
import { LoginForm } from 'src/pages/login/components';

export default function LoginPage() {
  return (
    <StyledRow>
      <Col xs={{ offset: 3, span: 18 }} md={{ offset: 8, span: 8 }} xxl={{ offset: 9, span: 6 }}>
        <LoginForm />
      </Col>
    </StyledRow>
  );
}
```

Рисунок 3.6 – Сторінка логіна

3.2 Тестування

Окрім мануального тестування на проєкті використовувалися бібліотеки: Jest, React Testing Library (юніт-тести), Cypress (автотести).

Візуальний контроль роботи додатка представлено на рисунках 3.7 та 3.8.

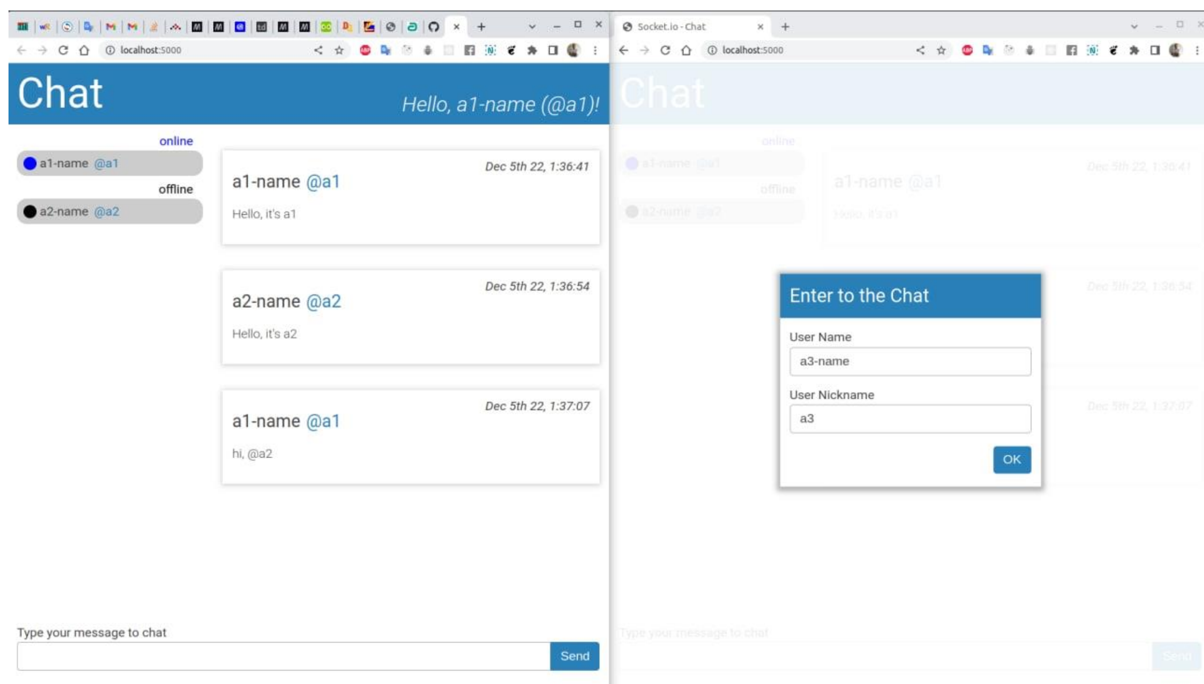

Рисунок 3.7 – Приклад роботи реєстрації

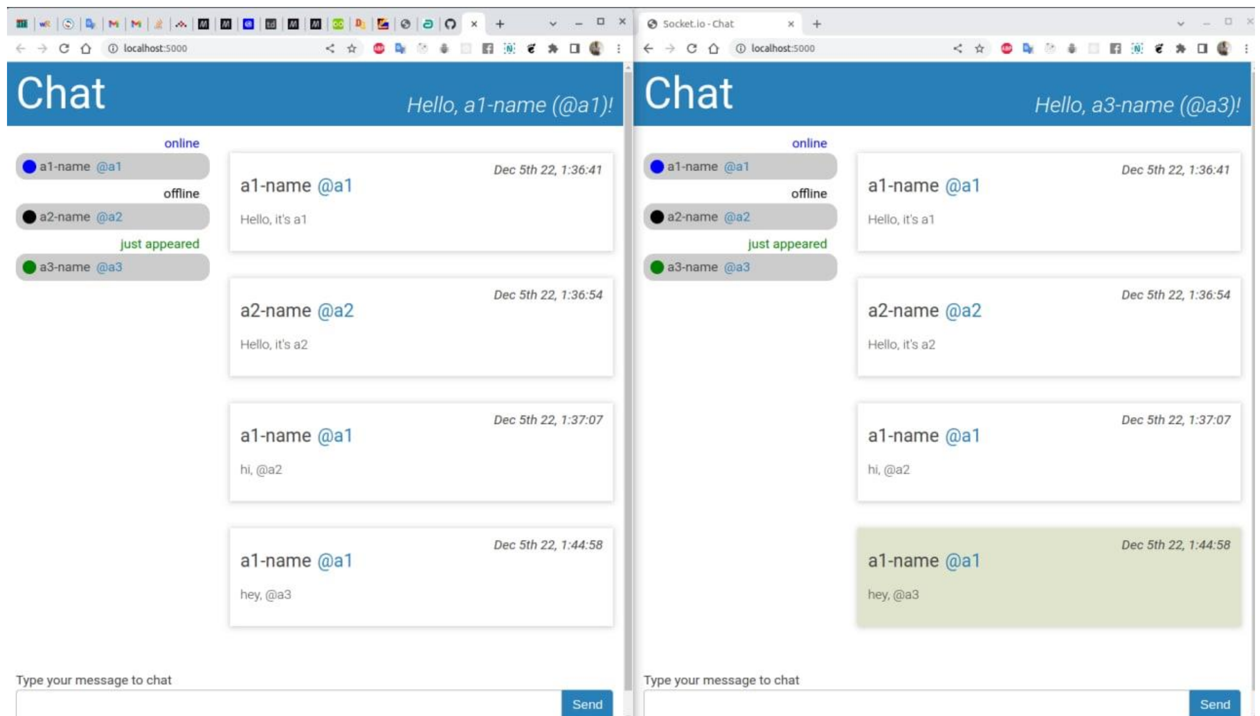

Рисунок 3.8 – Приклад згадування користовача у чаті

На рисунках 3.9 – 3.11 наведені фрагменти коду тестів (більш докладно – див. дод. А) [22-24].

```
it('should get the list of users. Function findAll', async () => {
  const user = {
    id: 'user id',
  };
  const result: any = { users: [], totalAmount: 0 };

  jest.spyOn(service, 'findAll').mockImplementationOnce(() => result);
  try {
    const response = await controller.findAll(user as any);
    expect(response).toBe(result);
  } catch (error) {
    expect(error).toBeUndefined();
  }
});
```

Рисунок 3.9 – Jest юніт-тест контролера користувача на боці сервера

```
it('should render properly with login form', () => {
   const { getByRole, getByLabelText } = render(<LoginPage />);
   // card heading
   getByRole('heading', { name: 'Login' });
   // form
   const form = getByRole('form');
   // form items
   expect(form.querySelectorAll('label')[1]).toHaveTextContent('Email');
   getByLabelText('Email', { selector: 'input' });
   expect(form.querySelectorAll('label')[2]).toHaveTextContent('Password');
   getByLabelText('Password', { selector: 'input' });
   // submit button
   expect(getByRole('button', { name: 'Login' })).toHaveAttribute('type', 'submit');
 });
```

Рисунок 3.10 – React Testing Library юніт-тест на боці клієнта

```
Cypress.Commands.add('login', (username, password) => {
  cy.visit('/login');

  cy.get('input[name=username]').type(username);

  cy.get('input[name=password]').type(`${password}{enter}`, { log: false });

  // we should be redirected to /dashboard
  cy.url().should('include', '/chat');

  // our auth cookie should be present
  cy.getCookie('your-session-cookie').should('exist');

  // UI should reflect this user being logged in
  cy.get('h3').should('contain', username);
});
```

Рисунок 3.11 – Cypress автотест

3.3 Впровадження

Для разгортання проєкта у якості сервера було обрано сервіси AWS, а саме екземпляри Amazon Elastic Compute Cloud (Amazon EC2) у групі автомасштабування [25].

На проєкті застосовуються практики DevOps для більш швидкого впровадження інновації шляхом автоматизації та оптимізації процесів розробки програмного забезпечення та управління інфраструктурою через безперервну інтеграцію та безперервна доставка (CI/CD).

Практика CI/CD скорочує час, необхідний для випуску нових оновлень програмного забезпечення, завдяки автоматизації розгортання. Для реалізації цієї практики доступно багато інструментів. Хоча AWS має набір власних інструментів для досягнення ваших цілей CI/CD, він також пропонує гнучкість і розширюваність для інтеграції з численними сторонніми інструментами.

GitHub Actions – це функція на популярній платформі розробки GitHub, яка допомагає вам автоматизувати робочі процеси розробки програмного забезпечення там же, де ви зберігаєте код і співпрацюєте над запитами на отримання та проблемами [26]. Ви можете написати окремі завдання, які називаються діями, а потім поєднати їх, щоб створити настроюваний робочий процес.

Робочі процеси – це спеціальні автоматизовані процеси, які можна налаштувати у своєму сховищі для створення, тестування, пакетування, випуску чи розгортання будь-якого проєкту коду на GitHub.

AWS CodeDeploy – це служба розгортання, яка автоматизує розгортання додатків в екземплярах Amazon EC2, локальних екземплярах, безсерверних функціях AWS Lambda або службах Amazon Elastic Container Service (Amazon ECS).

Рішення використовує такі сервіси:

– GitHub Actions – інструмент оркестровки робочого процесу, який

розміщуватиме конвеєр;

– AWS CodeDeploy – служба AWS для керування розгортанням у Amazon EC2 Autoscaling Group;
– автоматичне масштабування AWS – служба AWS, яка допомагає підтримувати доступність і еластичність програм шляхом автоматичного додавання або видалення екземплярів Amazon EC2;
– Amazon EC2 – цільовий обчислювальний сервер для розгортання програми;
– AWS CloudFormation – служба інфраструктури AWS як коду (IaC), яка використовується для розгортання початкової інфраструктури на стороні AWS;
– постачальник ідентифікаційних даних IAM OIDC – об’єднана служба автентифікації для встановлення довіри між GitHub і AWS, щоб дозволити розгортати GitHub Action на AWS без збереження секретів і облікових даних AWS;
– Amazon Simple Storage Service (Amazon S3) – Amazon S3 для зберігання артефактів розгортання.

На рисунку 3.12 представлена схема впровадження проєкту.

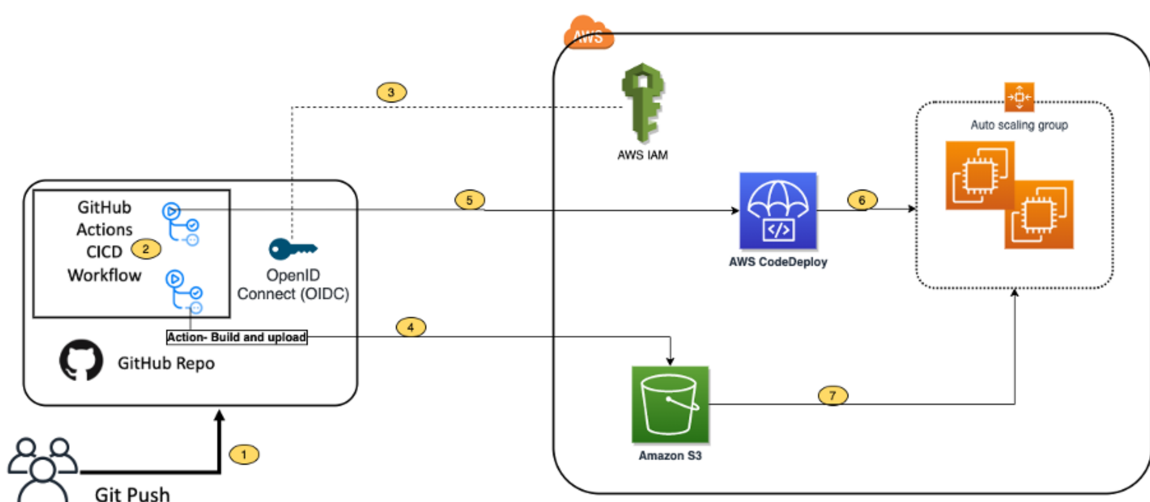

Рисунок 3.12 – Схема впровадження проєкту

На представленій діаграмі показано архітектуру рішення:

1) розробник фіксує зміни коду зі свого локального сховища в репозиторій GitHub;
2) GitHub Actions запускає етап збірки;
3) Open ID Connector GitHub (OIDC) використовує маркери для автентифікації в AWS і доступу до ресурсів;
4) GitHub Actions завантажує артефакти розгортання в Amazon S3;
5) GitHub Actions викликає CodeDeploy;
6) CodeDeploy ініціює розгортання в екземплярах Amazon EC2 у групі автомасштабування;
7) CodeDeploy завантажує артефакти з Amazon S3 і розгортає в екземплярах Amazon EC2.

ВИСНОВКИ

В межах даної роботи зроблено огляд сучасних технологій для розробки онлайн чата, зокрема протоколу WebSocket, проаналізовані вимоги к додатку та виконане проєктування.

На базі вимог, був реалізований та впроваджений додаток онлайн чату на базі протокола WebSocket.

В якості інстументів розробки було обрано сучасні JavaScript фреймворки, що мають широке росповсюдження в комерційній розробці. При впровадженні були використанні підходи CI/CD.

ПЕРЕЛІК ПОСИЛАНЬ

1. Jones S. PLATO – computer-based education system. URL: https://www.britannica.com/topic/PLATO-education-system (дата звернення: 20.10.2022).
2. Design and Implementation of a Multilingual Chat Application. URL: https://nairaproject.com/projects/3835.html (дата звернення: 20.10.2022).
3. McKinney M. Sell a Couch or Make a New Friend: Broadcast Provides Potential Mind Games and Hookups. The Wooster Voice. 1998. Vol. CXV, Issue 11. P. 8.
4. Shaddel P. Understand and Implement Long – Polling and Short Polling in Node.js. URL: https://levelup.gitconnected.com/understand-and-implement-long-polling-and-short-polling-in-node-js-94334d2233f3 (дата звернення: 22.10.2022).
5. What is Long Polling and Short Polling? URL: https://www.geeksforgeeks.org/what-is-long-polling-and-short-polling/ (дата звернення: 19.10.2022).
6. WebSockets Standard. URL: https://websockets.spec.whatwg.org/ (дата звернення: 01.11.2022).
7. The WebSocket API. URL: https://www.w3.org/TR/2021/NOTE-websockets-20210128/Overview.html (дата звернення: 01.11.2022).
8. Burns K. WebSockers vs Long Polling. URL: https://dev.to/kevburnsjr/websockets-vs-long-polling-3a0o#:~:text=WebSockets%20are%20Full%2DDuplex%20meaning,communicate%20something%20to%20the%20server (дата звернення: 01.11.2022).
9. WebSocket API. URL: https://caniuse.com/?search=websocket (дата звернення: 10.11.2022).
10. Awosan O. Top WebSocket libraries for Node.js in 2022. URL: https://blog.logrocket.com/top-websocket-libraries-nodejs-2022/ (дата

звернення: 25.10.2022).

11. Socket.IO Documentation. URL: https://socket.io/docs/v4/ (дата звернення: 20.11.2022).
12. McLaughlin B. Head first object-oriented analysis and design. Beijing: O'Reilly, 2007. 600 p.
13. Вендоров А. Проектирование программного обеспечения экономических информационных систем. М.: Финансы и статистика, 2006. 544 с.
14. Бублик В. Об’єктно-орієнтоване програмування. К.: ІТкнига, 2015. 624 с.
15. Booch G. Object-Oriented Analysis and Design with Applications. San Jose: Addison-Wesley, 2007. 691 p.
16. Fowler M. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002. 560 p.
17. NestJS Documentation. URL: https://docs.nestjs.com/ (дата звернення: 14.10.2022).
18. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 14.10.2022).
19. React Documentation. URL: https://reactjs.org/docs/getting-started.html (дата звернення: 20.10.2022).
20. Sass Documentation. URL: https://sass-lang.com/documentation/ (дата звернення: 20.10.2022).
21. Material UI Documentation. URL: https://mui.com/material-ui/getting-started/overview/ (дата звернення: 20.10.2022).
22. Jest Documentation. URL: https://jestjs.io/docs/getting-started (дата звернення: 25.10.2022).
23. React Testing Library Documentation. URL: https://testing-library.com/docs/react-testing-library/intro (дата звернення: 06.11.2022).
24. Cypress Documentation. URL: https://docs.cypress.io/guides/overview/why-cypress (дата звернення: 15.11.2022).
25. AWS Documentation. URL: https://docs.aws.amazon.com/ (дата звернення: 15.11.2022).

26. GitHub Actions Documentation. URL: https://docs.github.com/en/actions (дата звернення: 12.11.2022).

ДОДАТОК А

Приклади коду

А.1 Контролер користувача

```
import { Controller, Get, UseGuards } from '@nestjs/common'
import {
  ApiOperation,
  ApiTags,
  ApiBearerAuth,
  ApiUnauthorizedResponse,
  ApiInternalServerErrorResponse,
  ApiForbiddenResponse,
} from '@nestjs/swagger'
import { UserService } from './user.service'
import { UsersResponseDTO } from './dto/user.response.dto'
import { AuthGuard } from '../../shared/guards/auth.guard'
import {
  ForbiddenError,
  InternalServerError,
  UnauthorizedError,
} from '../../shared/dto/errors'
import { ListRolesEnum } from '../../config/enums'
import { RolesDecorator } from '../../shared/decorators/role.decorator'
import { RolesGuard } from '../../shared/guards/role.guard'

@ApiTags('Users')
@Controller('users')
export class UserController {
```

```
  constructor(private readonly service: UserService) {}

  @ApiBearerAuth()
  @ApiOperation({ description: 'Get list of users' })
  @ApiForbiddenResponse({ type: ForbiddenError })
  @ApiUnauthorizedResponse({ type: UnauthorizedError })
  @ApiInternalServerErrorResponse({ type: InternalServerError })
  @RolesDecorator(ListRolesEnum.ADMIN)
  @UseGuards(AuthGuard, RolesGuard)
  @Get()
  findAll(): Promise<UsersResponseDTO> {
    return this.service.findAll()
  }
}
```

А.2 Сервіс користувача

```
import { Injectable, HttpException, HttpStatus } from '@nestjs/common'
import { InjectRepository } from '@nestjs/typeorm'
import { Repository } from 'typeorm'
import { UserEntity } from '../../shared/entities'
import { UserNotificationInfo } from '../../shared/interfaces/user-notification-info'
import { UsersResponseDTO, UserResponseDTO } from './dto/user.response.dto'

@Injectable()
export class UserService {
  constructor(
    @InjectRepository(UserEntity)
```

```
  private readonly userRepo: Repository<UserEntity>,
) {}

async findAll(): Promise<UsersResponseDTO> {
  const [users, totalAmount] = await this.userRepo.findAndCount({
    relations: ['role'],
    select: [
      'id',
      'email',
      'name',
      'nickName',
      'createdAt',
      'updatedAt',
    ],
  })

  return {
    users,
    totalAmount,
  }
}

async findById(id: string): Promise<UserResponseDTO> {
  const user: UserResponseDTO = await this.userRepo.findOne(id)

  if (!user) {
    throw new HttpException('Not found User', HttpStatus.NOT_FOUND)
  }

  return user
```

```
  }

  async getUsersInfoByIds(ids: string[]): Promise<UserNotificationInfo[]> {
    return this.userRepo
      .createQueryBuilder('user')
      .select(['user.id', 'user.name', 'user.nickName', 'user.email'])
      .where('user.id IN (:...ids)', {
        ids,
      })
      .getRawMany()
  }

  async getUsersIds(): Promise<{ id: string }[]> {
    return this.userRepo
      .createQueryBuilder('user')
      .select(['user.id'])
      .getMany()
  }

  async getAmountUsers(): Promise<number> {
    return this.userRepo
      .createQueryBuilder('user')
      .select(['user.id'])
      .getCount()
  }
}
```

А.3 Робота WebSocket на боці сервера

```
const path = require('node:path');
const express = require('express');
const app = express();

const { Server } = require('socket.io');

const staticPath = path.normalize(__dirname + '/public');
app.use(express.static(staticPath));

app.get('/', function (req, res) {
   res.sendFile(__dirname + 'index.html');
});
app.get('/css/main.css', function (req, res) {
   res.sendFile(__dirname + 'css/main.css');
});
app.get('/js/main.js', function (req, res) {
   res.sendFile(__dirname + 'js/main.js');
});

const http = require('http');
const server = http.createServer(app).listen(5000, () => {
   console.log('listening on *:5000');
});

const io = new Server(server);

const users = {};
const messages = [];
```

```
io.on('connection', socket => {
  console.log('Client is connected');

  socket.on('new user', user => {
    user.status = 'appeared';
    user.id = socket.id;

    users[user.id] = user;

    socket.emit('my user id', user.id);

    io.emit('new user', user);

    setTimeout(() => {
      user.status = 'online';
      io.emit('change user status', user);
    }, 1000 * 60);
  });

  socket.on('new message', msg => {
    messages.push(msg);

    io.emit('new message', getMessageWithAuthor(msg));
  });

  socket.on('i am typing', name => {
    socket.broadcast.emit('someone is typing', name);
  });
```

```
  socket.on('disconnect', () => {
    const disconnectedUser = Object.values(users).filter(user => user.id ==
socket.id)[0];

    if(!disconnectedUser) return;

    disconnectedUser.status = 'left';

    io.emit('change user status', disconnectedUser);

    setTimeout(() => {
      disconnectedUser.status = 'offline';
      io.emit('change user status', disconnectedUser);
    }, 1000 * 60);
  });

  socket.emit('message history', getMessagesWithAuthors(messages));
  socket.emit('users list', users);

});

function getMessageWithAuthor(msg) {
  return Object.assign({}, msg, { user: users[msg.user] });
}

function getMessagesWithAuthors(messages) {
  if (messages.length > 100) {
    messages = messages.slice(messages.length - 100);
  }
```

```
   return messages.length ? messages.map(msg => getMessageWithAuthor(msg)) :
[];
}
```

А.4 Робота WebSocket на боці клієнта

```
import React, { useState, useEffect } from 'react';
import io from 'socket.io-client';
import UserList from './components/userList';
import MessageList from './components/messageList';
import MessageInput from './components/messageInput';
import { publicUsers, publicUser, publicPosts, publicPost } from './utils';

const socket = io();

function App() {
  const [isConnected, setIsConnected] = useState(socket.connected);
  const [messages, setMessages] = useState(null);
  const [users, setUsers] = useState(null);

  useEffect(() => {
    socket.on('connect', () => {
      setIsConnected(true);
    });

    socket.on('disconnect', () => {
      setIsConnected(false);
    });
```

```
  socket.on('users list', setUsers(publicUsers()));

  socket.on('new user', setUsers(publicUser()));

  socket.on('my user id', (userId) => {
    user.id = userId;
  });

  socket.on('message history', setMessages(publicPosts()));

  socket.on('new message', setMessages(publicPost()));

  return () => {
    socket.off('connect');
    socket.off('disconnect');
    socket.off('users list');
    socket.off('new user');
    socket.off('my user id');
    socket.off('message history');
    socket.off('new message');
  };
}, []);

const sendMessage = (message) => {
  socket.emit('new message', message);
};

return (
  isConnected && (
```

```
    <main>
      <div>
        <UserList users={users} />
        <MessageList messages={messages} />
      </div>

      <MessageInput onInput={sendMessage} />
    </main>
  )
 );
}

export default App;
```

А.5 Сторінка логіна

```
import React, { SyntheticEvent, FormEvent, useState } from 'react';
import { Card, FormItem, Input, Button, Row, Col } from 'elmo-elements';
import { findKey } from 'lodash';
import { StringMap } from 'src/types';
import { Fieldset } from 'src/components/FormElements/Fieldset';
import { StyledFormContainer as FormContainer } from './styles';
import { useAuth } from 'src/utils/auth';
import { TLoginFromStateType } from './types';

export const LoginForm = () => {
  const { login, isLoggingIn } = useAuth();
  const [state, setState] = useState<TLoginFromStateType>({
```

```
  form: {
    email: '',
    password: '',
  },
  errors: {},
});

const validate = () => {
  const { form } = state;
  let isValid = true;
  const errors: StringMap<string> = {};
  const emailRegexp = /\S+@\S+\.\S+/;

  if (form.email && !emailRegexp.test(form.email)) {
    isValid = false;
    errors['email'] = 'Wrong format of Email';
  }

  if (!form.email.length) {
    isValid = false;
    errors['email'] = 'Email is missing';
  }

  if (!form.password.length) {
    isValid = false;
    errors['password'] = 'Password is missing';
  }

  setState((prevState) => ({
    ...prevState,
```

```
    errors,
  }));

  return isValid;
};

const handleChange = (event: SyntheticEvent<HTMLInputElement>): void => {
  const { currentTarget } = event;

  if (!currentTarget) {
    return;
  }

  const { name, value } = currentTarget;

  setState((prevState) => ({
    ...prevState,
    form: { ...prevState.form, [name]: value },
    errors: { ...prevState.errors, [name]: undefined },
  }));
};

const handleSubmit = async (event: FormEvent) => {
  event.preventDefault();

  if (validate()) {
    await login(state.form);
  }
};
```

```
return (
  <Card heading="Login">
    <Fieldset isDisabled={isLoggingIn}>
      <form method="POST" name="login" onSubmit={handleSubmit}>
        <FormContainer>
          <FormItem label="Email" message={state.errors.email}
status={!!state.errors.email ? 'error' : undefined}>
            <Input
              ariaLabel="Email"
              name="email"
              htmlType="email"
              onChange={handleChange}
              value={state.form.email}
              status={!!state.errors.email ? 'error' : undefined}
            />
          </FormItem>

          <FormItem
            label="Password"
            message={state.errors.password}
            status={!!state.errors.password ? 'error' : undefined}
          >
            <Input
              ariaLabel="Password"
              name="password"
              htmlType="password"
              onChange={handleChange}
              value={state.form.password}
              status={!!state.errors.email ? 'error' : undefined}
            />
```

```
          </FormItem>

          <FormItem>
            <Row className="d-flex align-items-center">
              <Col xs={{ offset: 12, span: 12 }}>
                <Button
                  type={!findKey(state.errors, (err) => !!err) ? 'primary' : 'danger'}
                  htmlType="submit"
                  className="float-right"
                >
                  Login
                </Button>
              </Col>
            </Row>
          </FormItem>
        </FormContainer>
      </form>
    </Fieldset>
  </Card>
 );
};
```

A.6 Jest юніт-тест контролера користувача

```
import { Test, TestingModule } from '@nestjs/testing';
import { UserController } from './user.controller';
import { UserService } from './user.service';
import { TokenService } from '../authentication/token/token.service';
```

```
import { UserEntity } from '../../shared/entities';

describe('UserController', () => {
  let controller: UserController;
  let service: UserService;
  let tokenService: TokenService;

  beforeEach(async () => {
    service = module.get<UserService>(UserService);
    tokenService = module.get<TokenService>(TokenService);
    controller = module.get<UserController>(UserController);
  });

  it('should be defined', () => {
    expect(controller).toBeDefined();
    expect(service).toBeDefined();
    expect(tokenService).toBeDefined();
  });

  describe('User Controller', () => {
    describe('findAll tests', () => {
      it('should get the list of users. Function findAll', async () => {
        const user = {
          id: 'user id',
        };
        const result: any = { users: [], totalAmount: 0 };

        jest.spyOn(service, 'findAll').mockImplementationOnce(() => result);
        try {
          const response = await controller.findAll(user as any);
```

```
        expect(response).toBe(result);
      } catch (error) {
        expect(error).toBeUndefined();
      }
    });
  });
 });
});
```

А.7 React Testing Library юніт-тест на боці клієнта

```
import React from 'react';
import { render } from '@testing-library/react';
import LoginPage from './LoginPage';
import { useAuth } from 'src/utils/auth';

jest.mock('src/utils/auth');

describe('LoginPage', () => {
  beforeEach(() => {
    (useAuth as jest.Mocked<any>).mockReturnValueOnce({ user: null });
  });

  afterEach(jest.clearAllMocks);

  it('should render properly with login form', () => {
    const { getByRole, getByLabelText } = render(<LoginPage />);
    // card heading
```

```
    getByRole('heading', { name: 'Login' });
    // form
    const form = getByRole('form');
    // form items
    expect(form.querySelectorAll('label')[1]).toHaveTextContent('Email');
    getByLabelText('Email', { selector: 'input' });
    expect(form.querySelectorAll('label')[2]).toHaveTextContent('Password');
    getByLabelText('Password', { selector: 'input' });
    // submit button
    expect(getByRole('button', { name: 'Login' })).toHaveAttribute('type', 'submit');
  });
});
```