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

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

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: Дослідження засобів розробки програмного забезпечення для обміну повідомленнями в реальному часі з аутентифікацією користувачів

Виконав: студент VI курсу, групи САм-61
спеціальності 124 Системний аналіз
(шифр і назва спеціальності)

_____ Бачинський А.В.
(підпис) (прізвище та ініціали)

Керівник _____ Марценюк В.П.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Дуда О.М.
(підпис) (прізвище та ініціали)

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

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2024

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

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

Кафедра комп'ютерних наук
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Боднарчук І.О.
(підпис) (прізвище та ініціали)
«____»_____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Магістр
(назва освітнього ступеня)

за спеціальністю 124 Системний аналіз
(шифр і назва спеціальності)

Студенту Бачинському Артуру Васильовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Дослідження засобів розробки програмного забезпечення для обміну повідомленнями в реальному часі з аутентифікацією користувачів

Керівник роботи Марценюк Василь Петрович, доктор технічних наук, професор кафедри КН
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*29*» *листопада* 2024 року № *4/7-1140*

2. Термін подання студентом завершеної роботи 19 грудня 2024р.

3. Вихідні дані до роботи Документаційні інтернет-джерела щодо використання веб технологій

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз існуючих методів та середовищ для розробки програмного забезпечення. 2 Аналіз вибраних технологій і засобів для реалізації серверної частини програмного забезпечення. 3 Проектування і реалізація клієнтської частини застосунку, та інтеграція з сервером. 4 Охорона праці та безпека в надзвичайних ситуаціях. Висновки. Додатки

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Охорона праці	Сенчишин В.С., доцент		
Безпека в надзвичайних ситуаціях	Клепчик В.М., ст. викладач		

7. Дата видачі завдання 19 грудня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Ознайомлення з завданням до кваліфікаційної роботи	14.11.24-15.11.24	*Виконано*
2.	Підбір наукових джерел по темі роботи	16.11.24-20.11.24	*Виконано*
3.	Переклад та опрацювання наукових джерел	20.11.24-23.11.24	*Виконано*
	по темі кваліфікаційної роботи		
4.	Виконання дослідження щодо теми	24.11.24-10.12.24	*Виконано*
	Кваліфікаційної роботи		
5.	Оформлення першого розділу	11.12.24-12.10.24	*Виконано*
6.	Оформлення другого розділу	12.12.24-13.12.24	*Виконано*
7.	Оформлення третього розділу	13.12.24-14.12.24	*Виконано*
8.	Виконання завдання до підрозділу «Охорона праці»	08.12.24-09.11.24	*Виконано*
9.	Виконання завдання до підрозділу «Безпека в	10.12.24-12.12.24	*Виконано*
	надзвичайних ситуаціях»		
10.	Оформлення кваліфікаційної роботи	12.12.24-31.12.24	*Виконано*
11.	Нормоконтроль	14.12.24-15.12.24	*Виконано*
12.	Перевірка на плагіат	15.12.24	*Виконано*
13.	Попередній захист кваліфікаційної роботи	16.12.24	*Виконано*
14.	Захист кваліфікаційної роботи	23.12.2024	

Студент _____ Бачинський Артур Васильович
(підпис) (прізвище та ініціали)

Керівник роботи _____ Марценюк Василь Петрович
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дослідження засобів розробки програмного забезпечення для обміну повідомленнями в реальному часі з аутентифікацією користувачів // Кваліфікаційна робота освітнього рівня «Магістр» // Бачинський Артур Васильович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра комп'ютерних наук, група САм-61 // Тернопіль, 2024 // С. 53, рис. – 35, бібліогр. – 14.

Ключові слова: контролер, середовище, веб-розробка, модель, express, сервер, аутентифікація, хешування

Кваліфікаційна робота присвячена дослідженню засобів розробки програмного забезпечення для обміну повідомленнями в реальному часі з аутентифікацією користувачів.

В першому розділі кваліфікаційної роботи досліджено існуючі методи для розробки програмного забезпечення. Висвітлено сучасні технології. Розглянуто середовища для написання коду.

В другому розділі кваліфікаційної роботи обґрунтовано вибір технологій та середовище для реалізації сервера. Було спроектовано та реалізовано серверну частину. Створено допоміжну функцію для захисту приватних запитів від неавторизованих користувачів.

В третьому розділі кваліфікаційної роботи було налаштовано інструменти та сучасні бібліотеки для розробки. Створено інтерфейс клієнтської частини програмного забезпечення. Реалізовано функціонал для авторизації і реєстрації користувачів та обміну повідомленнями.

В четвертому розділі кваліфікаційної роботи було розглянуто питання щодо охорони праці, та безпеки в надзвичайних ситуаціях.

ANNOTATION

Research of software development tools for real-time messaging with user authentication // The educational level "Master" qualification work // Bachynskiy Artur // Ternopil Ivan Puluj National Technical University, Faculty of Computer Information Systems and Software Engineering, Department of Computer Science, SAm-61 group // Ternopil, 2024 // P. 53, fig. – 35, references -14

Keywords: controller, environment, web-development, model, express, server, authentication, hashing

The qualification work is devoted to the study of software development tools for real-time messaging with user authentication.

In the first section of the qualification work, the existing methods of software development were investigated. Modern technologies are highlighted. Considered environments for writing code.

The second section of the qualification work substantiates the choice of technologies and the environment for server implementation. The server part was designed and implemented. Added a helper feature to protect private requests from unauthorized users.

In the third section of the qualification work, tools and modern libraries for development were set up. The interface of the client part of the software was created. The functionality for authorization and registration of users and messaging was implemented.

The fourth section of the qualification work addressed the issues of labor protection and safety in emergency situations.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

JS – JavaScript.

React – JavaScript бібліотека для створення клієнтських інтерфейсів.

JSON – JavaScript object notation.

REST – Методологія для створення API

Model – модель об'єкта.

NPM – Пакетний менеджер

Context – засіб для керування глобальними даними.

DOM – об'єктна модель документа.

WebStorm – редактор коду.

JWT – токен для безпечної передачі даних між клієнтом і сервером.

Controller – функція для обробки вхідних запитів.

Socket – програмний інтерфейс для обміну даних між процесорами.

ЗМІСТ

ВСТУП

Актуальність теми. Задачами досліджень в області створення програмних продуктів займається багато українських та закордонних вчених. Зокрема, в ТНТУ ім. Івана Пулюя можна згадати роботи [1], [2], [3]. Предметом досліджень в них є задачі оптимізації процесів забезпечення якості на ранніх етапах розробки. Підходи з цих робіт можна використати в цій кваліфікаційній роботі для вибору засобів розробки програмного забезпечення, та платформи для реалізації механізмів двосторонньої аутентифікації.

Мета і задачі дослідження. Метою даної кваліфікаційної роботи освітнього рівня «Магістр» є розробка програмного забезпечення для обміну повідомленнями в реальному часі, та дослідження вибраних технологій для реалізації сервера та клієнтської частини. Для досягнення поставленої мети потрібно виконати ряд завдань, зокрема:

– Дослідити існуючі методи розробки веб-сторінок.
– Проаналізувати обрані технології та методи для розробки.
– Розробити серерну частину програмного забезпечення.
– Реалізувати інтерфейс та функціонал клієнтської частини.
– Розмістити веб-застосунок на віддаленому сервері.

Об’єкт дослідження. Процеси аутентифікації, методи і технології, що використовуються для підтвердження особи користувача, методи для обміну повідомленнями.

Предмет дослідження. Методи розробки програмного забезпечення для обміну повідомленнями в реальному часі.

Наукова новизна одержаних результатів кваліфікаційної роботи полягає у тому, що розроблений веб-месенджер забезпечує безпечний обмін повідомленнями в реальному часі завдяки впровадженню новітніх методів аутентифікації та шифрування даних. Це підвищує рівень безпеки користувачів і відповідає сучасним вимогам до захисту інформації.

Практичне значення одержаних результатів. Створено безпечний веб-месенджер, який забезпечує ефективний обмін повідомленнями і може бути

впроваджений у різних сферах, з можливістю подальшого вдосконалення та масштабування.

Апробація результатів магістерської роботи. Основні результати проведених досліджень обговорювались на XII науково-технічній конференції «Інформаційні моделі, системи та технології» Тернопільського національного технічного університету імені Івана Пулюя, м. Тернопіль, 2024 р. (Додаток А)

1 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ ТА СЕРЕДОВИЩ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Огляд існуючих середовищ для веб-розробки

Вибір правильного редактора є критично важливим. Ефективний редактор може значно підвищити швидкість роботи розробника. Функції, такі як автозавершення, підсвітка синтаксису та рефакторинг коду, дозволяють зменшити час на виконання рутинних завдань. Інтуїтивно зрозумілий інтерфейс полегшує навчання нових технологій і зменшує час, витрачений на налаштування середовища. Розробники можуть зосередитися на написанні коду, а не на пошуку функцій редактора.

У світі веб-розробки середовище коду відіграє ключову роль у процесі створення ефективних і зручних додатків. Правильний вибір редактора може суттєво підвищити продуктивність розробника, забезпечити зручність у роботі та спростити процес налагодження коду. Сучасні редактори коду пропонують широкий спектр функцій, таких як підсвітка синтаксису, інтеграція з системами контролю версій і можливості для налагодження. Ось кілька з найпопулярніших середовищ для веб-розробки:

- Visual Studio Code.
- Sublime Text.
- Atom.
- WebStorm.
- Brackets.
- Eclipse.
- PhpStorm.

Сучасні редактори часто підтримують численні мови програмування та фреймворки. Це дає можливість працювати з різними проектами без необхідності змінювати середовище. Багато редакторів мають вбудовану інтеграцію з системами контролю версій, системами управління проектами та іншими інструментами. Це спрощує командну роботу та дозволяє ефективніше

керувати проектами. Потужні функції налагодження допомагають швидко виявляти та виправляти помилки. Це зменшує час, витрачений на тестування та вирішення проблем, що підвищує загальну якість коду.

Можливість налаштування редактора під власні потреби — це важлива перевага. Розробники можуть додавати плагіни або налаштовувати інтерфейс, щоб зробити середовище роботи максимально комфортним. Деякі редактори підтримують функції спільної роботи, що дозволяє командам розробників працювати разом у реальному часі. Це покращує комунікацію і пришвидшує процес розробки.

Усе це робить вибір правильного редактора важливим кроком у створенні успішних веб-додатків. Правильний інструмент може не лише полегшити процес роботи, але й суттєво підвищити продуктивність і якість кінцевого продукту.

1.1.1 Аналіз середовища VS Code

Visual Studio Code – це безкоштовний, потужний редактор коду, розроблений компанією Microsoft. Він швидко здобув популярність серед розробників завдяки своїй гнучкості, простоті використання та широкому набору функцій. Цей редактор підтримує безліч мов програмування, що робить його універсальним інструментом для веб-розробки.

Завдяки великій бібліотеці плагінів, користувачі можуть легко налаштувати редактор під свої потреби. Від інтеграції з фреймворками до додаткових тем і шрифтів — можливості практично безмежні. VS Code має вбудовані інструменти для налагодження, що дозволяє розробникам легко знаходити та виправляти помилки у коді.

Інтеграція з Git та іншими системами контролю версій спрощує роботу в команді, дозволяючи легко відстежувати зміни в коді. Також є вбудований термінал який дозволяє виконувати команди безпосередньо з редактора, що підвищує продуктивність.

Окрім плюсів даного редактора, присутні також мінуси, наприклад його вимога до ресурсів.

VS Code може бути важким для старих або слабких комп'ютерів, оскільки він споживає значну кількість оперативної пам'яті та процесорних ресурсів, особливо з багатьма розширеннями. Хоча VS Code має безліч розширень, деякі функції можуть бути менш інтегрованими або зручними в порівнянні з повноцінними IDE, такими як Visual Studio або IntelliJ IDEA.

У великих проєктах іноді виникають проблеми з конфігураціями або налаштуваннями, які не завжди очевидні. Також є обмеженість в роботі з деякими мовами. VS Code підтримує багато мов, для деяких специфічних фреймворків або мов може не бути достатньо потужних інструментів.

У порівнянні з повноцінними IDE, VS Code може не мати деяких функцій, таких як глибока рефакторизація або підтримка специфічних тестових фреймворків.

Нестабільність розширень. Не всі розширення якісні, деякі можуть бути нестабільними або мати проблеми з оновленнями. Новачкам може бути важко налаштувати середовище та знайти потрібні розширення, що може призвести до витрат часу.

Ці мінуси можуть варіюватися в залежності від потреб користувача, але їх варто враховувати при виборі інструменту для розробки. Незважаючи на деякі мінуси, цей редактор може підійти як новачків завдяки простому інтерфейсу і великій кількості ресурсів для навчання, VS Code є чудовим вибором для тих, хто тільки починає вивчати програмування. Так і для професіональних розробників.

Даний редактор доступний на Windows, macOS та Linux, що робить його зручним для користувачів різних операційних систем.

В досвідчених розробників, які працюють над великими проєктами, VS Code може бути дуже корисним завдяки своїй гнучкості, можливості кастомізації та підтримці сучасних технологій.

VS Code підходить як для новачків, так і для професіоналів. Його універсальність та можливості налаштування роблять його популярним вибором серед різних категорій розробників.

1.1.2 Аналіз середовища WebStorm

WebStorm – це комерційний редактор коду, розроблений компанією JetBrains, спеціалізований на JavaScript та сучасних веб-технологіях. Цей інструмент здобув велику популярність серед веб-розробників завдяки своїм багатофункціональним можливостям і зручному інтерфейсу. Основні можливості даного середовища:

– Підтримка JavaScript і TypeScript.
– Інтеграція з системами контролю версій.
– Потужний налагоджувач.
– Автозавершення та рефакторинг коду.
– Тестування.
– Системи управління пакетами.
– Розширення і плагіни.

WebStorm є потужним інструментом для розробників, які працюють з сучасними веб-технологіями, надаючи всі необхідні функції для ефективної розробки, тестування та налагодження коду. Його зручність та функціональність роблять його ідеальним вибором для професіоналів у сфері веб-розробки.

Це інтегроване середовище для розробки на JavaScript та пов'язаних з ним технологій (React, Vue, Angular) і інші. На рисунку 1.1 показано інтерфейс середовища.

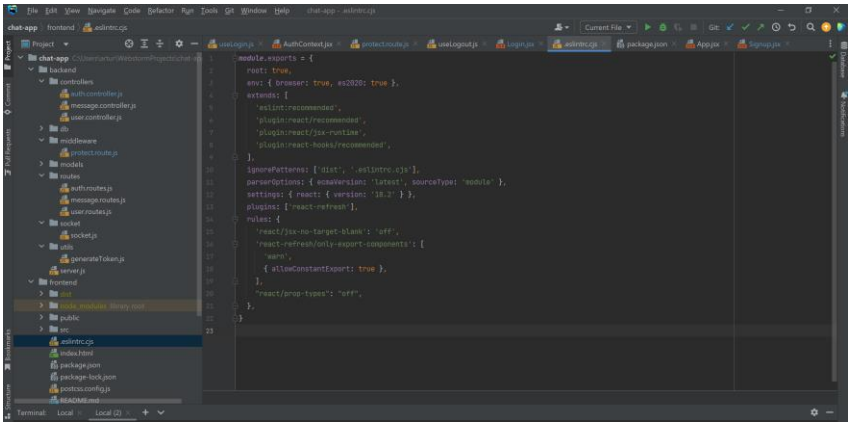

Рисунок 1.1 – Інтерфейс середовища WebStrom

Основним плюсом даного середовища як для JavaScript розробників являється глибока інтеграція. WebStorm пропонує потужну інтеграцію з JavaScript, TypeScript, HTML та CSS, що робить його ідеальним для веб-розробки.

Також варто сказати про розширені інструменти рефакторингу, він має зручні інструменти для рефакторингу коду, що допомагає підтримувати чистоту та організацію проєкту.

WebStorm підтримує популярні фреймворки для тестування (Jest, Mocha, Jasmine тощо), що спрощує написання та запуск тестів, високоякісне автозавершення та інтелектуальні підказки для коду, що допомагає швидше писати без помилок.

Дуже зручним являється те, що це середовище має вбудовані інструменти для управління версіями, що робить колективну роботу більш зручною.

Даний редактор підходить для професійних веб розробників завдяки своїм потужним функціям і зручному інтерфейсу.

WebStorm підходить для досвідчених розробників, які займаються складними проєктами. Також завдяки вбудованим інструментам для співпраці та управління версіями, WebStorm підходить для командної роботи.

Найбільше підходить для розробників які працюють з JavaScript / TypeScript, допомагаючи отримати максимальну продуктивність.

Із небагатьох мінусів являється вартість даного середовища, на відмінно від VS Code, він є платним продуктом, і для індивідуальних розробників або малих команд це може бути значною витратою.

Новачкам може бути важко адаптуватися до всіх функцій та налаштувань, оскільки середовище може здаватися перевантаженим та підтримка інших мов може бути менш глибокою в порівнянні з IDE, які спеціалізуються на цих мовах.

Також для активації ліцензії і доступу до деяких функцій потрібен стабільний інтернет, що може бути проблемою у випадку відключення.

Хоча WebStorm має безліч можливостей для кастомізації, це може бути і плюсом, і мінусом — іноді налаштування можуть бути складними і заплутаними.

1.2 Сучасні методи написання серверної частини ПЗ

Існує безліч мов програмування, фреймворків і бібліотек, які використовуються для створення серверної частини веб-додатків. Вибір конкретних технологій залежить від вимог проєкту, переваг команди та специфіки задач. Нижче буде наведено огляд основних мов і фреймворків, а також сучасних методів написання серверів.

Node.js дозволяє використовувати JavaScript для написання серверного коду, що забезпечує однорідність технологій на стороні клієнта і сервера. Це підходить для розробки реальних додатків у реальному часі (чати, ігри) завдяки подієво-орієнтованій архітектурі.

Python відомий за свою простоту і легкість у вивченні, Python часто використовується для веб-розробки за допомогою фреймворків, таких як Django і Flask. Django пропонує повнофункціональне рішення для розробки, в той час як Flask є легким мікрофреймворком, що дозволяє створювати прості API.

Java залишається популярною мовою для серверної розробки завдяки стабільності та безпеці. Фреймворки, такі як Spring Boot, забезпечують швидку і зручну розробку мікросервісів.

PHP довгий час використовувався для веб-розробки і залишається популярним завдяки фреймворкам, таким як Laravel і Symfony, які спрощують написання коду та підвищують продуктивність.

Також для кожної мови програмування існує безліч бібліотек і фреймворків для комфортної та продуктивнішої роботи. Кожна бібліотека це набір вже готових функцій, правил, стилів, тощо. За допомогою яких можна використовувати окремі компоненти для окремих завдань.

Express.js – Мінімалістичний фреймворк для Node.js, який дозволяє швидко створювати веб-додатки і API. Підходить для побудови RESTful сервісів.

Django – Фреймворк для Python, який забезпечує комплексні можливості для розробки, включаючи ORM, автентифікацію та адмін-панель.

Laravel – Сучасний фреймворк для PHP, що має великий набір функцій для створення складних веб-додатків, включаючи маршрутизацію, ORM та

валідацію даних. Вище було оглянуто мови та бібліотеки для створення серверної частини програмного забезпечення. Також потрібно уважно підходити до правил та методів написання сервера, існує декілька методів та способів, наприклад «REST» (Representational State Transfer): Методологія для створення програмного інтерфейсу, що базується на принципах HTTP. RESTful API дозволяють зручно взаємодіяти з ресурсами за допомогою стандартних методів (GET, POST, PUT, DELETE). GraphQL – це альтернатива REST, яка дозволяє клієнтам запитувати лише ті дані, які їм потрібні. Це забезпечує більш ефективну передачу даних та зменшує кількість запитів.

WebSockets – це протокол, який дозволяє двосторонню комунікацію між клієнтом і сервером. Використовується для створення додатків у реальному часі, таких як чати та ігри.

Сучасні методи написання серверів у веб-розробці пропонують великий вибір технологій, фреймворків і підходів, які можуть бути адаптовані під специфіку проєкту. Обрання правильних інструментів та методів є ключовим фактором для створення ефективних, масштабованих і безпечних веб-додатків.

1.3 Порівняння баз даних, та огляд інструментів для роботи з ними

Бази даних є невід’ємною частиною сучасних веб-додатків, оскільки вони забезпечують зберігання, управління і доступ до даних. Існує безліч різних типів баз даних, і вибір між ними може суттєво вплинути на продуктивність та масштабованість проєкту. Нижче наведено типи баз даних:

1. Реляційні бази даних.
2. Нереляційні бази даних.
3. Графові бази даних.
4. Системи управління даними в пам’яті.

Реляційні бази даних зберігають дані в таблицях, де кожна таблиця містить рядки та стовпці. Вони підтримують SQL (Structured Query Language) для управління даними. Ці бази підходять для структурованих даних з чіткою схемою.

Нереляційні бази даних дозволяють зберігати дані у форматах, що не потребують чіткої схеми. Вони добре підходять для неструктурованих даних або даних, що швидко змінюються.

Графові бази даних спеціалізуються на зберіганні даних у вигляді графів. Вони корисні для моделювання складних зв’язків між даними, таких як соціальні мережі.

Системи управління даними зберігають дані в пам’яті для забезпечення надшвидкого доступу, що робить їх ідеальними для кешування та високопродуктивних додатків.

Якщо порівнювати реляційні та нереляційні бази даних за схемами, то можна сказати, що реляційні бази мають чітку схему, що може ускладнити зміни. Нереляційні бази більш гнучкі щодо структури.

Якщо мова йде про запити, то SQL для реляційних баз даних є потужним інструментом для запитів. Нереляційні бази можуть використовувати API або специфічні мови запитів, які можуть бути менш універсальними.

За скалярністю нереляційні бази часто легше масштабувати горизонтально (додавання нових серверів), тоді як реляційні бази зазвичай масштабуються вертикально (покращення існуючого сервера).

Інструменти для роботи з базами даних є одним з основних моментів у розробці сервера, тому що при роботі із великою кількістю даних, дуже легко втратити концентрацію, та розгубитись, для того щоб можна було легше керувати даними у базі, використовуються спеціальні інструменти, через які можна отримувати доступ до бази даних.

pgAdmin – це офіційний інтерфейс для роботи з PostgreSQL, що забезпечує зручний доступ до бази даних та інструменти для її адміністрування.

MySQL – це інструмент для проектування, моделювання та управління базами даних MySQL. Має функції для оптимізації запитів і адміністрування.

Для роботи з MongoDB використовується MongoDB Compass, це графічний інтерфейс, який дозволяє візуалізувати дані, виконувати запити та аналізувати структуру бази.

Також дає змогу видаляти, змінювати дані та сортувати таблиці, не використовуючи при цьому веб-сервіс (див. рис. 1.2).

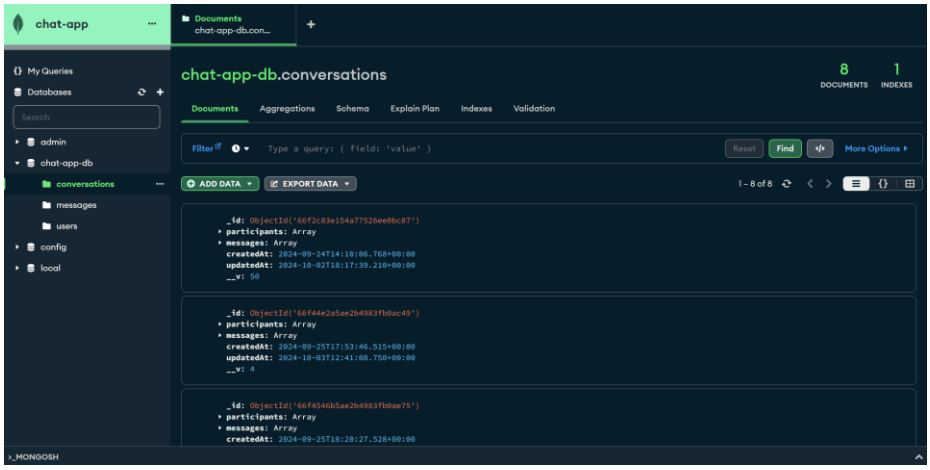

Рисунок 1.2 – Графічний інтерфейс MongoDB Compass

Отже, вибір бази даних та відповідних інструментів для роботи з ними є важливим етапом у розробці веб-додатків. Розуміння різниць між реляційними та нереляційними базами даних, а також доступних інструментів, допоможе приймати обґрунтовані рішення щодо архітектури проектів.

1.4 Висновок до першого розділу

У першому розділі кваліфікаційної роботи було всебічно розглянуто сучасні середовища та методи веб-розробки, що охоплює редактори коду, серверні технології та бази даних. Було проаналізовано важливість вибору відповідного редактора коду.

Серед таких популярних рішень, як Visual Studio Code і WebStorm, було відзначено їх функціональність, підтримку плагінів та інтеграцію з системами контролю версій, що дозволяє підвищувати продуктивність та зосереджуватись на створенні якісного коду.

Проаналізовано методи за засоби для написання серверної частини програмного забезпечення, включаючи основні мови програмування, такі як JavaScript, Python, Java, PHP, а також популярні фреймворки. Було підкреслено, що ці технології забезпечують гнучкість та масштабованість, необхідні для

розробки сучасних веб-додатків, а також описано переваги використання різних архітектурних підходів, таких як REST, GraphQL і мікросервіси.

Також було здійснено порівняння реляційних і нереляційних баз даних, з акцентом на їх особливості, переваги та сценарії використання. Розглянуті інструменти для роботи з базами даних, які сприяють зручному управлінню даними та оптимізації запитів, та дозволяють ефективно інтегрувати бази даних у проекти.

2 АНАЛІЗ ВИБРАНИХ ТЕХНОЛОГІЙ І ЗАСОБІВ ДЛЯ РЕАЛІЗАЦІЇ СЕРВЕРНОЇ ЧАСТИНИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Обґрунтування вибору мови Node js для реалізації сервера

Для реалізації серверної частини було обрано Node js, обґрунтовано це тим, що Node використовує неблокуючу архітектуру, що дозволяє обробляти кілька запитів одночасно без блокування виконання коду. Це є особливо важливим для веб-додатків, які повинні забезпечувати високу продуктивність і швидкість реагування. Завдяки асинхронній обробці запитів, Node може ефективно керувати великою кількістю одночасних з'єднань, що робить його ідеальним для створення реальних додатків.

Великий плюс цієї мови у тому, що вона дозволяє використовувати JavaScript для розробки як клієнтської, так і серверної частини. Це спрощує робочий процес, оскільки не потрібно вивчати нову мову для створення серверної логіки, що підвищує продуктивність і знижує ймовірність помилок.

Node.js має велику екосистему пакетів завдяки npm, що дозволяє легко інтегрувати сторонні модулі і бібліотеки у проект. Це значно скорочує час розробки, оскільки багато рішень уже реалізовані та доступні для використання.

Варто зазначити високу продуктивність мови, яка використовує V8 JavaScript Engine, що забезпечує високу продуктивність за рахунок компіляції JS у машинний код. Це робить Node швидким у виконанні, що є важливим фактором для серверних додатків.

Для реалізації серверної частини на Node було обрано фреймворк Express js, він вважається одним із найпопулярніших фреймворків для Node. Цей мікрофреймворк надає широкий спектр функцій для створення веб-додатків і API, роблячи процес розробки ще більш зручним і ефективним. Основними його перевагами являються:

1) Простота використання.
2) Гнучкість.
3) Широка екосистема.

4) Підтримка RESTful API.

5) Швидкість і продуктивність.

Отже, вибір Node.js як мови для реалізації сервера обґрунтований її високою продуктивністю і широкою екосистемою. Використання Express.js підвищує ефективність розробки, дозволяючи створювати гнучкі, масштабовані та продуктивні веб-додатки.

2.2 Ініціалізація сервера за допомогою фреймворка Express js

Сервер було створено за допомогою express, у кореневому файлі налаштовано усі маршрути та імпортовані потрібні бібліотеки для налагодження та запуску сервера. На рисунку 2.1 показано функцію та налаштування для запуску сервера.

Рисунок 2.1 – Кореневий файл сервера

Після ініціалізації та налаштування, було додано необхідні програми, інструменти та бібліотеки для розробки серверного функціоналу, а саме для хешування даних, генерації унікального токену, сокет для обміну даними у реальному часі, та бібліотека для підключення бази даних.

Спочатку було розроблено логіку маршрутів, для зручного та читабельного коду було поділено адреса маршрутів на декілька (див. рис. 2.2).

```
app.use('/api/auth', authRoutes)
app.use('/api/messages', messageRoutes)
app.use('/api/users', userRoutes)
```

Рисунок 2.2 – Адресні шляхи запитів

По кожному шляху, в залежності від того по якому користувач відсилає запит, буде спрацьовувати файл з потрібними роутами. В залежності який саме шлях спрацює по запиту, буде виконуватись функція ендпоінт, де виконується уся логіка.

Отже коли налаштовано серверну маршрутизацію та додано усі необхідні бібліотеки для розробки, наступним ділом було створено та підключено базу даних.

2.3 Створення та підключення бази даних MongoDB до сервера

Для збереження та керуванням даними було обрано MongoDB, оскільки вона є документно-орієнтованою базою, що дозволяє зберігати дані у форматі JSON. Це спрощує роботу з різними структурами даних без потреби попереднього визначення схеми.

Вона також забезпечує високу продуктивність завдяки використанню індексів та оптимізації запитів, що робить її ідеальною для роботи з великими обсягами неструктурованих даних, характерних для сучасних веб-додатків. На рисунку 2.3 продемонстровано функцію підключення та запуску бази даних.

```
import mongoose from "mongoose";

const connectToMongoDB = async () => {
    try {
        await mongoose.connect(process.env.MONGO_DB_URL)
        console.log('connected')
    } catch (error) {
        console.log(error.message)
    }
}

export default connectToMongoDB
```

Рисунок 2.3 – Функція підключення бази даних

За допомогою бібліотеки mongoose було реалізовано підключення до бази. Усі приватні дані, ключі та посилання на базу даних з паролем для підключення знаходиться в прихованому файлі “.env” для безпеки.

2.4 Реалізація контролерів та функції для захисту приватних запитів від неавторизованих користувачів

Першим контролером було розроблено реєстрацію користувачів. За допомогою вище описаних бібліотек та інструментів є можливість безпечно зареєструвати користувача, зберегти його дані в базу даних, та надійно зашифрувати пароль, на випадок втрати даних, пароль не буде видимий.

Перед тим як сервер відправить відповідь на клієнт, буде проведена валідація, перевірка на заповнення усіх полів. Також є поле для підтвердження паролю, перевірка на те, чи існує вже такий користувач у базі даних.

У випадку якщо хоча б одна з цих перевірок не пройде, функцію буде повернено з повідомленням про помилку. На рисунку 2.4 наведено функцію яка відповідає за реєстрацію.

```
export const signup = async (req, res) => {
    try {
        const { fullName, username, password, confirmPassword, gender } = req.body

        if(!fullName || !username || !password || !confirmPassword || !gender) {
            return res.status(400).json({ error: 'Усі поля обов’язкові' })
        }
        if(password !== confirmPassword) {
            return res.status(400).json({ error: 'Паролі не співпадають' })
        }

        const user = await User.findOne({ username })
        if(user) {
            return res.status(400).json({ error: 'Користувач з таким логіном вже існує' })
        }
        const salt = await bcrypt.genSalt(10)
        const hashPassword = await bcrypt.hash(password, salt)

        const boyProfilePic = `https://avatar.iran.liara.run/public/boy?username=${username}`
        const girlProfilePic = `https://avatar.iran.liara.run/public/girl?username=${username}`
```

Рисунок 2.4 – Логіка реєстрації

Після того як функція пройшла усі перевірки, потрібно зашифрувати пароль який користувач відправив на сервер.

Для цього було створено окрему змінну, у якій за допомогою бібліотеки bcrypt js було використано метод hash, та в параметри було передано пароль користувача, та сіль, яка додається до зашифрованого паролю для додаткової безпеки. Це випадковий алгоритм даних який створює унікальний хеш, який практично неможливо взламати.

Після шифрування паролю, було проведено ще одну логіку для генерації аватару користувачів в залежності від їх статті. При реєстрації після заповнення усіх даних, у формі є вибір статі, і в залежності від вибору користувачем його статі, йому буде згенеровано фотографію в стилі 2d, на якій буде або жінка, або чоловік.

У функції можна помітити після шифрування паролю, логіку яка відповідає за генерацію фотографій. Після того як вся логіка була виконана, та користувач може пройти реєстрацію, було створено змінну в яку поміщено усі дані про користувача.

Перед збереженням у базу даних зареєстрованого користувача, була виконана ще одна логіка для генерації унікального токена.

JWT – JSON Web Token це стандарт, який дозволяє безпечно передавати інформацію між сторонами у формі об'єкта JSON. Token у бібліотеці JWT використовується для аутентифікації та авторизації. Вони зручні тим, що їх можна використовувати в різних середовищах (веб, мобільні додатки) і вони легко масштабуються.

Для зручності було створено окремо функцію, яка приймає два параметри, це унікальний ID користувача і відповідь сервера (response). Для генерації унікального токена потрібно внести дані користувача, це може бути пошта, айді, або цілий об’єкт. Для генерації хватає хоча б одного ключа, у даному випадку це був айді. Далі за допомогою методу sign у параметри передано айді користувача, секретний ключ, який являється набором символів, та термін життя цього токену, тобто якщо вказати тривалість життя 10 днів, то після авторизації користувач може не виходити з свого акаунту, а буде авторизований 10 днів, після закінчення цього терміну токен буде прострочений і користувачу потрібно буде заново

авторизуватись, щоб оновити токен. У даному випадку термін дії токену триває 15 днів (див. рис. 2.5).

```
const generateTokenAndSetCookie = (userId, res) => {
    const token = jwt.sign({ userId }, process.env.JWT_SECRET_KEY, { expiresIn: '15d' })
    res.cookie('jwt', token, {
        maxAge: 15 * 24 * 60 * 60 * 1000,
        httpOnly: true,
        sameSite: 'strict',
        secure: process.env.NODE_ENV !== 'development'
    })
}
```

Рисунок 2.5 – Функція генерації токена

Після того як користувач пройшов усі необхідні перевірки, валідацію даних та отримав токен, його було збережено у базу даних.

Наступним ділом було розроблено авторизацію користувача, на відмінно від реєстрації, функція авторизації є меншою та легшою, оскільки в ній не потрібно зберігати якісь дані у базу, відбувається валідація даних, також по унікальному імені користувача (username) відбувається перевірка у чи є такий логін у базі даних, якщо такий логін не існує, то сервер поверне помилку з текстом про те, що такого користувача не існує.

Якщо усі дані пройшли перевірку, то користувачу надається токен та повідомлення про успішну авторизацію.

Після того як успішно було реалізовано авторизації та реєстрацію користувача, було реалізовано функцію для виходу з додатку (див. рис. 2.6).

```
export const logout = (req, res) => {
    try {
        res.cookie('jwt', '', { maxAge: 0 })
        res.status(200).json({ message: 'Logged out successfully' })
    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}
```

Рисунок 2.6 – Функція logout

У функції виходу виконується очищення cookie, як показано на рисунку вище по ключу ‘jwt’ в значення передається пуста стрічка, отже в момент коли користувач вийде з акаунту, його токен буде замінений на пусту стрічку і в такому випадку при умові обов’язкової наявності токена, його буде перенаправленно на сторінку авторизації.

Наступним етапом було розроблено контролер для реалізації функції отримання усіх користувачів, для подальшого використання на клієнтській частині. Отримується список зареєстрованих користувачів для відображення їх у списку, окрім авторизованого користувача, тому що авторизований користувач не повинен бачити себе у списку потенційний співрозмовників (див. рис. 2.7).

```
export const getUsersForSidebar = async (req, res) => {
    try {
        const loggedInUserId = req.user._id

        const filteredUsers = await User.find({ _id: { $ne: loggedInUserId } }).select('-password')

        res.status(200).json(filteredUsers)

    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}
```

Рисунок 2.7 – Функція для отримання усіх користувачів

При методі find в налаштуваннях вказано умову в якій авторизований користувач не повинен повернутись у списку, тому як на клієнтській частині, коли буде відображено список усіх користувачів, авторизований юзер не повинен бачити себе у загальному списку.

Наступним етапом була реалізація обміну повідомленнями. При відправці або отриманні повідомлення на сервер приходить запит з даними, модель яких є описана, дані відправника, дані отримувача, та саме повідомлення. Усі з цих об’єктів є обов’язковими. На рисунку 2.8 показано як виглядає модель повідомлення.

```
import mongoose from "mongoose";

const messageSchema = new mongoose.Schema({
    senderId: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User',
        required: true
    },
    receiverId: {
        type: mongoose.Schema.Types.ObjectId,
        ref: 'User',
        required: true
    },
    message: {
        type: String,
        required: true
    }
}, { timestamps: true })

const Message = mongoose.model('Message', messageSchema)
```

Рисунок 2.8 – Модель повідомлення

Також додано ключ який зберігає та показує час, в який було відправлено або отримано повідомлення, щоб потім була змога відобразити час на клієнтській частині, для повноцінного інтерфейсу.

Коли було описано модель документа, далі було створено модель бесіди, де зберігається масив з повідомленнями обох учасників, і людина з якою триває бесіда.

Тобто якщо користувач авторизований і проводить обмін повідомленнями з іншим користувачем, то в базі даних зберігається модель де є усі повідомлення та дані співбесідника (див. рис. 2.9).

```
const conversationSchema = new mongoose.Schema({
    participants: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'User'
        }
    ],
    messages: [
        {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'Message',
            default: []
        }
    ]
}, { timestamps: true })
```

Рисунок 2.9 – Модель бесіди

Після створення та описання всіх моделей, було реалізовано функціонал для відправки повідомлення. Було створено окремий контролер для реалізації функціоналу з повідомленнями, та в ньому функцію, яка відповідає за відправку повідомлення.

Спочатку було отримано повідомлення з запиту яке прийшло від клієнта, далі було виявлено айді користувача йому прийшло повідомлення, для того аби його знайти, достатньо звернутись до параметрів, при авторизації в параметрах зберігається ID користувача.

Далі було збережено у змінну ID відправника, ці всі дані потрібні для того, щоб надалі можна було зберегти їх у базу даних. Після того як дані про відправника, отримувача, та саме повідомлення було збережено в окремі змінні, було реалізовано перевірку на те, чи існує дана бесіда в базі даних, чи це нова бесіда яку потрібно добавити у базу.

Завдяки ID користувачів, можна легко виявити чи існує така бесіда в базі. Далі відбувається перевірка, в якій логіка заключається в тому, якщо такої бесіди немає у базі даних, то її потрібно створити, де в масив співбесідників вносяться айді відправника та отримувача повідомлення.

Як тільки бесіда була створена, наступним ділом було створення масиву для зберігання усіх повідомлень. Щоб зберегти повідомлення потрібно внести дані про відправника, отримувача, та текст повідомлення. Якщо немає ніяких помилок, виконується наступна логіка в якій нове повідомлення методом масиву ‘push’ вноситься у масив повідомлень даної бесіди.

Наступним чином для реалізації обміну повідомленнями в реальному часі було використано бібліотеку Socket.io, це бібліотека, яка дозволяє реалізувати двосторонню, реального часу комунікацію між веб-клієнтом і сервером. Вона особливо корисна для додатків, де важлива швидка передача даних, таких як чати, ігри або спільні редактори.

Коли клієнт (наприклад, браузер) завантажує веб-сторінку, він може підключитися до сервера через Socket.IO. Для цього на стороні сервера зазвичай запускається HTTP сервер, який також обробляє WebSocket з'єднання.

Socket.IO використовує WebSocket для забезпечення постійного з'єднання між клієнтом і сервером. Це дозволяє обмінюватися даними без повторних запитів, що робить комунікацію швидшою. Якщо WebSocket не підтримується, Socket.IO автоматично переходить на інші протоколи (наприклад, Long Polling), щоб забезпечити роботу в більшості браузерів і мережевих умов.

Socket.IO працює на основі подій. Клієнт і сервер можуть "слухати" події і "відправляти" їх. Наприклад, коли користувач надсилає повідомлення, клієнт відправляє подію message, і сервер обробляє її, а потім розсилає повідомлення іншим клієнтам. На рисунку 2.10 показано підключення та налаштування сокета.

```
io.on('connection', (socket) => {
    console.log('user connected', socket.id)
    const userId = socket.handshake.query.userId
    if(userId !== 'undefined') userSocketMap[userId] = socket.id

    io.emit('getOnlineUsers', Object.keys(userSocketMap))
    socket.on('disconnect', () => {
        console.log('user disconnected', socket.id)
        delete userSocketMap[userId]
        io.emit('getOnlineUsers', Object.keys(userSocketMap))
    })
})
```

Рисунок 2.10 – Підключення Socket.io

За допомогою даної бібліотеки є можливість не тільки відправляти та отримувати повідомлення в реальному часі, але і реалізувати можливість показувати онлайн користувачів, тобто якщо один користувач знаходиться в мережі, і другий аналогічно з ним, на клієнтській частині відобразити це різними методами, наприклад як статус користувача, в мережі або був в мережі певний час тому.

Таким чином було реалізовано функціонал для відправки повідомлень в реальному часі (див. рис. 2.11).

```
const newMessage = new Message({
    senderId,
    receiverId,
    message
})
if(newMessage) {
    conversation.messages.push(newMessage._id)
}
await Promise.all([conversation.save(), newMessage.save()]);

const receiverSocketId = getReceiverSocketId(receiverId)
if(receiverSocketId) {
    io.to(receiverSocketId).emit('newMessage', newMessage)
}

res.status(201).json(newMessage)
```

Рисунок 2.11 – Логіка відправки повідомлення

Далі була реалізована ще одна функція, яка відповідає за повернення усіх повідомлень на клієнтську частину, це обов'язковий момент, коли чат відкривається, потрібно щоб було видно не тільки нові повідомлення, а і старі, що логічно.

При відкритті одного з чату, буде відкриватись вікно з чатом де якщо є повідомлення, вони будуть показані, а якщо повідомлень ще нема, буде текст з пропозицією відправити повідомлення користувачу.

Останнім етапом у розробці серверної частини програмного забезпечення, була реалізація функції посередника (middleware) – це функція, яка обробляє запити перед тим, як вони досягнуть основного обробника маршруту. Middleware може виконувати різноманітні задачі, такі як:

- Обробка запитів.
- Парсинг даних.
- Обробка помилок.
- Логування.

Middleware може бути глобальним (застосовується до всіх маршрутів) або специфічним для окремого маршруту. У Express.js їх можна підключати за допомогою методу .use(), а також передавати як параметри до маршрутів. Її можна реалізувати як функцію, яка спочатку перевіряє дані, а потім, у разі успішної перевірки, дозволяє виконання наступних функцій.

Такий підхід дозволяє зберігати логіку перевірки даних окремо від основних обробників, роблячи код чистішим і легшим для обслуговування. Спочатку у функції виконується збереження токену у змінну, його можна отримати за допомогою файлів cookie в яких збережено цей самий токен.

За допомогою метода verify який надає бібліотека jwt, було розшифровано токен, в якому був зашифрований ID користувача. Для розшифрування потрібно надати сам токен, та секретний ключ.

Після розшифрування відбувається перевірка чи існує такий токен, якщо існує, тоді відбувається пошук користувача за цим токеном. Якщо користувача не знайдено, сервер відповість помилкою з повідомленням що такого користувача не знайдено.

Коли усі перевірки успішно пройдуть, в параметри користувача буде присвоєно сам користувач який на даний момент авторизований. Після того буде виконуватись наступна функція, яка може знаходитись в любому місці. На рисунку 2.12 продемонстровано функцію middleware.

```
const protectRoute = async (req, res, next) => {
    try {
        const token = req.cookies.jwt

        if(!token) {
            return res.status(401).json({ error: 'Не авторизований' })
        }
        const decoded = jwt.verify(token, process.env.JWT_SECRET_KEY)

        if(!decoded) {
            return res.status(401).json({ error: 'Не авторизований' })
        }
        const user = await User.findById(decoded.userId).select('-password')

        if(!user) {
            return res.status(401).json({ error: 'Користувач не знайдений' })
        }
        req.user = user
        next()
    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}
```

Рисунок 2.12 – Функція middleware

Функція ‘next’ це потенційна функція, яка буде виконуватись після того, як усі умови та перевірки будуть пройдені у даній middleware.

Оскільки допоміжна функція для захисту приватних роутів була реалізована, наступним етапом був виклик у тих місцях де необхідно проводити перевірку користувача на аутентифікацію (див. рис. 2.13).

```
router.get('/:id', protectRoute, getMessages)
router.post('/send/:id', protectRoute, sendMessage)
```

Рисунок 2.13 – Виклик middleware

В такому випадку, при відправці повідомлення, спочатку буде виконуватись перевірка на те, чи дійсно користувач авторизований, якщо так, його дані зберігаються у параметри, після чого будуть використовуватись там де це потрібно і сама функція викличе всередині себе функцію next, які і є цими самими функціями після перевірки.

Отже якщо користувач пройде перевірку, буде виконана логіка що йде після неї, а саме відправка або завантаження повідомлень. Це стосується і інших роутів які також є захищеними завдяки middleware.

2.5 Висновок до другого розділу

В другому розділі кваліфікаційної роботи було обґрунтовано вибір описаних технологій. Було створено сервер за допомогою фреймворка express js, та продемонстровано підключення до бази даних.

Також реалізовано функціонал для обміну повідомленнями, реєстрації та авторизації користувача, разом з цим було створено допоміжну функцію яка відповідала за додатковий захист приватних запитів, та не дозволяла серверу продовжувати виконання запиту якщо користувач неавторизований.

3 ПРОЕКТУВАННЯ І РЕАЛІЗАЦІЯ КЛІЄНТСЬКОЇ ЧАСТИНИ ЗАСТОСУНКУ, ТА ІНТЕГРАЦІЯ З СЕРВЕРОМ

3.1 Ініціалізація та проектування клієнтської частини на Reactю.js

Реалізація клієнтської частини була створена на базі React JS з використанням Vite.

Vite – це сучасний інструмент для розробки фронтенду, який забезпечує швидке налаштування проекту та швидкий перезапуск завдяки технології "модульного завантаження".

Він особливо популярний серед розробників, які працюють з React. Vite дозволяє швидко створити проект з React завдяки простій команді в CLI та підтримує гаряче перезавантаження (HMR), що дозволяє бачити зміни в реальному часі без перезавантаження сторінки.

При збірці проекту Vite використовує Rollup для оптимізації фінального бандлу, що робить його малим і швидким. Також підтримує ES-модулі, TypeScript, JSX та інші сучасні технології без додаткових налаштувань. Vite має велику екосистему плагінів, що дозволяє легко інтегрувати різні функції, наприклад, для CSS, графіки та анімацій.

При збірці Vite стискає файли, щоб вони займали менше місця, а також забезпечує легше транспортування та автоматично ділить код на частини, які завантажуються за потребою.

При збірці проекту він створить папку dist, де зберігатимуться всі оптимізовані файли, готові до розгортання. Отже, Vite робить процес збору проектів простим і ефективним.

Для проектування додатку було створено файлову структуру проекту, де були розподілені компоненти у окремі файли, та деяка логіка винесена в окремі хуки для багаторазового використання.

Наступним кроком було завантаження усіх необхідних інструментів та бібліотек для комфортної та швидкої розробки клієнтської частини.

3.1.1 Налаштування сторонніх інструментів для розробки

Після того як було створено проект, та реалізована файлова структура, наступним етапом було підключення до проекту усіх необхідних інструментів. Завдяки пакетному менеджеру npm є змога завантажувати усе напряму в проект, користуючись внутрішнім терміналом, який надає середовище.

Для стилізації інтерфейсу було використано фреймворк Tailwind CSS – це утилітно-орієнтований CSS-фреймворк, який дозволяє швидко створювати адаптивні дизайни без написання великих обсягів стилів. Він пропонує набір класів, які можна комбінувати прямо в HTML, що полегшує процес стилізації елементів. Завдяки йому можна отримати клас для кожного стилю, наприклад, для кольору фону, відступів, шрифтів тощо, що дає змогу швидко і гнучко компонувати елементи.

Tailwind підтримує медіа-запити, що дозволяє легко створювати адаптивний дизайн, використовуючи просто класи, такі як md:bg-blue-500 для змінювання кольору фону на середніх екранах. Також дозволяє налаштовувати теми, кольори, шрифти та інші параметри через конфігураційний файл, що робить його гнучким для різних проектів (див. рис. 3.1).

```
/** @type {import('tailwindcss').Config} */
export default {
  content: [
    "./index.html",
    "./src/**/*.{js,ts,jsx,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [require('daisyui')],
}
```

Рисунок 3.1 – Конфігураційний файл tailwind

Існує безліч плагінів, які можна підключати для розширення функціональності, таких як анімації або формати зображень.

Одним з таких плагінів було додано бібліотеку daisyui, вона використовувалась для створення окремих компонентів.

DaisyUI – це компонентна бібліотека, яка побудована на базі Tailwind CSS. Вона надає готові до використання компоненти інтерфейсу, такі як кнопки, модальні вікна, карточки, форми і багато інших елементів. DaisyUI дозволяє швидко створювати стилізовані інтерфейси, використовуючи простий і зрозумілий синтаксис.

Також було налаштовано конфігураційний файл Vite, де вказано порт на якому буде запуск сервера, та порт для запуску клієнтської частини. Після того було додано бібліотеку для роботи з іконками, та react-router-dom для маршрутизації додатку.

3.1.2 Розробка авторизації і реєстрації користувача

Першим етапом розробки клієнтської частини була реалізація авторизації та реєстрації користувача. На рисунку 3.2 показано форму для реєстрації.

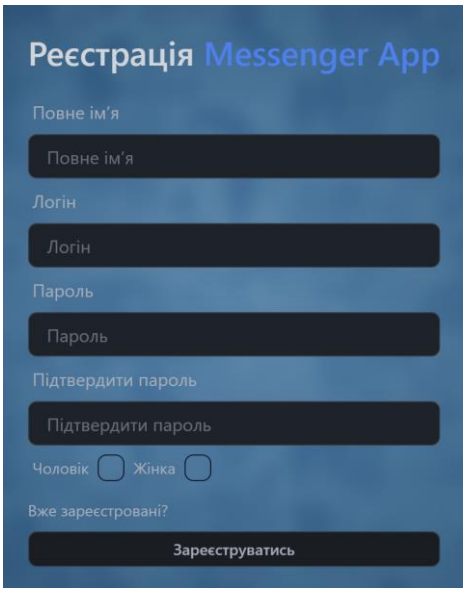

Рисунок 3.2 – Форма реєстрації користувача

Кожне поле є обов'язковим для заповнення, також як було сказано раніше присутнє поле для підтвердження паролю, такий метод потрібний для того, щоб користувач запам'ятав свій пароль, а не ввів його один раз з помилкою яку він

може не замітити і при наступній авторизації отримає повідомлення, що його пароль є невірним.

Також присутня валідація форми, користувач не зможе зареєструватись, якщо хоча б одне з полів не буде заповнено, у такому випадку користувач отримає повідомлення з помилкою. Якщо користувач введе паролі які не співпадають між собою, при спробі реєстрації появиться помилка як показано на рисунку 3.3.

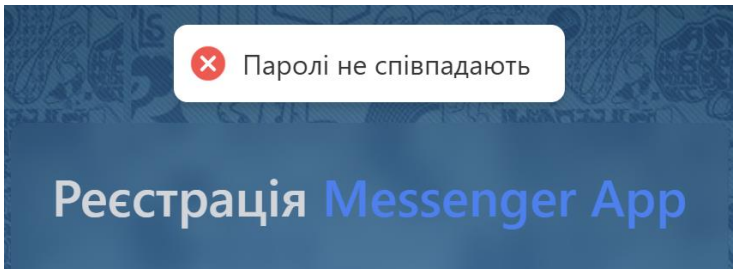

Рисунок 3.3 – Помилка при реєстрації

Коли користувач заповнить вірно усі поля, та підтвердить пароль, йому потрібно буде вибрати свою стать, для генерації аватару.

Для того щоб не навантажувати сервер додатковими запитами, валідація була реалізовано ще на клієнтській частині, при помилці функція реєстрації не відправляла запит на сервер, а зупиняла виконання ще на клієнтській стороні. На рисунку 3.4 показано функцію для валідації даних форми.

```
const handleInputError = ({ fullName, username, password, confirmPassword, gender }) => {
    if(!fullName || !username || !password || !confirmPassword || !gender) {
        toast.error('Будь ласка, заповніть усі поля')
        return false
    }

    if(password !== confirmPassword) {
        toast.error('Паролі не співпадають')
        return false
    }

    if(password.length < 6) {
        toast.error('Пароль повинен містити більше 6 символів')
        return false
    }

    return true
}
```

Рисунок 3.4 – Перевірка форми реєстрації

Таким чином коли користувач заповнив всі поля, його дані перевіряються спочатку на клієнтській частині та автоматично відправляються на сервер для обробки запиту, та при відсутності помилок користувача буде збережено у базу даних. Після реєстрації дані користувача зберігається не тільки у базу даних, а й у локальному сховищі (див. рис. 3.5).

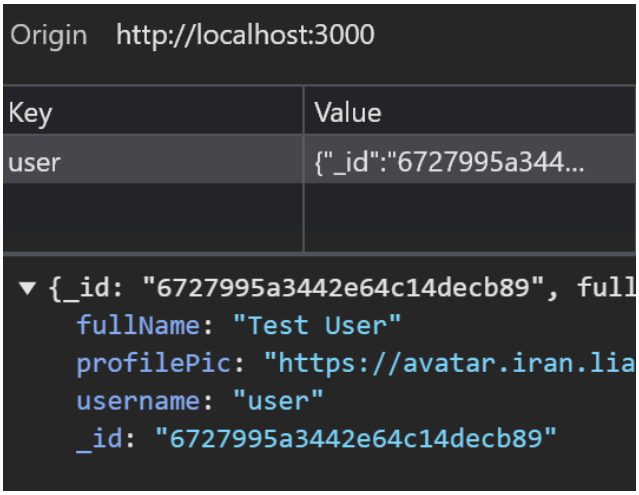

Рисунок 3.5 – Локальне сховище

Local Storage – це частина веб API, яка дозволяє зберігати дані у браузері користувача. Це простий механізм для зберігання пар або ключ-значення, що дозволяє зберігати інформацію навіть після закриття вкладки або перезавантаження сторінки. Дані з Local Storage доступні на тому ж домені та протоколі, що й веб-сайт, що дозволяє зберігати інформацію для всіх сторінок одного веб-додатку. Також може зберігати дані у форматі рядків (strings) до 5-10 МБ, в залежності від браузера.

Після реєстрації було реалізовано вхід у застосунок. Логіка входу та інтерфейс форми авторизації практично схожі до реєстрації за виключенням того, що при вході потрібно ввести лиш логін та пароль, також присутня валідація та перевірка даних.

Якщо користувач ввів усі дані без помилок, та база даних знайшла такого користувача, його буде переадресовано на головну сторінку застосунку. На рисунку 3.6 продемонстрована функція яка виконує логіку авторизації.

```
const login = async (username, password) => {
    const success = handleInputError({ username, password })
    if(!success) return

    setLoading(true)
    try {
        const response = await fetch('/api/auth/login', {
            method: 'POST',
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({ username, password })
        })
        const data = await response.json()
        if(data.error) {
            throw new Error(data.error)
        }
        localStorage.setItem('user', JSON.stringify(data))
        setAuthUser(data)
        toast.success(`Ви ввійшли як ${data?.fullName}`)
    } catch (error) {
        toast.error(error.message)
    } finally {
        setLoading(false)
    }
}
```

Рисунок 3.6 – Функція авторизації

Аналогічно з реєстрацією, після входу дані користувача будуть збережені у локальному сховищі. На рисунку видно змінну ‘authUser’ у яку передаються дані авторизованого користувача, для наступних дій у застосунку.

Було реалізовано функцію за допомогою хука який надає реакт – context. Context Hook у React – це спосіб управління глобальним станом у додатку без необхідності передавати дані через props на кожному рівні компонентів.

Context API дозволяє створювати контексти, які можна використовувати для зберігання даних, які повинні бути доступними багатьом компонентам. Хук зменшує необхідність передавати дані через пропси на кількох рівнях компонентів, та підходить для зберігання глобального стану, такого як дані користувача.

Таким чином колі потрібно передати дані з одного компонента в інший, не потрібно це робити через кожен компонент по сходинці вниз пропсами, а за допомогою хука useContext викликати його у потрібному компоненті, та використовувати дані які у ньому збережені.

Такий механізм схожий до стейт-менеджера Redux, однак між ними є певні відмінності у концепції, архітектурі та способі використання. Також, якщо говорити про невеликі проекти то зовнішні стейт-менеджери можна замінити

контекстом ректу, він призначений для управління невеликими або середніми обсягами глобального стану. Це хороший варіант для простих додатків або для передачі даних між компонентами. Тоді як Redux створений для управління складними глобальними станами у великих додатках.

У даній розробці обсяг даних невеликий, тому було прийнято рішення використовувати контекст який надає реакт, без додаткових завантажень та підключень (див. рис. 3.7).

```
import {createContext, useContext, useState} from "react";

export const AuthContext = createContext()

export const useAuthContext = () => {
    return useContext(AuthContext)
}

export const AuthContextProvider = ({ children }) => {
    const [authUser, setAuthUser] = useState(JSON.parse(localStorage.getItem('user')) || null)

    return <AuthContext.Provider value={{ authUser, setAuthUser }}>{ children }</AuthContext.Provider>
}
```

Рисунок 3.7 – Функція для керування глобальними даними

У функції збережено дані про користувача, які беруться з локального сховища, де вони потрапили після реєстрації або авторизації користувача, таким чином дані будуть доступні у любому місці проекту.

Також у локальному сховищі дані зберігаються у форматі JSON, для того щоб було зручно працювати з такими даними, потрібно їх перевести у звичайний JavaScript об’єкт, отже за допомогою метода parse() було перетворено формат рядка у звичайний об’єкт, після цього можна використовувати його для роботи з окремими ключами.

Останнім кроком у створенні аутентифікації було реалізовано функцію logout для виходу з додатку. Коли користувач натискає на кнопку виходу з застосунку, спочатку відправляється запит на сервер, після того як сервер поверне успішний статус код, з локального сховища будуть видалені дані про користувача, та в контекст буде передано замість об’єкту з даними, значення null. В такому випадку користувача буде переадресовано на сторінку авторизації завдяки функції яка відповідає за захист від неавторизованих користувачів.

3.2 Реалізація інтерфейсу та функціоналу для обміну повідомленнями в реальному часі

Наступним етапом розробки програмного забезпечення була реалізація інтерфейсу та функціоналу для обміну повідомленнями. Компонентна структура проекту була реалізована так, що в основному файлі знаходилось два компонента, це компонент бокової панелі, та сторінка з повідомленнями.

У компоненті SideBar так знаходяться розділені компоненти, це поле для пошуку, список усіх зареєстрованих користувачів, в самому низу знаходиться кнопка для виходу з акаунта та ім’я авторизованого на даний момент користувача. На рисунку 3.8 зображено інтерфейс головної сторінки.

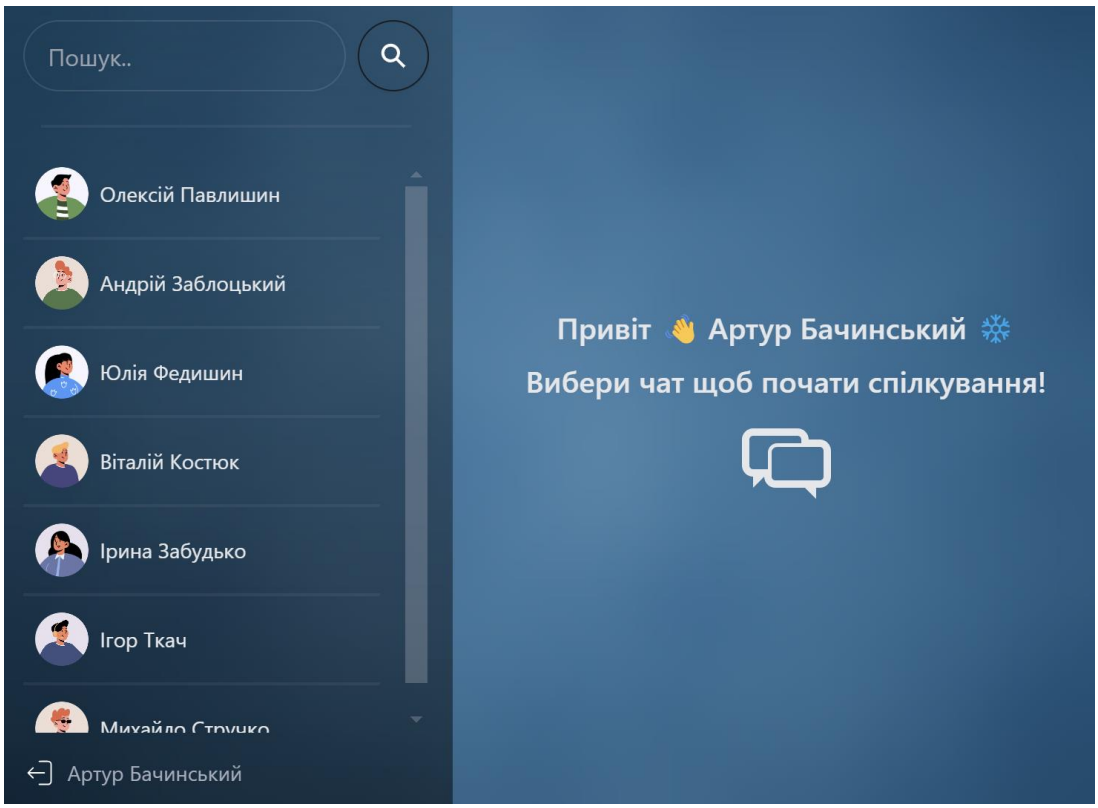

Рисунок 3.8 – Інтерфейс домашньої сторінки

При вході у застосунок користувач бачить блок з чатом у такому стилі тому, що чат не може буде вибраним коли користувач тільки увійшов, тому йому показано повідомлення з привітанням та його іменем який він вказував при реєстрації.

Для того щоб домашня сторінка переключались на сторінку з повідомленнями, потрібно вибрати чат з любим користувачем.

Для демонстрації повного функціоналу та інтерфейсу було зареєстровано декілька користувачів. Першим кроком було реалізовано запит на сервер для виведення усіх користувачів у список, далі створено функціонал для пошуку одного або декількох користувачів.

При пошуку конкретного користувача, його було виділено за допомогою заднього фону, та відкривався чат з ним. Також як можна замітити у кожного користувача є своя фотографія, яка була згенеровано на основі того, яку стать вказує користувач при реєстрації.

Функція яка виконує запит на сервер для отримання усіх користувачів, знаходиться в хуку useEffect.

UseEffect – це один із основних хуків у React, який дозволяє виконувати побічні ефекти в функціональних компонентах. Побічні ефекти можуть включати запити до API, маніпуляції з DOM, таймери та інше. Функція приймає два аргумента, перший аргумент — це функція, яка містить код для виконання, а другий аргумент – масив залежностей, від зміни яких залежить виконання цього коду. Якщо передати пустий масив, код виконається тільки один раз після первинного рендеру.

Якщо вказати змінні в масиві залежностей, код буде виконуватись щоразу, коли ці змінні змінюються. А якщо не вказати масив залежностей, useEffect буде виконуватись після кожного рендеру компонента.

Функція, яка повертається з useEffect, буде викликатись перед наступним виконанням ефекту або перед демонтажем компонента. Це дозволяє очищати ресурси (наприклад, скидання таймерів або скасування запитів).

Тому для реалізації запиту на сервер щоб вивести усіх користувачів було застосовано саме цей хук, тому як користувач який зареєструвався або просто увійшов у додаток, не повинен у пошуку шукати користувачів яких не виведено на бокову панель, а має зразу бачити список усіх інших користувачів.

Тому як було сказано, ця функція виконує логіку запиту як тільки відбувся рендер компоненту (див. рис. 3.9).

```
useEffect(() => {
    const getConversation = async () => {
        setLoading(true)
        try {
            const res = await fetch('/api/users')
            const data = await res.json()
            if(data.error) {
                throw new Error(data.error)
            }
            setConversations(data)
        } catch (error) {
            toast.error(error.message)
        } finally {
            setLoading(false)
        }
    }
    getConversation()
}, [])
return { loading, conversations }
}
```

Рисунок 3.9 – Функція useEffect

Після виведення користувачів на боковій панелі, було реалізовано логіку для пошуку за іменем. Для пошуку користувача використовувався метод масиву find, який проводить ітерацію по масиву та повертає його.

Такий метод дозволяє знайти перший елемент масиву, який відповідає умові, заданій у функції. Якщо елемент знайдено, метод повертає його, а якщо ні поверне undefined. Метод find зупиняє пошук, як тільки знаходить перший елемент, що відповідає умові.

У даному випадку пошук відбувався за іменем користувача (див. рис. 3.10).

```
const conversation = conversations.find((c) => c.fullName.toLowerCase().includes(search.toLowerCase()))
```

Рисунок 3.10 – Застосування методу find

У змінну ‘conversation’ вноситься результат виконання даного методу, далі метод застосовується до масиву користувачів який було отримано за допомогою вище описаної функції всередині хука useEffect.

Умова пошуку була така, що ім’я користувача який є у списку, повинно бути порівнянне з текстом який ввів користувач у поле для пошуку, у випадку якщо функція знайде користувача, то збереже результат у змінну.

Також було застосовано метод стрічки ‘toLowerCase’ для імені користувача до якого застосовується пошук, та текстом який ввів користувач у поле пошуку. Така логіка буде шукати користувачів не залежно від того з якої букви пише користувач, малої чи великої.

Метод toLowerCase повертає новий рядок, у якому всі великі літери перетворені на малі. Оригінальний рядок не змінюється, оскільки рядки в JavaScript є незмінними. Якщо рядок вже містить лише малі літери, метод не змінює його.

Якщо користувача було знайдено, то за умовою буде мінятись колір заднього фону, та відкриється чат із знайденим користувачем (див. рис. 3.11).

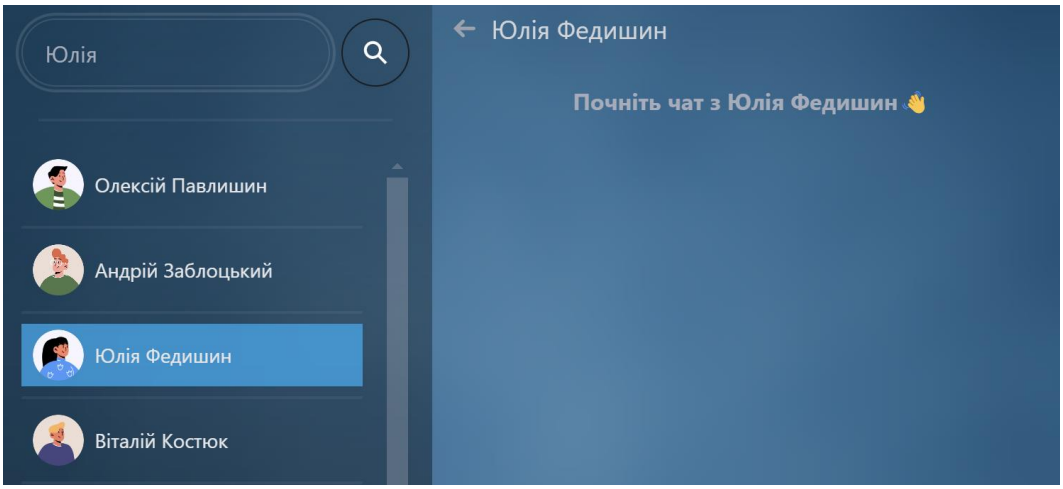

Рисунок 3.11 – Результат пошуку користувача

Після того як потрібного користувача було знайдено, автоматично відкривається чат з ним, та якщо немає ніяких повідомлень, замість пустого чату буде показано повідомлення з пропозицією розпочати чат.

При тому, якщо користувача не знайдено за пошуком, буде появлятись повідомлення про те, що такого користувача не знайдено. Такий підхід потрібний для того, щоб авторизований користувач не думав що сталась помилка на сервері, або пошук не відбувся, а побачив одразу повідомлення в якому буде сказано що такого користувача не знайдено.

Наступним кроком була розробка функціоналу та інтерфейсу чат-контейнера, де розмітка розділена на окремі компоненти, такі як верхня панель з

іменем користувача з яким відкритий чат, та іконка для повернення на головну сторінку.

Компонент який знаходиться посередині є блоком для самих повідомлень. В самому низу знаходиться поле для вводу повідомлення, з кнопкою для відправлення повідомлення.

Далі було реалізовано функціонал для відправки повідомлення в реальному часі за допомогою спеціального сокета.

У компоненті з повідомленнями реалізовано декілька функцій для відправки та отримання повідомлень з сповіщенням. Спочатку було викликано сокет який був занесений в контекст для незалежного виклику у любому місці проекту.

За допомогою сокета, коли сервер відправляє повідомлення, клієнт отримує його без необхідності перезавантажувати сторінку або виконувати повторні запити до сервера.

Це означає, що будь-які зміни на сервері миттєво відображаються на клієнтському боці. Коли клієнт або сервер більше не хоче підтримувати з'єднання, з'єднання можна закрити за допомогою методу disconnect().

Після отримання повідомлення сервер може відправити відповідь назад до клієнта або всім підключеним клієнтам.

Після того, як з'єднання встановлено, клієнт може надсилати повідомлення до сервера або між клієнтами через сервер, використовуючи події.

Socket.IO використовує події для організації обміну повідомленнями. Кожне повідомлення має свою подію, що дозволяє точно визначити, яке саме повідомлення обробляється в кожен момент часу.

Таким чином було реалізовано кастомний хук для отримання повідомлень в реальному часі, та оновлення масиву з повідомленнями. На рисунку 3.12 показано логіку виконання даної функції.

```
const useListenMessages = () => {
    const {socket} = useSocketContext()
    const {messages, setMessages} = useConversation()

    useEffect(() => {
        socket?.on('newMessage', (newMessage) => {
            newMessage.shouldShake = true
            const sound = new Audio(notificationSound)
            sound.play()
            setMessages([...messages, newMessage])
        })

        return () => socket?.off('newMessage')
    }, [socket, setMessages, messages])
}
```

Рисунок 3.12 – Функція для отримання повідомлень

Завдяки масиву залежностей, при змінах у масиві повідомлень, наприклад коли появилось нове повідомлення, або відбулись дії у сокеті, буде відбуватись перерендер сторінки, для оновлення масиву з повідомленнями.

Таким чином відбувається отримання повідомлень в реальному часі, без додаткових перезавантажень сторінки. Цей хук викликається у компоненті з повідомленнями.

Далі за допомогою функції setTimeout яка викликається всередині хука useEffect, було реалізовано функціонал для скролу чату вниз, до останніх повідомлень. Через те, що при заповненому чаті повідомлень, нові повідомлення з’являються внизу сторінки, і їх не видно, саме тому було застосовано такий метод, щоб при рендері сторінки, або при отриманні нового повідомлення, чат автоматично спускався вниз. На рисунку 3.13 зображено функцію яка реалізовує дану логіку.

```
useEffect(() => {
    setTimeout(() => {
        lastMessageRef.current?.scrollIntoView({ behavior: 'auto' })
    }, 0)
}, [messages])
```

Рисунок 3.13 – Функція для автоматичного скролу сторінки

Наступним етапом була розробка інтерфейсу самих повідомлень, вони повинні відрізнятись за стилем на основі того, відправлене воно чи отримане, також відправлене повідомлення повинно знаходитись по правій стороні контейнера, а отримане у лівій, як прийнято позиціонувати їх у всіх чат-месенджерах.

Збоку від повідомлення знаходиться аватар користувача, знизу розташовано час відправки повідомлення. Час було відформатовано за допомогою спеціальних методів для роботи з датами у JavaScript так, щоб була тільки година та хвилина. На рисунку 3.14 зображено інтерфейс повідомлень у чаті.

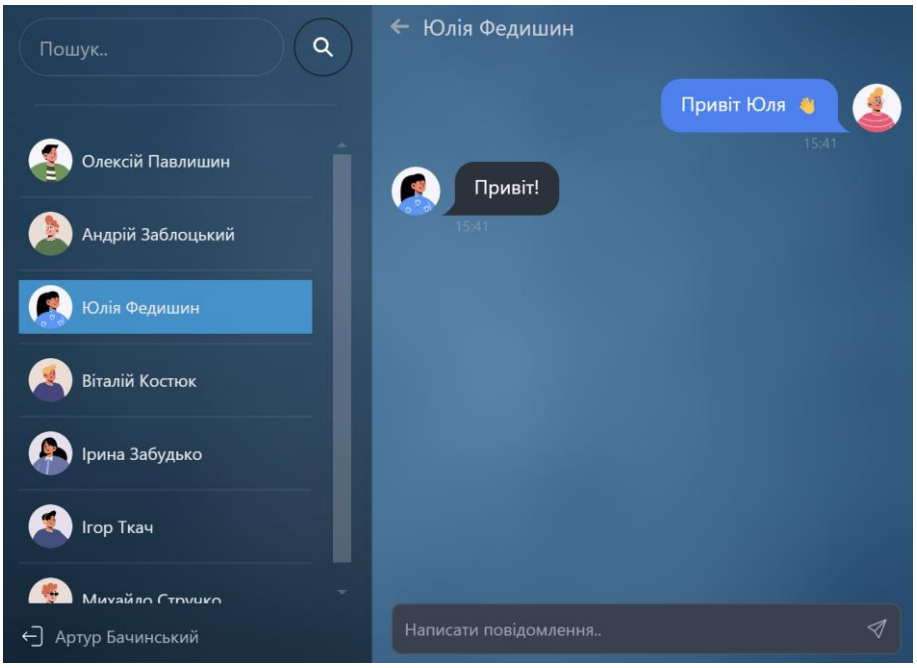

Рисунок 3.14 – Інтерфейс повідомлень

Для відправки повідомлення було реалізовано функцію, яка робить запит на сервер, вказуючи ID вибраного чату, та в тіло запиту передається саме повідомлення, яке за допомогою методу stringify переводить JavaScript об’єкт в рядок JSON, через те що у базі даних все зберігається в JSON форматі, а з клієнтської частини передається як звичайний JavaScript об’єкт.

Після того як повідомлення було успішно збережене в базу даних та повернено на клієнт, у масив додається нове повідомлення, яке автоматично

відображається на клієнтській частині без перезавантаження сторінки (див. рис. 3.15).

```
const sendMessage = async (message) => {
    setLoading(true)
    try {
        const res = await fetch(`/api/messages/send/${selectedConversation._id}`, {
            method: 'POST',
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify({message})
        })

        const data = await res.json()

        if(data.error) {
            throw new Error(data.error)
        }

        setMessages([...messages, data])
    } catch (error) {
        toast.error(error.message)
    } finally {
        setLoading(false)
    }
}
```

Рисунок 3.15 – Функція відправки повідомлення

Також було реалізовано компонент який відпрацьовує у той момент, коли йде завантаження сторінки з повідомленнями.

Коли чат завантажує старі повідомлення або історію чату, замість того, щоб показати порожнє місце або анімацію завантаження, можна показати попередній ескіз розташування повідомлень, з місцями для аватарок або заголовків.

Користувачі менше відчувають затримки в завантаженні, оскільки skeleton-екран відразу показує, де будуть повідомлення.

Якщо skeleton з'являється дуже швидко, а реальний контент заповнює місце впродовж наступних секунд, користувач вважає, що чат вже готовий, навіть якщо завантаження ще триває.

Якщо нові повідомлення надходять у чат, skeleton-екран може бути використаний для індикації того, що нове повідомлення завантажується. Це дає користувачеві зрозуміти, що нове повідомлення буде на його місці, поки воно не з'явиться. Такий метод і було обрано під час завантаження сторінки чату (див. рис. 3.16).

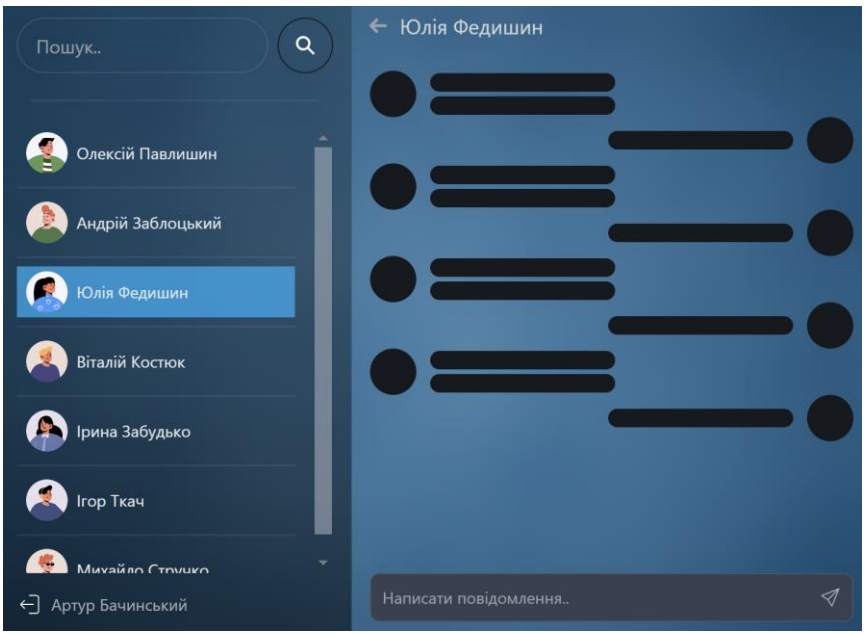

Рисунок 3.16 – Загрузка сторінки з чатом

Після реалізації функціоналу для відправки повідомлень, було розроблено видимість користувачів, які разом з іншими авторизовані в реальному часі. На рисунку 3.17 зображено реалізацію даного метода.

Рисунок 3.17 – Статус онлайн користувача

Якщо два користувача в один час знаходяться в мережі, вони будуть бачити онлайн статус один одного, такий підхід дозволяє реалізувати socket.io, завдяки якому також відбувається автоматичне оновлення даних, без зайвого перезавантаження сторінки.

Таким чином був реалізований інтерфейс, та функціонал для обміну повідомленнями в реальному часі.

Наступним та останнім етапом був деплой веб-застосунка на віддалений сервер, для можливості перегляду, та користування іншими користувачами у загальному доступі.

3.3 Розміщення веб-застосунку на віддаленому сервері

Додаток було розміщено на веб-сервісі Render, це платформа яка дозволяє розміщувати фул-стек додатки швидко і безкоштовно.

Render – це сучасна платформа для хостингу та деплою веб-застосунків і сервісів, яка забезпечує простоту використання, автоматизацію процесів та високу продуктивність. Render дає змогу легко розгортати проекти без необхідності управляти інфраструктурою або серверними налаштуваннями вручну. Підтримує автоматичний деплой з репозиторіїв GitHub або GitLab. Після кожного пушу в репозиторій зміни автоматично відображаються на продакшн сервері.

Це означає, що навіть після деплою можна внести якісь зміни у проекті, зробити пуш проекту у гітхаб, і сервер сам оновить застосунок.

Render дозволяє деплоїти різні типи застосунків, статичні сайти, серверні додатки, API, бази даних, фонові процеси тощо. Також платформа підтримує популярні мови програмування та фреймворки, такі як Node.js, Python, Go, Ruby, Java, PHP, та інші.

В залежності від навантаження на сайт чи додаток, Render автоматично масштабує ресурси, забезпечуючи оптимальну продуктивність. Мається на увазі не тільки горизонтальне масштабування, але й можливість настроїти кількість інстансів чи розмір ресурсів. Платформа також підтримує зовнішні бази даних і дозволяє легко управляти їх доступом і надає підтримку для різних інтеграцій, таких як Redis, Docker, Custom Domains, а також SSL сертифікати для забезпечення безпеки додатку.

Завдяки інтеграції з Git-репозиторіями та CI/CD, деплой стає майже повністю автоматизованим. Платформа автоматично масштабує ресурси в залежності від навантаження, що робить її ідеальним варіантом для проектів різних розмірів.

Наприклад, для деплою сайту на Next.js можна просто вказати репозиторій і Render згенерує та оптимізує статичні файли, після чого вони будуть доступні на домені.

Для початку було створено репозиторій на гітхабі, після чого можна було створювати новий проект на сервісі для розміщення на віддалений сервер (див. рис. 3.18).

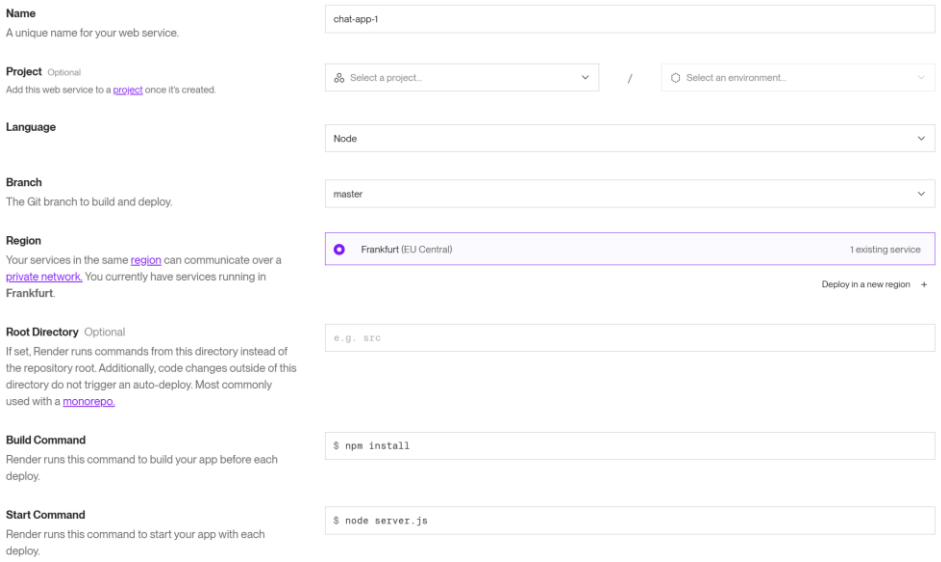

Рисунок 3.18 – Створення і налаштування віддаленого сервера

Спочатку потрібно було вибрати ім’я проекту, після чого мову на базі якої написана серверна частина. Після чого вибрати основну гілку гітхаб репозиторія, яку потрібно задеплоїти. Далі йде список регіонів в яких буде запускатись віддалений сервер, було вибрано центральний регіон в Європі.

Далі було додано змінні середовища, які знаходяться у файлі .env, який є прихованим і використовується для зберігання приватних даних, такий як назва та пароль до бази даних, секретний ключ токена, тощо (див. рис. 3.19).

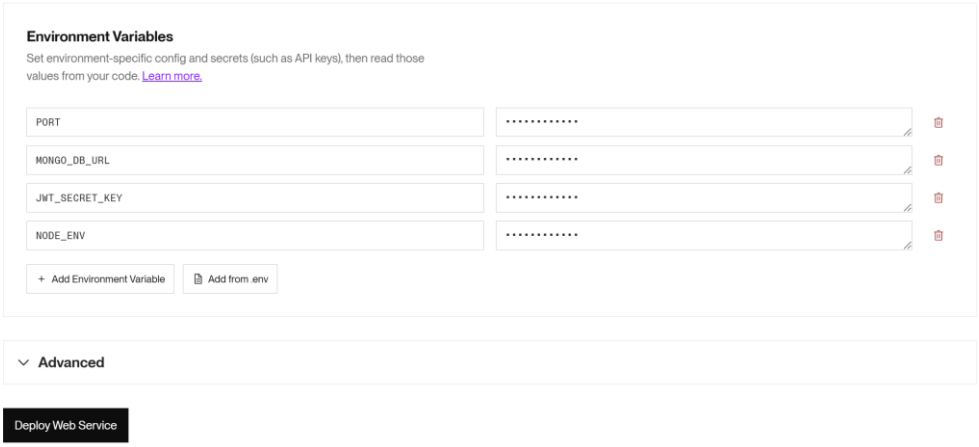

Рисунок 3.19 – Збереження змінних середовища

Коли було налаштовано сервер та додано усі приватні змінні, можна було розпочинати деплой проекту на сервер, після кнопки підтвердження, сервіс переходить на нову сторінку де показує процес деплою, де можна побачити усі логи (див. рис. 3.20).

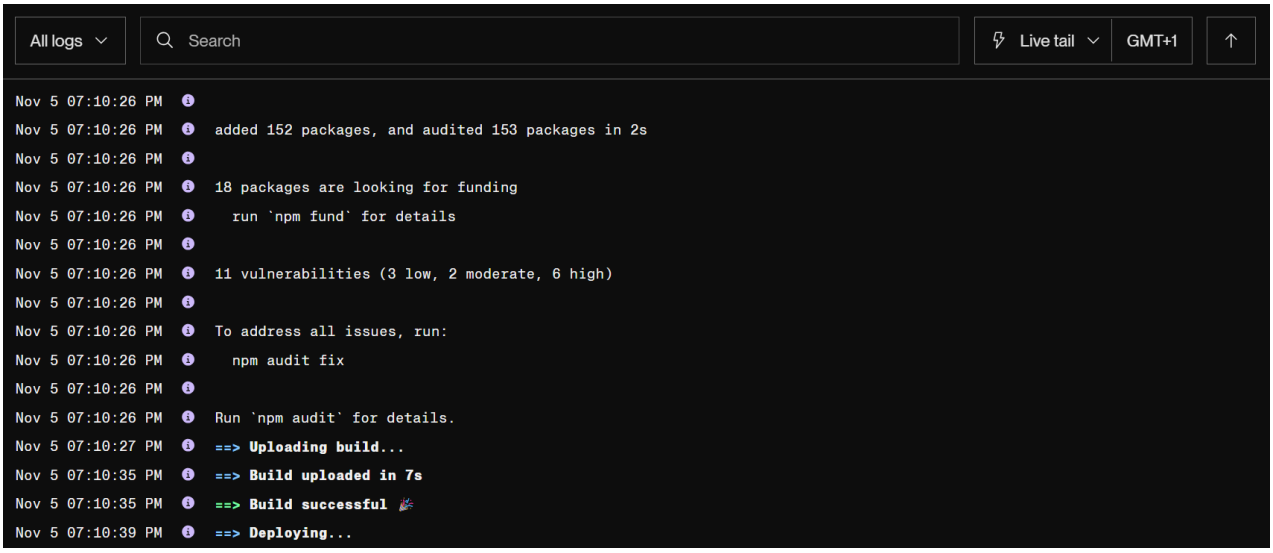

Рисунок 3.20 – Деплой проекту

В логах показана вся інофрмація та процес завантаження проекту на сервер. Можна побачити скільки пакетів було додано, та процес білдингу.

Після успішного деплою, сервіс створює та надає посилання на готову веб-сторінку, яку можна повноцінно використовувати.

3.4 Висновок до третього розділу

У третьому розділі кваліфікаційної роботи було налаштовано інструменти для створення клієнтської частини.

Було створено логіку авторизації і реєстрації користувачів на стороні клієнта, та функціонал для обміну повідомленнями в реальному часі.

Описано можливі методи для реалізації клієнтської частини. Також було розглянуто метод для розміщення застосунку на віддаленому сервері.

4 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

4.1 Питання щодо охорони праці

Інформаційні технології – це галузь, яка швидко розвивається і змінюється, а також одна з найбільших за масштабами впливу на сучасне суспільство. Проте, незважаючи на те, що більшість процесів у цій галузі відбуваються віртуально, питання охорони праці та безпеки співробітників залишаються важливими. Оскільки розробка програмного забезпечення, зокрема чат-систем у реальному часі, вимагає високої концентрації уваги, роботи з комп'ютерними технологіями та участі в командній роботі, охорона праці в IT-сфері має багато аспектів.

Незважаючи на відсутність фізичних ризиків, типових для деяких інших галузей, робота в IT-секторі не є безпечною для здоров’я працівників. Основні фізичні проблеми, з якими стикаються розробники програмного забезпечення, включають: Охорона зору, робота за комп'ютером протягом тривалого часу може викликати різноманітні проблеми із зором, такі як втома очей, сухість очей або навіть серйозніші порушення зору, зокрема синдром комп’ютерного зору (CVS).

Для запобігання цим проблемам важливо дотримуватись режиму відпочинку (правило 20-20-20, тобто кожні 20 хвилин робити перерву на 20 секунд і дивитися на об'єкт, що знаходиться на відстані 20 футів). Також рекомендується використовувати екрани з антибліковим покриттям і налаштовувати контрастність та яскравість екрану відповідно до умов освітлення в робочому просторі.

Проблеми з поставою: Постійне сидіння перед комп'ютером може спричинити проблеми з поставою, болі в спині та шиї. Це особливо актуально для розробників програмного забезпечення, які працюють у сидячому положенні понад 6 годин на день.

Важливо використовувати ергономічні стільці, налаштовувати робоче місце так, щоб екран перебував на рівні очей, а клавіатура — на зручній висоті для запобігання напруженню в руках і спині.

У розробці чат-систем в реальному часі програмістам потрібно активно працювати з кодом, вирішувати складні задачі, співпрацювати з іншими членами команди та підтримувати високий рівень комунікації. Все це може призводити до психологічного навантаження. Часто у процесі розробки програмного забезпечення можуть виникати ситуації, коли терміни здачі проекту стають все більш обмеженими. Це може викликати стрес та вигорання у працівників.

Для зменшення стресу рекомендується використовувати методи тайм-менеджменту, організовувати робочі процеси в межах реалістичних термінів і створювати комфортні умови для командної роботи. Важливу роль відіграє підтримка і допомога з боку керівництва та колег. Психоемоційне вигорання: Робота в IT-сфері може бути виснажливою через постійну необхідність адаптуватися до нових технологій і змінюваних вимог. Вигорання може статися, коли програмісти працюють у надмірно інтенсивному режимі без належного відпочинку.

Важливо забезпечити гнучкий графік роботи, давати можливість для професійного розвитку та підтримувати здоровий баланс між роботою та особистим життям. Регулярні перерви і фізична активність допомагають зменшити напругу.

Розробка чат-систем в реальному часі зазвичай передбачає активну командну роботу. Це включає програмістів, тестувальників, дизайнерів інтерфейсів, а також інших фахівців. Така діяльність пов’язана з постійною комунікацією та координацією.

Взаємодія з іншими членами команди може бути як джерелом мотивації, так і стресу. Правильне управління проектами, чітке визначення завдань і вміння ефективно вирішувати конфлікти є важливими аспектами забезпечення здорового клімату в команді. Часто IT-фахівці повинні брати участь у численних нарадах, як онлайн, так і офлайн. Перевантаження наради можуть спричиняти втому та дратівливість. Оптимізація комунікаційних процесів, використання відповідних інструментів для спілкування (наприклад, Slack, Microsoft Teams) дозволяють знизити цей стрес.

Розробка чат-систем, особливо тих, що функціонують в реальному часі, має свої особливості, які безпосередньо впливають на умови праці. Програмісти повинні працювати з великими обсягами даних, взаємодіяти з різними протоколами передачі інформації і тестувати взаємодію між користувачами в реальному часі.

Важливим аспектом є забезпечення безпеки даних, захист від хакерських атак, безпечне зберігання даних користувачів. Це вимагає постійного оновлення знань, уважності до змін у галузі безпеки та застосування найкращих практик програмування.

Також важливим є оптимізація програмного забезпечення для забезпечення високої продуктивності при одночасній роботі великої кількості користувачів. Для цього розробники повинні мати доступ до сучасних інструментів і технологій, що дозволяють зменшити ризики програмних помилок, збоїв і перегрузок серверів.

Охорона праці в ІТ-сфері є важливим аспектом, який стосується не тільки фізичного, а й психологічного стану працівників. Профілактика стресу, підтримка здоров'я співробітників та створення комфортних умов для ефективної роботи є основними чинниками успіху в розробці програмного забезпечення. У випадку розробки чат-систем у реальному часі особлива увага повинна приділятися як технічній безпеці, так і здоров’ю працівників, що дозволяє забезпечити не лише ефективну, але й безпечну розробку програмного продукту.

4.2 Питання щодо безпеки в надзвичайних ситуаціях

Надзвичайні ситуації – це події, що призводять до порушення нормальної життєдіяльності людей, забруднення навколишнього середовища, значних економічних і соціальних втрат. У світі існує велика кількість різноманітних надзвичайних ситуацій, що можуть виникнути внаслідок техногенних катастроф, природних катастроф, радіаційних, хімічних або біологічних інцидентів, а також військових конфліктів.

Надзвичайні ситуації можна класифікувати за різними ознаками:

– За характером: природні (землетруси, повені, урагани) та техногенні (аварії на виробництві, техногенні катастрофи).

– За масштабами: локальні, регіональні, національні та глобальні.

– За типами загрози: хімічні, радіаційні, біологічні, екологічні.

Кожен з типів НС має специфічні методи захисту, що залежать від виду загрози. Найбільш поширеними НС є техногенні катастрофи, що включають аварії на атомних електростанціях, хімічні вибухи, аварії на транспорті, які спричиняють значні загрози для здоров’я та життя людей.

Захист населення від наслідків надзвичайних ситуацій вимагає комплексного підходу, що включає інформаційне забезпечення. Важливим аспектом є своєчасне інформування населення про загрозу, використовуючи засоби масової інформації, спеціальні попереджувальні сигнали, сирени, телевізійні та радіопрограми.

В разі виникнення загрози на об'єкті необхідно організувати евакуацію людей в безпечні місця, забезпечити укриття в укриттях, що мають відповідну захист від радіаційних, хімічних або біологічних загроз.

У випадку НС необхідно організувати роботу медичних пунктів, забезпечити постраждалих швидкою допомогою, а також вжити заходів для запобігання епідеміям і хворобам. Важливо організувати роботу психологів та волонтерів, які допомагають населенню подолати стресові ситуації, що виникають під час катастроф.

Для ефективного захисту населення необхідно проводити постійну підготовку як для населення, так і для спеціалізованих служб: Проводяться тренування, інструктажі та курси для населення, щоб люди знали, як діяти в разі НС (як евакуюватися, куди йти, як захистити себе від певних загроз).

Спеціалізовані служби повинні мати навички надання першої медичної допомоги, організації евакуації, управління панікою та збереження безпеки громадян. Окремі структури повинні мати спеціальні засоби для надання першої допомоги, санітарної обробки постраждалих та лікування.

Радіаційний захист є важливою складовою охорони праці на підприємствах, що мають справу з радіоактивними матеріалами. Працівники, що виконують роботи в умовах радіаційного забруднення або потенційно небезпечних для здоров’я умовах, підлягають особливому захисту від радіаційного випромінювання.

Захист від радіації включає низку технологічних та організаційних заходів, що спрямовані на запобігання впливу радіації на організм людини. До основних принципів радіаційного захисту відносяться:

– Зменшення часу перебування на забруднених територіях.
– Збільшення відстані від джерела радіації.
– Захист за допомогою бар'єрів.

Працівники, які виконують роботи в умовах можливого радіаційного впливу, повинні проходити регулярне медичне обстеження та інструктажі щодо дій у разі надзвичайних ситуацій. Окрім того, необхідно дотримуватись усіх вимог. Робітники повинні проходити медичні огляди, щоб визначити наявність або відсутність впливу радіації на їхній організм. Окремо проводяться обстеження на наявність радіаційних захворювань. На виробничих об’єктах має проводитись моніторинг рівня радіації, а також оцінка потенційних ризиків для працівників.

Для роботи в зонах з підвищеною радіаційною небезпекою працівники повинні мати відповідний одяг, який забезпечує бар'єр від радіації, а також прилади для контролю за рівнем радіації. На виробничих об’єктах, що працюють з радіоактивними матеріалами, повинні бути організовані спеціальні режими захисту.

Визначення максимально дозволених доз для працівників, контроль за їх виконанням. Використання технологічного обладнання, яке обмежує або усуває викиди радіації, а також має системи моніторингу.

Доступ на об'єкти, де присутні радіоактивні матеріали, повинний бути обмежений для сторонніх осіб. На таких об’єктах повинна бути розроблена чітка система допуску для працівників та проведення навчань.

4.3 Висновок до четвертого розділу

У четвертому розділі кваліфікаційнох роботи розглянуто основні аспекти охорони праці в ІТ-індустрії, зокрема ризики, пов'язані з роботою в офісах та при розробці програмного забезпечення, таких як чат-системи в реальному часі.

Особлива увага приділена фізичним та психологічним факторам, що впливають на здоров'я працівників: тривала робота за комп’ютером, стресові навантаження та вигорання.

Проаналізовано заходи для запобігання цим проблемам, такі як організація комфортних умов праці, використання сучасних технологій для захисту даних та підтримка психологічного здоров’я співробітників.

Подано рекомендації щодо створення безпечних умов для працівників в ІТ-сфері, акцентуючи на важливості регулярних медичних оглядів, психосоціальної підтримки та технічних заходів для забезпечення кібербезпеки.

Розглянуто питання захисту населення в разі виникнення надзвичайних ситуацій (НС), зокрема організацію евакуації, укриттів та медичної допомоги, а також заходи щодо психологічної підтримки постраждалих.

Проаналізовано основні типи НС, такі як природні, техногенні, хімічні, радіаційні та біологічні загрози, та визначено ключові елементи захисту, що включають своєчасне інформування громадян і належну підготовку спеціалізованих служб.

Подано рекомендації щодо радіаційного захисту працівників та службовців, що працюють у зонах з підвищеною радіаційною небезпекою. Це включає використання засобів індивідуального захисту, проведення медичних оглядів та дотримання стандартів безпеки на виробничих об'єктах.

ВИСНОВКИ

В першому розділі кваліфікаційної роботи було всебічно розглянуто сучасні середовища та методи веб-розробки, що охоплює редактори коду, серверні технології та бази даних. Було проаналізовано важливість вибору відповідного редактора коду.

Серед таких популярних рішень, як Visual Studio Code і WebStorm, було відзначено їх функціональність, підтримку плагінів та інтеграцію з системами контролю версій, що дозволяє підвищувати продуктивність та зосереджуватись на створенні якісного коду.

Проаналізовано методи за засоби для написання серверної частини програмного забезпечення, включаючи основні мови програмування, такі як JavaScript, Python, Java, PHP, а також популярні фреймворки. Було підкреслено, що ці технології забезпечують гнучкість та масштабованість, необхідні для розробки сучасних веб-додатків, а також описано переваги використання різних архітектурних підходів, таких як REST, GraphQL і мікросервіси.

В другому розділі кваліфікаційної роботи було обґрунтовано вибір описаних технологій. Було створено сервер за допомогою фреймворка express js, та продемонстровано підключення до бази даних.

Також реалізовано функціонал для обміну повідомленнями, реєстрації та авторизації користувача, разом з цим було створено допоміжну функцію яка відповідала за додатковий захист приватних запитів, та не дозволяла серверу продовжувати виконання запиту якщо користувач неавторизований.

В третьому розділі кваліфікаційної роботи було створено логіку авторизації і реєстрації користувачів на стороні клієнта, та функціонал для обміну повідомленнями в реальному часі.

Було налаштовано інструменти для створення клієнтської частини. Описано можливі методи для реалізації клієнтської частини. Також було розглянуто метод для розміщення застосунку на віддаленому сервері.

ПЕРЕЛІК ДЖЕРЕЛ

1 Bodnarchuk, I., Duda, O., Kharchenko, A., Kunanets, N., Matsiuk, O., & Pasichnyk, V. (2020). Choice Method of Analytical Platform for Smart City (No. 4374). EasyChair.

2 Kharchenko, A., Halay, I., Zagorodna, N., & Bodnarchuk, I. (2015, September). Trade-off optimal decision of the problem of software system architecture choice. In 2015 Xth International Scientific and Technical Conference" Computer Sciences and Information Technologies"(CSIT) (pp. 198-205). IEEE.

3 Kharchenko, A., Bodnarchuk, I., Raichev, I., & Morar, Y. (2014). The Method of Software Architecture Design Accounting the Quality Requirements Change.

4 Моделі потоків даних (DFD-моделі): призначення, місце застосування в системному аналізі, правила побудови, приклади [Електронний ресурс] – Режим доступу: http://victoria.lviv.ua/html/wp/index.html. (19.11.2024)

5 Веб-технології та веб-дизайн [Електронний ресурс] – Режим доступу: http://victoria.lviv.ua/html/wp/index.html. (19.11.2024)

6 Документація щодо роботи з TypeScript [Електронний ресурс] – доступу до ресурсу: docs/handbook/typescript-in-5-minutes.html.

7 React Documentation [Online Resource] — Access: https://reactjs.org/docs/getting-started.html. (19.11.2024)

8 Node.js Documentation [Online Resource] — Access: https://nodejs.org/en/docs/ Getting Started with Node.js [Online Resource] — Access: https://www.digitalocean.com/community/tutorials/first-nodejs-application. (19.11.2024)

9 Розробка програми з використанням Socket.IO [Електронний ресурс] Режим доступу ресурсу: https://www.digitalocean.com/community/tutorials/socket-io-chat-application. (19.11.2024)

10 Docker compose up | Docker documentation. URL: https://docs.docker.com/engine/reference/commandline/compose_up. (19.11.2024)

11 Building Token-based Authentication in Web Applications [Online Resource] — Access: https://www.smashingmagazine.com/2021/01/authentication-token-based-architecture/ (19.11.2024)

12 REST API with JWT Authentication [Online Resource] — Access: https://jwt.io/introduction/ (19.11.2024)

13 Using JSON Web Tokens for Authentication in Node.js [Online Resource] — Access: https://www.digitalocean.com/community/tutorials. (19.11.2024)

14 Express Documentation [Online Resource] — Access: https://expressjs.com/en/starter/installing.html. (19.11.2024)

ДОДАТКИ

Додаток А

Тези доповіді на конференції

Додаток Б

Програмний код

СЕРВЕР

```
import express from "express";
import dotenv from "dotenv";
import connectToMongoDB from "./db/connectToMongoDB.js";
import cookieParser from 'cookie-parser'
import authRoutes from "./routes/auth.routes.js";
import messageRoutes from "./routes/message.routes.js";
import userRoutes from "./routes/user.routes.js";
import {app, server} from "./socket/socket.js";
import * as path from "path";

const PORT = process.env.PORT || 5000

const __dirname = path.resolve()

dotenv.config()
app.use(express.json())
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser())

app.use('/api/auth', authRoutes)
app.use('/api/messages', messageRoutes)
app.use('/api/users', userRoutes)

app.use(express.static(path.join(__dirname, '/frontend/dist')))
app.get('*', (req, res) => {
    res.sendFile(path.join(__dirname, 'frontend', 'dist', 'index.html'))
})

server.listen(PORT, () => connectToMongoDB())
```

АВТОРИЗАЦІЯ

```
import User from "../models/user.model.js";
import bcrypt from 'bcryptjs'
import generateTokenAndSetCookie from "../utils/generateToken.js";

export const signup = async (req, res) => {
    try {
        const { fullName, username, password, confirmPassword, gender } = req.body

        if(!fullName || !username || !password || !confirmPassword || !gender) {
            return res.status(400).json({ error: 'All fields are required' })
        }

        if(password !== confirmPassword) {
            return res.status(400).json({ error: 'Passwords don’t match' })
        }

        const user = await User.findOne({ username })

        if(user) {
            return res.status(400).json({ error: 'Username already exists' })
        }

        const salt = await bcrypt.genSalt(10)
        const hashPassword = await bcrypt.hash(password, salt)

        const                          boyProfilePic                                =
`https://avatar.iran.liara.run/public/boy?username=${username}`
```

```
        const                          girlProfilePic                                =
`https://avatar.iran.liara.run/public/girl?username=${username}`

        const newUser = new User({
            fullName,
            username,
            password: hashPassword,
            gender,
            profilePic: gender === 'male' ? boyProfilePic : girlProfilePic
        })

        if(newUser) {
            generateTokenAndSetCookie(newUser._id, res)
            await newUser.save()
            res.status(201).json({
                _id: newUser._id,
                fullName: newUser.fullName,
                username: newUser.username,
                profilePic: newUser.profilePic
            })
        } else {
            res.status(400).json({ error: 'Invalid user data' })
        }
    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}

export const login = async (req, res) => {
    try {
        const { username, password } = req.body

        if(!username || !password) {
            return res.status(400).json({ error: 'All fields are required' })
        }

        const user = await User.findOne({ username })

        if(!user) {
            return res.status(400).json({ error: 'Invalid login or password' })
        }

        const isPasswordCorrect = await bcrypt.compare(password, user.password)

        if(!isPasswordCorrect) {
            return res.status(400).json({ error: 'Invalid login or password' })
        }
        generateTokenAndSetCookie(user._id, res)
        res.status(201).json(user)
    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}

export const logout = (req, res) => {
    try {
        res.cookie('jwt', '', { maxAge: 0 })
        res.status(200).json({ message: 'Logged out successfully' })
    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}
```

Контролер роботи з коритсувачами

```
import User from "../models/user.model.js";

export const getUsersForSidebar = async (req, res) => {
    try {
        const loggedInUserId = req.user._id

        const filteredUsers = await User.find({ _id: { $ne: loggedInUserId }
}).select('-password')

        res.status(200).json(filteredUsers)

    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}

export const getCurrentUser = async (req, res) => {
    try {
        const userId = req.user._id
        const currentUser = await User.findById(userId)

        res.status(200).json(currentUser)

    } catch (error) {
        console.log(error.message)
        res.status(500).json({ error: error.message })
    }
}
```