Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _інформаційних радіотехнологій та технічного захисту інформації_
(повна назва)

Кафедра ___медіаінженерії та інформаційних радіоелектронних систем_______
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти ___________другий (магістерський)______________________
(позначення документа)

Розробка веб-додатку SPA з використанням фреймворку React
(тема)

Виконав:
студент _2_ курсу, групи ____МІм-22-1___
_____________Здор О.В.________________
(прізвище, ініціали)

Спеціальність ____________________________
__172 Телекомунікації та радіотехніка_____
(код і повна назва спеціальності)
Тип програми ___освітньо-професійна____
(освітньо-професійна або освітньо-наукова)
Освітня програма _____Медіаінженерія___
(повна назва освітньої програми)

Керівник ______доц. Цехмістро Р.І.______
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри ________________ _____Володимир КАРТАШОВ____
(підпис) (прізвище, ініціали)

2023 р.

Харківський національний університет радіоелектроніки

Факультет інформаційних радіотехнологій та технічного захисту інформації
Кафедра медіаінженерії та інформаційних радіоелектронних систем
Рівень вищої освіти другий (магістерський)
Спеціальність 172 Телекомунікації та радіотехніка
(код і повна назва)
Тип програми освітньо-професійна
Освітня програма "Медіаінженерія"
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри ______________
(підпис)
«____»____________ 20 ___ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Студентові Здору Олегу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка веб-додатку SPA з використанням фреймворку React.

затверджена наказом університету від 20 10 2023 р. № 1224 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 10.01.2024 р.
3. Вихідні дані до роботи
1. Провести аналіз методів та технологій які будуть використовуватися у розробці проекту.
2. Спроектувати інтерфейс користувача додатку, побудувати логічну структуру його роботи.
3. Розробити front-end частину додатку та його анімації.
4. Розробити серверну логіку та налаштувати роботу хмарних сервісів та баз даних.
4. Перелік питань, що потрібно опрацювати в роботі
ВСТУП
1. Аналітичний огляд методів та технологій які будуть використовуватися у розробці.
2. Проектування інтерфейсу користувача додатку та логічної структури його роботи.
3. Розроблення front-end частини додатку та його анімацій.
4. Розроблення серверної логіки, налаштування роботи хмарних сервісів та баз даних.
ВИСНОВКИ
ПЕРЕЛІК ПОСИЛАНЬ
ДОДАТКИ

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри):

1. Постановка задачі (1 слайд).
2. Порівняння SPA та звичайного веб-сайту (1 слайд).
3. Фреймворк React (1 слайд).
4. Переваги та недоліки React (1 слайд).
5. Компонентний підхід розробки (1 слайд).
6. Переваги NoSQL над SQL базами даних (1 слайд).
7. Платформа Firebase (1 слайд).
8. Висновки (1 слайд).

6. Консультанти розділів роботи (п.6 включається до завдання за наявності консультантів згідно з наказом, зазначеним у п.1)

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Терміни виконання етапів роботи	Примітка
1	Аналітичний огляд літературних джерел	01.09.23–27.09.23	
2	Огляд документації, проектування SPA додатку	28.09.23–11.10.23	
3	Вибір інструментів. Розробка SPA додатку та підключення бази даних	12.10.23–10.11.23	
4	Тестування додатку, розроб	11.11.23–13.11.23	
5	Графічна частина проекту	13.11.23–03.12.23	
6	Перевірка керівником проекту	06.12.23–07.12.23	
7	Перевірка нормоконтролем	02.01.24–04.01.24	
8	Перевірка на академічний плагіат	04.12.23–05.01.24	
9	Перевірка завідувачем кафедри, рецензування	06.01.24–09.01.24	

Дата видачі завдання 20.10.2023 р.

Студент ____________ Олег ЗДОР
(підпис)

Керівник роботи ____________ Роман ЦЕХМІСТРО
(підпис)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи має: 69 сторінки, 33

рисунки, 2 додатки, 10 джерел.

ОДНОСТОРІНКОВИЙ ДОДАТОК, REACT, ФРЕЙМВОРК, БАЗА ДАНИХ, FIREBASE, JAVASCRIPT, ІНТЕРФЕЙС КОРИСТУВАЧА.

Об'єкт дослідження – Процеси створення SPA додатків з використанням новітніх JavaScript фреймворків та back-end платформ

Предмет дослідження – JavaScript фреймворк React та Firebase, програмні засоби для створення веб додатку та налаштування його для подальшої взаємодії між користувачами.

Мета кваліфікаційної роботи – створити односторінковий додаток (SPA) додаток використовуючі новий підхід розробки який базується на NoSQL базах даних. У додаток інтегрувати функціонал створення унікального аккаунту користувача, авторизацію користувача. створення чату між двома користувачами для обміну повідомленнями у реальному часі. Розробити функціонал, що забезпечить взаємодію з моделлю у додатку, впроваджуючи анімації та досягаючи готового вигляду продукту, який можна легко інтегрувати в інші додатки.

Методи дослідження – Аналіз теоретичних аспектів, статистична обробка даних, вивчення документації, застосування практичних навичок для створення односторінкового додатку (SPA) та подальша інтеграція функціоналу в інтерфейс користувача, а також налаштування бази даних.

У процесі роботи було висвітлено етапи створення односторінкового додатку (SPA) за допомогою сучасних веб-технологій, використання NoSQL баз даних, фреймворків та бібліотек. Це дозволило розробити функціонал додатку, який також може бути легко розширений у майбутньому.

ABSTRACT

Explanatory note of the qualification work has: 69 pages., 33 drawing, 2

appendix, 10 sources.

SINGLE PAGE APPLICATION, REACT, FRAMEWORK, REALTIME DATABASE, FIREBASE, FULLSTACK, JAVASCRIPT, USER INTERFACE.

Object of research is the processes of creating SPA applications using the latest JavaScript frameworks and back-end platforms

Subject of research is JavaScript framework React and Firebase, software tools for creating a web application and customizing it for further interaction between users.

The purpose of the work is to create a single-page application (SPA) application using a new development approach based on NoSQL databases. To integrate the functionality of creating a unique user account, user authorization, creating a chat between two users for real-time messaging into the application. Develop functionality that will provide interaction with the model in the application, implementing animations and achieving a finished product that can be easily integrated into other applications.

Research methods is analysis of theoretical aspects, statistical data processing, study of documentation, application of practical skills to create a single-page application (SPA) and further integration of functionality into the user interface, as well as setting up a database.

In the course of the work, the stages of creating a single-page application (SPA) with the help of modern web technologies, the use of NoSQL databases, frameworks and libraries were highlighted. This allowed us to develop the application's functionality, which can also be easily expanded in the future.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.................... 8
ВСТУП ....**Ошибка! Закладка не определена.**
1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ**Ошибка! Закладка не определена.**
1.1 Фреймворк React ....**Ошибка! Закладка не определена.**
1.2 Компонентна розробка SPA додатків у React**Ошибка! Закладка не определена.**
1.3 Використання Vite у React додатках 15
1.4 Принцип роботи SPA додатків 16
1.5 SQL та NoSQL бази даних.. 18
1.6 Хмарні сервіси 20
1.7 Firebase та її компоненти 21
2 СТВОРЕННЯ FRONT-END ЧАСТИНИ ДОДАТКУ 23
2.1 Створення форми Авторизації та Реєстрації 23
2.2 Створення компонентів та їх інтеграція у додаток 26
3 СТВОРЕННЯ BACK-END ЛОГІКИ ТА ІНТЕГРАЦІЯ БАЗИ ДАНИХ 41
3.1 Налаштування та підключення Firebase 41
3.2 Створення логіки авторизації та створення аккаунту 44
3.3 Створення логіки пошуку користувачів та їх взаємодії.......................... 49
3.4 Створення логіки обміну даними між користувачами 52
3.5 Тестування роботи додатку та бази даних 58
ВИСНОВКИ.. 66
ПЕРЕЛІК ПОСИЛАНЬ 68
ДОДАТКИ.. 69
ДОДАТОК А 70

ДОДАТОК В 75

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

SPA – Single page application. Односторінковий додаток;

JS – Язик програмування JavaScript;

Framework – програмне середовище яке є каркасом додатку у фізуальній частині;

UI – User interface. Набір рішень що забезпечують розуміння функціоналу продукту і дає користувачу інтуїтивне розуміння взаємодії з додатком;

JSX – JavaScript XML. Розширення синтаксису JavaScript що дозволяє писати HTML у фреймворку React;

SQL – Structed Query Language. Стандартна мова запитів що використовується для взаємодії з реляційними базами даних;

NoSQL – Not Only SQL. Група баз даних які відрізняються від SQL баз даних способом організації та представлення даних;

API – Application Programming Interface. Набір протоколів взаємодії та засобів створення програмного забезпечення;

DOM – Document Object Model. Програмний інтерфейс що забезпечує представлення документу у вигляді структурованої групи вузлів та об’єктів які мають свої властивості та методи;

SASS – Syntactically Awesome Style Sheets. Метамова що інтерпретується в звичайний CSS після компіляції додатку;

NPM – Node Packages Manager. Пакетний менеджер NodeJS для завантаження бібліотек та інших пакетів;

REACT – JavaScript фреймворк який використовується для створення веб та мобільних додатків.

Firebase – Платформа яка використовується для розробки додатків, створенням та керуванням базами даних та модулями які можна інтегрувати у додатки.

ВСТУП

В епоху сучасних технологій, де вимоги до веб-додатків постійно зростають, розробка стає надзвичайно важливим етапом в створенні продуктів, які не лише задовольняють очікування користувачів, але й перевершують їх. У цьому контексті використання фреймворку React та бази даних Firebase від Google визнається ключовим стратегічним вибором, який трансформує веб-розробку в захоплюючу та результативну подорож.

React, як фреймворк з відкритим вихідним кодом, завоював серця розробників своєю ефективністю та гнучкістю. Здатний до створення вражаючих інтерфейсів, він полегшує взаємодію з DOM та реагує на зміни стану швидше, ніж багато інших альтернатив. Його компонентний підхід дозволяє розробникам структурувати код в елегантний спосіб, спрощуючи розробку та підтримку проекту. Використання JSX, що нагадує HTML, робить код React зрозумілим та легким для читання, що стає неоціненим плюсом в колективній розробці.

У поєднанні з React, Firebase, хмарна платформа від Google, стає каталізатором для інновацій та швидкого розвитку. Firebase не обмежується лише зберіганням даних; вона пропонує синхронізацію в реальному часі, ефективну систему аутентифікації та потужні інструменти інтеграції з іншими сервісами Google. Це відкриває перед розробниками не лише можливості для створення сучасних SPA, але й робить інтеграцію з іншими додатками безперешкодною та ефективною.

Це захоплююче союз React та Firebase створює не просто веб-додатки, але технологічні шедеври, здатні перетворити уявлення про користувацький досвід та забезпечити справжню цифрову інновацію. Давайте спрямуємось у цей захоплюючий світ розробки, де React та Firebase об'єднують свої сили, щоб втілити найаудаційніші веб-додатки у реальність.

1 АНАЛІТИЧНИЙ ОГЛЯД ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1.1 Фреймворк React

React - це фреймоврк JavaScript, яка надає готові рішення для створення SPA додатків. За допомогою React можна швидко розробляти, розширяти, запускати та тестувати SPA різної складності.

Під час використання цієї бібліотеки розробникам не потрібно приймати багато дизайнерських рішень, оскільки React вже надає кілька розумних рішень. Це дозволяє розробникам зосереджуватися на написанні ефективної програмної логіки, оскільки багато речей вже вирішено в рамках фреймворку. Однак, для досвідчених розробників, які працюють над великими кодовими базами, існують певні недоліки використання фреймворків.

Фреймворки часто не дозволяють достатню гнучкість, оскільки вимагають слідування певному стилю коду. Вони можуть боротися з розробником, якщо той вирішить відхилитися від стандартних підходів. Крім того, фреймворки зазвичай мають великий обсяг і включають в себе багато функцій, навіть якщо потрібно використовувати лише частину з них.

React вирішує ці питання, дотримуючись філософії Unix і фокусуючись на конкретному завданні - створенні інтерфейсів користувача. UI або інтерфейс користувача представляє собою все, що користувач бачить і взаємодіє з веб-сайтом або програмою. React дозволяє описувати цей інтерфейс декларативно, що означає вказівку бажаного стану інтерфейсу, а не того, як його досягти.

React використовує об'єктну модель документа DOM для взаємодії з веб-сторінками. Проте, він відмінюється від інших бібліотек, використовуючи декларативний підхід до опису інтерфейсів. Замість того, щоб відправляти в браузер шаблони, React передає дерево об'єктів через своє

API, яке визначає необхідні операції DOM для створення відповідного інтерфейсу.

React надає розробникам потужний інструментарій для створення інтерфейсів, що визначаються декларативно та для ефективного управління їх станом. Це робить процес розробки односторінкових додатків (SPA) більш простим та зрозумілим. У порівнянні з HTML-шаблонами, які розглядаються як рядок, React опрацьовується як дерево об'єктів, що дозволяє ефективно організувати та керувати компонентами.

JSX, як розширення синтаксису JavaScript, візуально асоціюється з XML та надає можливість трансформувати його в React-код. Незважаючи на те, що JSX на перший погляд може здатися мовою шаблонів, це насправді розширення JavaScript. Воно дозволяє представляти дерево об'єктів React, використовуючи синтаксис, що нагадує HTML.

Його синтаксис, полегшує читання та написання коду, забезпечуючи при цьому всі переваги JavaScript. Важливо підкреслити, що і сам React, і браузер не обов'язково працюють з JSX, оскільки воно обробляється на етапі компіляції. Код, який потрапляє в браузер, вже не містить JSX і залишається чистим від будь-яких шаблонів.

Компонент у React діє як шаблон, план чи глобальне визначення, і може бути представлений як функцією або класом. Елемент React, який представляє об'єкт, повертається компонентами, практично описує вузли DOM. Групування логіки та розділення на компоненти роблять код більш зручним для управління та тестування.

Однією з найважливіших переваг React є можливість легко створювати односторінкові додатки (SPA), які автоматично оновлюються у реальному часі, без очікування відповіді від сервера. Розподіл коду на компоненти сприяє ефективному управлінню та масштабуванню додатків.

Структура React-додатку дуже зручна та легко зрозуміла, а використання JSX у шаблонах компонентів додає гнучкості та зрозумілості

коду, роблячи його легко перевикористовуваним у будь-якому розділі додатку.

1.2 Компонентна розробка SPA додатків у React

Сучасні веб-додатки мають таку ж складність, як і будь-які інші програми, і часто розробляються командами розробників, які спільно працюють над фінальним продуктом. У таких умовах для підвищення ефективності природно використовувати стратегії розподілу роботи з мінімальними перетинами між розробниками та підсистемами. Компонентний підхід зазвичай використовується для досягнення цієї мети.

SPA, або односторінковий додаток, представляє собою веб-програму, яка завантажує лише один документ, а потім динамічно оновлює його вміст за допомогою JavaScript API, таких як XMLHttpRequest або Fetch. Це відбувається без повторного завантаження всієї сторінки, щоб показати новий вміст, що сприяє більш плавному та динамічному користувацькому досвіду.

Основна мета веб-компонентів - зменшення складності за рахунок ізоляції пов'язаних груп коду на HTML, SCSS та JavaScript для виконання загальної функціональності в межах контексту однієї сторінки. Це все добре реалізовано у React бо там фактично об’єднані HTML та JavaScript у форматі JSX.

JSX – це розширення JavaScript дозволяє писати код, який нагадує HTML, прямо всередині JavaScript-файлів. JSX спрощує створення інтерфейсів, адже його синтаксис подібний до HTML, і в ньому можна вставляти вирази JavaScript для динамічного визначення вмісту. Різницю між методами відображення HTML використовуючи JSX можна подивитись на рис. 1.1.

```
1  import React from 'react'
2
3  const CodeWithJSX = () => {
4  return (
5  <div>
6      <h1>Hello World!</h1>
7  </div>
8  )}
9
10 export default CodeWithJSX;
11
```

```
1  import React from 'react'
2
3  const CodeWithoutJSX = () => {
4      return React.createElement("div", null,
       React.createElement("h1", null, "Hello
       World!"))
5
6  }
7
8  export default CodeWithoutJSX;
9
```

Рисунок 1.1 – Приклад відображення HTML у компоненті JSX

Необхідно підкреслити, що JSX, який використовується в React, є розширенням синтаксису JavaScript і вимагає попередньої компіляції до стандартного JavaScript перед тим, як може бути використаний в браузері. Цей процес зазвичай виконується за допомогою інструментів, таких як Babel, що конвертують JSX в еквівалентний код JavaScript.

У React, компоненти є основною будівельною одиницею інтерфейсу. Вони допомагають структурувати додаток, розбиваючи його на невеликі та самостійні частини. Компоненти можна розглядати як вузли, які спільно формують структуру додатку, полегшуючи його розробку та підтримку.

У фреймворку React існують два основних типи компонентів: функціональні та класові. Функціональні компоненти є простими функціями, які приймають вхідні дані (props) і повертають елемент React. З іншого боку, класові компоненти є класами, що розширюють React.Component і можуть мати внутрішній стан.

Властивості (props) використовуються для передачі даних в компоненти, забезпечуючи їх конфігурованістю та можливістю використання в різних контекстах. Класові компоненти також можуть утримувати внутрішній стан, який дозволяє зберігати та оновлювати дані в межах компонента.

Метод render у React компонентах визначає, як саме компонент повинен відображатися на екрані. Класові компоненти також мають методи життєвого циклу, такі як componentDidMount і componentDidUpdate, які дозволяють виконати код на різних етапах життєвого циклу компонента.

Важливою особливістю React є можливість вкладення компонентів один в одного, утворюючи ієрархію. Це підтримує повторне використання та полегшує управління кодом, сприяючи читабельності та скальованості додатку.

1.3 Використання Vite у React додатках

Vite – це потужний інструмент для швидкого та ефективного розроблення веб-додатків. Заснований на сучасному JavaScript і обладнаний вбудованим сервером розробки, він дозволяє вам створювати свій додаток значно швидше, ніж інші інструменти. Розроблений з використанням технології Vue.js, Vite має низку переваг, що роблять його ідеальним вибором для веб-розробки.

Фактично Vite це міст між Twig/Craft CMS та інструментом для створення фронтенду нового покоління Vite.js. Vite дозволяє здійснювати гарячу заміну модулів (HMR) JavaScript, CSS та Twig під час розробки та оптимізації виробничих збірок. Vite створює необхідні теги для синхронного та асинхронного завантаження JavaScript і CSS, також має рівень кешування для підвищення продуктивності.

Основна перевага Vite - це швидкість. Завдяки вбудованому серверу розробки, гарячій заміні модулів HMR та оптимізації їх завантаження. Vite використовує сучасні модулі ECMAScript, це дозволяє використовувати імпорти та експорти у коді, спрощуючи структуру та організацію проекту. При використанні Vite для розробки, він запускає сервер, який компілює та обслуговує необхідні залежності через модулі ECMAScript. Цей підхід дозволяє Vite обробляти та надавати лише той код, який потрібний в даний момент. Таким чином, Vite обробляє значно менше коду при запуску сервера та під час оновлення коду, швидкість роботи приведено на рис. 1.2.

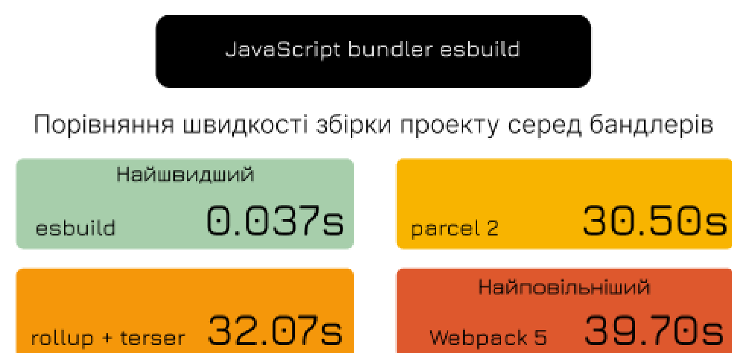

Рисунок 1.2 – Порівняння швидкості Javascript бандлерів.

Vite має простий конфігураційний файл, який можна легко налаштовувати відповідно до потреб вашого проекту, також є підтримка використання плагінів для додавання додаткової функціональності та підтримує TypeScript.

З переваг також можна віднести швидкий час розробки завдяки швидкому відгуку та можливості переглядати зміни без перезавантаження сторінки та модульну структуру. Оптимізація завантаження забезпечує ефективну продуктивність у виробничому середовищі та завантажує тільки потрібні модулі.

1.4 Принцип роботи SPA додатків

У контексті SPA додатків, всі функціональні можливості об'єднуються в єдину веб-сторінку. Цей підхід дозволяє витягти рівень представлення за межі сервера, надаючи браузеру контроль над ним. Кожен запит на отримання контенту, такого як HTML-сторінка, подається до сервера, який відповідає на нього, взаємодіючи з браузером у двох напрямках.

Коли на клієнтській стороні виникає потреба у свіжих даних, відправляється запит на сервер. На стороні сервера цей запит обробляється об'єктом контролера в межах представницького рівня. Контролер взаємодіє з модельним рівнем через сервісний шар, що визначає необхідні компоненти

для виконання завдання на рівні моделі даних. Після отримання необхідних даних від об'єкта доступу до даних або сервісного агенту, будь-які зміни вносяться в дані за допомогою бізнес-логіки на рівні бізнес-моделі.

Такий підхід розподіленої архітектури дозволяє підтримувати консистентність та ефективно керувати додатком, зменшуючи необхідність повторних завантажень сторінок і забезпечуючи динамічний та відзивчивий користувацький інтерфейс Приклад різниці між SPA та звичайним Web сайтом наведено на рис. 1.3.

Рисунок 1.3 – Різниця між SPA та звичайним Web сайтом

Керування передається назад на рівень представлення, де обирається відповідне представлення. Логіка представлення диктує, як щойно отримані дані будуть представлені у вибраному поданні. Часто результуюче подання починається як вихідний файл із заповнювачами, куди потрібно вставити дані. Цей файл діє як своєрідний шаблон того, як буде виглядати контент, коли контролер надсилає до нього запит.

Перенесення процесу створення та керування поданнями в інтерфейс користувача відокремлює його від сервера. З архітектурної точки зору, це дає SPA перевагу. Якщо ви не виконуєте частковий рендеринг сторінки на сервері, сервер більше не повинен брати участь у представленні даних.

Оскільки логіка представлення здебільшого знаходиться на стороні клієнта в SPA, завдання поєднання HTML і даних переноситься з сервера в браузер. Як і на стороні сервера, вихідний HTML містить заповнювачі, куди потрібно вставити дані (і, можливо, інші інструкції з рендерингу). Цей шаблон на стороні клієнта використовується як основа для штампування нових представлень у клієнті. Однак це не шаблон HTML для всієї сторінки. Він призначений лише для частини сторінки, яку представляє подання.

Все це відбувається без необхідності оновлювати оболонку. Таким чином, замість того, щоб отримувати нову статичну сторінку для кожного навігаційного запиту, SPA може відображати новий вміст без перешкод для користувача. Для певної частини екрана вміст одного подання просто замінюється вмістом іншого подання. Це створює ілюзію, що сама сторінка змінюється під час навігації користувача. Навігація без перезавантаження є ключовою особливістю SPA додатку.

1.5 Бази даних SQL та NoSQL

Реляційні та нереляційні бази даних пропонують різні підходи до збереження та організації інформації, розкриваючи широкий спектр можливостей для вирішення варіативних вимог у сфері зберігання даних.

Реляційні бази даних, засновані на табличній моделі з фіксованою схемою, використовують мову запитів SQL та гарантують чітку та передбачувану структуру даних. В цьому підході кожна таблиця наперед визначається зі своєю унікальною структурою, включаючи типи даних та взаємозв'язки. Така система є ефективною для великих проектів з чіткою та сталою структурою, де важлива стійкість схеми даних.

Нереляційні бази даних, з іншого боку, пропонують гнучкий підхід, де структура даних може бути менш жорстко визначеною. Вони дозволяють використовувати різноманітні формати для зберігання інформації, що робить

їх ідеальними для випадків, де схема даних може змінюватися з часом або коливатися в залежності від різних типів інформації.

Мови запитів для нереляційних баз даних можуть варіюватися від одного типу до іншого, і вони часто розроблені для відповіді на конкретні потреби певного типу даних. Однак рівень підтримки транзакцій у нереляційних базах даних може бути меншим, що важливо враховувати при виборі системи для конкретного проекту більш зрозуміло їх різницю наведено на рис. 1.4.

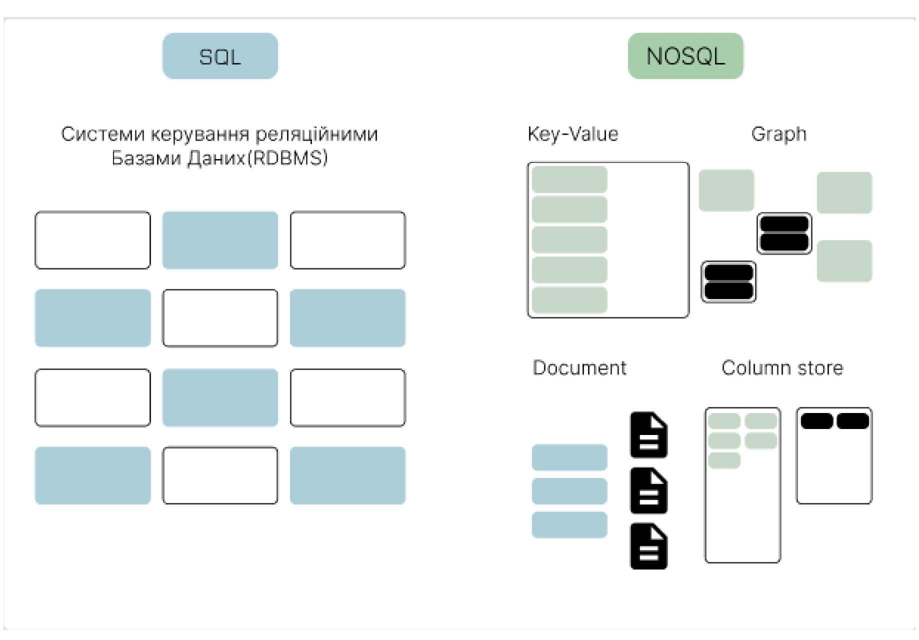

Рисунок 1.4 – Різниця між SQL та NoSQL базами даних

SQL розшифровується як "Structured Query Language”. Він є мовою програмування, що активно використовується для управління даними в реляційних системах керування базами даних з 1970-х років.

В наш час SQL залишається популярним для формулювання запитів до реляційних баз даних, де дані організовані у вигляді рядків та таблиць, які взаємодіють між собою. Записи однієї таблиці можуть мати зв'язки з іншими, будучи пов'язаними з одним або кількома іншими записами. Ці реляційні бази даних надають швидке зберігання та відновлення даних, і вони здатні

обробляти об'ємні та складні SQL-запити. великі обсяги даних і складні SQL-запити.

NoSQL є нереляційною базою даних, що вказує на те, що вона дозволяє використовувати інші структури, ніж база даних SQL (без рядків і стовпців). Також вона має більшу гнучкість у використанні формату, який найкраще відповідає конкретним даним. Термін "NoSQL" був введений на початку 2000-х років. Важливо відзначити, що це не означає відсутність використання SQL, оскільки деякі команди SQL іноді підтримуються базами даних NoSQL. Іншими словами, "NoSQL" іноді розглядається як "не тільки SQL".

1.6 Хмарні сервіси

Хмарні сервіси - це різновид послуг та ресурсів, доступних через Інтернет для використання, управління та доступу до різноманітних обчислювальних ресурсів. Основний принцип полягає в тому, що ці сервіси надаються віддалено через мережу, дозволяючи отримувати доступ до ресурсів з будь-якого місця та пристрою без необхідності фізичної власності чи управління обчислювальними ресурсами.

Хмарні сервіси можуть бути розділені на кілька категорій, таких як Інфраструктура як сервіс – IaaS, платформа як сервіс – PaaS та програмне забезпечення як сервіс – SaaS:

- IaaS забезпечує віртуальні обчислювальні ресурси;
- PaaS включає в себе платформу для розробки та розгортання додатків;
- SaaS дозволяє користувачам отримувати доступ до готових застосунків та програм через хмару.

Основна різниця між хмарними сервісами та звичайними NoSQL базами даних полягає в розташуванні та управлінні інфраструктурою. Звичайні NoSQL бази даних можуть бути розгорнуті на локальних серверах або власних обчислювальних ресурсах. У такому випадку, розробник повністю відповідає за установку, конфігурацію та управління базою даних.

Хмарні сервіси пропонують готові рішення для роботи з NoSQL базами даних у хмарних середовищах. Інфраструктура, така як сервери, мережеві з'єднання та безпека, управляється хмарним провайдером. Розгортання, масштабування та управління базою даних стають автоматизованими та спрощеними завдяки хмарним сервісам.

Інша важлива різниця - це масштабованість та доступність. Хмарні сервіси забезпечують можливість легко масштабувати ресурси за потребою, а також гарантують високу доступність через розташування даних в різних регіонах та автоматичне виявлення та відновлення випадків відмов.

Захист даних також здійснюється на рівні хмарного сервісу, надаючи різні інструменти для забезпечення конфіденційності та безпеки інформації.

Таким чином, хмарні сервіси роблять роботу з NoSQL базами даних більш простою, масштабованою та доступною, забезпечуючи розробникам готові рішення для ефективної роботи у хмарних середовищах.

1.7 Firebase та її компоненти

Firebase - це інтегрована платформа для розробки веб-додатків, яка пропонує широкий набір хмарних сервісів і інструментів для ефективного розвитку та управління проектами. Вона стала важливим інструментом для розробників, які прагнуть полегшити та прискорити процес створення веб-застосунків.

Однією з ключових переваг Firebase є його відмінна інтеграція з різними технологіями та бібліотеками. Це робить платформу універсальною та придатною для використання в різноманітних веб-проектах. Firebase пропонує реальну базу даних, яка автоматично синхронізує дані між усіма клієнтами в реальному часі. Це стає важливим аспектом для веб-додатків, які вимагають миттєвого оновлення та спільної роботи над даними. Основні компоненти наведені на рис. 1.5.

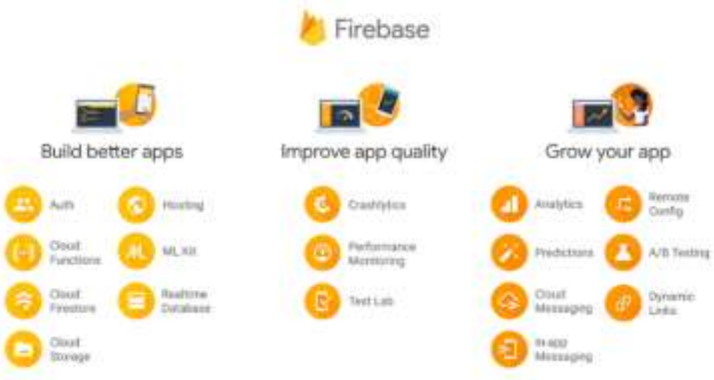

Рисунок 1.5 – Компоненти та сервіси Firebase.

Система аутентифікації Firebase надає розробникам гнучкість у впровадженні різних методів аутентифікації, такі як електронна пошта, Google, Facebook та інші. Це забезпечує високий рівень безпеки та дозволяє ефективно управляти доступом користувачів до ресурсів додатка.

Хмарні функції Firebase дозволяють розгортати власний код на сервері, що дає розробникам гнучкість налаштовувати та оптимізувати додаток відповідно до його вимог. Завдяки цьому підходу, Firebase стає ідеальним вибором для веб-додатків різних розмірів та обсягів.

Хмарне зберігання Firebase дозволяє ефективно та масштабовано управляти мультимедійним контентом, таким як зображення чи відео. Його засоби для аналізу дозволяють відстежувати та аналізувати використання додатка, що допомагає виробникам приймати обґрунтовані рішення щодо його оптимізації.

Firebase також надає інструменти для тестування функціоналу додатка та зручні засоби для його розгортання в виробничому середовищі. Завдяки простому та інтуїтивно зрозумілому інтерфейсу, Firebase дозволяє розробникам швидко розгортати проекти та ефективно керувати різними аспектами розробки веб-додатків.

Firebase універсальним для різноманітних веб-проектів. Важливими особливостями є реальний час синхронізації даних, гнучка система

аутентифікації, можливість розгортання власного коду на сервері та ефективне управління мультимедійним контентом. Firebase стає ідеальним вибором для веб-додатків різних масштабів та вимог, надаючи зручні інструменти для тестування, аналізу та розгортання проектів в хмарному середовищі.

2 СТВОРЕННЯ FRONT-END ЧАСТИНИ ДОДАТКУ

2.1 Проектування макету додатку у Figma

Мета: Спроектувати базовий макет додатку та його логіку, спроектувати UI додатку, розділити на модулі і компоненти, написати код Front-end частини додатку.

Використовуючи Figma ми можемо спроектувати макет додатку. Пріоритетом буде створення інтуїтивно зрозумілого додатку, тож почнемо працювати з його схемою яку наведено на рис. 2.1.

Рисунок 2.1 – Базова логічна схема додатку

Маючи базову схему ми можемо побудувати макет форми логіну та реєстрації користувача. У цій формі нам потрібно отримати дані від користувача, тож вона буде мати три поля введення та одне поле завантаження, форма логіну буде мати два поля введення, електронної пошти та пароля вистачить щоб авторизувати користувача. Макет наведено на рис. 2.2.

Рисунок 2.2 – Макет сторінок авторизації та реєстрації

Спроектуємо головну сторінку додатку на яку будемо потрапляти після авторизації, вона має містити функціонал пошуку користувачів, створення нових чатів та відправці повідомлень, тож буде мати ділення на дві секції: зліва – секція з вибором чатів та пошуком нових, справа – секція взаємодії з обраним користувачем. Це буде основна структура нашого додатку, вона є зручною і зрозумілою новим користувачам. Макет наведено на рис. 2.3.

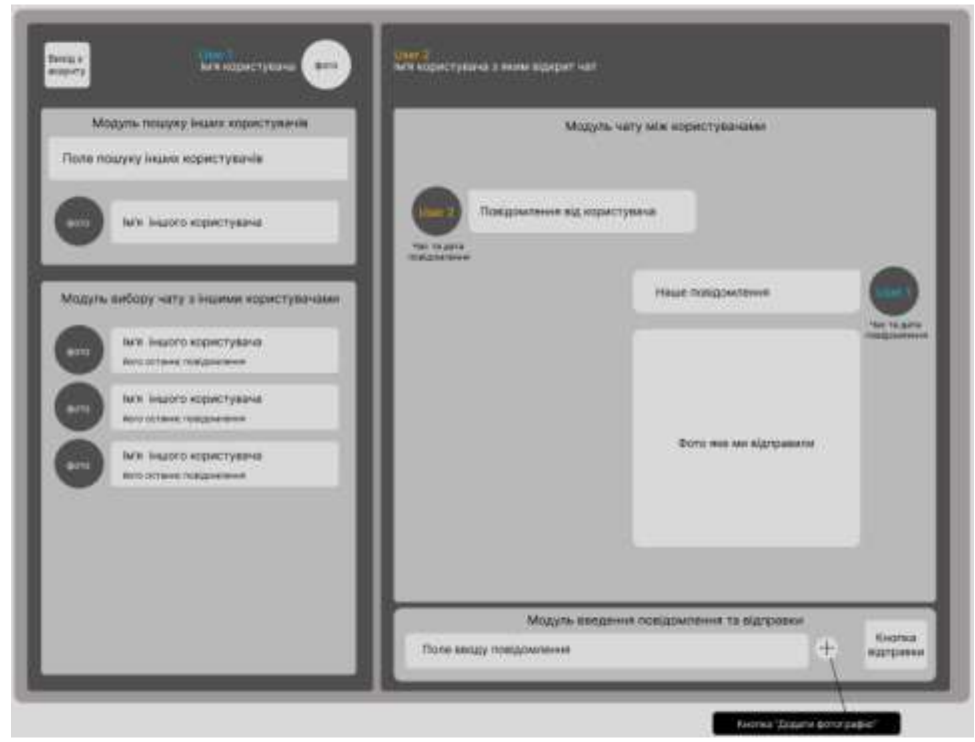

Рисунок 2.3 – Макет головної сторінки.

Також важливою частиною є підбір кольорів які ми будемо використовувати. Кольорів та відтінків на невеличкий додаток краще використовувати не більше п’яти тож візьмемо.

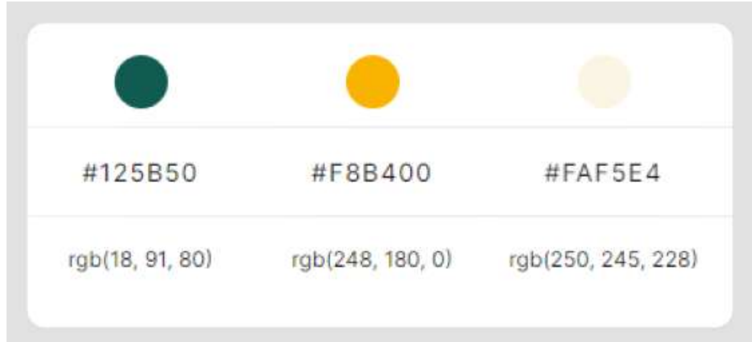

Рисунок 2.4 – Палітра кольорів додатку

2.2 Налаштування проекту, створення форм Авторизації та Реєстрації

Мета: Ініциалізувати та налаштувати проект. Підключити потрібні бібліотеки. Створити дві форми використовуючи JSX та SCSS занести у проект їх як окремі модулі щоб в подальшому інтегрувати логіку і роутінг.

Почнемо зі створення проекту по перше нам потрібно мати середу розробки(IDE), я буду використовувати Webstorm. Треба мати встановлений NodeJS та Npm. У IDE відкриваємо термінал та пишемо:

$npm create vite@latest app01 -- --template react.

Також треба підключити роутінг тож одразу завантажимо пакет:

$npm i react-router-dom.

Ця команда ініціалізує новий проект, структура якого буде трохи відрізнятися від звичайного React. Vite є більш спрощеною системою тож там не буде деяких модулів. Одразу створимо у корні проекту папку pages у якій створимо три файли з майбутніми сторінками у форматі JSX.

Рисунок 2.5 – Основні сторінки додатку

Почнемо зі сторінок “Login” та “Register”. По перше створимо компонент Register в якому пропишемо базові стилі та створимо поки що пусту функцію handleSubmit, також пропишемо роутінг і прив’яжемо сторінку до шляху “/login”:

```
import React, {useState} from "react"
import addImgIcon from "../img/add_photo.svg"
import {useNavigate, Link } from "react-router-dom";
```

```
const Register = () => {
    const navigate = useNavigate()
    const handleSubmit = async (e) => {}
return (
        <div className='formContainer'>
            <div className="formWrapper">
                <span className="logo"></span>
                <span className="title"></span>
                <form onSubmit={handleSubmit}>
                    <input type="text" placeholder="Логін"/>
                    <input type="email" placeholder="Email"/>
                    <input type="password" placeholder="Пароль"/>
                    <input style={{display:"none"}} type="file" id="file"/>
                    <label htmlFor="file">
                        <img src={addImgIcon} alt=""/>
                        <span>Додати фотографію</span>
                    </label>
                    <button>Зареєструватися</button>

               </form>
                <p>Вже маєте аккаунт?  <Link to="/login">Увійти</Link></p>
            </div>
        </div>
    )
    };

export default Register.
```

Створимо універсальний файл стилів style.scss. Також якщо необхідно встановимо пакет, пишемо в терміналі:

$npm install sass.

Напишемо універсальні стилі для наших сторінок:

```
.formContainer {
  background-color: #f2f2f3;
  height: 100vh;
  display: flex;
  align-items: center;
  justify-content: center;

  .formWrapper {
    background-color: white;
    box-shadow: 0px 0px 30px 1px rgba(0,0,0,0.42);
    padding: 20px 60px;
    border-radius: 10px;
    display: flex;
    flex-direction: column;
    gap: 10px;
    align-items: center;

    .logo {
      color: #5d5b8d;
      font-weight: bold;
      font-size: 24px;
```

```
    }

    .title {
      color: #5d5b8d;
      font-size: 12px;
    }

    form {
      display: flex;
      flex-direction: column;
      gap: 15px;

      input {
        padding: 15px;
        border: none;
        width: 250px;
        border-bottom: 1px solid #495464;
        &::placeholder {
          color: rgb(175, 175, 175);
        }
      }

      button {
        background-color: #125B50;
        color: white;
        padding: 10px;
        border-radius: 10px;
        font-weight: bold;
        border: none;
        cursor: pointer;
        transition: all .3s;
        &:hover {
          background-color: #F8B400;
          border-radius: 10px;
          color:#125B50 ;
        }
      }

      label {
        display: flex;
        align-items: center;
        gap: 10px;
        color: #495464;
        font-size: 12px;
        cursor: pointer;

        img {
          width: 25px;
        }
      }
    }
    p {
      color: #495464;
      font-size: 12px;
      margin-top: 10px;
    }
  }
}.
```

Для сторінки Login нам достатньо прописати схоже але видалити поле вводу логіну та додавання фотографії код виглядатиме так:

```
import React, {useState} from "react"
import {useNavigate, Link } from "react-router-dom";

const Login = () => {
    const navigate = useNavigate()
    const handleSubmit = async (e) => {}

  return (
        <div className='formContainer'>
            <div className="formWrapper">
                <span className="logo"></span>
                <span className="title"></span>
                <form onSubmit={handleSubmit}>
                    <input type="email" placeholder="Email"/>
                    <input type="password" placeholder="Пароль"/>
                    <button>Увійти</button>
                </form>
                {err && <span>Щось пішло не так</span>}
                <p>Все ще не маєте аккаунту? <Link
to="/register">Зареєструватися</Link></p>
            </div>
        </div>
    )
};

export default Login.
```

Одразу підключимо роутінг. Перейдемо в App.jsx підключимо react-router-dom та де пізніше створимо функцію захищеного роутінга:

```
import Register from "./pages/Register.jsx";
import Login from "./pages/Login.jsx";
import Home from "./pages/Home.jsx";
import "./style.scss"
import {
  BrowserRouter,
  Routes,
  Route, Navigate,
} from "react-router-dom";
import {useContext} from "react";

function App() {



  return (
      <BrowserRouter>
        <Routes path="/">
          <Route index element={
            <ProtectedRoute>
              <Home />
```

```
            </ProtectedRoute>
          }/>
        <Route path="login" element={<Login/>}/>
        <Route path="register" element={<Register/>}/>
      </Routes>
    </BrowserRouter>
  )
}

export default App;.
```

На виході будемо мати повноцінні сторінки реєстрації та логіну які будуть прив’язані до відповідного шляху у посиланні на додаток. Створені форми наведено на рис. 2.6.

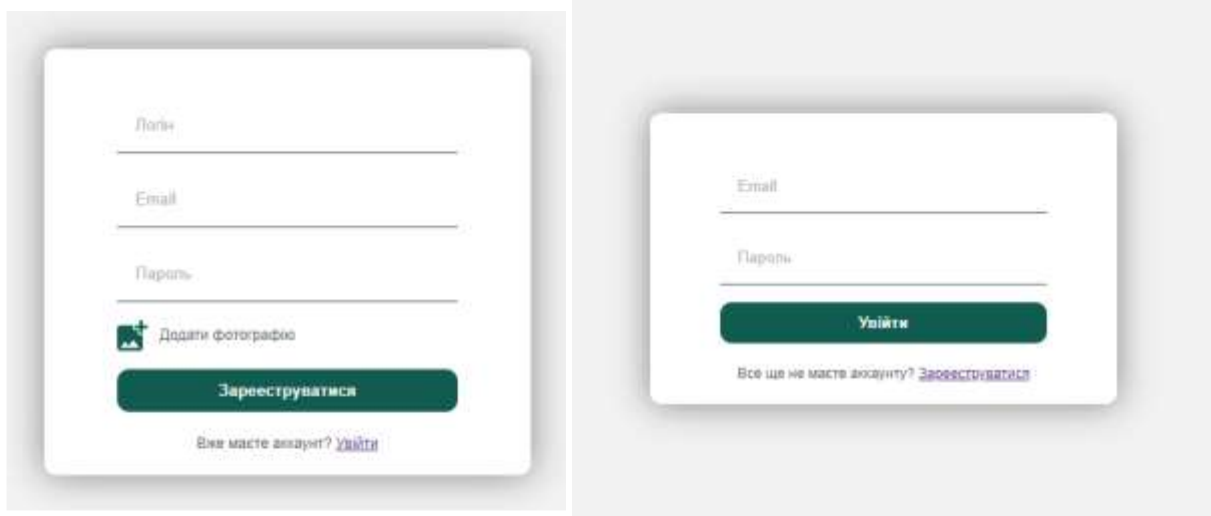

Рисунок 2.6 – Стилізовані сторінки

Наступним кроком буде створення візуальної і базової логічної частини до основної сторінки додатку.

2.2 Створення компонентів та їх інтеграція у додаток

Мета: Створити головну сторінку додатку, перенести візуальну частину та базову логіку усіх модулів та компонентів які були спроектовані у макеті.

Спочатку створимо достатню кількість компонентів у директорії components та будемо створювати їх виходячи із макету.

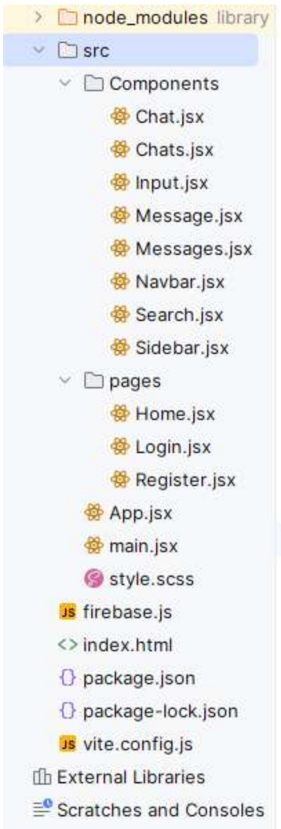

Рисунок 2.7 – Директорія з компонентами до головної сторінки

Перейдемо до файлу Home.jsx та створимо головний контейнер для модулів чату, також імпортуємо декілька компонентів:

```
import React from "react";
import Sidebar from "../Components/Sidebar.jsx";
import Chat from "../Components/Chat.jsx";

const Home = () => {
    return (
        <div className='home'>
            <div className="container">
                <Sidebar/>
                <Chat/>
            </div>
        </div>
    )
}

export default Home.
```

Тепер створимо компоненту Sibebar.jsx. вона буде відповідати за наш лівий модуль додатку який відповідає за навігацію між чатами, модуль пошуку користувачів та відображення нашого зареєстрованого акаунту:

```
import React from "react";
import Navbar from "./Navbar.jsx";
import Search from "./Search.jsx";
import Chats from "./Chats.jsx";

const Sidebar = () => {
    return(
        <div className='sidebar'>
            <Navbar/>
            <Search/>
            <Chats/>
        </div>
    )
}

export default Sidebar.
```

Спочатку закінчимо ліву секцію тож переходимо до Navbar.jsx:

```
import React from "react";
import logoutIcon from "../img/logout.svg"

const Navbar = () => {
    return(
        <div className="navbar">
            <button onClick={}>
                <img src={logoutIcon} alt="logout"/>
            </button>

                <div className="user">
                    <img src={} alt=""/>
                    <span>{}</span>

                </div>
        </div>
    )
}

export default Navbar.
```

Тепер напишемо пошукову систему у компоненті Search.jsx :

```
import React, from "react";

const Search = () => {
return (
        <div className="search">
            <div className="searchForm">
```

```
                <input
                    type="text"
                    placeholder="Знайти користувача"
                    onKeyDown={handleKey}
                    onChange={(e) => setUsername(e.target.value)}
                    value={username}
                />
            </div>
            {err && <span>Користувача не знайдено</span>}
            {user && (
                <div className="userChat" onClick={handleSelect}>
                    <img src={user.photoURL} alt="" />
                    <div className="userChatInfo">
                        <span>{user.displayName}</span>
                    </div>
                </div>
            )}
        </div>
    );
};

export default Search.
```

Залишилось лише прописати візуальну частину для блоку Chats.jsx у якому ми будемо відображати чати з іншими користувачами якщо вони існують:

```
import React, from "react";

const Chats = () => {
return (
        <div className="chats">

                <div
                    className="userChat"
                    key={chat[0]}
                    onClick()}
                >
                    <img src={ } alt="" />
                    <div className="userChatInfo">
                        <span>{ }</span>
                        <p>{ }</p>
                    </div>
                </div>
            ))}
        </div>
    );
};
export default Chats.
```

Тепер зробимо праву частину макету яка буде включати в себе відображення користувача з яким ми відкрили чат, історія повідомлень, та поле вводу повідомлень та кнопка «Додати фото». Назвемо цей модуль Chat.jsx і запишемо його структуру:

```
import React, {useContext} from "react";
import Messages from "./Messages.jsx";
import Input from "./Input.jsx";

const Chat = () => {

    return (
        <div className="chat">
            <div className="chatInfo">
                 <span>{}</span>
                <div className="chatIcons">
                    <img src="../img/cam.png" alt="" />
                </div>
            </div>
            <Messages />
            <Input/>
        </div>
    );
};

export default Chat.
```

Тепер запишемо компоненту Messages.jsx яка буде відображати повідомлення, запишемо її:

```
import React from "react";

const Messages = () => {

return (
        <div className="messages">
            {messages.map((m) => (
                <Message message={m} key={m.id} />
            ))}
        </div>
    );
};

export default Messages.
```

В ній ми створимо ще одну компоненту Message.jsx яка буде вкладати в себе структуру повідомлень:

```
import React from "react";


  return(
        <div className={`message owner`}>
            <div className="messageInfo">
                <img src={}
                     alt=""/>
                <div className="messageDate">
                    <span>{}</span>
```

```
                    <span>{}</span>
                </div>
            </div>
            <div className="messageContent">
                <p>{}</p>
                 <img src={ } alt=""/>
            </div>
        </div>
    )
}

export default Message.
```

Перш ніж почати роботу з серверною частиною нам потрібно дописати компоненту Input.jsx яка буде відповідати за введення повідомлення та його відправку, запишемо код:

```
import React from "react";

const Input = () => {
   return (
        <div className="input">
            <input
                type="text"
                placeholder="Напишіть щось..."
                onChange={(e) => setText(e.target.value)}
                value={text}
            />
            <div className="send">
                <input
                    type="file"
                    style={{ display: "none" }}
                    id="file"
                    onChange={(e) => setImg(e.target.files[0])}
                />
                <label htmlFor="file">
                    <img src={add_photo} alt="Додати фотографію" />
                </label>
                <button onClick={handleSend}>надіслати</button>
            </div>
        </div>
    );
};

export default Input.
```

Після цього допишемо стилі в style.scss:

```
.home {
  background-color: #f2f2f3;
  height: 100vh;
  display: flex;
  align-items: center;
  justify-content: center;
```

```
.container {
  box-shadow: 0px 0px 30px 1px rgba(0,0,0,0.42);
  border-radius: 10px;
  width: 65%;
  height: 80%;
  display: flex;
  overflow: hidden;
  @include tablet {
    width: 90%;
  }

  .sidebar {
    flex: 1;
    background-color: rgba(18, 91, 80, 0.9);
    position: relative;

    .navbar {
      display: flex;
      align-items: center;
      background-color: #125B50;
      height: 50px;
      padding: 10px;
      justify-content: space-between;
      color: #f2f2f3;

      button {
        background-color: #125B50;
        padding: 5px;
        border-radius: 5px;
        border: none;
        cursor: pointer;
        @include tablet {
          position: absolute;
          bottom: 10px;
        }
        img{
          width: 24px;
        }
      }
      }

      .user {
        display: flex;
        align-items: center;
        gap: 10px;

        img {
          background-color: #ddddf7;
          height: 30px;
          width: 30px;
          border-radius: 50%;
          object-fit: cover;
        }

      }
    }
    .search {
      border-bottom: 1px solid rgba(18, 91, 80, 0.9);

      .searchForm {
        padding: 10px;

        input {
          background-color: transparent;
```

```
        border: none;
        color: white;
        outline: none;

        &::placeholder {
          color: white;
        }
      }
    }
  }

  .userChat {
    padding: 10px;
    display: flex;
    align-items: center;
    gap: 10px;
    color: white;
    cursor: pointer;
    transition: all .3s;

    &:hover {
      background-color: #F8B400;
      color: #395248;
    }

    img {
      width: 50px;
      height: 50px;
      border-radius: 50%;
      object-fit: cover;
    }

    .userChatInfo {
      max-width: 450px;
      span {
        font-size: 15px;
        font-weight: bold;
      }
      p {
        display: block;
        overflow: hidden;
        text-overflow: ellipsis;
        font-size: 12px;
        padding-top: 5px;
        color: rgba(242, 242, 243, 0.71);
        width: 100px;

      }
    }
  }
}
.chat {
  flex: 2;

  .chatInfo {
    height: 50px;
    background-color: rgba(18, 91, 80, 0.9);
    display: flex;
    align-items: center;
    justify-content: space-between;
    padding: 10px;
    color: #F8B400;
  }
```

```
.chatIcons {
  display: flex;
  gap: 10px;

  img {
    height: 24px;
    cursor: pointer;
  }
}

.messages {
  background-color: #f2f2f3;
  padding: 10px;
  height: calc(100% - 160px);
  overflow-y: scroll;

  .message {
    display: flex;
    flex-wrap: wrap;
    text-overflow: ellipsis;
    gap: 20px;
    margin-bottom: 20px;

    .messageInfo {
      display: flex;
      flex-direction: column;
      color: gray;
      font-weight: 300;

      img {
        width: 40px;
        height: 40px;
        border-radius: 50%;
        object-fit: cover;
      }
      .messageDate {
        padding-top: 5px;
        display: flex;
        flex-direction: column;
        span {
          font-size: 11px;
        }
      }
    }
    .messageContent {
      max-width: 80%;
      display: flex;
      flex-direction: column;
      gap: 10px;

      p {
        word-break: break-word;
        background-color: white;
        padding: 10px 20px;
        border-radius: 0px 10px 10px 10px;
        max-width: max-content;
        box-shadow: 0px 0px 10px 1px rgba(0,0,0,0.2);
      }

      img {
        width: 50%;
      }
    }
```

```
          &.owner {
            flex-direction: row-reverse;

            .messageContent {
              align-items: flex-end;
              p {
                color: #000000;
                border-radius: 10px 0px 10px 10px;
              }
            }
          }
        }
      }

      .input {
        height: 50px;
        background-color: white;
        padding: 10px;
        display: flex;
        align-items: center;
        justify-content: space-between;

        input {
          width: 100%;
          border: none;
          outline: none;
          color: #2f2d52;
          font-size: 16px;

          &::placeholder {
            color: rgba(18, 91, 80, 0.44);
          }
        }

        .send {
          display: flex;
          align-items: center;
          gap: 10px;

          img {
            height: 24px;
            cursor: pointer;
          }

          button {
            border: none;
            padding: 10px 15px;
            color: white;
            background-color: #125B50;
            cursor: pointer;
            border-radius: 5px;
            transition: all .3s;

            &:hover {
              background-color: #F8B400;
              color: #125B50;
            }
          }
        }
      }
    }
  }
}.
```

На виході будемо мати такий результат будови нашого проекту:

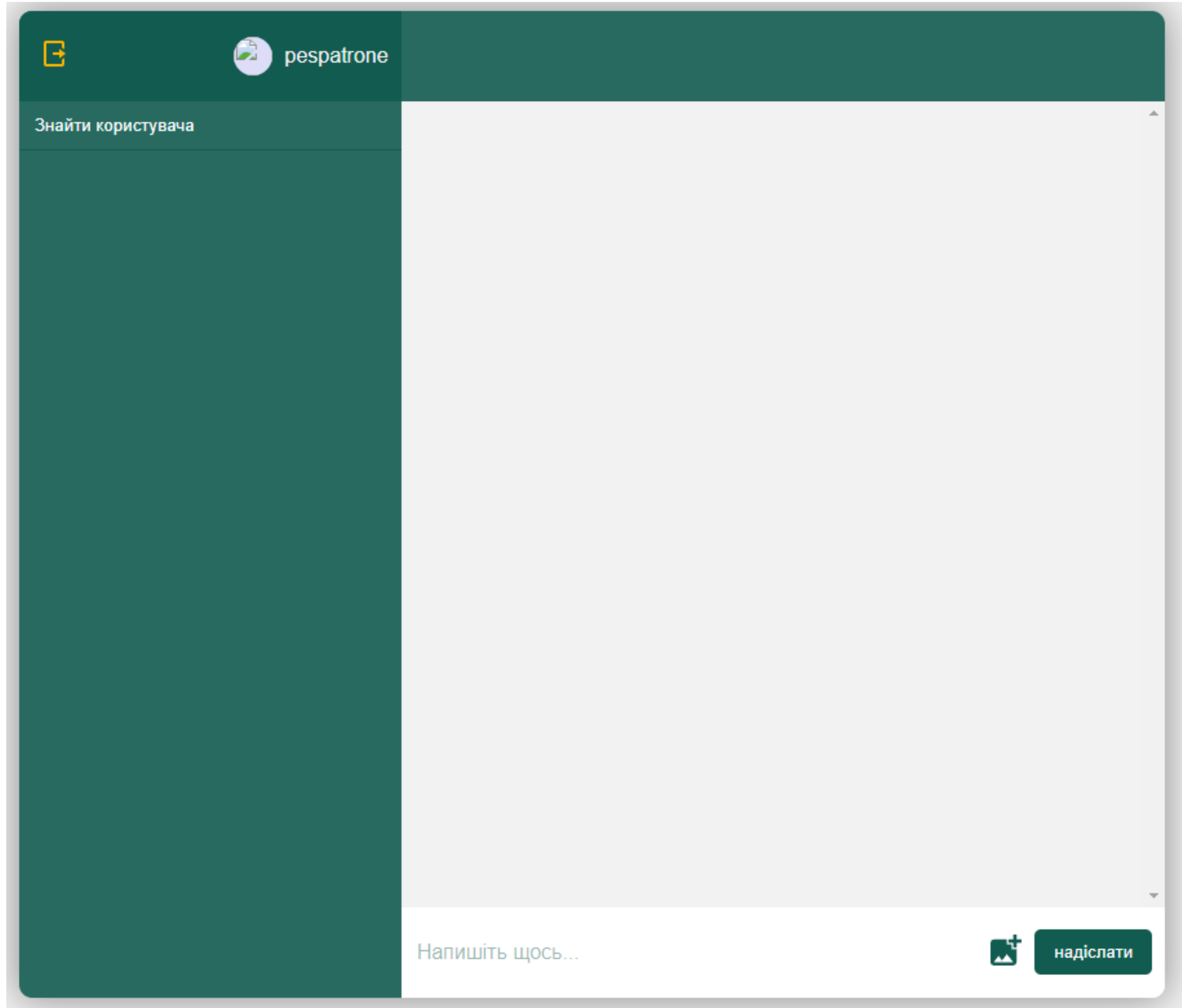

Рисунок 2.8 – Стилізована головна сторінка додатку.

У цьому розділі ми спроектували структуру проекту та його блок схему з базовою логікою, створили каркас проекту побудувавши його у Figma та зробили Front-end частину додатку зверставши його використовуючи компонентний підхід. Щоб додаток функціонував та взаємодіяв з користувачем потрібно прописати логіку та налаштувати серверну частину, поєднавши Front-end з Back-end ми отримаємо додаток що буде працювати та оновлюватись в реальному часі.

3 ІНТЕГРАЦІЯ БАЗИ ДАНИХ ТА СТВОРЕННЯ BACK-END ЛОГІКИ

3.1 Налаштування та підключення Firebase

Мета: Підключити Firebase до проекту, налаштувати потрібні модулі та підключити потрібні пакети до проекту.

Для того щоб налаштувати базу даних нам потрібно зайти на офіційний сайт Firebase через акаунт Google, так як це їх продукт є багато спрощених і зручних функцій які можна використати та інтегрувати до проекту.

Спершу створимо новий проект, дамо йому ім’я та почекаємо хвилину коли він створиться та нам надійде повідомлення як на рис 3.1.

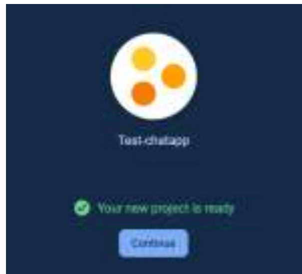

Рисунок 3.1 – Сповіщення про готовність проекту до роботи

Далі ми потрапимо у меню де нам запропонує обрати тип нашого додатку, Firebase автоматично адаптується під наш вибір, обираємо Web як показано на рис 3.2.

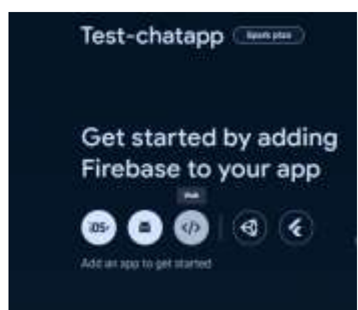

Рисунок 3.2 – Панель вибору типу додатку

Записуємо ім'я та реєструємо додаток. Після цього ми отримаємо ключ, це доступ до бази даних, приклад цього ключу наведено на рис. 3.3.

```
// Import the functions you need from the SDKs you need
import { initializeApp } from "firebase/app";
// TODO: Add SDKs for Firebase products that you want to use
// https://firebase.google.com/docs/web/setup#available-libraries

// Your web app's Firebase configuration
const firebaseConfig = {
  apiKey: "AIzaSyDHnjwEWqrPL5KV8p0YrfT_jo2mncbHYds",
  authDomain: "test-chatapp-117ab.firebaseapp.com",
  projectId: "test-chatapp-117ab",
  storageBucket: "test-chatapp-117ab.appspot.com",
  messagingSenderId: "461552246566",
  appId: "1:461552246566:web:965353fe553593979f2be7"
};

// Initialize Firebase
const app = initializeApp(firebaseConfig);
```

Рисунок 3.3 – Унікальні параметри проекту в Firebase

Також є додатковий імпорт, тож нам потрібно встановити firebase локально на наш проект:

$ npm install --save firebase.

Після цього в корні проекту створимо файл firebase.js та запишемо в нього дані нашої бази а також додамо модулі і експортуємо їх:

```
import { initializeApp } from "firebase/app";
import { getAuth } from "firebase/auth"
import { getStorage } from "firebase/storage"
import { getFirestore } from "firebase/firestore"

const firebaseConfig = {
  apiKey: "AIzaSyDHnjwEWqrPL5KV8p0YrfT_jo2mncbHYds",
  authDomain: "test-chatapp-117ab.firebaseapp.com",
  projectId: "test-chatapp-117ab",
  storageBucket: "test-chatapp-117ab.appspot.com",
  messagingSenderId: "461552246566",
  appId: "1:461552246566:web:965353fe553593979f2be7"

};

export const app = initializeApp(firebaseConfig);
export const auth = getAuth();
export const storage = getStorage();
export const db = getFirestore().
```

Після того як створили файл, почнемо створювати БД яка оновлюється у реальному часі, БД під аутентифікацію та БД під сховище для фотографій. Для цього перейдемо у вкладку “Build” та створимо декілька сховищ. Нам потрібні «Authentication», «Firestore Database» та «Storage» усі функції можна подивитись на рис. 3.4.

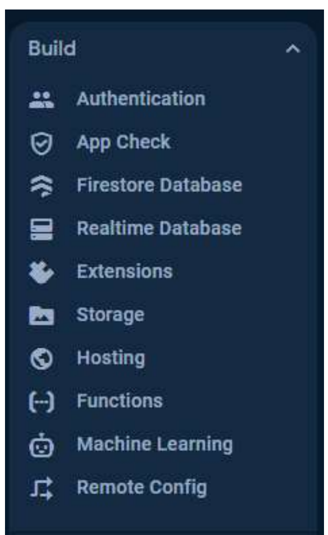

Рисунок 3.4 – Доступні функції Firebase

Обираємо звичайний метод по пошті та паролю, але є багато інших варіантів які можна зробити, методи показано на рис. 3.5.

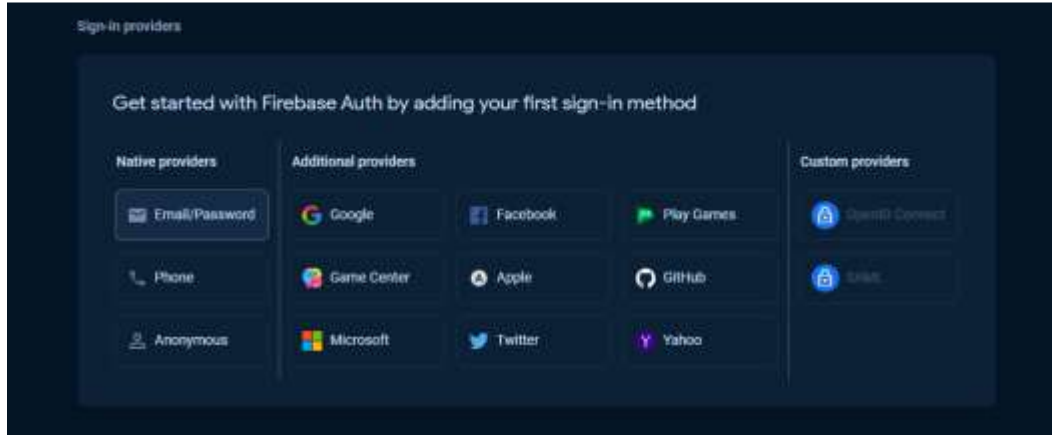

Рисунок 3.5 – Методи Аунтефікації Firebase

Далі створюємо Local Storage в тестовому режимі, для того щоб зберігати фотографії які ми будемо відправляти або завантажувати при реєстрації. Після тестів ми зможемо перевести його у production, приклад на рис. 3.6

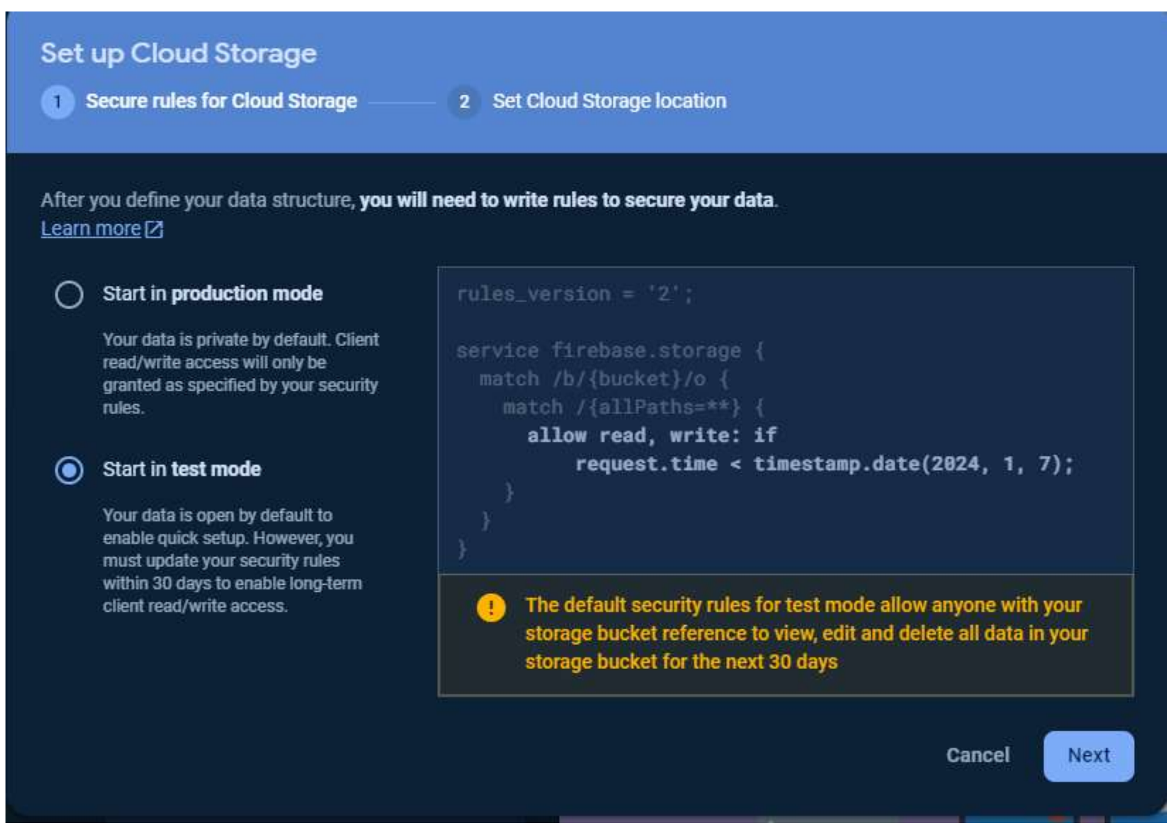

Рисунок 3.6 – Створення Local Storage

Створивши Firestore Database ми поки що не маємо ніяких колекцій. Ми можемо декількома способами створити коллекції у БД.

3.2 Створення логіки авторизації та створення аккаунту

Мета: Підключити Firebase модуль Authentication та налаштувати його, написати логіку авторизації, зробити систему логіна/реєстрації користувача через електронну пошту.

Почнемо зі створення файлу AuthContext.jsx який буде відповідати за перевірку контексту нашого користувача, якщо користувач буде у системі то ми будемо пропускати його до додатку, якщо ні то будемо направляти на сторінку з логіном. Важливим моментом є те що ми будемо постійно перевіряти state через метод useState. І якщо ми будемо мати зміну стану (наприклад користувач вийшов з акаунту) то будемо направляти його на сторінку з логіном код у цьому файлі буде виглядати так:

```
import {createContext, useEffect, useState,} from "react";
import { auth } from "../../firebase.js";
import { onAuthStateChanged } from "firebase/auth"
export const AuthContext = createContext()
export const AuthContextProvider = ({children}) => {

    const [currentUser, setCurrentUser] = useState ({})

    useEffect(() => {
        const unsub = onAuthStateChanged(auth,(user)=> {
            setCurrentUser(user)
            console.log(user)
        })
        return () => {
            unsub()
        }
    }, []);

    return (
        <AuthContext.Provider value={{ currentUser }}>
            {children}
        </AuthContext.Provider>
    )

}.
```

Використовуючи цей контекст, перепишемо наш файл App.jsx реалізуємо захищений роутінг який постійно буде перевіряти state за допомогою функції ProtectRoute та контексту:

```
import Register from "./pages/Register.jsx";
import Login from "./pages/Login.jsx";
import Home from "./pages/Home.jsx";
import "./style.scss"
import {
  BrowserRouter,
  Routes,
  Route, Navigate,
} from "react-router-dom";
import {useContext} from "react";
```

```
import {AuthContext} from "./context/AuthContext.jsx";

function App() {
  const {currentUser} = useContext(AuthContext)

  const ProtectedRoute = ({children}) => {
    if(!currentUser) {
      return <Navigate to="/login"/>
    }

    return children
  }


  return (
      <BrowserRouter>
        <Routes path="/">
          <Route index element={
            <ProtectedRoute>
              <Home />
            </ProtectedRoute>
           }/>
          <Route path="login" element={<Login/>}/>
          <Route path="register" element={<Register/>}/>
        </Routes>
      </BrowserRouter>
  )
}

export default App.
```

Тепер допишемо компоненту Register.jsx яка відповідає за сторінку реєстрації. Додамо логіку та асинхронну функцію яка буде введені нами дані передавати до Firestore і зберігати їх там, щоб у подальшому використати для аунтефікації. Три параметри які вводить користувач будуть зберігатися у БД та використовуватись для сторінки входу:

```
import React, {useState} from "react"
import { createUserWithEmailAndPassword, updateProfile} from "firebase/auth";
import { auth, storage, db} from "../../firebase.js";
import { ref, uploadBytesResumable, getDownloadURL } from "firebase/storage";
import { doc, setDoc } from "firebase/firestore";
import {useNavigate, Link} from "react-router-dom";
import addImgIcon from "../img/add_photo.svg"

    const Register = () => {
        const [err,setErr] = useState(false)
        const navigate = useNavigate()
        const handleSubmit = async (e) => {
            e.preventDefault()
            const displayName = e.target[0].value;
            const email = e.target[1].value;
            const password = e.target[2].value;
            const file = e.target[3].files[0];

    try {
```

```
        const res = await createUserWithEmailAndPassword(auth, email,
password)
        const storageRef = ref(storage, displayName);
        const uploadTask = uploadBytesResumable(storageRef, file);


        uploadTask.on(
            (error) => {
                setErr(true)
            },
            () => {
                getDownloadURL(uploadTask.snapshot.ref).then( async
(downloadURL) => {
                    await updateProfile(res.user,{
                        displayName,
                        photoURL: downloadURL,
                    })
                    await setDoc(doc(db, "users", res.user.uid),{
                        uid: res.user.uid,
                        displayName,
                        email,
                        photoURL: downloadURL,
                    })
                    await setDoc(doc(db, "userChats", res.user.uid), {})
                    navigate("/")
                });
            }
        )

    } catch (err) {
        setErr(true)
    }
    }

    return (
        <div className='formContainer'>
            <div className="formWrapper">
                <span className="logo"></span>
                <span className="title"></span>
                <form onSubmit={handleSubmit}>
                    <input type="text" placeholder="Логін"/>
                    <input type="email" placeholder="Email"/>
                    <input type="password" placeholder="Пароль"/>
                    <input style={{display:"none"}} type="file" id="file"/>
                    <label htmlFor="file">
                        <img src={addImgIcon} alt=""/>
                        <span>Додати фотографію</span>
                    </label>
                    <button>Зареєструватися</button>
                    {err && <span>Щось пішло не так</span>}
                </form>
                <p>Вже маєте аккаунт?  <Link to="/login">Увійти</Link></p>
            </div>
        </div>
    )
    };

export default Register.
```

Доповнимо компоненту Login.jsx функцією яка відправляє запит до БД та отримує відповідь. Цей функціонал реалізовано через метод signInWithEmailAndPassword який ми імпортуємо з пакету Firebase:

```
import React, {useState} from "react"
import {useNavigate, Link } from "react-router-dom";
import { signInWithEmailAndPassword } from "firebase/auth";
import { auth } from "../../firebase.js";

const Login = () => {
    const [err,setErr] = useState(false)
    const navigate = useNavigate()
    const handleSubmit = async (e) => {
        e.preventDefault()
        const email = e.target[0].value;
        const password = e.target[1].value;

        try {
            await signInWithEmailAndPassword(auth, email, password)
            navigate("/")
        } catch (err) {
            setErr(true)
        }
    }
    return (
        <div className='formContainer'>
            <div className="formWrapper">
                <span className="logo"></span>
                <span className="title"></span>
                <form onSubmit={handleSubmit}>
                    <input type="email" placeholder="Email"/>
                    <input type="password" placeholder="Пароль"/>
                    <button>Увійти</button>
                </form>
                {err && <span>Щось пішло не так</span>}
                <p>Все ще не маєте аккаунту? <Link
to="/register">Зареєструватися</Link></p>
            </div>
        </div>
    )
};

export default Login.
```

Отже ми маємо повноцінну систему логіну з захищеним роутінгом яку можна масштабувати у майбутніх версіях додатку та доповнити досить цікавими можливостями Firebase Authentication такими як, розширення методів за якими користувач проходить реєстрацію або входить до акаунту.

3.3 Створення логіки пошуку користувачів та їх взаємодії

Мета: Написати логіку пошуку користувачів у системі та логіку додавання нових користувачів.

Тепер попрацюємо з лівою частиною додатку. Почнемо з написання логіки до компоненти Navbar.jsx. Створимо змінну currentUser яка буде слідкувати за станом користувача. Якщо користувач буде залогінений у додаток то ми використаємо дані з контексту щоб побачити фотографію, кому належить цей акаунт, щоб це реалізувати допишемо:

```
import React, {useContext} from "react";
import {signOut} from "firebase/auth"
import {auth} from "../../firebase.js";
import {AuthContext} from "../context/AuthContext.jsx";
import logoutIcon from "../img/logout.svg"

const Navbar = () => {
    const {currentUser} = useContext(AuthContext)
    return(
        <div className="navbar">
            <button onClick={()=>signOut(auth)}>
                <img src={logoutIcon} alt="logout"/>
            </button>

                <div className="user">
                    <img src={currentUser.photoURL} alt=""/>
                    <span>{currentUser.displayName}</span>

                </div>
        </div>
    )
}

export default Navbar.
```

Допишемо логіку системі пошуку яка буде шукати користувачів за їх ім’ям яке було указано при реєстрації. Після кліку по користувачу якого ми шукаємо ми будемо створювати окремий чат який буде відображатися у компоненті Chats.jsx. У кожному новому чаті базуючись на двох унікальних Id користувачів буде створюватися окремий чат в якому буде масив messages який буде складатись з повідомлень. Система буде базуватися на різних методах з Firestore:

```
import React, {useContext, useState} from "react";
import {collection, query, where, getDoc, getDocs, doc, setDoc, updateDoc,
serverTimestamp } from "firebase/firestore";
import {db} from "../../firebase.js";
import {AuthContext} from "../context/AuthContext.jsx";


const Search = () => {
    const [username, setUsername] = useState("");
    const [user, setUser] = useState(null);
    const [err, setErr] = useState(false);

    const { currentUser } = useContext(AuthContext);

    const handleSearch = async () => {
        const q = query(
            collection(db, "users"),
            where("displayName", "==", username)
        );

        try {
            const querySnapshot = await getDocs(q);
            querySnapshot.forEach((doc) => {
                setUser(doc.data());
            });
        } catch (err) {
            setErr(true);
        }
    };

    const handleKey = (e) => {
        e.code === "Enter" && handleSearch();
    };

    const handleSelect = async () => {
        //check whether the group(chats in firestore) exists, if not create
        const combinedId =
            currentUser.uid > user.uid
                ? currentUser.uid + user.uid
                : user.uid + currentUser.uid;
        try {
            const res = await getDoc(doc(db, "chats", combinedId));

            if (!res.exists()) {
                //create a chat in chats collection
                await setDoc(doc(db, "chats", combinedId), { messages: [] });

                //create user chats
                await updateDoc(doc(db, "userChats", currentUser.uid), {
                    [combinedId + ".userInfo"]: {
                        uid: user.uid,
                        displayName: user.displayName,
                        photoURL: user.photoURL,
                    },
                    [combinedId + ".date"]: serverTimestamp(),
                });

                await updateDoc(doc(db, "userChats", user.uid), {
                    [combinedId + ".userInfo"]: {
                        uid: currentUser.uid,
                        displayName: currentUser.displayName,
                        photoURL: currentUser.photoURL,
                    },
                    [combinedId + ".date"]: serverTimestamp(),
```

```
                });
            }
        } catch (err) {}

        setUser(null);
        setUsername("")
    };
    return (
        <div className="search">
            <div className="searchForm">
                <input
                    type="text"
                    placeholder="Знайти користувача"
                    onKeyDown={handleKey}
                    onChange={(e) => setUsername(e.target.value)}
                    value={username}
                />
            </div>
            {err && <span>Користувача не знайдено</span>}
            {user && (
                <div className="userChat" onClick={handleSelect}>
                    <img src={user.photoURL} alt="" />
                    <div className="userChatInfo">
                        <span>{user.displayName}</span>
                    </div>
                </div>
            )}
        </div>
    );
};

export default Search.
```

Треба виводити чати користувачів яких ми обрали. для цього допишемо логіку до модулю Chats.jsx:

```
import React, {useContext, useEffect, useState} from "react";
import { doc, onSnapshot } from "firebase/firestore";
import {AuthContext} from "../context/AuthContext.jsx";
import {ChatContext} from "../context/ChatContext.jsx";
import {db} from "../../firebase.js";
const Chats = () => {
    const [chats, setChats] = useState([]);

    const { currentUser } = useContext(AuthContext);
    const { dispatch } = useContext(ChatContext);

    useEffect(() => {
        const getChats = () => {
            const unsub = onSnapshot(doc(db, "userChats", currentUser.uid),
(doc) => {
                setChats(doc.data());
            });

            return () => {
                unsub();
            };
        };
```

```
        currentUser.uid && getChats();
    }, [currentUser.uid]);

    const handleSelect = (u) => {
        dispatch({ type: "CHANGE_USER", payload: u });
    };

    return (
        <div className="chats">
            {Object.entries(chats)?.sort((a,b) => b[1].date -
a[1].date).map((chat) => (
                <div
                    className="userChat"
                    key={chat[0]}
                    onClick={() => handleSelect(chat[1].userInfo)}
                >
                    <img src={chat[1].userInfo.photoURL} alt="" />
                    <div className="userChatInfo">
                        <span>{chat[1].userInfo.displayName}</span>
                        <p>{chat[1].lastMessage?.text}</p>
                    </div>
                </div>
            ))}
        </div>
    );
};
export default Chats.
```

Ми повністю доповнили лівий модуль додатку. Тепер система пошуку та додавання нового чату і вибір між ними повністю працює. Також реалізована функція виходу з аккаунту яка стирає дані пристрою, коли AuthContext.jsx не бачить користувача відправляє його на сторінку з логіном. Тож увесь функціонал.

3.4 Створення логіки обміну даними між користувачами

Мета: Написати логіку чату між двома користувачами, забезпечити можливим обмін повідомленнями та фотографіями між ними.

Щоб наш чат міг оновлюватися у реальному часі нам потрібно постійно дивитися на контекст при кожній зміні контенту перемальовувати UI додатку. Тож як і з AuthContext.jsx нам потрібен ще один контекст який буде слідкувати за зміною state у чатах між користувачами. Тож давайте створимо новий контекст який назвемо ChatContext.jsx туди запишемо таку логіку:

```
import {createContext, useEffect, useReducer, useState,useContext} from
"react";
import { auth } from "../../firebase.js";
import { onAuthStateChanged } from "firebase/auth"
import {AuthContext} from "./AuthContext.jsx";
import chat from "../Components/Chat.jsx";
export const ChatContext = createContext()


export const ChatContextProvider = ({children}) => {
    const {currentUser} = useContext(AuthContext)
    const INITIAL_STATE = {
        chatId:"null",
        user: {}
    }

    const chatReducer = (state, action) => {
        switch (action.type) {
            case "CHANGE_USER":
                return {
                    user: action.payload,
                    chatId:
                        currentUser.uid > action.payload.uid
                            ? currentUser.uid + action.payload.uid
                            : action.payload.uid + currentUser.uid,
                };

            default:
                return state;
        }
    };

    const [state, dispatch] = useReducer(chatReducer, INITIAL_STATE);

    return (
        <ChatContext.Provider value={{ data:state, dispatch }}>
            {children}
        </ChatContext.Provider>
    )

}.
```

Цей контекст трохи іншим чином слідкує за станом наших чатів, він взаємодіє з AuthContext.jsx постійно слідкуючи за станом. Тож передаємо цей контекст у нашу компоненту Chat.jsx, запишемо:

```
import React, {useContext} from "react";
import Messages from "./Messages.jsx";
import Input from "./Input.jsx";
import {ChatContext} from "../context/ChatContext.jsx";

const Chat = () => {
    const { data } = useContext(ChatContext);

    return (
        <div className="chat">
            <div className="chatInfo">
                 <span>{data.user?.displayName}</span>
```

```
                <div className="chatIcons">
                    <img src="../img/cam.png" alt="" />
                </div>
            </div>
            <Messages />
            <Input/>
        </div>
    );
};

export default Chat.
```

Реалізуємо блок повідомлень у якому будемо перебирати повідомлення у масиві та виводити їх у наш UI. Перейдемо до компоненти Messages.jsx

```
import React, {useContext, useState, useEffect} from "react";
import Message from "./Message.jsx";
import {ChatContext} from "../context/ChatContext.jsx";
import { doc, onSnapshot } from "firebase/firestore";
import {db} from "../../firebase.js";

const Messages = () => {
    const [messages, setMessages] = useState([]);
    const { data } = useContext(ChatContext);

    useEffect(() => {
        const unSub = onSnapshot(doc(db, "chats", data.chatId), (doc) => {
            doc.exists() && setMessages(doc.data().messages);
        });

        return () => {
            unSub();
        };
    }, [data.chatId]);
    console.log(messages)

    return (
        <div className="messages">
            {messages.map((m) => (
                <Message message={m} key={m.id} />
            ))}
        </div>
    );
};

export default Messages.
```

Також допишемо компоненту Message.jsx яка пізніше буде виводитися у великий кількості. Повідомлення також мають параметри дати та часу які нам потрібно вивести. За ці параметри відповідає serverTimestamp, повідомлення фільтруються по даті щоб правильно відображатися у масиві:

```
import React, {useContext, useEffect, useRef} from "react";
import {AuthContext} from "../context/AuthContext.jsx";
import {ChatContext} from "../context/ChatContext.jsx";

const Message = ({message}) => {

    const {currentUser} = useContext(AuthContext)
    const {data} = useContext(ChatContext)

    const ref = useRef()
    const date = message.date.toDate().getDate() + '.' +
(message.date.toDate().getMonth() + 1) + '.' +
message.date.toDate().getFullYear()
    const time = message.date.toDate().getHours() + ':' +
message.date.toDate().getMinutes()
        // + ':' + message.date.toDate().getSeconds()
    useEffect(() => {
        ref
    }, []);

   return(
        <div className={`message ${message.senderId === currentUser.uid &&
"owner"}`}>
            <div className="messageInfo">
                <img src={message.senderId === currentUser.uid ?
currentUser.photoURL : data.user.photoURL}
                     alt=""/>
                <div className="messageDate">
                    <span>{date}</span>
                    <span>{time}</span>
                </div>
            </div>
            <div className="messageContent">
                <p>{message.text}</p>
                {message.img &&  <img src={message.img} alt=""/>}
            </div>
        </div>
    )
}

export default Message.
```

Допишемо логіку до компоненту Input.jsx , також реалізуємо завантаження фотографій у Firebase storage. Для цього компоненту нам також знадобиться uuid. Спочатку завантажимо його через npm:

$npm install uuid

Тепер імпортуємо до проекту та допишемо модуль завантаження картинки, модуль оновлення масиву даних повідомленнями з власним uniqueID. З прив’язкою до часу який ми раніше конвертували у JavaScript в компоненті Message.jsx, напишемо код:

```
import React, {useContext, useState} from "react";
import {AuthContext} from "../context/AuthContext.jsx";
import {ChatContext} from "../context/ChatContext.jsx";
import add_photo from "../img/add_photo.svg";
```

```
import {
    arrayUnion,
    doc,
    serverTimestamp, setDoc,
    Timestamp,
    updateDoc,
} from "firebase/firestore";
import {db, storage} from "../../firebase.js";
import {v4 as uuid} from "uuid"
import { getDownloadURL, ref, uploadBytesResumable } from "firebase/storage";
import {updateProfile} from "firebase/auth";


const Input = () => {
    const [text, setText] = useState("");
    const [img, setImg] = useState(null);

    const { currentUser } = useContext(AuthContext);
    const { data } = useContext(ChatContext);

    const handleSend = async () => {
        if (img) {
            const storageRef = ref(storage, uuid());

            const uploadTask = uploadBytesResumable(storageRef, img);

            uploadTask.on(
                (error) => {
                    //TODO:Handle Error
                },
                () => {
                    getDownloadURL(uploadTask.snapshot.ref).then(async
(downloadURL) => {
                        await updateDoc(doc(db, "chats", data.chatId), {
                            messages: arrayUnion({
                                id: uuid(),
                                text,
                                senderId: currentUser.uid,
                                date: Timestamp.now(),
                                img: downloadURL,
                            }),
                        });
                    });
                }
            );
        } else {
            await updateDoc(doc(db, "chats", data.chatId), {
                messages: arrayUnion({
                    id: uuid(),
                    text,
                    senderId: currentUser.uid,
                    date: Timestamp.now(),
                }),
            });
        }

        await updateDoc(doc(db, "userChats", currentUser.uid), {
            [data.chatId + ".lastMessage"]: {
                text,
            },
            [data.chatId + ".date"]: serverTimestamp(),
        });

        await updateDoc(doc(db, "userChats", data.user.uid), {
```

```
            [data.chatId + ".lastMessage"]: {
                text,
            },
            [data.chatId + ".date"]: serverTimestamp(),
        });

        setText("");
        setImg(null);
    };

    return (
        <div className="input">
            <input
                type="text"
                placeholder="Напишіть щось..."
                onChange={(e) => setText(e.target.value)}
                value={text}
            />
            <div className="send">
                <input
                    type="file"
                    style={{ display: "none" }}
                    id="file"
                    onChange={(e) => setImg(e.target.files[0])}
                />
                <label htmlFor="file">
                    <img src={add_photo} alt="Додати фотографію" />
                </label>
                <button onClick={handleSend}>надіслати</button>
            </div>
        </div>
    );
};

export default Input.
```

Отже ми повністю реалізували функціонал додатку. Захищена авторизація, пошук користувачів, створення нових чатів, оновлення чатів у реальному часі без перезавантаження сторінки, можливість відправляти фотографії та повідомлення різної довжини, також реалізовано функціонал правильного відображення повідомлень які ми копіюємо та заносимо у поле введення, при відправленні вони правильно відображають повідомлення різної довжини та конотексту. Структура бази даних наведена на рис. 3.7

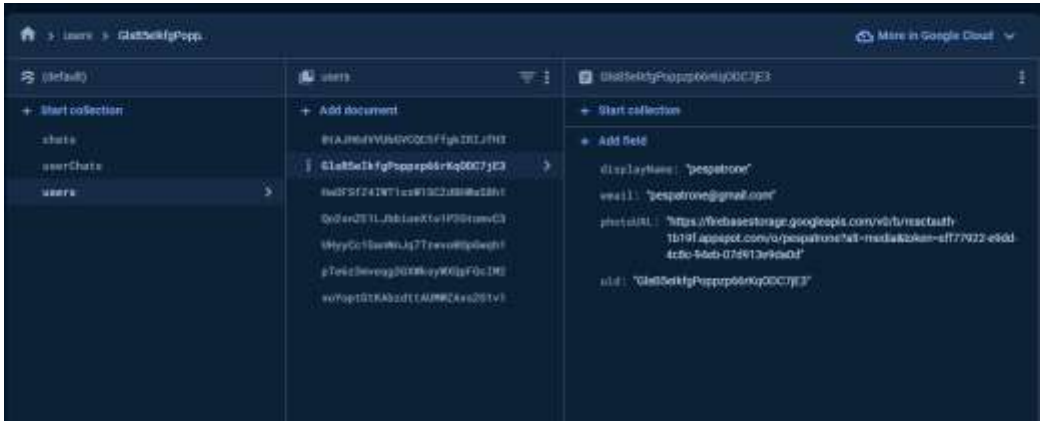

Рисунок 3.7 – Створення Local Storage

Як ми можемо побачити під час розробки усі колекції створилися в правильну структуру та мають унікальні ідентифікатори та props.

3.5 Тестування роботи додатку та бази даних.

Мета: Протестувати роботу додатку створивши двох нових користувачів та забезпечивши між ними обмін повідомленнями.

Перейдемо до тестування роботи додатку та бази даних шляхом створення двох користувачів, створення їх чату та тесту обміну різними варіантами повідомлень.

Для початку перевіримо як працює реєстрація нового користувача. Заповнимо дві форми та додамо різні картинки щоб легше було розлічати між собою. Записуємо у різні форми два різних акаунти. Натискаємо зареєструватися. Приклад заповненої форми на рис. 3.8.

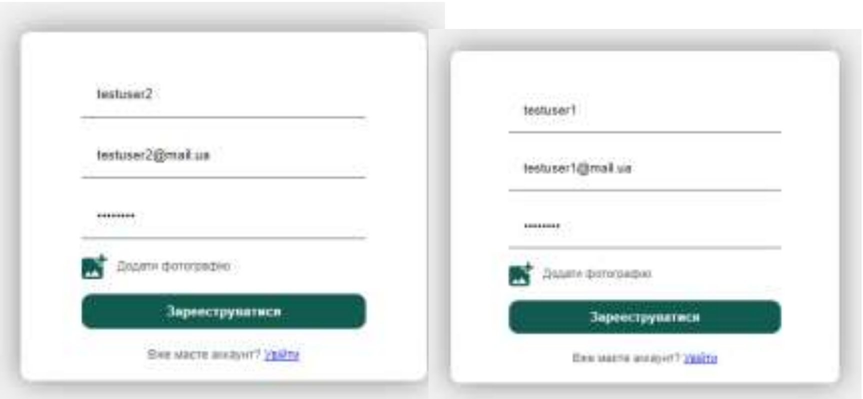

Рисунок 3.8 – Приклад реєстрації двох користувачів

Після натискання кнопки дані було відправлено. Натискаємо кнопку «зареєструватися». Якщо дані що ми ввели нові то нам відкриється нове вікно користувача у якому ми вже будемо мати унікальне Id користувача і зможемо почати роботу з додатком. Якщо наш акаунт вже зареєстровано ми отримаємо повідомлення о помилці реєстрації. Коли наш id підходить по даним, React автоматично перерисовує потрібні компоненти не перезавантажуючи додаток.
Результат роботи додатку приведено на рис. 3.9.

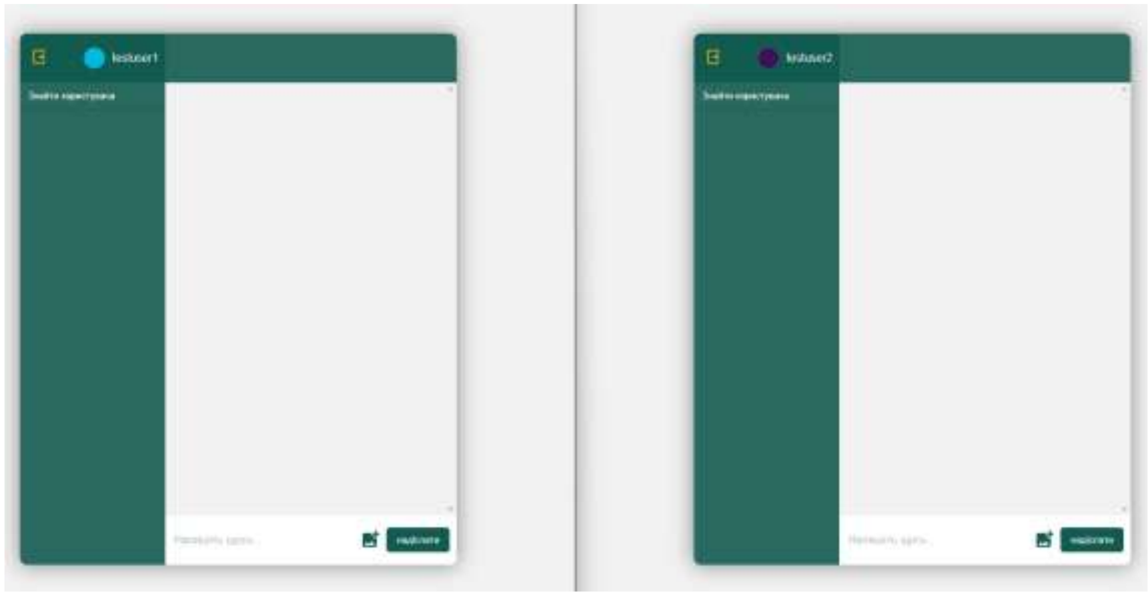

Рисунок 3.9 – Результат роботи додатку після реєстрації

У консолі ми можемо побачити дані з AuthContext.jsx, також можемо побачити ім'я, пошту, uid та інші важливі дані. Якщо замість даних будемо мати null то користувач буде відправлений на головну сторінку, якщо це null то користувач буде відправлений на сторінку логіну приклад даних наведено на рис. 3.10.

Рисунок 3.10 – Дані користувача які використовуються для авторизації

Тепер подивимося у firebase, у вкладці authentication будемо бачити зареєстрованих користувачів додатку, також зможемо подивитись дату їх реєстрації, унікальний Id та останню дату логіну. Як ми бачимо наші 2 тестових акаунти потрапили до бази даних вони є у списку на рис 3.11.

Рисунок 3.11 – Список користувачів в Firebase authentication

Так як ми додатково вносили параметри користувача до БД нам треба перевірити чи були ці користувачі записані туди переходимо до Firestore

Database та шукаємо uid який лежить у колекції users, усі користувачі були записані у БД можемо побачити це на рис. 3.12.

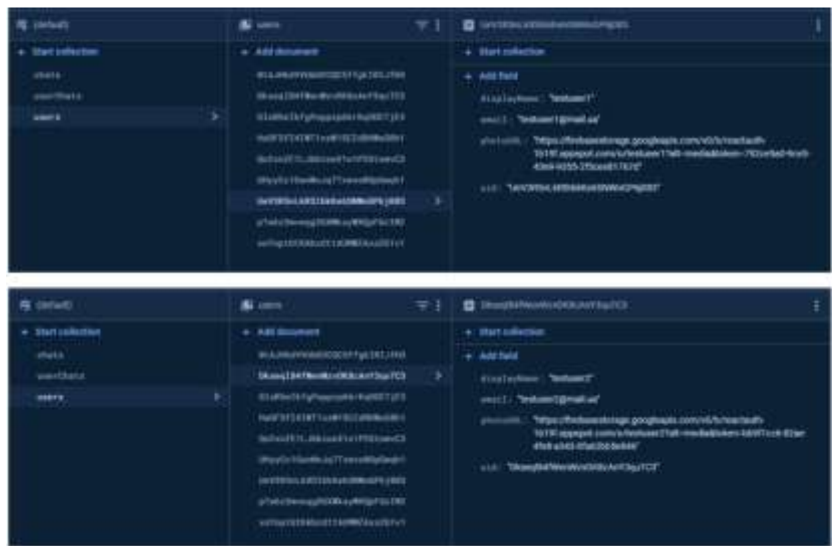

Рисунок 3.12 – Відображення користувачів у Firestore Database.

Спробуємо пошукати різних користувачів. На першому користувачі я буду шукати іншого, а на другому ще один тестовий акаунт, вбиваємо ім’я користувача та бачимо що користувачів знайдено, приклад роботи наведено на рис 3.13.

Рисунок 3.13 – Приклад роботи пошукової системи додатку

Натиснемо на чат першим користувачем, як ми можемо побачити чат з’явився у обох користувачів. Також чат одразу було занесено до БД де створилася окрема таблиця яка показує які користувачі мають чати один з іншим , приклад як це виглядає візуально наведено на рис 3.14.

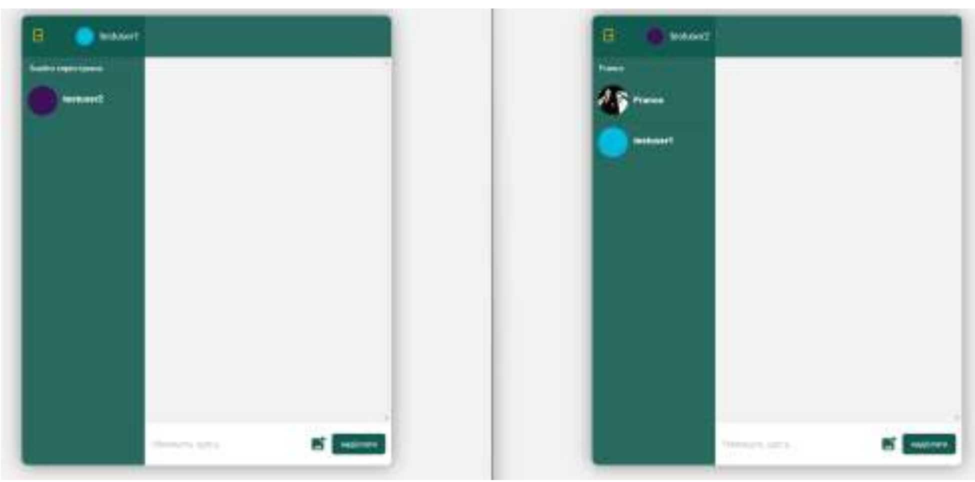

Рисунок 3.14 – Приклад додавання нового чату

Якщо ми подивимося у БД то побачимо що такий чат було створено, але масив повідомлень ще пустий. Напишемо різні повідомлення з обох акаунтів та відправимо фотографії подивимося як себе поводять більш великі повідомлення, приклад роботи чату на рис 3.15.

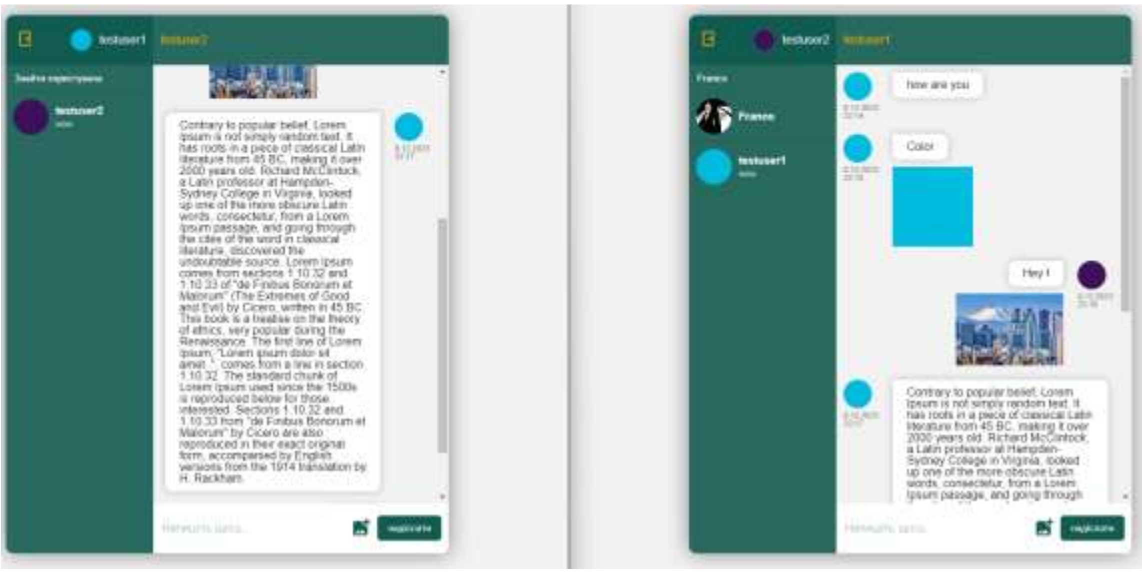

Рисунок 3.15 – Приклад діалогу в чатах обох користувачів

Бачимо що все відображається правильно. Структура блоку повідомлення не дозволяє йому виходити за рамки, фотографії автоматично

підстроюються та не виходять за максимально заданий розмір, дата та час на повідомленнях відображаються правильно, зліва у модулю зі списком чату можемо побачити чат та останнє повідомлення у цьому чаті.

Тепер подивимося як це виглядає у БД, перейдемо до Firestore Database та виберемо колекцію userChats, далі нам треба знайти серед всіх унікальних чатів, чат в якому будуть потрібні користувачі. Можемо побачити параметри дати, останнього повідомлення, ім’я користувачів та їх uid. Приклад відображення у БД наведено на рис 3.16

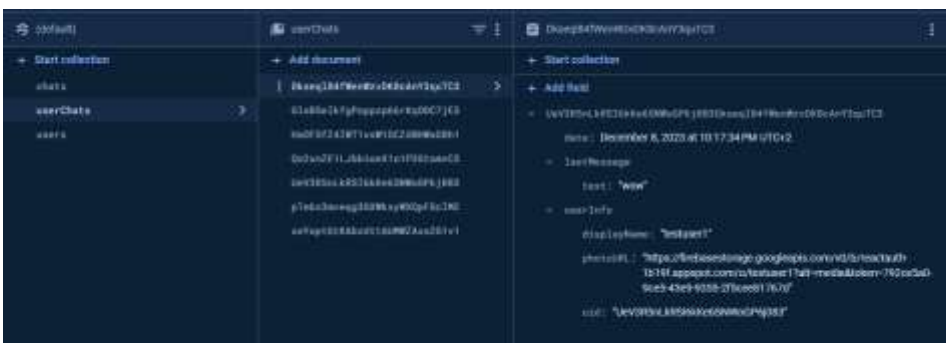

Рисунок 3.16 – Відображення чату користувачів у Firestore Database.

Ми побачили які параметри має чат у БД тож тепер можемо подивитись як виглядає масив повідомлень, перейдемо до Firestore Database та виберемо колекцію chats. Основними параметрами повідомлення є його дата, унікальний ідентифікатор повідомлення, посилання на фото, id користувача який відправив це повідомлення та його контент. Кожний такий об’єкт повідомлення є окремим елементом масиву messages. Тож на кожний унікальний id чату є унікальний масив повідомлень який оновлюється у реальному часі, також останнє повідомлення у чаті передається до масивів чату щоб після цього бути відображеним у модулю з чатами. Приклад відображення у БД наведено на рис 3.17 та 3.18

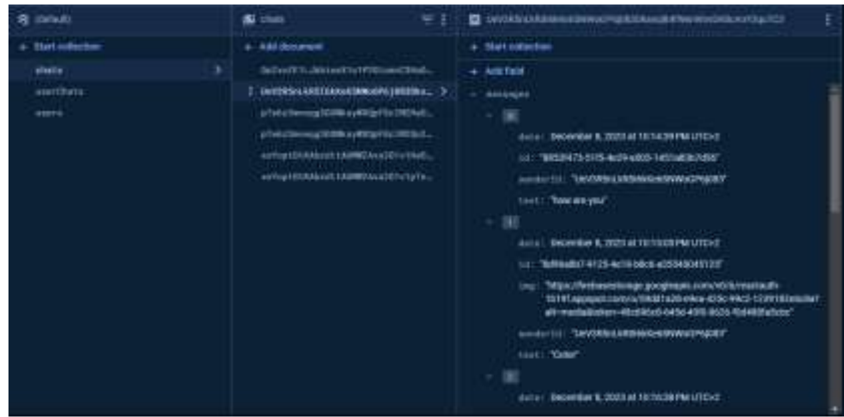

Рисунок 3.17 – Відображення повідомлень у Firestore Database.

Рисунок 3.18 – Відображення повідомлень у Firestore Database.

При тестуванні чату та реєстрації користувача ми відправляли фото. Ці можна знайти у вкладці storage. Фото які користувачі відправляли при реєстрації акаунту будуть названі ім’ям користувача, фото які були відправлені в чатах будуть названі як id повідомлення. Тож перейдемо до Firebase Storage де ми бачимо ім’я картинки, її розмір, тип, та дату відправлення. Приклад наведено на рис 3.19

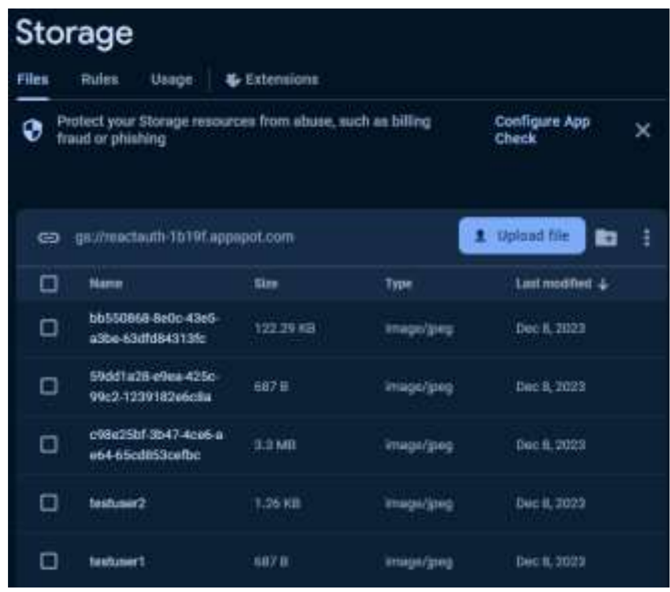

Рисунок 3.19 – Відображення фото у Firebase Storage.

У цьому розділі ми дописали внутрішню логіку додатку, підключили систему авторизації по пошті яка перевіряє чи існують акаунти або створює нові. Створили та налаштували роботу баз даних для користувачів, доступних чатів користувача та масиву повідомлень. Налаштували збереження фотографій у хмарному сервісі та реалізували можливість відправляти їх у чаті. Отримали працюючий додаток який добре виконує усі свої функції починаючи від авторизації користувачів та додавання чатів, закінчуючи збереженням повідомлень у масивах з оновленням у реальному чаті. Це можливо завдяки функціям React та Firebase, у майбутньому цей додаток можна розширювати та доробляти майже необмеженим функціоналом.

ВИСНОВКИ

В ході дослідження було проведено аналіз та реалізація створення SPA-додатку з використанням Javascript-бібліотеки React. Проект включав інтеграцію функцій авторизації та створення чатів, забезпечуючи повноцінний обмін повідомленнями між користувачами. Дані у чаті автоматично оновлювалися при будь-яких змінах, а також надавалася можливість відправляти фотографії та інші файли.

Весь процес розробки відбувався в середовищі редактора WebStorm, використовуючи стандартний термінал PowerShell та файловий менеджер npm. Версія React на час розробки: 18.2.0, версія Firebase для JavaScript: 10.7.1.

Firebase виявилася надзвичайно зручною та ефективною для інтеграції в мій React-проект, оптимізований за допомогою Vite. Під час процесу інтеграції не виникло жодних проблем чи труднощів, все пройшло дуже плавно та без помилок.

Порівнюючи стандартний SQL з NoSQL базою даних у Firebase, слід відзначити, що NoSQL виявилася не тільки зручною, але й дуже потужною з вражаючим функціоналом. Це надає більшу гнучкість у роботі з даними та значно полегшує їхню обробку в процесі розробки.

Використання додаткових пакетів значно спростило розробку, дозволяючи покращити продуктивність та забезпечити безперервну роботу програми. Ці пакети додають до проекту додатковий функціонал, не ускладнюючи при цьому код та дозволяючи зосередитися на основному завданні розробки.

В цілому використання Firebase та його модулів у поєднанні з React та Vite стало важливим етапом у створенні продуктивного та функціонального додатку.

Прив’язав деякий функціонал Firebase такий як: firestore, authentication та storage до react додатку було реалізовано авторизацію, створення нових та

вибір існуючих чатів повідомлення у яких оновлюються реальному часі у всіх користувачів та завантаження файлів і їх збереження у хмарі.

На мою думку для невеликих додатків React є найзручнішим інструментом для створення невеликих додатків у сфері веб та мобільного розроблення. Його розгалужений функціонал дозволяє необмежено розширювати можливості та створювати надзвичайно складні компоненти. Бібліотека продовжує активно оновлюватися та впроваджувати нові можливості, роблячи її перспективним вибором для створення додатків. Щоразу, коли відбувається мажорне оновлення (кожні два роки), React отримує ще більше функцій та покращень, що додає йому конкурентні переваги в ринку розробки програмного забезпечення. Таким чином, його стабільність та актуальність роблять його перевіреним і ефективним інструментом для розробників, які шукають зручність та високий рівень функціональності в своїх проектах.

Під час теоретичного дослідження було висвітлено такі ключові аспекти: переваги односторінкових додатків (SPA) над звичайними веб-сайтами, важливість модульності у порівнянні із традиційною структурою, а також переваги використання NoSQL баз даних і сервісів порівняно з застарілими SQL.

Отримані результати можуть слугувати основою для аналізу недоліків застарілих підходів у веб-розробці. Використання SPA з JavaScript-бібліотекою React та інтеграція з Firebase та NoSQL базами даних в WEB-просторі відкриває можливості для поліпшення існуючих веб-додатків або створення нових.

Спостерігається потенціал для подальшого розвитку у напрямку створення адаптивних додатків. Цей напрямок має значні перспективи, оскільки відзначається тенденцією до використання більш простих і ефективних рішень. Майбутнє визначається спрощенням використання для всіх користувачів без додаткового навантаження на їх систе

ПЕРЕЛІК ПОСИЛАНЬ

1. С.М. Порошин, В.М. Карташов, В.В. Усик, Р.І. Цехмістро, І.С.Бєліков. Технології створення складових мультимедійного контенту. Анімація та web-анімація. Навчальний посібник –Харків, НТУ.ХПІ, 2022.- 314с.

2. М.А. Омаров Основи технологій сучасної web-анімації. [Текст] Навчальний посібник /М.А. Омаров, В.М. Карташов, Р.І. Цехмістро, В.В. Усик – Харків ХНУРЕ- 2022р.- 214с

3. Хольцшлаг Моллі. Використання HTML 4. Пер з англ.: навч. допомога. М: Вид. будинок "Вільямс", 2001. -1008 с.

4. Томсон Л., Веллінг Л. Розробка Web-додатків на PHP та MySQL: Пер. з англ. - К.: Вид-во "ДіаСофт", 2002. - 672 с.

5. Порселло Е., Бенкс А. React: сучасні шаблони для розробки додатків 2-е видання: Вид." O'Reilly", 2022. – 360 с.

6. Хоффман Е. Безпека веб-додатків: Вид." O'Reilly", 2021. – 336 с.

7. Адам С. Розробка на JavaScript. Побудова кросплатформених додатків за допомогою GraphQL, React, React Native і Electron: Вид. " O'Reilly ", 2021. -320 с.

8. Крис А, Тодд Г. Front-end. Клієнтська розробка для професіоналів. Node.js, ES6, REST: Вид. " O'Reilly ", 2017. – 520 с.

9. Juntao Qiu. Test-Driven Development with React: Apply Test-Driven Development in Your Applications. 2021. – 220p.

10. Nabendu Biswas. Beginning React and Firebase: Create Four Beginner-Friendly Projects Using React and Firebase. 2022. – 184p.