МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

КВАЛІФІКАЦІЙНА МАГІСТЕРСЬКА РОБОТА

на тему:

«Чат додаток з використанням React JS + Node JS»

Завідувач випускаючої кафедри **Довбиш А.С.**

Керівник роботи **Петров С.О.**

Студента групи ІН.мз – 91с **Крамаренко В.Ю.**

СУМИ 2020

_____

(назва вузу)

Факультет _____ Кафедра _____

Спеціальність _____

Затверджую:

зав.кафедрою _____

_____

“_____” _____ 20___р.

ЗАВДАННЯ

НА ДИПЛОМНИЙ ПРОЕКТ (РОБОТУ) СТУДЕНТОВІ

_____

(прізвище, ім’я, по батькові)

1. Тема проекту (роботи)_____

_____

затверджую наказом по інституту від “____” _____ 20 ___ р. № ___

2. Термін здачі студентом закінченого проекту (роботи)

_____

3. Вхідні данні до проекту (роботи)

_____

_____

_____

_____

_____

4. Зміст розрахунково-пояснювальної записки (перелік питань, що їх належить розробити)

_____

_____

_____

_____

_____

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

_____

_____

_____

_____

_____

6. Консультанти до проекту (роботи), із значенням розділів проекту, що стосується їх

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання

_____

Керівник _____

(підпис)

Завдання прийняв до виконання _____

(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів дипломного проекту (роботи)	Термін виконання проекту (роботи)	Примітка

Студент – дипломник _____

(підпис)

Керівник проекту _____

(підпис)

РЕФЕРАТ

Записка: 46 стор., 27 рис., 9 джерел.

Об’єкт дослідження — Чат додаток з використанням React JS та Node JS.

Мета роботи — Розробка чату для спілкування в режимі реального часу

Методи дослідження — метод функціонально-статистичних випробувань.

Результати — Розроблено чат додаток з використанням React JS та Node, також допоміжними системами виступають socket.io та в якості бази даних використано MongoDB. За допомогою всіх цих технологій реалізовано чат в якому можна отримувати та відправляти повідомлення в режимі реального часу.

ЗМІСТ

ВСТУП

В епоху стрімкого розвитку соціальних медіа безперервне формування глобальних процесів і створення нових форм соціальної інтеграції є повсякденною картиною дня. Можливість власного контенту та обміну інформацією різного характеру в режимі реального часу дозволяє соціальним медіа стати невід'ємною частиною життя людини, набуваючи нових видів та форм.

Вплив мережі Інтернет на різні аспекти соціальної взаємодії на сьогоднішній день незаперечно. Можливості мобільних додатків дозволяють інтернет-користувачам споживати і передавати великий обсяг інформації в режимі реального часу за допомогою своїх смартфонів. Тенденції, що формуються в рамках широкого поширення інформаційних технологій, говорять про актуальні сьогодні мобільних додатках - месенджерах.

Месенджери - це новий спосіб комунікації між людьми, незалежно від географічних особливостей, за допомогою обміну миттєвими повідомленнями. В умовах розвитку Інтернету як публічного простору з можливістю доступу до різної персональної інформації у користувачів виникає потреба в усвідомленому споживанні і приватноій взаємодії. Тому і зростає актуальність серед додатків, що надають подібні можливості.

1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Перед тим як почати розробку нашого додатку, спочатку потрібно визначитись за допомогою чого ми будемо це робити. В роботі обрано такі системи та бібліотеки як React JS, Node JS, MongoDB та socket.io. React JS дозволить нам, завдяки своїм можливостям створити неперевершений інтерфейс, Node JS буде виступати в ролі сервера, socket.io дозволить нам обмінюватися повідомленнями в реальному часі, а всі наші дані буде зберігати MongoDB. Далі ми розглянемо всі ці інструменти більш детально

2. ОПИС ВИКОРИСТАНИХ ІНСТРУМЕНТІВ ПІД ЧАС РОБОТИ

2.1. React JS

React - це JavaScript-бібліотека для розробки інтерфейсу. React спочатку був спроектований так, щоб його можна було впроваджувати поступово. Іншими словами, ми можемо почати з малого і використовувати тільки ту функціональність React, яка необхідна нам в даний момент. Інформація в цьому розділі буде корисна в будь-якій ситуації: при першому знайомстві з React, при створенні простої динамічної HTML-сторінки і навіть при проектуванні складного React-додатка.

Для впровадження React не треба нічого переписувати. Його можна використовувати як для маленької кнопки, так і для цілого додатка. Можливо, ми захочемо трохи оживити нашу сторінку. React-компоненти підходять для цього якнайкраще. Більшість сайтів в Інтернеті є звичайними HTML-сторінками. Навіть якщо ваш сайт не відноситься до односторінкових додатків, ви можете додати на нього React, написавши всього кілька рядків коду без будь-яких інструментів збірки. Залежно від цілей, можна поступово перенести на React весь сайт або переписати лише кілька віджетів.

Властивості передаються від батьківських компонентів до дочірніх. Компоненти отримують властивості як безліч незмінних (англ. Immutable) значень, тому компонент не може безпосередньо змінювати властивості, але може викликати зміни через callback-функції. Такий механізм називають «властивості вниз, події вгору».

React використовує віртуальний DOM (англ. Virtual DOM). React створює кеш-структуру в пам'яті, що дозволяє обчислювати різницю між попереднім і поточним станами інтерфейсу для оптимального оновлення DOM браузера. Таким чином програміст може працювати зі сторінкою, вважаючи, що вона оновлюється вся, але бібліотека самостійно вирішує, які компоненти сторінки необхідно оновити. React використовується не тільки для відтворення HTML в браузері. Наприклад, Facebook має динамічні графіки, які знаходяться в тезі <canvas>. Netflix і PayPal використовують ізоморфні завантаження для відтворення ідентичного HTML на сервері і клієнті.

Компонент React - це, якщо по-простому, ділянка коду, який представляє частину веб-сторінки. Кожен компонент - це JavaScript-функція, яка повертає шматок коду, що представляє фрагмент сторінки. Для формування сторінки ми викликаємо ці функції в певному порядку, збираємо разом результати викликів і показуємо їх користувачеві. React використовує мову програмування JSX, яка схожа на HTML, але працює всередині JavaScript, що відрізняє її від HTML.

Класи компонентів повинні містити функцію, звану render (). Ця функція повертає JSX-код компонента. Їх можна використовувати так само, як функціональні компоненти, наприклад, звертаючись до них за допомогою конструкцій. У тому випадку, якщо вас цікавлять компоненти без стану, перевагу слід віддати функціональним компонентам, їх, зокрема, легше читати.

React може використовуватися для розробки односторінкових і мобільних додатків. Його мета - надати високу швидкість, простоту і масштабованість. Як бібліотеки для розробки призначених для користувача інтерфейсів React часто використовується з іншими бібліотеками, такими як MobX, Redux і GraphQL.

Також існує React Native. За назвою може здатися, що це той самий інструмент, але це не так. Для чого потрібен React Native - для створення мобільних додатків, що створюються під обидві популярні платформи (iOS і Android) одночасно. Обидві версії програми будуть відповідати рекомендаціям Apple і Google, при цьому розробка відбувається швидше і задіє одну і ту ж команду розробників. Тому розробка додатків на React Native дозволяє досягти максимальних результатів в найкоротші терміни.

2.2. Node JS

Node або Node.js - програмна платформа, заснована за допомогою двигунаV8 (здійснює трансляцію JavaScript в машинний код), що перетворює

JavaScript з вузькоспеціалізованої мови в мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями введення-виведення через свій API який написаний на C ++, підключати інші зовнішні бібліотеки, написані на різних мовах, забезпечуючи виклики до них з JavaScript-коду. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні додатки. (за допомогою NW.js, AppJS або Electron для Linux, Windows і macOS) і навіть програмувати мікроконтролери (наприклад, tessel, low.js і espruino). В основі Node.js лежить орієнтоване і асинхронне (або реактивне) програмування з неблокуючим введенням / виведенням.

Для розробки під Node JS досить найпростішого текстового редактора, зокрема, Notepad ++. Також можна використовувати більш витончені редактори типу Atom, Sublime, Visual Studio Code, або середовища розробки, які підтримують роботу з Node.JS, наприклад, Visual Studio або WebStorm.

З Node простіше масштабироваться. При одночасному підключенні до сервера тисяч користувачів Node працює асинхронно, тобто ставить пріоритети і розподіляє ресурси грамотніше. Java ж, наприклад, виділяє на кожне підключення окремий потік.

2.3. Socket.io

Socket.IO - JavaScript-бібліотека для веб-додатків і обміну даними в реальному часі. Складається з двох частин: клієнтської, яка запускається в браузері і серверної для node.js. Обидва компоненти мають схожу API. Подібно node.js, Socket.IO орієнтована. Socket.IO головним чином використовує протокол WebSocket, але якщо потрібно, використовує інші технології, наприклад Flash Socket, AJAX Long Polling, AJAX Multipart Stream, надаючи той же самий інтерфейс. Крім того, що Socket.IO може бути використана як оболонка для WebSocket, вона містить багато інших функцій, включаючи мовлення на кілька гнізд, зберігання даних, пов'язаних з кожним

клієнтом, і асинхронний ввід / вивід. Може бути встановлена через npm (node package manager).

За допомогою Socket.IO можна реалізувати аналітику в реальному часі, розраховані на багато користувачів гри, обмін миттєвими повідомленнями і спільну роботу з документами в реальному часі.

Socket.IO досить популярний, його використовують Microsoft, Yammer, Zendesk, Trello і багато інших організацій для створення систем реального часу.

Socket.IO полегшує життя, нам не потрібно піклуватися про проблеми, пов'язані з балансуванням навантаження, розривом з'єднань або розсилкою повідомлень, але також клієнтська бібліотека Socket.IO важить більше, ніж пакети React, Redux і React-Redux разом узяті.

Основний простір імен «/» є простором імен за замовчуванням, до якого приєднуються клієнти, якщо клієнтський простір вказує на простір імен при підключенні до сервера. Всі підключення до сервера з використанням клієнтської сторони об'єкта сокета виконуються в просторі імен за замовчуванням

2.4. MongoDB

MongoDB - документоорієнтована система управління базами даних, яка не потребує опису схеми таблиць. Вважається одним з класичних прикладів NoSQL-систем, використовує JSON-подібні документи і схему бази даних. Застосовується в веб-розробці, зокрема, в рамках JavaScript-орієнтованого стека MEAN. MongoDB реалізує новий підхід до побудови баз даних, де немає таблиць, схем, запитів SQL, зовнішніх ключів і багатьох інших речей, які притаманні об'єктно-реляційним базам даних.

Система підтримує ad-hoc-запити: вони можуть повертати конкретні поля документів і призначені для користувача JavaScript-функції. Підтримується

пошук за регулярними виразами. Також можна налаштувати запит на повернення випадкового набору результатів. Є підтримка індексів.
На відміну від реляційних баз даних MongoDB пропонує документо-орієнтовану модель даних, завдяки чому MongoDB працює швидше, має кращу масштабованість, а також її легше використовувати.

Але, навіть враховуючи всі недоліки традиційних баз даних і плюси MongoDB, важливо розуміти, що завдання бувають різні і методи їх вирішення бувають різні. В якійсь ситуації MongoDB дійсно поліпшить продуктивність вашої програми, наприклад, якщо треба зберігати складні за структурою дані. В іншій же ситуації краще буде використовувати традиційні реляційні бази даних. Крім того, можна використовувати змішаний підхід: зберігати один тип даних в MongoDB, а інший тип даних - в традиційних БД.

Вся система MongoDB може представляти не тільки одну базу даних, що знаходиться на одному фізичному сервері. Функціональність MongoDB дозволяє розташувати кілька баз даних на декількох фізичних серверах, і ці бази даних зможуть легко обмінюватися даними і зберігати цілісність.

MongoDB написана на C ++, тому її легко перенести на найрізноманітніші платформи. MongoDB може бути розгорнута на платформах Windows, Linux, MacOS. Можна також завантажити вихідний код і самому скомпілювати MongoDB, але рекомендується використовувати бібліотеки.

Якщо реляційні бази даних зберігають рядки, то MongoDB зберігає документи. На відміну від рядків документи можуть зберігати складну за структурою інформацію. Документ можна уявити як сховище ключів і значень. Ключ являє просту мітку, з яким асоційоване певний шматок даних. Однак при всіх відмінностях є одна особливість, яка зближує MongoDB і реляційні бази даних. У реляційних СУБД зустрічається таке поняття як первинний ключ. Це поняття описує якийсь стовпець, який має унікальні значення. У MongoDB для кожного документа є унікальний ідентифікатор,

який називається _id. І якщо явно не вказати його значення, то MongoDB автоматично згенерує для нього значення. Кожному ключу створюється певне значення. Але тут також треба враховувати одну особливість: якщо в реляційних базах є чітко окреслена структура, де є поля, і якщо якесь поле не має значення, йому (в залежності від налаштувань конкретної бд) можна поставити значення NULL. У MongoDB все інакше. Якщо якомусь ключ не порівнювати значення, то цей ключ просто опускається в документі і не вживається.

3. РЕАЛІЗАЦІЯ ЧАТУ ДЛЯ СПІЛКУВАННЯ В РЕЖИМІ РЕАЛЬНОГО ЧАСУ

3.1. Створення нового проекту React JS

Давайте скористуємося CLI create-react-app для налаштування нашого додатку. Якщо на вашому комп'ютері не встановлено CLI create-response-app, виконайте наступну команду, щоб встановити його глобально. Після установки CLI create-react-app для створення нового проекту React виконайте команду нижче. Ця команда створить всі необхідні файли, завантажить всі необхідні зовнішні залежності і виконає всі налаштування за нас.

npm install -g create-react-app

Дуже важливо, щоб ви розуміли, які папки та файли ви будете створювати в цьому додатку. Ви можете побачити всі папки та файли на зображенні нижче.

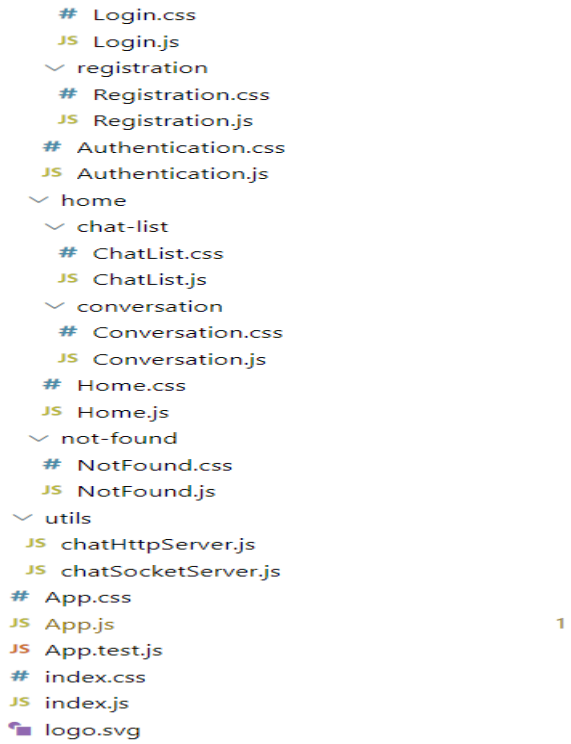

Рис. 1 – Структура папок React

Наведене вище зображення являє собою /src папку з декількома папками і файлами всередині. Ми створили кілька папок, давайте розберемося з мотивом кожного файлу і папки.

- / pages.: В цій папці знаходяться всі сторінки. У цьому додатку у нас всього дві сторінки з декількома компонентами всередині. Хоча назва кожного компонента React зрозуміло. Але тим не менш, перерахуємо кожен компонент разом з його використанням.
- Authentication Component: Ми будемо використовувати його для входу в систему і для реєстрації.
- Chat-list Component: Як випливає з назви, цю папку будемо використовувати для відображення списку чату в реальному часі.
- Conversation Component: Цей компонент використовується для відображення повідомлень.
- Home Component: Це буде компонент вузла чату і компонента бесіди.

- NotFound Component: Цей компонент буде використовуватися, коли користувач вводить невірну URL-адресу, яка не визначена в нашому додатку.
- / utils.: Ця папка містить два файли, які будуть використовуватися для виконання HTTP-запиту і подій Socket.
- ChatHttpServer class: У цьому класі ми будемо писати всі HTTP-запити. У цьому додатку ми будемо робити HTTP-виклики.
- ChatSocketServer class: У цьому класі ми напишемо пов'язаний з Socket код для отримання і відправки подій в реальному часі.

3.2. Як реагує додаток на маршрутизацію?

У цьому розділі ми налаштуємо маршрутизацію нашого додатку. Тут ми будемо використовувати response-router-dom для створення маршрутів React. Разом у нас буде два маршрути / і /home маршрут. Коли це додаток завантажується в браузері, ми перенаправляємо користувача на / маршрут. Цей маршрут відобразить сторінку аутентифікації, яка є не чим іншим, як Authentication компонентом.

```
import React, { Component } from 'react';

import {
  BrowserRouter as Router,
  Route,
  Switch
} from "react-router-dom";

import Authentication from './pages/authentication/Authentication';
import Home from './pages/home/Home';
import NotFound from './pages/not-found/NotFound';

import './App.css';

class App extends Component {
  render() {
    return (
      <Router>
        <Switch>
          <Route path="/" exact component={Authentication} />
          <Route path="/home/" component={Home} />
          <Route component={NotFound} />
        </Switch>
      </Router>
    );
  }
}

export default App;
```

Рис. 2 – App.js файл

3.3. Створення нового проекту Node JS

До цієї миті ми вже створили і закінчили налаштування додатку React. На даний момент додаток React являє собою не що інше, як чистий аркуш. У цьому розділі ми створимо шаблон API Nodejs. Чому ми використовуємо саме Node.js?

- По-перше, нам знадобиться сервер, на якому ми зможемо зберігати і отримувати дані і повідомлення користувача.
- По-друге, нам необхідно реалізувати обмін повідомленнями в реальному часі, для чого нам знадобиться веб-сокет.

- По-третє, це кращий вибір, нам взагалі не потрібно нічого вивчати, крім Javascript.

Як бачите, на зображенні нижче, в цьому додатку п'ять папок; Тут ми будемо створювати чотири папки, очікувати одну папку, тобто / node_modules. На даний момент наш додаток не дуже великій, тому структура папок дуже мінімальна і проста для розуміння.

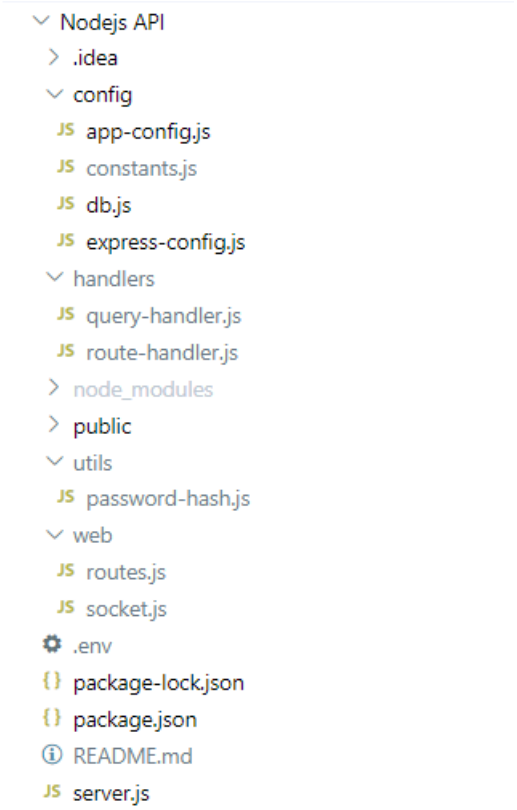

Рис. 3 – Структура папок Node

Щоб зрозуміти призначення кожної папки, розглянемо їх детальніше:

- /config: Всього в цій папці у нас є чотири файли, як показано на зображенні вище. У цих файлах ми розглянемо додаток і висловимо відповідну конфігурацію разом з підключенням до бази даних.
- /handlers: Тут ми створимо файли, які будуть включати виконання запитів MongoDB і обробники експрес-маршрутів.

- /utils: Тут ми будемо створювати файли, які в основному використовуються для виконання інших операцій. В цьому випадку хешування пароля для входу і реєстрації.
- /web: Всі Express Routes і Socket Events будуть тут.

Тепер давайте створимо проект Nodejs, виконавши команду нижче.

npm init

Ця команда створить файл package.json,. Тепер давайте подивимося на package.json


```
{
  "name": "rest-chat",
  "version": "0.0.1",
  "description": "React Chat",
  "main": "server.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [
    "rest",
    "api"
  ],
  "author": "Vladyslav Kramarenko",
  "license": "MIT",
  "dependencies": {
    "bcrypt": "^3.0.8",
    "body-parser": "^1.15.2",
    "cors": "^2.8.1",
    "dotenv": "^5.0.1",
    "express": "^4.14.0",
    "mongodb": "^2.2.19",
    "socket.io": "^1.7.2"
  }
}
```


Рис. 4 – package.json

3.4. Створення сервера Node JS

Ми знайомі зі структурою папок, перейдемо до створення сервера Nodejs. Ми запустимо свій Nodejs в server.js файлі. Переглянемо код файлу server,js.

```
'use strict';
const express = require("express");
const http = require('http');
const socketio = require('socket.io');

const socketEvents = require('./web/socket');
const routes = require('./web/routes');
const appConfig = require('./config/app-config');


class Server{

    constructor(){
        this.app = express();
        this.app.use(express.static("public"));
        this.http = http.Server(this.app);
        this.socket = socketio(this.http);
    }

    appConfig(){
        new appConfig(this.app).includeConfig();
    }

    /* Including app Routes starts*/
    includeRoutes(){
        new routes(this.app).routesConfig();
        new socketEvents(this.socket).socketConfig();
    }
    /* Including app Routes ends*/

    appExecute(){
        this.appConfig();
        this.includeRoutes();

        const port =  process.env.PORT || 4000;
        const host = process.env.HOST || `localhost`;

        this.http.listen(port, host, () => {
            console.log(`Listening on http://${host}:${port}`);
        });
    }

}

const app = new Server();
app.appExecute();
```

Рис. 5 – server.js

Сервер запускається в три етапи, які перераховані нижче:

- По-перше, додаток починається з виконання методу конструктора при створенні об'єкта Server класа.
- По-друге, в appConfig () методі ми включаємо всі свої конфігурації.
- Останній крок - ми повинні включити Routes і Socket Event.

І все це виконується, коли ми запустимо aapExecute() метод, створивши об'єкт сервера.

3.5. Створення файлів конфігурації

Цей розділ присвячений написанню файлу конфігурації, тому ми в основному будемо мати справу з файлами всередині папки /config. Почнемо з express-config.js файла. Коли наш додаток розростеться, нам потрібно буде виконати експрес-налаштування. Отже, давайте поглянемо на файл express-config.js.

```
class ExpressConfig{

    constructor(app){
        // Setting .html as the default template extension
        app.set('view engine', 'html');

        //Files
        app.use(require('express').static(require('path').join('public')));
    }
}
module.exports = ExpressConfig;
```

Рис. 6 - express-config.js.

Другий і наш основний файл конфігурації - це app-config.js файл, в якому ми запишемо всі зміни, пов'язані з додатком, а також ExpressConfig. Відкриємо app-config.js.

```
const expressConfig = require('./express-config');
const bodyParser = require('body-parser');
const cors = require('cors');
const dotenv = require('dotenv');

class AppConfig{

    constructor(app){
        dotenv.config();
        this.app = app;
    }

    includeConfig() {
        this.app.use(
            bodyParser.json()
        );
        this.app.use(
            cors()
        );
        new expressConfig(this.app);
    }

}
module.exports = AppConfig;
```

Рис. 7 - app-config.js

Оскільки додаток використовує .env файл для зберігання конфігурацій, ми отримаємо модуль dotenv. Усередині includeConfig () ми будемо в основному використовувати bodyparser і активувати CORS для маршрутів додатку.

3.6. Підключення до бази даних (MongoDB)

Вся система MongoDB може представляти не тільки одну базу даних, що знаходиться на одному фізичному сервері. Функціональність MongoDB дозволяє розташувати кілька баз даних на декількох фізичних серверах, і ці бази даних зможуть легко обмінюватися даними і зберігати цілісність.

Створимо файл db.js в папці /config і запишемо наведений нижче код. Тут використовуємо модуль MongoDB, хоча можемо використовувати схему mongo.

```
"use strict";
const mongodb = require('mongodb');
const assert = require('assert');

class Db{

    constructor(){
        this.mongoClient = mongodb.MongoClient;
        this.ObjectID = mongodb.ObjectID;
    }

    onConnect(){
        const mongoURL = process.env.DB_URL;
        return new Promise( (resolve, reject) => {
            this.mongoClient.connect(mongoURL, (err, db) => {
                if (err) {
                    reject(err);
                } else {
                    assert.equal(null, err);
                    resolve([db,this.ObjectID]);
                }
            });
        });
    }
}
module.exports = new Db();
```

Рис. 8 – файл db.js

3.7. Налаштування сторінки аутентифікації

У цьому розділі спочатку ми налаштуємо сторінку аутентифікації. У нас є вкладки для входу в систему і реєстрації. Якщо ви натиснете вкладку «Вхід», ви побачите форму входу в систему, а якщо ви натиснете на сторінку «Реєстрація», ви побачите форму реєстрації. Тут ми будемо реагувати на завантаження, щоб реалізувати вкладки. У контейнері з цих вкладок, ми

будемо надавати Logini Registration компоненту. Код нижче зробить те ж саме, відкриємо файл Authentication.js і поглянемо на його вміст як все реалізовано.

```
import React, { Component } from 'react';
import {Tabs, Tab} from 'react-bootstrap'

import Login from './login/Login';
import Registration from './registration/Registration';

import './Authentication.css';

class Authentication extends Component {
  state = {
    loadingState: false
  }

  setRenderLoadingState = (loadingState) => {
    this.setState({
      loadingState: loadingState
    });
  }

  render() {
    return (
      <div className="container">
        <div className = {`overlay auth-loading ${this.state.loadingState ? '' : 'visibility-hidden'}`}>
          <h1>Loading</h1>
        </div>
        <div className="authentication-screen">
          <Tabs variant="pills" defaultActiveKey = "login" >
            <Tab eventKey="login" title="Login">
              <Login loadingState={this.setRenderLoadingState}/>
            </Tab>
            <Tab eventKey="registration" title="Registration">
              <Registration loadingState={this.setRenderLoadingState}/>
            </Tab>
          </Tabs>
        </div>
      </div>
    );
  }
}

export default Authentication;
```

Рис. 9 – Authentification.js

3.8. Реалізація функції реєстрації

Почнемо з реалізації функції реєстрації. Спочатку напишемо код на стороні клієнта. Пізніше ми напишемо API вузли і весь пов'язаний з цим код.

Отже, першим кроком є написання MarkUp, тому переглянемо записаний код в render() методі в файлі Registration.js

```
render() {
  return (
    <Form className="auth-form">
      <Form.Group controlId="formUsername">
        <DebounceInput
          className="form-control"
          placeholder = "Enter username"
          minLength={2}
          debounceTimeout={300}
          onChange={this.checkUsernameAvailability} />
        <Alert className={{
          'username-availability-warning' : true,
          'visibility-hidden': this.state.usernameAvailable
        }}  variant="danger">
          <strong>{this.state.username}</strong> is already taken, try another username.
        </Alert>
      </Form.Group>

      <Form.Group controlId="formPassword">
        <Form.Control
          type = "password"
          name = "password"
          placeholder = "Password"
          onChange = {
            this.handleInputChange
          }
        />
      </Form.Group>
      <Button variant="primary" type="submit" onClick={this.handleRegistration}>
        Registration
      </Button>
    </Form>
  );
  }
}
```

Рис. 10 – Registration.js

Щоб наша розмітка працювала, нам потрібно буде написати код всередині нашого класу компонента. Відкриваємо Registration.js і запишемо туди код наведений нижче.

У наведеному нижче коді ми будемо використовувати клас ChatHttpServer для реєстрації нового користувача.

Додаток перенаправить користувача на домашню сторінку, як тільки користувач завершить процес реєстрації.

```
import React, { Component } from 'react';
import { Alert, Form, Button } from 'react-bootstrap';
import { withRouter } from 'react-router-dom';
import { DebounceInput } from 'react-debounce-input';

import ChatHttpServer from '../../../utils/ChatHttpServer';
import './Registration.css';

class Registration extends Component {

  constructor(props) {
    super(props);
    this.state = {
      username: '',
      password: '',
      usernameAvailable: true
    };
  }

  handleRegistration = async (event) => {
    event.preventDefault();
    this.props.loadingState(true);
    try {
      const response = await ChatHttpServer.register(this.state);
      this.props.loadingState(false);
      if (response.error) {
        alert('Unable to register, try after some time.')
      } else {
        ChatHttpServer.setLS('userid', response.userId);
        this.props.history.push(`/home`);
      }
    } catch (error) {
      this.props.loadingState(false);
      alert('Unable to register, try after some time.')
    }
  }

  checkUsernameAvailability = async (event)  => {
    if(event.target.value !== '' && event.target.value !== undefined) {
      this.setState({
        username: event.target.value
      });
      this.props.loadingState(true);
      try {
        const response = await ChatHttpServer.checkUsernameAvailability(this.state.username);
        this.props.loadingState(false);
        if(response.error) {
          this.setState({
            usernameAvailable: false
          });
        } else {
          this.setState({
            usernameAvailable: true
          });
        }
      } catch (error) {
        this.props.loadingState(false);
        this.setState({
          usernameAvailable: false
        });
      }
    } else if (event.target.value === '') {
      this.setState({
        usernameAvailable: true
      });
    }
  }

  handleInputChange = (event) => {
    this.setState({
      [event.target.name]: event.target.value
    });
  }
```

Рис. 11 – Продовження файлу Registration.js

Пояснення:

- Почнемо спочатку зверху, ми імпортували всі необхідні компоненти зі сторонніх бібліотек.
- Потім ми імпортували ChatHttpServer об'єкт класу.
- В об'єкті стану у нас є дві властивості: ім'я користувача і пароль. Ми безпосередньо відправимо цей об'єкт стану на сервер при реєстрації.
- У handleRegistration () методі спочатку ми викличемо loadingState () метод за допомогою React props
- ChatHttpServer.register () зареєструє користувача, зробивши HTTP-виклик на сервер, тут ми безпосередньо передаємо об'єкт стану. У об'єкта стану є usernamei password всередині нього з оновленим значенням.
- Після успішної реєстрації ми переспрямовуємо користувача на / home сторінку. Крім того, ми будемо зберігати userId в локальному сховищі, щоб ми могли використовувати його пізніше в додатку.
- У checkUsernameAvailability () і handleInputChange () методі оновить ключі стану об'єкта відповідно.

Для завершення процесу реєстрації нам також потрібно буде написати код на Nodejs. По суті, написання коду Nodejs буде для нас легкою справою, бо нам нічого писати. Давайте перейдемо до нього, відкриэмо routes.js і додамо маршрут нижче.

```
this.app.post('/register', routeHandler.registerRouteHandler);
```

Внаслідок цього нам потрібно написати кілька рядків коду всередині registerRouteHandler (), просто для обробки запиту / відповіді і виклику методу, який просто вставляє дані в базу даних.

Додамо наведений нижче код в файл route.handler.js. registerRouteHandler () подбає про / register маршрути.

```
async registerRouteHandler(request, response){
    const data = {
        username : (request.body.username).toLowerCase(),
        password : request.body.password
    };
    if(data.username === '') {
        response.status(CONSTANTS.SERVER_ERROR_HTTP_CODE).json({
            error : true,
            message : CONSTANTS.USERNAME_NOT_FOUND
        });
    }else if(data.password === '') {
        response.status(CONSTANTS.SERVER_ERROR_HTTP_CODE).json({
            error : true,
            message : CONSTANTS.PASSWORD_NOT_FOUND
        });
    } else {
        try {
            data.online = 'Y' ;
            data.socketId = '' ;
            data.password = passwordHash.createHash(data.password);
            const result = await queryHandler.registerUser(data);
            if (result === null || result === undefined) {
                response.status(CONSTANTS.SERVER_OK_HTTP_CODE).json({
                    error : false,
                    message : CONSTANTS.USER_REGISTRATION_FAILED
                });
            } else {
                response.status(CONSTANTS.SERVER_OK_HTTP_CODE).json({
                    error : false,
                    userId : result.insertedId,
                    message : CONSTANTS.USER_REGISTRATION_OK
                });
            }
        } catch ( error ) {
            response.status(CONSTANTS.SERVER_NOT_FOUND_HTTP_CODE).json({
                error : true,
                message : CONSTANTS.SERVER_ERROR_MESSAGE
            });
        }
    }
}
```

Рис. 12 – route-handler.js

Посянення:

- У наведеному вище коді ви спочатку перевірите валідацію. Якщо щось не так, ми відправимо правильний код помилки з повідомленням про помилку
- Потім ми згенеруємо хеш пароля для пароля, введеного користувачем.

- І кінець, якщо все вірно або, точніше кажучи, якщо запит дійсний, ніж ви зареєструєте користувача, визвавши registerUser ().

3.9. Перевірка унікального імені користувача

Дуже важливо перед реєстрацією перевірити унікальність імені користувача. Очевидно, що у вас не може бути двох користувачів з одним і тим же ім'ям користувача.

Насамперед давайте додамо розмітку в Registration.js. Тут писати особливо нічого, просто береться індикатор, який показує конкретний логін.

Після цього додамо наведений нижче код в клас компонента реєстрації. Наведений нижче код перевірить унікальність імені користувача, відправивши HTTP-запит на наш сервер Nodejs. Відкриємо Registration.js і оновимо код checkUsernameAvailability () методу,

```
checkUsernameAvailability = async (event)  => {
  if(event.target.value !== '' && event.target.value !== undefined) {
    this.setState({
      username: event.target.value
    });
    this.props.loadingState(true);
    try {
      const response = await ChatHttpServer.checkUsernameAvailability(this.state.username);
      this.props.loadingState(false);
      if(response.error) {
        this.setState({
          usernameAvailable: false
        });
      } else {
        this.setState({
          usernameAvailable: true
        });
      }
    } catch (error) {
      this.props.loadingState(false);
      this.setState({
        usernameAvailable: false
      });
    }
  } else if (event.target.value === '') {
    this.setState({
      usernameAvailable: true
    });
  }
}
```

Рис. 13 – Registration.js

Пояснення:

- Спочатку ми оновимо властивість стану username, потім викликаємо loadingState () метод за допомогою реквізиту React.
- Потім ми викличемо ChatHttpServer.checkUsernameAvailability () функцію, яка в кінцевому підсумку зробить HTTP-запит для перевірки унікальності імені користувача.
- І на основі відповіді HTTP-сервера ми встановимо значення usernameAvailablecвойства стану.

Це ще не все, нам ще потрібно написати код для частини Nodejs. Написання маршрутів Nodejs і його помічника буде таким же, як і в попередньому розділі. Фактично, він буде ідентичним для маршрутів Nodejs. Отже, перш за все, як завжди, відкриваємо routes.js і додаємо маршрут нижче.

```
this.app.post('/usernameAvailable', routeHandler.userNameCheckHandler);
```

3.10. Реалізація функціїї входу в систему

Реалізація функції входу в систему буде досить простою і практично ідентично реалізованою відповідно вище зазначених функцій. Спочатку ми почнемо з розмітки, потім компонента і, в кінці, напишемо кілька NodeJS, як ми це робили в останніх двох розділах.

Наведена нижче розмітка відобразить форму входу в систему, яка буде мати два поля: ім'я користувача і пароль. Відкриємо Login.js і додамо в render () метод наведений нижче:

```
render() {
  return (
    <Form className="auth-form">
      <Form.Group controlId="loginUsername">
        <Form.Control
          type = "text"
          name = "username"
          placeholder = "Enter username"
          onChange = {
            this.handleInputChange
          }
        />
      </Form.Group>

      <Form.Group controlId="loginPassword">
        <Form.Control
        type = "password"
        name = "password"
        placeholder = "Password"
        onChange = {
          this.handleInputChange
        }
        />
      </Form.Group>
      <Button variant="primary" type="submit" onClick={this.handleLogin}>
        Login
      </Button>
    </Form>
  );
}
```

Рис. 14 – Login.js

Пояснення:

- Form, Form.Group, From.ControlI Button компоненти є частиною бібліотеки – реагує на Самозавантаження.
- handleInputChange () методи оновлять стан React з відповідними деталями.
- handleLogin () методи зроблять HTTP-запит до / login останньой точкою, використовуючи ChatHTTPServerclass.

Тепер доробимо інше, якщо код залишений в компоненті Login. Наведений нижче код буде обробляти надсилання форми входу в систему. Відкриваємо Login.js і запишемо наведений нижче код,

```
import React, { Component } from 'react';
import { Form, Button } from 'react-bootstrap';
import { withRouter } from 'react-router-dom';

import ChatHttpServer from '../../../utils/ChatHttpServer';
import './Login.css';

class Login extends Component {

  constructor(props) {
    super(props);
    this.state = {
      username: '',
      password: ''
    };
  }

  handleLogin = async (event) => {
    event.preventDefault();
    this.props.loadingState(true);
    try {
      const response = await ChatHttpServer.login(this.state);
      this.props.loadingState(false);
      if(response.error) {
        alert('Invalid login details')
      } else {
        ChatHttpServer.setLS('userid', response.userId);
        this.props.history.push(`/home`)
      }
    } catch (error) {
      this.props.loadingState(false);
      alert('Invalid login details')
    }
  }

  handleInputChange = (event) => {
    this.setState({
      [event.target.name]: event.target.value
    });
  }

export default withRouter(Login)
```

Рис. 15 – Login.js

Пояснення:

- Почнемо спочатку зверху, ми імпортували всі необхідні компоненти з сторонніх бібліотек.
- Потім ми імпортували ChatHttpServerоб'єкт класу.

- В об'єкті стану у нас є дві властивості: ім'я користувача і пароль. Ми безпосередньо відправимо цей об'єкт стану на сервер при вході в систему.
- У handleLogin () методі спочатку ми викличемо loadingState () метод за допомогою React props
- ChatHttpServer.login () Метод зареєструє користувача, зробивши HTTP-виклик на сервер, тут ми безпосередньо передаємо об'єкт стану. У об'єкта стану є usernamei password всередині нього з оновленим значенням.

Після успішного входу в систему ми переспрямовуємо користувача на / home сторінку. Крім того, ми будемо зберігати userId в локальному сховищі, щоб ми могли використовувати його пізніше в додатку.

Ці handleInputChange () методи будуть оновлені ключі стану об'єкта відповідно.

Давайте напишемо частину Nodejs. Тут ми зробимо три речі. Спочатку ми додамо маршрут. По-друге, ми додамо метод для обробки запиту / відповіді маршруту і в кінці метод для виконання входу в систему. Так що давайте просто додамо маршрут, відкриємо routes.js. і додамо маршрут нижче.

```
this.app.post('/login', routeHandler.loginRouteHandler);
```

3.11. Реалізація списка чату в реальному

Розділимо цей розділ на підрозділи, оскільки розгляд всього в одному розділі було б дуже важко для розуміння. Створення списку чату включає перераховані нижче моменти:

- Створення ChatList компонента.
- Написання розмітки для списку чату
- ChatList компонент Class.
- Запис події Socket server.

- Оновлення онлайн-статусу користувача в ChatList
- Вибір користувача зі списку чату користувача.

3.11.1. Створення компонента ChatList

Навіщо потрібен новий компонент для ChatList? Хоча ми можемо написати всю розмітку і код всередині домашнього компонента. Але це не дуже гарна практика, і ваш клас Home Component буде важким і довгим в управлінні. Тому, перш за все, ми повинні створити ChatList компонент, в якому немає нічого особливого. Давайте імпортуємо ChatList компонент в HomeComponent.

3.11.2. Запис подій сервера сокета для видачі оновленого списка чату

Все раніше перераховане відбувається тільки в тому випадку, якщо у вас є Socket-сервер, який відповідає на оновлений список чатів. У будь-якому випадку, давайте поговоримо про що-небудь конструктивне. Тут ми напишемо socket.io на стороні сервера, і він буде визиватися коли його запросить клієнт.

Як тільки React запросить список чатів на сервері Socket, сокет запросить деталі з бази даних. Після запиту, в залежності від користувача, він видає результат.

Пояснення:

- Насамперед ми перевіримо валідацію. Якщо все йде добре, процес попереду, інакше буде видано відповідь з повідомленням про помилку.
- try-catch призначений для обробки невідомих винятків і необроблених відхилень.
- Потім всередині блоку try-catch ми спочатку отримуємо інформацію про користувача, визвавши йому getUserInfo ().

Чому ми генеруємо дві події сокета? Тому що, ми не хочеммо відправляти користувачеві весь список чату, замість цього відправляємо тільки інформацію про користувача, який виходить в онлайн / офлайн.

3.12. Отримання вхідних даних в компоненті Conversation з компонента ChatList

Тут ми не збираємося використовувати Redux або будь-які бібліотеки управління станом, оскільки область застосування програми не така вже й велика. Тут ми будемо використовувати прості властивості стану для передачі даних між компонентами. Щоб отримати дані з компонента ChatList, спочатку нам потрібно буде отримати дані в компоненті Home. Тут ми створимо стан який буде містити поточного обраного користувача. Цей стан буде оновлено функцією, яка буде передана як реквізит в компоненті ChatList. Створимо новий метод під назвою updateSelectedUser () Home Component і додамо в нього наведений нижче код.

```
updateSelectedUser = (user) => {
  this.setState({
    selectedUser: user
  });
}
```

Рис. 16 – Home.js

Створимо нову властивість, що викликається selectedUserв об'єктом стану, для зберігання значення обраного користувача. На цьому етапі властивість стану в домашньому компоненті налаштована на отримання оновленого користувача. Тепер давайте відправимо оновлені дані про користувача в компонент Conversation, використовуючи реквізити React.

3.13. Отримання зв’язку між користувачами та сервером

У цьому розділі ми зробимо HTTP-виклик для отримання зв’язку між користувачем та сервером. Для цього скористаємося getMessages () методом. Цей метод визначається всередині ChatHttpServer класу.

Отже, відкриваємо Conversation.js і запишемо наведений нижче код. Але спочатку додамо нижче ще дві властивості всередині об'єкту стану Conversation класу Component.

```
static getDerivedStateFromProps(props, state) {
  if (state.selectedUser === null || state.selectedUser.id !== props.newSelectedUser.id) {
    return {
      selectedUser: props.newSelectedUser
    };
  }
  return null;
}
```

Рис. 17 – Conversation.js

Кожного разу, коли Conversation буде отримувати дані нового користувача, він буде викликати службу HTTP для отримання зв’язку між двома користувачами. Для цього ми будемо використовувати два методи компонентів getDerivedStateFromProps () і ComponentDidUpdate () відповідно.

Тепер, щоб перевірити оновлені дані користувача, які ми будемо використовувати getDerivedStateFromProps(), які оновлюють стан за допомогою оновлених даних, тому компонент буде повторно відображати себе. Відкриваємо Conversation.js і запишемо наведений нижче код.

```
componentDidUpdate(prevProps) {
  if (prevProps.newSelectedUser === null || (this.props.newSelectedUser.id !== prevProps.newSelectedUser.id)) {
    this.getMessages();
  }
}
```

Рис.18 – Conversation.js

Як тільки компонент викличе метод render(), відразу після виконання ComponentDidUpdate () методу. У цьому методі ми зробимо HTTP-виклик для отримання зв’язку між двома користувачами. Відкриємо Conversation.js і запишемо наведений нижче код.

```
getMessages = async () => {
  try {
    const { userId, newSelectedUser} = this.props;
    const messageResponse = await ChatHttpServer.getMessages(userId,newSelectedUser.id);
    if (!messageResponse.error) {
      this.setState({
        conversations: messageResponse.messages,
      });
      this.scrollMessageContainer();
    } else {
      alert('Unable to fetch messages');
    }
    this.setState({
      messageLoading: false
    });
  } catch (error) {
    this.setState({
      messageLoading: false
    });
  }
}
```

Рис. 19 – Conversation.js

Пояснення:

- Метод ComponentDidUpdate () надасть попередні властивості, а ми отримаємо нові властивості у властивості props виклику.
- Використовуючи ці дві властивості, ми можемо визначити, чи вибрав той хто увійшов в систему іншого користувача в списку чату.
- Після отримання оновлених даних користувача ми викликаємо getMessages() метод класу.
- В getMessages() методі класу ми будемо використовувати ChatHttpServer клас.

Як тільки ми отримаємо нові повідомлення, getMessages(), метод оновить стан і відобразить зв’язок між двома користувачами. Відкриємо Conversation.js і запишемо наведений нижче код.

```
render() {
  const { messageLoading, selectedUser } = this.state;
  return (
    <>
      <div className={`message-overlay ${!messageLoading ? 'visibility-hidden' : ''}`}>
        <h3> {selectedUser !== null && selectedUser.username ? 'Loading Messages' : ' Select a User to chat.' }</h3>
      </div>
      <div className={`message-wrapper ${messageLoading ? 'visibility-hidden' : ''}`}>
        <div className="message-container">
          <div className="opposite-user">
            Chatting with {this.props.newSelectedUser !== null ? this.props.newSelectedUser.username : '----'}
          </div>
          {this.state.conversations.length > 0 ? this.getMessageUI() : this.getInitiateConversationUI()}
        </div>

        <div className="message-typer">
          <form>
            <textarea className="message form-control" placeholder="Type and hit Enter" onKeyPress={this.sendMessage}>
            </textarea>
          </form>
        </div>
      </div>
    </>
  );
```

Рис. 20 – Conversation.js

Пояснення:

- Тут ми викликаємо getMessages(), який визначений всередині ChatHttpServer класу.
- Для методу getMessages() потрібні userId (зареєстрований користувач) і toUserId (користувач, обраний зі списку чату)
- Як тільки ми отримаємо відповідь від сервера, ми оновимо стан, використовуючи conversation властивість.
- scrollMessageContainer метод, як випливає з назви він буде прокручувати вміст до самого низу.

3.14. Відображення бесіди як повідомлення чату

На даний момент ми маємо всі повідомлення між двома користувачами. Залишається тільки написати розмітку, щоб можна було відображати повідомлення.

```
sendMessage = (event) => {
  if (event.key === 'Enter') {
    const message = event.target.value;
    const { userId, newSelectedUser } = this.props;
    if (message === '' || message === undefined || message === null) {
      alert(`Message can't be empty.`);
    } else if (userId === '') {
      this.router.navigate(['/']);
    } else if (newSelectedUser === undefined) {
      alert(`Select a user to chat.`);
    } else {
      this.sendAndUpdateMessages({
        fromUserId: userId,
        message: (message).trim(),
        toUserId: newSelectedUser.id,
      });
      event.target.value = '';
    }
  }
}
```

Рис. 21 – Conversation.js

Пояснення:

- По-перше, на компонент розмови буде накладення, поки ми не виберемо будь-якого користувача зі списку чату.
- Як тільки ми виберемо користувача зі списку чату, ми покажемо повідомлення про це.
- Потім на основі об'єкта стану розмови ми отримаємо відповідний призначений для користувача інтерфейс (ми побачимо це відразу після

Ми використали два нових методи: this.getMessageUI () і this.getInitiateConversationUI (). This.getMessageUI () буде повертати розмови

у вигляді повідомлень між двома користувачами і this.getInitiateConversationUI () повертає заповнювач, якщо користувач не обраний зі списку чату.

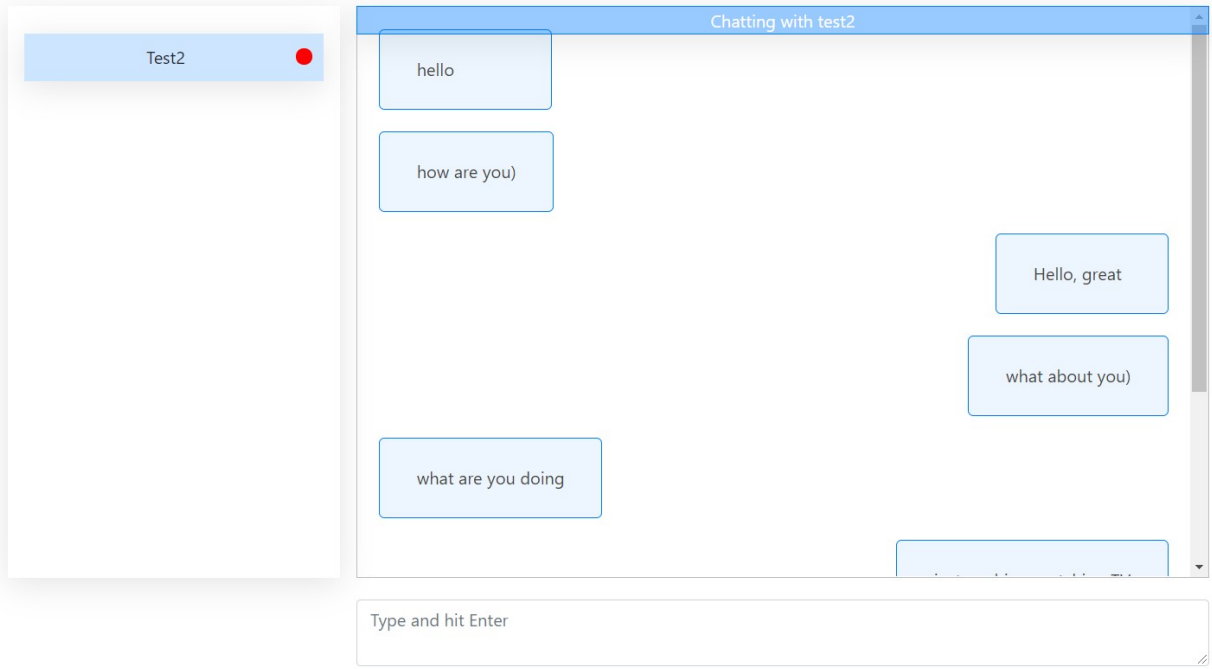

Рис. 22 – Інтерфейс

3.15. Відправлення та отримання повідомлень у реальному часі

Остання задача додатки, тут спочатку ми відправимо повідомлення обраному користувачу зі списку чату. А потім ми напишемо код для прослуховування вхідних повідомлень з сервера. Отже, давайте почнемо з розмітки, відкриваємо файл conversation.js і напишемо наведений нижче код під div тегом, який має .message-typer клас.

```
sendMessage = (event) => {
  if (event.key === 'Enter') {
    const message = event.target.value;
    const { userId, newSelectedUser } = this.props;
    if (message === '' || message === undefined || message === null) {
      alert(`Message can't be empty.`);
    } else if (userId === '') {
      this.router.navigate(['/']);
    } else if (newSelectedUser === undefined) {
      alert(`Select a user to chat.`);
    } else {
      this.sendAndUpdateMessages({
        fromUserId: userId,
        message: (message).trim(),
        toUserId: newSelectedUser.id,
      });
      event.target.value = '';
    }
  }
}
```

Рис.23 – conversation.js

Пояснення:

- Спочатку ми перевіримо валідацію. Якщо щось піде не так, ми покажемо попередження.
- Після цього ми всередині else блока, викликаємо інший метод sendAndUpdateMessages ().
- Крім того, ми помістимо ці дані в існуючий conversation масив станів.
- Потім ми фактично відправляємо ці дані на сервер Nodejs, викликаючи sendMessage () метод, який в основному генерує подію сокета.

Наведена нижче подія сокета буде прослуховувати подію сокета і в залежності від ідентифікатора користувача буде відправляти повідомлення одержувачам. Оскільки ця подія сокета, яку ми генеруємо з Angular, відкриємо файл socket.js і додамо наступний код,

```
socket.on(`add-message`, async (data) => {
    if (data.message === '') {
        this.io.to(socket.id).emit(`add-message-response`,{
            error : true,
            message: CONSTANTS.MESSAGE_NOT_FOUND
        });
    }else if(data.fromUserId === ''){
        this.io.to(socket.id).emit(`add-message-response`,{
            error : true,
            message: CONSTANTS.SERVER_ERROR_MESSAGE
        });
    }else if(data.toUserId === ''){
        this.io.to(socket.id).emit(`add-message-response`,{
            error : true,
            message: CONSTANTS.SELECT_USER
        });
    }else{
        try{
            const [toSocketId, messageResult ] = await Promise.all([
                queryHandler.getUserInfo({
                    userId: data.toUserId,
                    socketId: true
                }),
                queryHandler.insertMessages(data)
            ]);
            this.io.to(toSocketId).emit(`add-message-response`,data);
        } catch (error) {
            this.io.to(socket.id).emit(`add-message-response`,{
                error : true,
                message : CONSTANTS.MESSAGE_STORE_ERROR
            });
        }
    }
});
```

Рис. 24 – socket.js

3.16. Отримання повідомлень у реальному часі

Ця частина проста в реалізації. Тут ми створимо метод для прослуховування вхідних подій сокета. Після отримання події сокета ми помістимо нові повідомлення в conversation масив станів. Отже, ми створили receiveMessage () метод всередині ChatSocketServer класу. Цей метод буде підписуватися на нове вхідне повідомлення від сервера сокетів. Ми будемо викликати цей метод з componentDidMount () методу. Як тільки ми викликали цей метод, він буде прослуховувати вхідні повідомлення і автоматично поміщати нові повідомлення в conversation масив станів.

```
receiveSocketMessages = (socketResponse) => {
  const { selectedUser } = this.state;
    if (selectedUser !== null && selectedUser.id === socketResponse.fromUserId) {
      this.setState({
        conversations: [...this.state.conversations, socketResponse]
      });
      this.scrollMessageContainer();
    }
}
```

Рис. 25 – conversation.js

Пояснення:

- Цей метод спочатку перевіряє, чи надходять повідомлення від відповідного користувача, перевіряючи ідентифікатор користувача.
- Як тільки ми отримаємо нове повідомлення, ми додамо це повідомлення у conversation властивість стану Conversation класу.

3.17. Реалізація виходу

Реалізація виходу з системи, так, це останнє, що залишилося зробити. Так що тут ми в основному будемо працювати тільки в HomeComponent класом. Нам дійсно не потрібно давати огляд функцій виходу з системи. Отже, давайте просто перейдемо в home.js, додамо в render () метод розмітку нижче і переконаємося, що наш заголовок виглядає як розмітка нижче,

```
<header className="app-header">
  <nav className="navbar navbar-expand-md">
    <h4>Hello {this.state.username} </h4>
  </nav>
  <ul className="nav justify-content-end">
    <li className="nav-item">
      <a className="nav-link" href="#" onClick={this.logout}>Logout</a>
    </li>
  </ul>
</header>
```

Рис. 26 – home.js

Давайте додамо метод logout () в компонент Home, який змушує користувача вийти з системи.

```
logout = async () => {
  try {
    await ChatHttpServer.removeLS();
    ChatSocketServer.logout({
      userId: this.userId
    });
    ChatSocketServer.eventEmitter.on('logout-response', (loggedOut) => {
      this.props.history.push(`/`);
    });
  } catch (error) {
    console.log(error);
    alert(' This App is Broken, we are working on it. try after some time.');
    throw error;
  }
}
```

Рис. 27 – home.js

Пояснення:

- Усередині logout () методу ми викликали removeLS () метод, який визначається всередині ChatService класу. Цей метод видалить всі дані, що зберігаються в локальному сховищі, пов'язані з цим додатком.
- Потім після цього ми викликали метод виходу з ChatSocketServer класу. Цей метод відправляє на сервер подію сокета. Це змінить онлайн-статус на офлайн.
- І як тільки ми отримаємо відповідь про вихід з системи, ми просто переспрямуємо користувача на домашню сторінку.

ВИСНОВОК

В результаті роботи було створено чат додаток, для спілкування в режимі реального часу з використанням React JS, Node JS, socket.io, а також в якості бази даних було використано MongoDB. Так як зараз всі більшу частину свого часу проводять в соціальних мережах, то саме чат додатки і призначені для обміну повідомленнями між людьми.

Месенджери - це новий спосіб комунікації між людьми, незалежно від географічних особливостей, за допомогою обміну миттєвими повідомленнями. В умовах розвитку Інтернету як публічного простору з можливістю доступу до різної персональної інформації у користувачів виникає потреба в усвідомленому споживанні і приватноій взаємодії. Тому і зростає актуальність серед додатків, що надають подібні можливості.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Підручник: введення в React
[Електронний ресур]. – Режим доступу:
https://learn-reactjs.ru/tutorial#what-is-react

2. Node.js
[Електронний ресур]. – Режим доступу:
https://en.wikipedia.org/wiki/Node.js

3. MongoDB
[Електронний ресур]. – Режим доступу:
https://metanit.com/nosql/mongodb/1.1.php

4. React Overview and Walkthrough
[Електронний ресур]. – Режим доступу:
https://www.taniarascia.com/getting-started-with-react/

5. Що таке Node JS
[Електронний ресур]. – Режим доступу:
https://netology.ru/blog/node

6. Socket.io
[Електронний ресур]. – Режим доступу:
https://coderlessons.com/tutorials/kompiuternoe-programmirovanie/uznaite-socket-io/socket-io-kratkoe-rukovodstvo

7. MongoDB
[Електронний ресур]. – Режим доступу:
https://proselyte.net/tutorials/mongodb/introduction/

8. Node JS та NPM
[Електронний ресур]. – Режим доступу:
https://fructcode.com/ru/blog/what-nodejs-and-npm/

9. React JS
[Електронний ресур]. – Режим доступу:
https://metanit.com/web/react/2.3.php