МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
УНІВЕРСИТЕТ МИТНОЇ СПРАВИ ТА ФІНАНСІВ

ФАКУЛЬТЕТ ІННОВАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра комп`ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота бакалавра

на тему «Розробка вебсервісу для миттєвого обміну повідомленнями у реальному часі для невеликих компаній»

Виконав: студент групи ІПЗ19-2

Спеціальність 121 «Інженерія програмного забезпечення»

Погорілий Дмитро Станіславович

(прізвище та ініціали)

Керівник д.т.н., проф. Яковенко В.О.

(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент

доцент кафедри кібербезпеки та інформаційних технологій

(місце роботи)

к.т.н., доц. Флоров С.В.

(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2023

АННОТАЦІЯ

Погорілий Д.С. Створення вебсервісу для миттєвого обміну повідомленнями у реальному часі для невеликих компаній. – Кваліфікаційна робота на здобуття освітнього ступеня бакалавр на правах рукопису.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр, галузь знань 12 «Інформаційні технології», за спеціальністю 121 «Інженерія програмного забезпечення». – Університет митної справи та фінансів, Дніпро, 2023.

У результаті виконання кваліфікаційної роботи бакалавра була створена програма на мовах програмування TypeScript та JavaScript , серверна частина якої працює на фреймворку NestJS, а клієнтська частина на фреймворку Vue.js, із застосування сучасних бібліотек, шифруванням даних за стандартом RFC 7519, аутентифікацією за допомогою JSON Web Token, документо-орієнтованої системи керування базами даних MongoDB, об’єктно реляційної проекції Prisma.

Були використані можливості браузера та фреймворків: локальне сховище для зберігання даних, реактивність для зміни сторінок без перезапуску, гарди для захисту даних від сторонніх користувачів, роутери для звернення на API та переадресації між сторінками, сокети для постійного з’єднання клієнтської частини та серверної, динамічна зміна даних в клієнтській частині.

Ключові слова:

TypeScript, JavaScript, NestJS, Prisma, Vue.js, конструктор, алгоритм, API, WebSocket, вебсервіс.

ABSTRACT

Pogorelyi D.S. Creating a web service for instant messaging in real time for small companies - Qualification work for the degree of bachelor in the form of a manuscript.

Qualification work for the bachelor's degree, field of knowledge 12 "Information Technology", specialty 121 "Software Engineering." - University of Customs and Finance, Dnipro, 2023.

As a result of the bachelor's qualification work, a program was created in the programming languages TypeScript and JavaScript, the server part of which runs on the NestJS framework, and the client part on the Vue.js framework, using modern libraries, data encryption according to RFC 7519, authentication using JSON Web Token, document-oriented database management system MongoDB, object-relational projection Prisma.

The capabilities of the browser and frameworks were used: local storage for data storage, reactivity for changing pages without reloading, guards to protect data from third-party users, routers for API calls and redirects between pages, sockets for permanent connection between the client and server sides, dynamic data change in the client side.

Keywords:

TypeScript, JavaScript, NestJS, Prisma, Vue.js, constructor, algorithm, API, WebSocket, web service.

ЗМІСТ

ВСТУП

Актуальність дослідження. Сучасний світ змінюється з великою швидкістю. Зараз на ринку для продажу своєї продукції з'являється велика кількість компаній. Всі компанії унікальні, але є речі які їх поєднують. Одна з таких речей - це комунікація всередині компанії.

Комунікація – це взаємодія між двома або більше людьми. Якщо говорити про компанії та важливість комунікації в них, то можливо виділити два головних чинники:

- чим більше колектив комунікує та має гарні стосунки, тим краще їх продуктивність
- чим менше часу затрачує колектив на знаходження людини в компанії для розв'язання проблеми, тим швидше проблема вирішується

Час який колектив затрачує на знаходження людини може бути дуже великим. Наприклад, людина пішла по своїм справам, та навіть якщо їй напишуть листа на пошту, вона не одразу отримає повідомлення. Цей фактор, як і багато інших, негативно впливають на економічний стан компаній.

Мета роботи. Метою написання кваліфікаційної роботи є автоматизація процесу комунікації між людьми. Це можливо реалізувати завдяки створенню вебсервісу для online взаємодії між людьми та миттєвого обміну різної за типом інформації між ними.

Завдання роботи. В роботі потрібно буде виконати багато завдань, для реалізації поставленої мети:

- сформулювати задачу;
- дослідити та проаналізувати засоби реалізації
- спроектувати вебсервіс
- розробити інтерфейс користувача

- розробити серверну та клієнтську частину вебсервісу
- написати інструкцію користувача

Створення зрозумілого для співробітника інтерфейсу, миттєвого отримання повідомлення та швидкої обробки даних на сервері є пріоритетом.

Практична значимість. У наш час для кожної програми, яка містить алгоритм або декілька алгоритмів для реєстрації, ідентифікації користувача, шифрування даних, взаємодії в реальному часі та через запроси клієнта с сервером важливо використовувати нові стандарти захисту і новітні рішення які присутні на ринку інформаційних технологій. У даній роботі описується як раз один з варіантів програми яка має алгоритми авторизації, реєстрації, ідентифікації, надання різних типів доступу, шифрування даних, збереження даних в базу даних, збереження файлів в систему, валідація всіх даних які проходять через сервер. В системі, в якості сховища використовуються документо-орієнтована структурна система керування базами даних, а для слідкування за логікою програми використовуються бібліотеки, класи та два фреймворки.

Акутальність обраних методів. Використання мови програмування TypeScript зумовлює строгу типізацію. Виходячи з цього, в програмі не може бути помилок пов’язаних с передачею не того типу в переміну, функцію, клас.

Фреймворк NestJS створює сприятливі умови для багатьох аспектів серверної розробки вебсервісу:

- легке підключення RESTful Api та WebSocket;
- вилов помилок та створення однакового макету для їх виводу на клієнтській частині;
- легке логування помилок та запитів;
- валідація даних на різних етапах життєвого циклу запита (middleware, guards, interceptors, pipes);
- легкий імпорт зовнішніх бібліотек та їх функціоналу в типізованому вигляді [3-4].

Фреймворк Vue.js створює сприятливі умови для багатьох аспектів розробки клієнтської частини вебсервісу:

- велика структурованість даних;
- динамічність даних;
- реактивність;
- легке створення компоненту, який можливо використовувати в багатьох місцях;
- легкий імпорт бібліотек та підключення до серверної частини;
- більша оптимізація ніж у конкурентів

Об'єкт дослідження: шифрування даних, ідентифікації та автентифікації користувачів у системі, робота с сервером за допомогою посилань та зв'язку в реальному часі, реактивність та динамічність клієнтської частини, робота з файлами.

Предмет дослідження: шифрування за стандартом RFC 7519 та робота в системі за допомогою фреймворків NestJS та Vue.js, різних бібліотек, документо-орієнтованої системи керування базами даних MongoDB, об'єктно реляційної проекції Prisma та мов програмування TypeScript та JavaScript. Зв'язок серверної частини з клієнтською за допомогою робота с сервером за допомогою RESTful Api та WebSpcket.

Структура роботи. Структура роботи зумовлена поставленими задачами і складається з вступу, трьох розділів, загальних висновків по роботі та списку використаних джерел (21 - найменувань). Повний обсяг роботи становить 115 сторінки, з них основного тексту - 51 сторінка. У роботі налічується 55 рисунків та 1 додатку.

Перший розділ складається з обгрунтування актуальності теми для дослідження, аналізу наявних аналогів та аналізу їх переваг та недоліків. На основі проведеної роботи була сформульована мета та завдання програмного продукту, перелік необхідної функціональності для коректної роботи застосунку, а також обрано програмне забезпечення для створення вебсервісу.

У другому розділі розглядаються теоретичні питання

Вебсервісу на мові TypeScript та Javascript із використанням

технології NestJS та Vue.js. Також, описується технічне завдання, вимоги до даних, структуру проекту.

Третій розділ описує безпосередньо реалізацію проекту, розглядаються деталі розробки (написання класів та функцій), демонструються скріншоти реалізації програми та інструкцію користувача.

У висновках роботи були зазначені досягнення, здобуті в процесі розробки вебсервісу та аналіз негативних та позитивних сторін.

РОЗДІЛ 1. ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ ТА АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ

1.1 Формулювання задачі

Розробити вебсервіс для миттєвого обміну повідомленнями на мовах програмування TypeScript та JavaScript , серверна частина якої працюватиме на фреймворку NestJS, а клієнтська частина на фреймворку Vue.js, із застосування сучасних бібліотек, шифруванням даних за стандартом RFC 7519, аутентифікацією за допомогою JSON Web Token, документо-орієнтованої системи керування базами даних MongoDB, об’єктно реляційної проекції Prisma.

Веб-сервіс повинен забезпечити:

- Реєстрація користувача.
- Авторизація користувача.
- Шифрування паролю з використання новітніх бібліотек.
- Використання фреймворку NestJS для серверної частини веб-сервісу.
- Використання фреймворку Vue.js для публічної частини веб-сервісу.
- Періодична аутентифікація користувача за допомогою JSON Web Token за стандартом RFC 7519.
- Створення чату між користувачами.
- Відправка повідомлень в чаті.
- Миттєве отримання повідомлень.
- Можливість передавати в повідомлені не тільки текст, але і файли.

У веб-сервісі буде 4 сутностей, які будуть тісно пов’язані.

Сторінка для авторизації та регістрації.

Вхідними даними для функціонування сторінки є введення пошти та паролю при авторизації або ім’я, пошта та пароль при реєстрації .

Вихідними даними є успішна авторизація, створення токенів, та переадресація на наступну сторінку або в сценарії введення некоректних даних є вивід помилок.

Сторінка з списком чатів.

Вхідними даними для функціонування сторінки є інформація яка отримана при авторизації, список чатів з інформаціє про співрозмовника та виведеним останнім повідомленням, якщо є не прочитані повідомлення то виводиться їх кількість, якщо відправник останнього повідомлення користувач, то виводиться статус повідомлення (відправлено – одна галочка, доставлено – дві галочки). Якщо під час того як користувач знаходиться на цій сторінці йому відправляють нове повідомлення, виводиться повідомлення про цю подію.

Вихідними даними є вибір чату та переадресація на його сторінку.

Сторінка чату.

Вхідними даними для функціонування сторінки є отримані з серверної сторони дані про повідомлення в цьому чаті.

Вихідними даними є створення нового повідомлення або отримання нового повідомлення.

Сторінка для створення нового чату.

Вхідними даними для функціонування сторінки є показ вікна в якому треба записати id користувача з яким ви хотіли би створити чат.

Вихідними даними є створення чату.

Вимоги до інтерфейсу програмного забезпечення:

- зручний та зрозумілий інтерфейс користувача;
- простота використання;
- динамічність
- реактивність

Розробку програмного забезпечення виконати у вигляді окремого веб-сервісу який буде працювати на всіх браузерах.

WebSocket буде присутній на всіх сторінках де користувач авторизован. Головні вимоги:

- отримання інформації про надходження нового повідомлення;
- отримання інформації про помилки на серверній частині;
- підключення до WebSocket при авторизації;
- якщо користувач заходить з різних браузерів, всі браузери повинні одночасно працювати коректна;
- відключення від WebSocket у разі, якщо користувач намагається порушити справну роботу системи.

1.2 Методи та засоби розв’язування задачі

1.2.1 JSON Web Token

JSON Web Token - стандарт токена для доступу в вебсервісах та вебзастосунках на основі JSON. Має стандарт захисту RFC 7519.

Найбільше застосування в клієнт-серверних програмах для для аутентифікації в клієнт-серверних програмах при передачі даних.

Дії між клієнтською та серверною частиною:

1) сервер створює токен;
2) підписує секретним ключем;
3) передає клієнту (який використовую цей токен для підтвердження своєї особи) [5, 19].

При підтверджені особи відбуваються наступні дії:

1) токен передається з запитом на серверну частину
2) сервер перевіряє токен
3) сервер віддає відповідь якщо токен коректний або відправляє помилку з статусом 401 якщо токен не коректний.

Токени можуть ще використовуватись для перевірки дозволу на окремі дії, які доступні не всім користувачам, тоді з’являються додаткові дії:

1) сервер записує в мета-дату дозволи з якими дозволено виконувати запит
2) сервер перевіряє чи є ці права доступу у токена користувача
3) сервер віддає відповідь якщо користувач має дозволи на запит або відправляє помилку з статусом 403 якщо користувач не має дозволів.

Токен складається з трьох частин(див. рис. 1.1):

- заголовка (header)
- корисне навантаження (payload)
- підпис (key) [5]

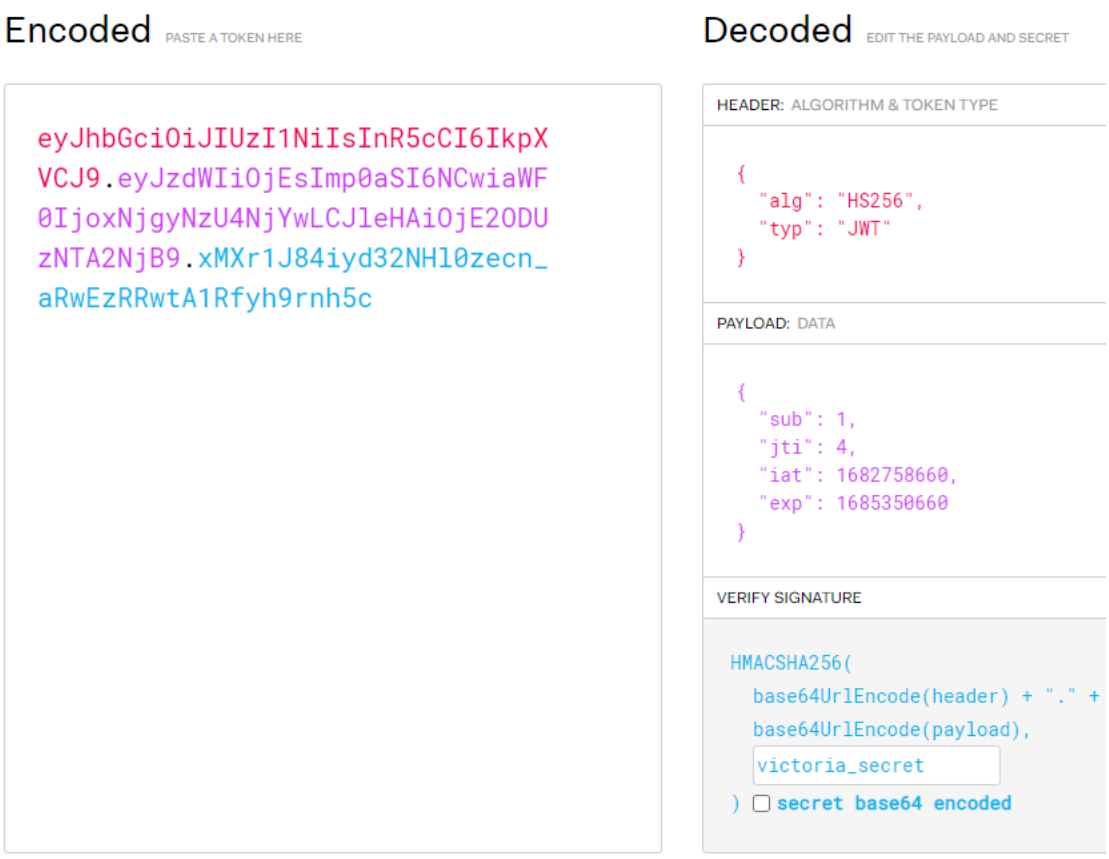

Рисунок 1.1 – Зашифрована та розшифрована інформація з токена

1.2.2 Vue3.js

Vue3.js – це фреймворк для розробки клієнтської частини на мові JavaScript або TypeScript (рисунок 1.2).

```
<div id="app">
  <button @click="count++">
    Count is: {{ count }}
  </button>
</div>
```

Рисунок 1.2 – Приклад запису реактивних даних за допомогою Vue.js

За допомогою цього фреймворку можна зручно організувати роботу на стороні клієнта, а саме створювати компоненти, які потім можливо використовувати багато разів в різних місцях, що дозволяє зменшити навантаження роботи програми.

Динамічна зміна даних – корисна функція, в фреймворку всі переміні на клієнтській частині можливо змінювати в реальному часі не перезавантажуючи сторінку, ця функція тісно. (див. рис. 1.3).

```
const Counter = {
  data() {
    return {
      counter: 0
    }
  },
  mounted() {
    setInterval(() => {
      this.counter++
    }, 1000)
  }
}
```

Рисунок 1.3 – Приклад реактивного відслідковування данних в перемінній “counter”

1.2.3 Бібліотека AXIOS

Бібліотека AXIOS – це бібліотека заснована на функціях HTTP-клієнта для браузера та Node.js на основі асинхроного програмування.

AXIOS – це проста у використанні бібліотеку. У невеликому пакеті є багато можливостей для розширення. Бібліотека була створена для виконання запитів на сервер з боку клієнта або на зовнішні RESTful API (див. рис. 1.4). Дозволяє зручно взаємодіяти з API для отриманння та надсилання даних на сервер без необхідності вручну прописувати HTTP-запити [2].

```
const axios = require('axios');

// Делаем запрос пользователя с данным ID
axios.get('/user?ID=12345')
  .then(function (response) {
    // обработка успешного запроса
    console.log(response);
  })
  .catch(function (error) {
    // обработка ошибки
    console.log(error);
  })
  .finally(function () {
    // выполняется всегда
  });
```

Рисунок 1.4 – Посилання запиту на отримання данних

1.2.4 Vuex Store

Vuex Store – це пакет для Vue3.js. Ця бібліотека реалізує зберігання та організовування пам’яті у додатку на стороні клієнта.

Бібліотека реалізує швидку логіку для оновлення та зберігання даних у пам’яті браузера, що робить можливим поєднання додатку в одну для всіх компонентів систему. За допомогою бібліотеки при перезавантаженні сторінки та переходу між ними дані не зникають.

При використані Vuex створюється блок пам'яті. "Блок пам'яті" - це місце, в якому зберігається всі реактивні дані вашого додатку (див. рис. 1.5). Є дві речі, які відрізняють Vuex від звичайного глобального блоку пам'яті:

- блок пам’яті Vuex є реактивними. Коли частини Vue беруть з нього інформацію, вони реактивно та швидко оновлюються, якщо інформація блоку пам'яті змінюється.
- ви не можете власноруч змінювати стан блоку пам'яті. Єдиний варіант змінити стан блоку пам'яті - це виявити зміні. Це гарантує, що кожне змінення будь-чого залишає інформацію по це, яку реально відслідкувати, і реалізує використовування технологій, які покращують наше розуміння в вебсервісах [7].

```
// Create a new store instance.
const store = createStore({
  state () {
    return {
      count: 0
    }
  },
  mutations: {
    increment (state) {
      state.count++
    }
  }
})
```

Рисунок 1.5 – Збереження та редагування зміної “count” в “ сховищі”

1.2.5 Vue Router

Router - це пакет для Vue3, який дозволяє організовувати маршрутизацію веб-сторінок [8].

За допомогою цього пакету можна налаштовувати адреси сторінок та поведінку додатку при переході між ними.

На рисунку 1.6 зображен файл с маршрутизацією на дві сторінки: “Home” та “About”.

```
// 1. Define route components.
// These can be imported from other files
const Home = { template: '<div>Home</div>' }
const About = { template: '<div>About</div>' }

// 2. Define some routes
// Each route should map to a component.
// We'll talk about nested routes later.
const routes = [
  { path: '/', component: Home },
  { path: '/about', component: About },
]

// 3. Create the router instance and pass the `routes`
// You can pass in additional options here, but let's
// keep it simple for now.
const router = VueRouter.createRouter({
  // 4. Provide the history implementation to use. We a
  history: VueRouter.createWebHashHistory(),
  routes, // short for `routes: routes`
})

// 5. Create and mount the root instance.
const app = Vue.createApp({})
// Make sure to _use_ the router instance to make the
// whole app router-aware.
app.use(router)

app.mount('#app')
```

Рисунок 1.6 – Файл для машрутизації на клієнській частині

1.2.6 SCSS

SCSS – це препроцесор CSS, він використовується для зручного та швидкого написання стилів для вебсервісів. SCSS реалізує використання змінних, міксінів та інших функціоналів для ефективної роботи зі стилями і зменшення повторювання коду.

1.2.7 NestJS

NestJS – це фреймворк для створення ефективних та масштабованих програм Node.js на стороні сервера (див. рисунок 1.7).

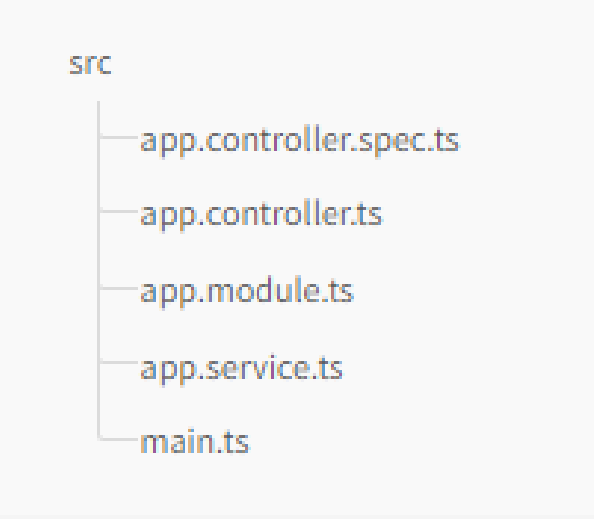

Рисунок 1.7 – Приклад початкової архітектури при створенні проекту

Фреймворк використовує дві мови програмування:

1) JavaScript;
2) TypeScript.

Фреймворк з’єднує елементи функціонального програмування, об’єктно-орієнтованого програмування і функціонально реактивного програмування.

NestJS має чудову архітектуру вебсервісу.

Архітектура дозволяє створювати вебсервіси з великим перевагами:

- легко тестувати;
- масштабовані модулі;
- слабозв’язан між собою;
- не тяжкі в обслуговуванні.

Архітектура дуже легка для розуміння і складається з таких частин:

- Провайдер. Назви таких файлів закінчуються на .service.ts, .service.js. Їх функція полягає в зберіганні функцій методів, їх логіки. Багато файлів в вебсервісі є як провайдери. Головна ціль провайдера це реалізація файлу та класу в ньому як залежності, тобто, компоненти створюють зв’язки один з одним, а Nest відповідає за їх будування (див. рис 1.8).

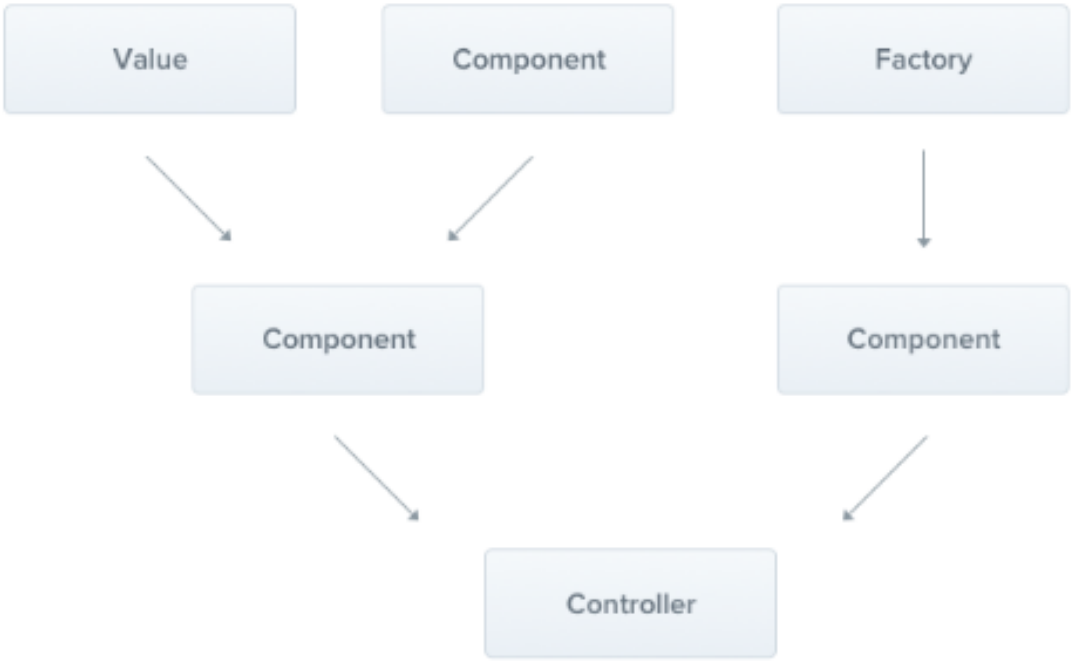

Рисунок 1.8 – Схема провайдера

- Контролер. Функції конролера – відправлення та отримання запитів для вебсервісу. Будова контролера відслідковує, який контролер прописаний для якого маршруту. Є багато випадків коли контролер пов’язанимй більше ніж з одним маршрутом, в свою чергу всі маршрути пов’язані з різними функціями в провайдері. Назви таких файлів закінчуються на .controller.ts, . controller.js. Їх функція полягає в зберіганні посилань на які

клієнтська частина може звертатись, прив’язка методів з провайдерів до посилань, навішування на посилання різних перевірок та логіки яка буде відбуватись під час життєвого циклу посилання (див. рис 1.9).

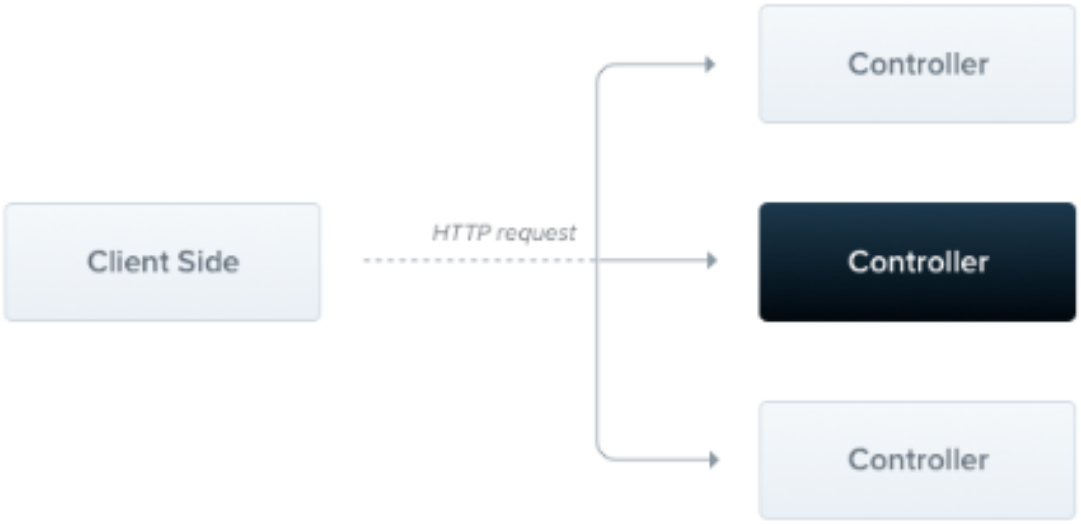

Рисунок 1.9 – Схема котролера

- Модуль. Вебсервіс повинен мати мінімум один модуль. Головний модуль є сутністю з якої вебсервіс починає свою роботу. Nest будує граф додатку - внутрішню архітектуру, потім Nest використовує цю архітектуру для визначення зв'язків між модулями та контролерами. В документації фреймворку написано, що модулі рекомендуються при написанні вашого вебсервісу. Для більшості застосунків в архітектурі буде налічуватись декілька модулів (див. рис 1.10).

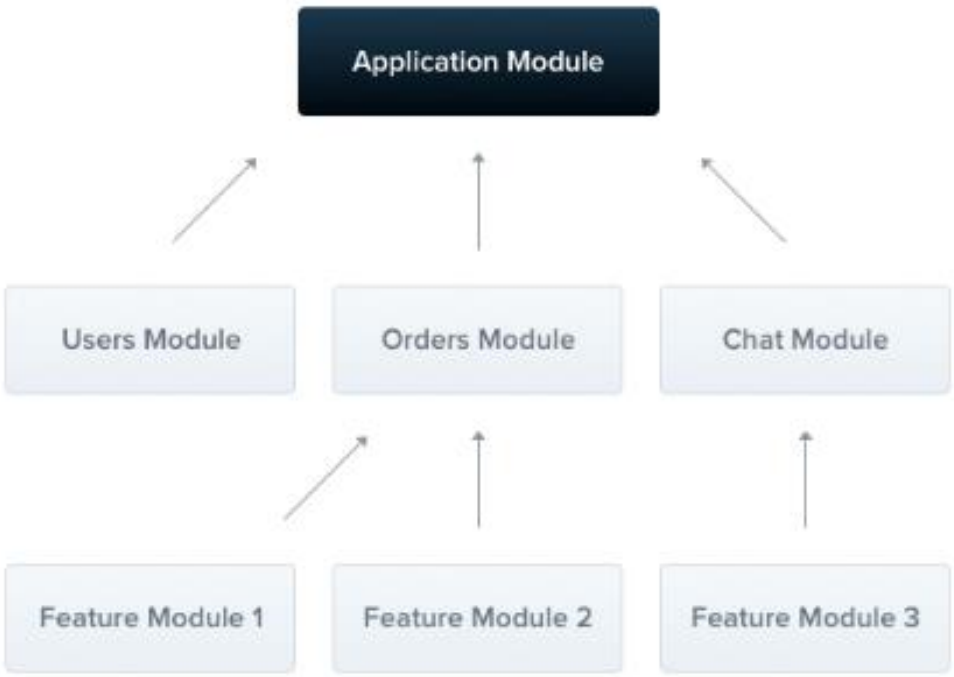

Рисунок 1.10 – Схема модуля

- Проміжне програмне забезпечення. Воно викликається коли клієнтська частина робе запит по маршруту, але провайдер ще не почав працювати. Воно може використовувати об'єкт з запиту і відповіді з запиту. Можливе паралельне та послідовне використання (див. рис 1.11).

Рисунок 1.11 – Схема проміжного програмного забезпечення

- Фільтр винятків. Nest має функція ґка реалізую винятки. Функція винятків робить дві дії у програмі. Перша дія – це захоплення помилок, які не зміг прорахувати розробник. Друга дія – це перетворення серверної помилки на зручну стуктуру для клієнтської частини. Коли виняток не обробляється кодом вашої програми, він перехоплюється цим рівнем, який автоматично надсилає відповідну зручну для користувача відповідь (див. рис 1.12) .

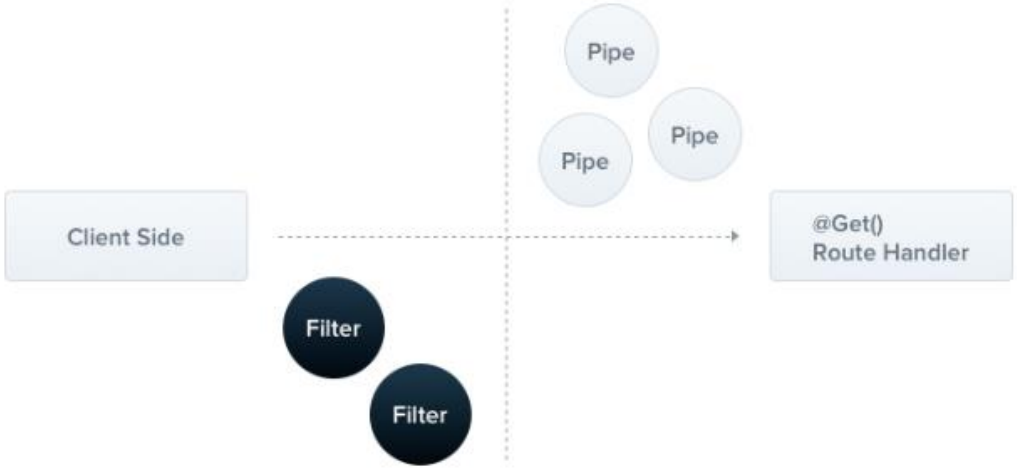

Рисунок 1.12 – Схема фільтра винятків та труб

- Труби (pipes). Труби використовуються у двох випадках: перетворення даних які йдуть по маршруту в зручнішу форму (наприклад, з бінарного файлу в об’єкт з описом цього фалу), перевірка даних які йдуть по маршруту і змінюють їх, якщо вони не відповідають вимогам, або у разі помилок створюють виключення. У всіх варіантах труби працюють з даними, які обробляються контролером. Nest по своїй архітектурі створює труби перед викликом методу, і труба отримує дані маршруту які були призначені для методу, та оброблює їх. Можливо використовувати паралельно та послідовно (див. рис 1.12).

- Гарди. Гарди несуть в собі єдину логічну функцію. Ця функція полянає в визначенні, чи можливо запиту пройти далі, чи має він достатньо прав дозволу, залежно від умов (наприклад, дозволів, ролей). У розробників – цю функцію часто називають аутентифікацією.

- Перехоплювач (див. рис 1.13). Перехоплювачі мають багато швидких та оптимізованих функцій, багато з них розробники фреймворку взяли з аспектно-орієнтованого програмування. Є багато цікавих функцій, таких як: додавання Функцій і логіки до початку або після роботи провайдера, змінна результату в потрібний формат і його відправка в провайдер, змінна структури виключення, яке з’явилося у провайдера. [4,11,18].

Рисунок 1.13 – Схема перехоплювача

1.2.8 MongoDB

MongoDB – це база даних. ЇЇ структура - документно-орієнтована база даних

MongoDB - постійно використовується для зберігання не структурованих даних (рисунок 1.14).

Замість таблиць і рядків, як у реляційних базах даних, MongoDB використовує колекції та документи. Документи складаються з пар ключ-значення, які є основною одиницею даних в MongoDB.

MongoDB - це база даних, яка з'явилася приблизно в середині 2000-х років [1].

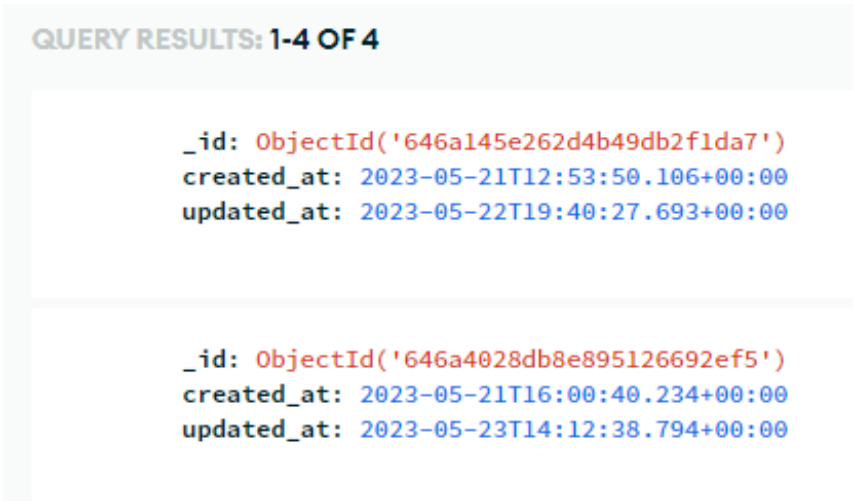

Рисунок 1.14 – Приклад колекції з MongoDB

Особливості MongoDB:

1) кожна база даних містить колекції;
2) кожна колекція містить документи;
3) кожен документ може відрізнятися різною кількістю полів;
4) розмір і зміст кожного документа можуть відрізнятися один від одного;
5) структура документа більше відповідає тому, як розробники будують свої класи та об'єкти у відповідних мовах програмування;
6) розробники часто кажуть, що їхні класи не є рядками і стовпчиками, а мають чітку структуру з парами ключ-значення;
7) документи не повинні мати схему, визначену заздалегідь. Натомість, поля можна створювати на льоту;
8) модель даних, доступна в MongoDB, дозволяє легше представляти ієрархічні зв'язки, зберігати масиви та інші більш складні структури;
9) середовище MongoDB є дуже масштабованим. Компанії по всьому світу створили кластери, деякі з них налічують понад 100 вузлів з мільйонами документів у базі даних.

1.2.9 Prisma

Prisma – це технологія програмування, яка зв'язує бази даних з концепцією об'єктно-орієнтованої мови програмування TypeScript та платформи Node.js, створюючи «віртуальну об'єктну базу даних».

Prisma відкриває новий рівень роботи розробників з базами даних завдяки великій кількості плюсів:

- зрозуміла модель даних, яка знаходиться в схемі;
- автоматизовані міграції;
- безпека типів
- автоматичне завершення роботи процесів.

Вона використовується як альтернатива написанню простого SQL або використанню інших інструментів доступу до баз даних, таких як конструктори SQL-запитів (наприклад, knex.js) або ORM (наприклад, TypeORM і Sequelize). Наразі Prisma підтримує PostgreSQL, MySQL, SQL Server, SQLite, MongoDB та CockroachDB (попередня версія).

Prisma можно використовувати з JavaScript але вона підтримує TypeScript і реалізує рівень безпеки введення, який виходить за рамки гарантій інших ORM в екосистемі TypeScript [10].

На рисунку 1.15 зображен код, який створює в базі даних дві таблиці, зв’язок між ними та типізацію.

```
// Define the `User` table in the database
model User {
  id    Int     @default(autoincrement()) @id
  name  String?
  email String  @unique
  posts Post[]
}

// Define the `Post` table in the database
 model Post {
   id        Int      @default(autoincrement()) @id
   published Boolean? @default(false)
   title     String
   content   String?
   author    User?    @relation(fields: [authorId], references: [id])
   authorId  Int?
 }
```

Рисунок 1.15 – Схема для створення двух таблиць

1.2.10 WebSocket

WebSocket – це протокол, який призначений для обміну інформації в режимі реального часу між клієнтською та серверною частиною вебсервісу.

Найбільш популярна бібліотека для використання WebSocket – це Socket.IO.

Socket.IO - це бібліотека, яка забезпечує двонаправлений зв'язок між клієнтом і сервером з низькою затримкою та на основі подій [18]. При її використанні є багато плюсів:

1) витрачається менше ресурсів
2) велика оптимізація процесів (див. рис 1.16);
3) якщо з'єднання з WebSocket неможливе, він повернеться до HTTP лонг-полінгу. А якщо з'єднання буде втрачено, бібліотека автоматично спробує відновити його заново;
4) можливо підключитись одразу до декількох клієнтських частин і легко надсилати події всім кому це потрібно;

```
//send message from sender to other user room
sendMessage(id: string, data) {
  this.server.to(id).emit('msgToClient', data);
}
```

Рисунок 1.16 – Подія “sendMessage” для відправлення повідомлення на клієнтську частину.

1.3 Висновки до першого розділу

Дослідження показало, що зараз для створення клієнтської частини вебсервісу не обійтись без потужного та зарекомендувавшого себе на ринку фрейморку. Вебсервіси які будуть написані на чистих мовах програмування без використання стороніх бібліотек та фреймворків не зможуть бути конкурентами іншим вебсервісам.

На підставі вищевикладеного та дослідження можна дійти невтішного висновку про серверні частини вебсервісів. Зараз на ринку існує всього два фреймворки, які можуть створювати один одному конкуренцію:

- Laravel
- NestJS

Ті вебсервіси які будуть написані на інших фреймворках, навіть використовуючи велику кількість сторонніх бібліотек, зазнають невдачу.

Проводячи дослідження в роботі з базами даних, було з’ясовано, що при використанні об’єктно реляційної системи керування, кращим варіантом буде «PostgreSQL». Ця система добре працює с таблицями та даними, які дуже структуровані та мають однакові за типом значення. В даній роботі така система не буде гарно працювати. Всі дані для вебсервісу – є не структуровані та можуть мати багато видів, виходячи з ситуації. Для роботи з такими даними краще використовувати документо-орієнтовну систему керування. Найкращим вибором в таких системах – є «MongoDB».

РОЗДІЛ 2. ПОЧАТКОВЕ ПРОЕКТУВАННЯ ВЕБСЕРВІСУ

2.1 Проектування веб-сервісу

Перед тим як писати код програми, необхідно правильно спроектувати логіку програми. Тобто необхідно продумати які функції виконуватиме програмне забезпечення, які потрібні додаткові функції, які обробники помилок потрібно передбачити, які типи даних необхідно використовувати.

Напишемо структуру програми і її функцій. У підсумку все виглядає наступним чином, як показано на рисунку 2.1. На цій діаграмі показані основні функції програми, на самому початку точка входу в додаток і сторінка логіна. Кожен з наступних блоків показує пункти основного меню.

Далі створимо прототип майбутньої структури. Для цього опишемо всі компоненти, які будуть присутні в нашій програмі:

- Клас axiosInstance: клас відповідає за логіку відправлення запитів на сервер с клієнтської частини програми за допомогою бібліотеки AXIOS.
- Темплейт RegistrationViews.vue: темплейт реалізує сторінку реєстрації та авторизації користувача.
- Темплейт HomeView.vue: темплейт реалізує сторінку з списком чатів, та чатами користувачів, можливість створення нового чату, можливість відправлення смс.
- Темплейт 404View.vue: реалізує сторінку попередження, якщо користувач ввів недійсне посилання.
- Клас router: клас відповідає за логіку переадресації користувача по сторінкам за допомогою бібліотеки Router Vue.

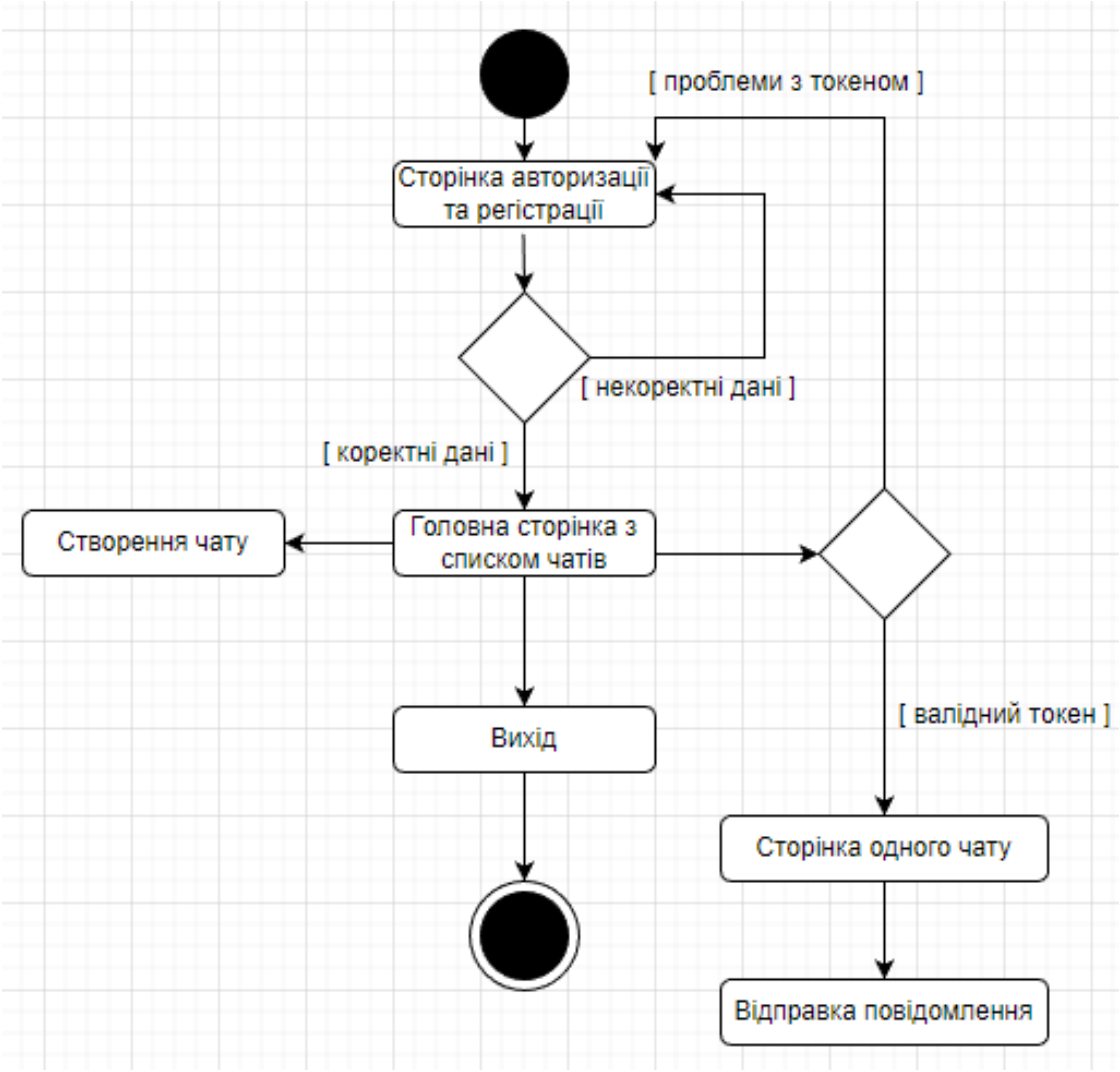

Рисунок 2.1 – Діаграма діяльності

- Клас createStore: клас відповідає за коректну логіку роботи локального збереження даних в браузері за допомогою бібліотеки VuexStore.
- Схема prisma: схема для створення бази даних, таблиць. Після пропису команд створюються також міграції по цій схемі.
- Папка node_modules: зберігає всі зовнішні бібліотеки, які були встановленні в вебсервісі.
- Папка dist: зберігає зібраний вебсервіс для запуску.
- Папка storage: зберігає файли, які були відправленні в повідомленнях.
- Клас LoginDto: клас для валідації даних при авторизації.

- Клас RegisterDto: клас для валідації даних при регістрації.
- Клас RefreshDto: клас з токеном для відновлення токена.
- Клас JwtAuthGuard: гард для валідації токенів в запиті до сервера.
- Клас LocalAuthGuard: гард для валідації юзера при авторизації.
- Стратегія JwtStrategy: доповнення логіки в гард JwtAuthGuard.
- Стратегія LocalStrategy: доповнення логіки в гард LocalAuthGuard
- Клас AuthModule: клас який збирає всю логіку про дії з авторизацією, аутентифікацією та створює робочий модуль.
- Клас AuthController: клас який реалізує створення посилань для кожної дії.
- Клас AuthService: клас який містить всі методи про дії пов’язані з авторизацією, аутентифікацією та регістрацією.
- Клас ChatsModule: клас який збирає всю логіку про дії з чатами.
- Клас ChatsController: клас який реалізує створення посилань для кожної дії.
- Клас ChatsService: клас який містить всі методи про дії пов’язані з створенням чатів, їх виводом, пошуком в чатів.
- Клас MessagesModule: клас який збирає всю логіку про дії з посиланнями с чату.
- Клас MessagesController: клас який реалізує створення посилань для кожної дії.
- Клас MessagesService: клас який містить всі методи про дії пов’язані з відправленням повідомлення, їх виводом.
- Клас AllExeptionFilter: клас який реалізує вилов всіх помилок, перебудову в гарний вигляд для клієнтської частини.
- Клас AppLoger: клас який реалізує запис критичних помилок до файлів, запис запитів, які були відправлені до бази даних.

2.2 Розробка інтерфейсу користувача

Дизайн користувальницького інтерфейсу є фактором, який впливає на три основні показники якості програмного продукту: його функціональність, естетику і продуктивність.

Функціональність є фактором, на який розробники додатків часто звертають основну увагу. Вони намагаються створювати програми так, щоб користувачі могли виконувати свої завдання і їм було зручно це робити. Функціональність важлива, але, тим не менш, це не єдиний показник, який повинен враховуватися в ході розробки.

Вікно авторизації (див. рис. 2.2) містить поля пошта та пароль. Ці поля, як і всі інші перевіряються на клієнтській частині, якщо перевірка не проходить, кнопка стає сірою (див. рис. 2.3).

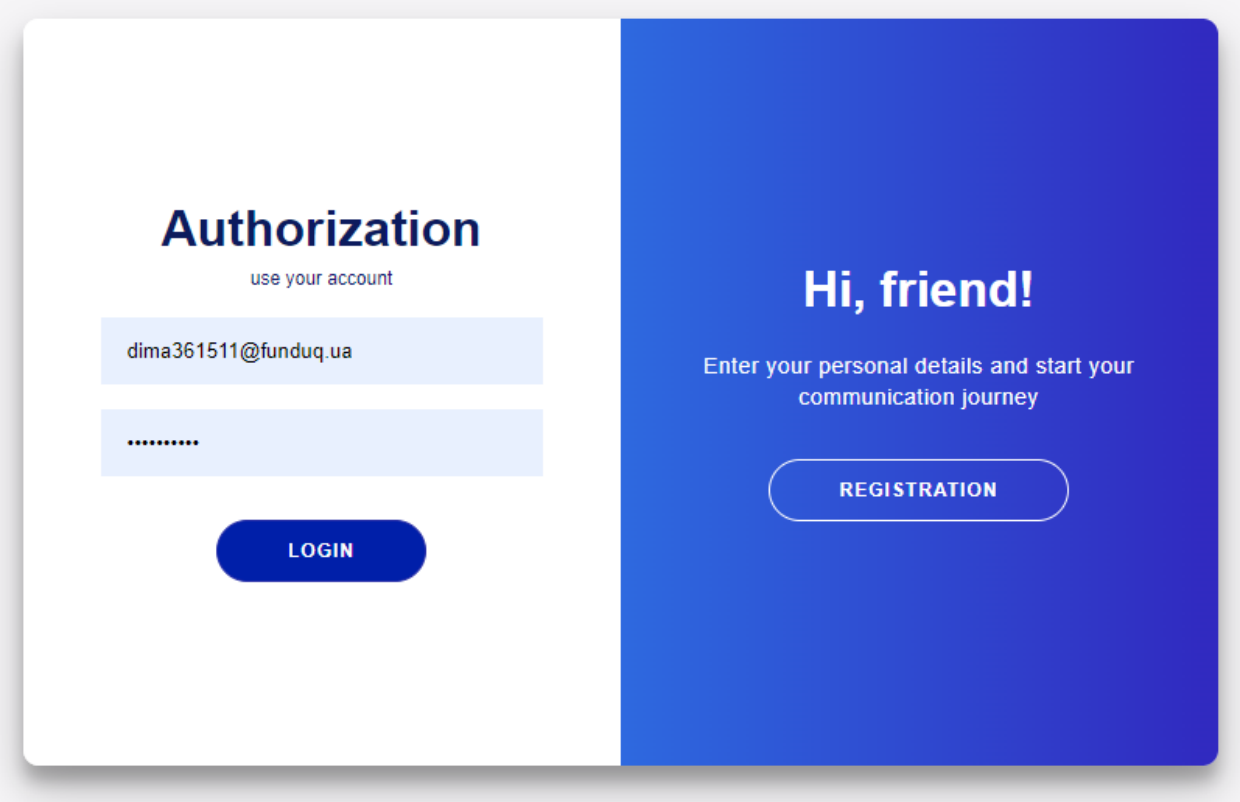

Рисунок 2.2 – Вікно авторизації

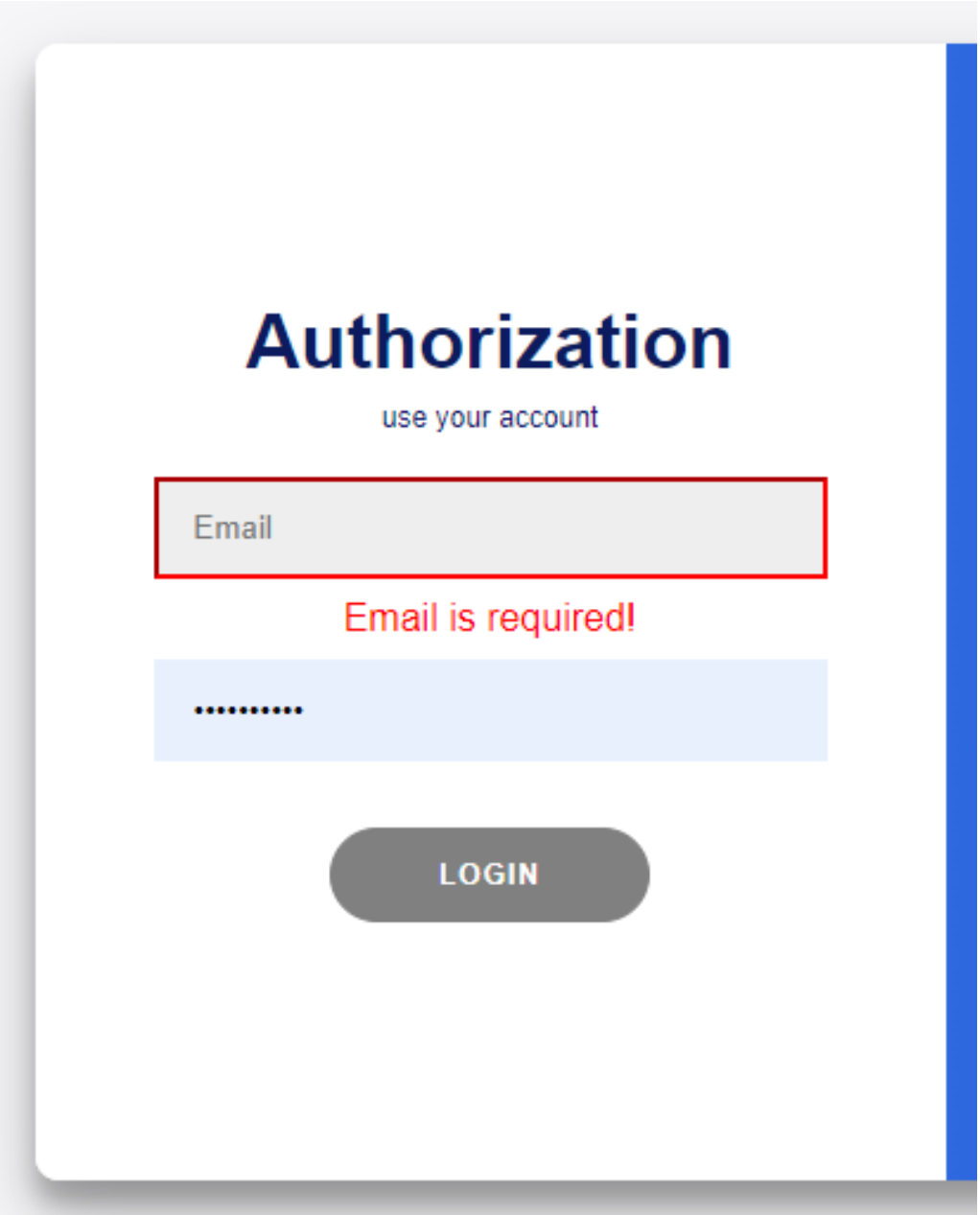

Рисунок 2.3 – Вікно авторизації з некоректно введеними даними

Вікно реєстрації (див. рис. 2.4) містить поля пошта, ім’я та пароль. Ці поля, як і всі інші перевіряються на клієнтській частині, якщо перевірка не проходить, кнопка стає сірою.

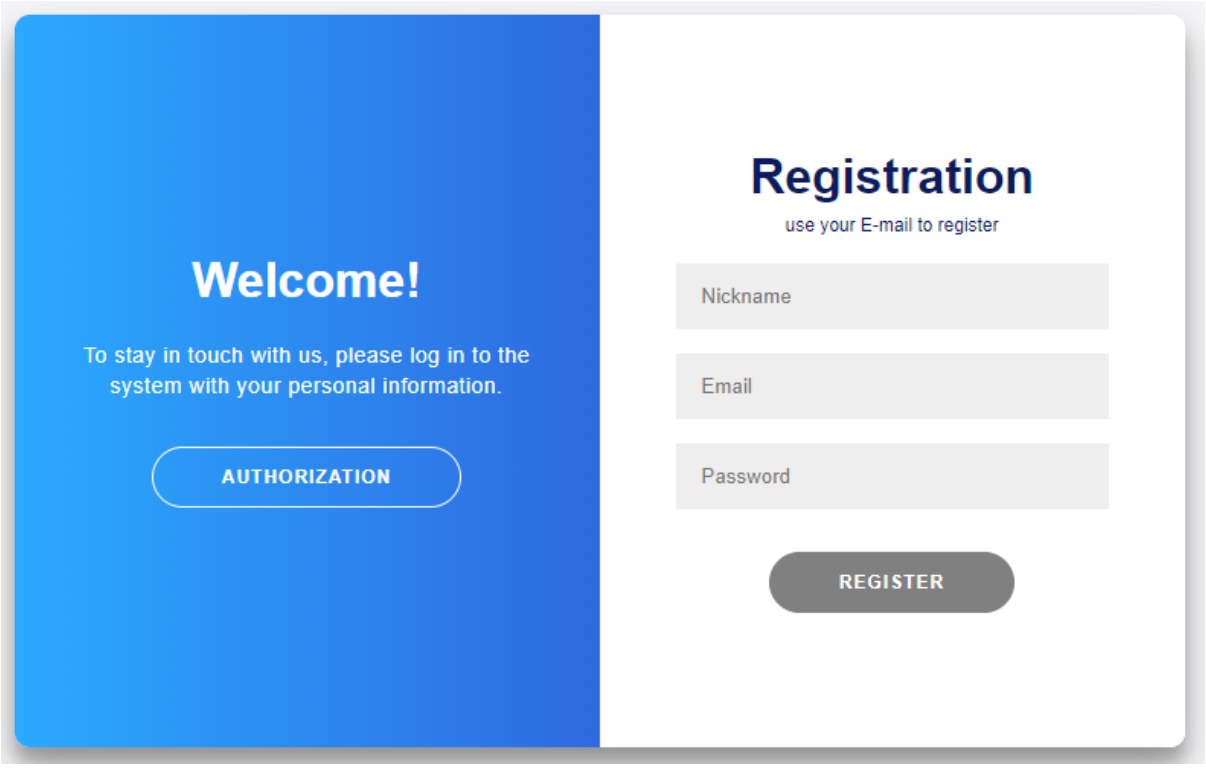

Рисунок 2.4 – Вікно реєстрації

При коректній реєстрації, не треба авторизуватись, користувач одразу переходить до головного меню.

Головне меню клієнтської частини при запуску виглядає як на рисунку 2.5:

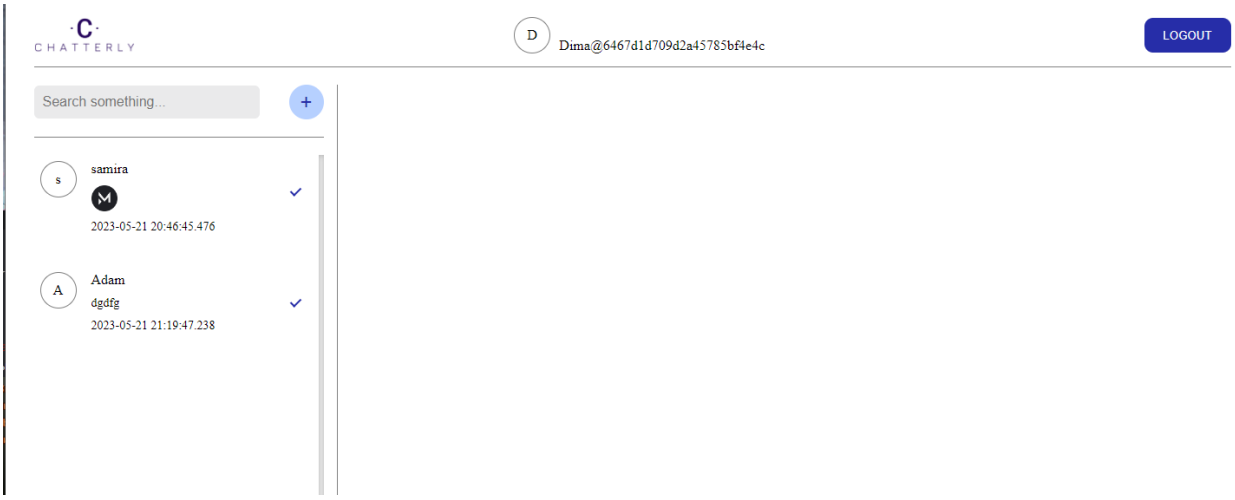

Рисунок 2.5 – Вигляд головного меню вебсервісу при запуску

Зліва користувач зможе побачити список доступних чатів (див. рис. 2.6), текстове поле для пошуку (див. рис. 2.7) та кнопку додати новий чат.

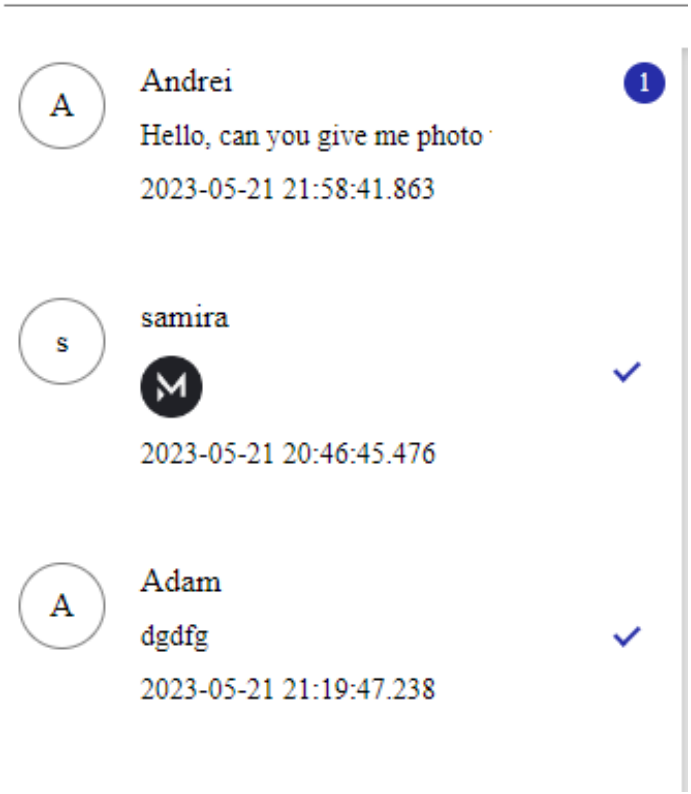

Рисунок 2.6 – Список доступних чатів

Рисунок 2.7 – Текстове поле для пошуку

При натисканні на кнопку додати новий чат, з’являється вікно (див. рис. 2.8), де користувач повинен ввести id людини з якою він хочу вести діалог.

Якщо чат вже був створений, нічого не відбудеться.

Рисунок 2.8 – Створення нового чату

При натисканні на чат зі списку чатів, з’являється чат зі всіма повідомленнями, які були відправленні в цьому чаті (див. рис 2.9).

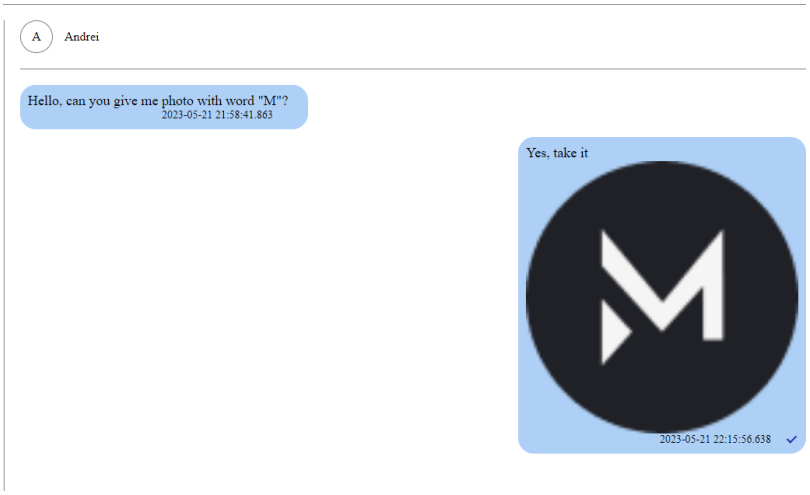

Рисунок 2.9 – Сторінка чату с користувачем Andrei

На рисункк можна побачити такі компоненти повідомлення:

- текст;
- зображення;
- час відправлення.

Статуси у повідомлення:

- відправлено – одна галочка (якщо користувач ще не прочитав);
- доставлено – дві галочки (якщо користувач прочитав).

2.3 Висновки до другого розділу

Початкове проєктування вебсервісу закінчене. При проєктуванні до уваги брався досвід відомих проєктів, таких як «Telegram» та «Facebook». При використані цих проєктів та їх дослідженні було виявлено декілька проблем, наприклад:

- напис про те що користувач пише повідомлення іноді не зникає;
- людина виходить з вебсервісу, але статус онлайн зникає не завжди.

Якщо брати до уваги дизайн клієнтської частини, то «Telegram» набагато зручніший. В ньому поєднані багато технологічних рішень, які простому користувачеві важко помітити:

- проєкт перейшов на нову базу даних, але старі повідомлення не видаляв, натомість дійшовши до них, користувача перекидає на такий самий чат, але його дії обмежені.
- якщо вікно чату буде дуже великим, всі повідомлення здвигаються на одну сторону, так користувачу буде легше розуміти їх послідовність.

При проектуванні була розроблена мінімальна логіка дій для користувача. Ця логіка знаходиться в діаграмі діяльності (див. рис. 2.1).

Розроблена структура для майбутніх файлів, папок та схем. При її розробці увага приділялась модульній архітектурі (див. рис. 2.10). Така архітектура – є зручною та її легко інтегрувати в інші більші модулі та сервіси.

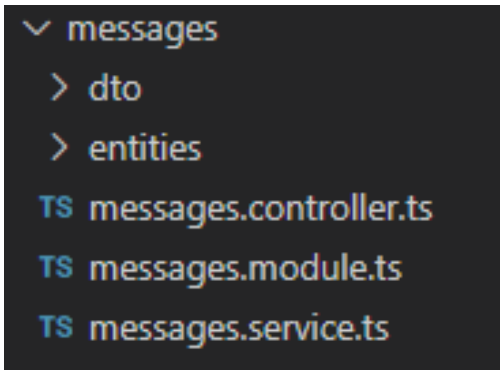

Рисунок 2.10 – Приклад модуля

Повністю описано про логіку роботи кожного файлу в вебсервісі. Майже всі файли мають класи, як основу даних. Також велика увага приділялась тим файлам, які відповідають за захист даних та аутентифікації користувача.

Розроблений початковий інтерфейс клієнтської частини. При розробці інтерфейсу брались до уваги наступні критерії:

- зручність та зрозумілість при використанні
- швидкість завантаження даних та сторінок

Розглядалось два варіанти інтерфейсу:

1) Окремі сторінки для списку чатів та чата;
2) Одна сторінка, де буде поєднано список чатів та обраний чат

Вибір був зроблений на другий варіант, оскільки користувач може швидко переходити між чатами та бачити нову інформацію по всім чатам.

РОЗДІЛ 3. РОЗРОБКА ВЕБСЕРВІСУ

3.1 Розроблення програми та її опис

3.1.1 Опис програми

Створюваний веб-сервіс призначений для реалізації комунікації між людьми в реальному часі. Даний веб-сервіс можливо інтегрувати як мікро-сервіс в інші веб-сервіси більших масштабів. Також підходить як стартова модель для чатів великих середовищ.

Клієнтська частина писалася на мові програмування JavaScript на фреймворку Vue.js. Дана мова дозволяє розробляти легкі за розміром додатки.

Серверна частина писалася на мові програмування TypeScript на фреймворку NestJS. Дана мова дозволяє розробляти ООП додатки, а також додатки, які займають значно меншу кількість оперативної пам'яті. Ця мова широко використовується для розробки веб-сервісів, будучи однією з найпопулярніших мов програмування. Область її застосування включає створення різноманітних прикладних програм, драйверів пристроїв, додатків для вбудованих систем, високопродуктивних серверів, а також ігор. Існує безліч реалізацій мови TypeScript, як безкоштовних, так і комерційних і для різних платформ.

Для розробки програми необхідне середовище розробки Visual Studio Code. В даному випадку використовувалася версія Stable Build. Чому саме це середовище і ця версія? Тому що з її допомогою легко додавати додаткові пакети під час розробки: «Auto Close Tag», «Code Spell Checker», «Docker», «ESLint», «GitLens», «IntelliCide», «npm Intellisense», «Prettier» та багато інших. Також легко інтегрувати її з системою контролю версій.

3.1.2 Авторизація або регістрація користувача

Після вводу даних з клієнтської частини посилається запит на сервер та отримується відповідь. Якщо ця відповідь з кодом 201 то користувачу приходить токени та мінімальна інформація (див. рис. 3.1).

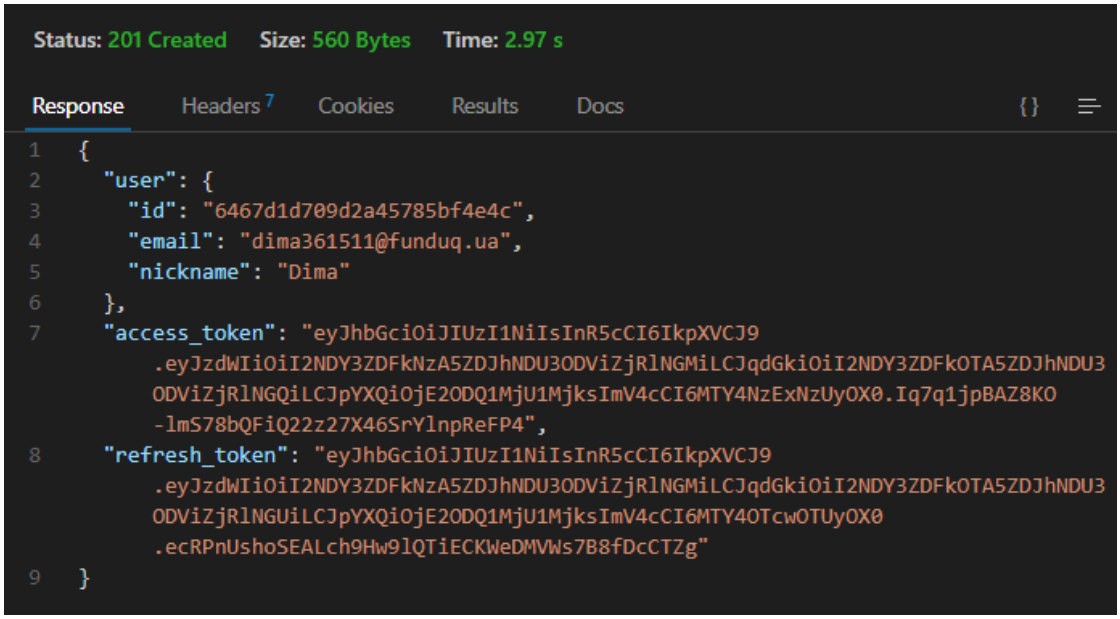

Рисунок 3.1 – Типи та структура відданої інформація при реєстрації або авторизації на клієнтську частину вебсервісу

Вся отримана інформація була збережена у localstorage та store vuex (див. рис. 3.2). Localstorage є механізмом для зберігання даних на стороні клієнта в браузері. Використовуючи localstorage, зберігалась інформація про користувача, список дозволів, та токени для роботи програми.

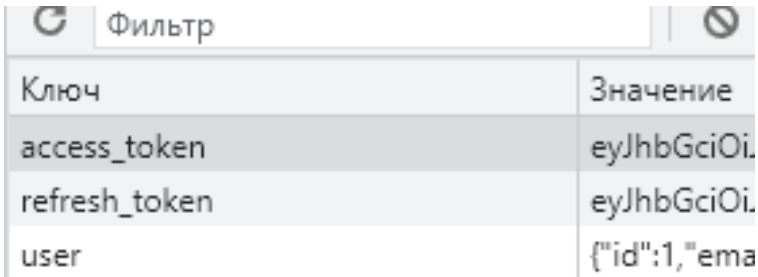

Рисунок 3.2 – Localstorage та дані збережені в ньому

3.1.3 Аутентифікація користувача

Аутентифікація проходить на серверній частині при запиті користувача на захищені від зовнішнього втручання посилання. При запиті на сервер передається токен ,який повністю перевіряється на сервері (див. рис. 3.3, 3.4, 3.5).

```
You, last week | 1 author (You)
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor() {
    super();
  }

  handleRequest(err, user) {
    if (err) {
      throw err;
    }
    if (user) return user;
    throw new UnauthorizedException('Invalid token');
  }
}
```

Рисунок 3.3 – Клас який створює Guard для перевірки токена.

```
async validateAccessToken(payload: IPayload) {
  const { user_id, ...access_token } =
    await this.prisma.userAccessToken.findUniqueOrThrow({
      where: { id: payload.jti },
      select: {
        user_id: true,
        id: true,
        expires_at: true,
      },
    });
  if (access_token.expires_at.valueOf() < Date.now().valueOf()) {
    throw new UnauthorizedException('Access token is expired');
  }
  if (user_id !== payload.sub) {
    throw new UnauthorizedException('Invalid access token');
  }
  return {
    (property) token_id: string
    token_id: payload.jti,
  };
}
```

Рисунок 3.4 – Метод для перевірка токену

```
@UseGuards(JwtAuthGuard)
@Post()
createChat(
```

Рисунок 3.5 – Використання класу JwtAuthGuard на посиланні “linkOnWebsite/chats”

3.1.4 Функції головного меню програми

В головному меню реалізовані функції для користувача:

- вивід списку чатів (див. рис. 3.6)

```
async getChats(      You, last week • Chats module …
  user_id: string,
  { skip = 0, take = 10, universal }: GetChatsQueryDto,
) {
  //create where
  const where: Prisma.ChatWhereInput = {···
  };
  if (universal) {···
  }
  //get chats
  const chats = await this.prisma.chat.findMany({···
  });
  return chats.map(({ messages, id, user_messages, user_chats }) => ({
    id,
    user: user_chats[0].user,
    messages:
      messages.length > 0
        ? {
            created_at: messages[0].created_at,
            text: messages[0].text,
            files: messages[0].files,
            unread_count: user_messages.length,
            sent:
              user_id === messages[0].user_id &&
              messages[0].user_messages[0].read === false
                ? true
                : undefined,
            delivered:
              user_id === messages[0].user_id &&
              messages[0].user_messages[0].read === true
                ? true
                : undefined,
          }
        : null,
  }));
}
```

Рисунок 3.6 – Метод для отримання списка чатів

- вивід списку чатів після універсального пошуку (співпадіння написаного в пошуку з нікнеймом або з текстом повідомлення написаного в чаті) (див. рис. 3.7).

```
//create where
const where: Prisma.ChatWhereInput = {
  user_chats: { some: { user_id } },
};
if (universal) {
  where.OR = [
    {
      user_chats: { some: { user: { nickname: { contains: universal } } } }
    },
    {
      messages: {
        some: { text: { contains: universal } },
      },
    },
  ];
}
```

Рисунок 3.7 – Утворення інструкції для універсального пошуку чатів

- створення чату між двома користувачами - користувач повинен ввести id другого користувача (див. рис. 3.8).

```
async createChat(user_id: string, { id }: CreateChatDto) {
  const ids = [user_id, id];
  //chat was created?
  const createdChat = await this.prisma.chat.findFirst({
  });
  //return chat_id if chat was created
  if (createdChat) {
    return { status: 'success', id: createdChat.id };
  }
  //create chat
  const chat = await this.prisma.chat.create({···
  });
  return { status: 'success', id: chat.id };
}
```

Рисунок 3.8 – створення чату

- отримання повідомлення з чату та відправлення (див. рис. 3.9)

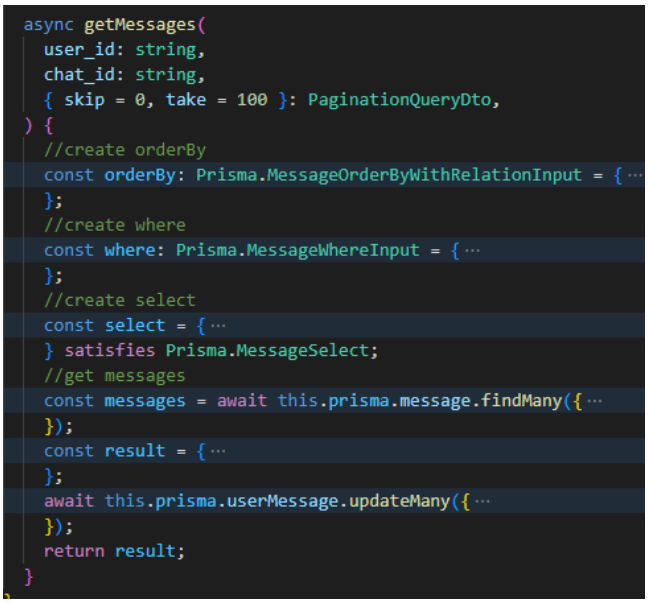

Рисунок 3.9 – Отримання повідомлення з чату

- робота з WebSocket для отримання повідомлень у реальному часі, а саме підключення до WebSocket (див. рис. 3.10), відключення від WebSocket (див. рис. 3.11), отримання помилок з сервера (див. рис. 3.12), отримання повідомлення про написання повідомлення яким-то іншим користувачем (див. рис. 3.13).

Рисунок 3.10 – подія при підключенні до WebSocket

```
//socket disconnect
handleDisconnect(client: Socket) {
  console.log(`Client disconnected: ${client.id}`);
}
```

Рисунок 3.11 – подія при відключенні від WebSocket

```
client.emit('error', { message: 'Invalid token' });
```

Рисунок 3.12 – подія для отримання помилок

```
//send message from sender to other user room
sendMessage(id: string, data) {
  this.server.to(id).emit('msgToClient', data);
}
```

Рисунок 3.13 – подія для повідомлення від іншого користувача

- схема бази даних (див. рис. 3.14, 3.15)

```
model User {
  id            String            @id @default(auto()) @map("_id") @db.ObjectId
  email         String            @unique
  nickname      String
  password      String
  createdAt     DateTime          @default(now())
  access_tokens UserAccessToken[]
  UserChat      UserChat[]
  Message       Message[]
  UserMessage   UserMessage[]
}
```

Рисунок 3.14 – модель користувача

```
model Message {
  id            String        @id @default(auto()) @map("_id") @db.ObjectId
  user_id       String        @db.ObjectId
  chat_id       String        @db.ObjectId
  text          String?
  files         String[]
  created_at    DateTime      @default(now())
  user          User          @relation(fields: [user_id], references: [id],
  chat          Chat          @relation(fields: [chat_id], references: [id],
  user_messages UserMessage[]
}
```

Рисунок 3.15 - модель повідомлення

3.1.5 Запобігання помилок

В програмі створено механізм роботи при отриманні помилок з сервера. Коли користувач пише не коректні дані в поля, йому с сервера виводяться помилки (див. рис. 3.16, 3.17) які в клієнтській частині з’являються в дружелюбному для користувача вигляді.

```
You, last week | 1 author (You)
export class LoginDto {
  @Transform(({ value }) =>
    typeof value === 'string' ? value.toLowerCase() : value,
  )
  @IsDefined({ message: 'Email is required' })
  @IsEmail({}, { message: 'Incorrect email address' })
  @ApiProperty({ example: 'adam@gmail.com' })
  email: string;
  @IsDefined({ message: 'Password is required' })
  @IsString({ message: 'Password must be a string' })
  @ApiProperty({ example: 'MySr0ngP@ssw0rd' })
  password: string;
}
```

Рисунок 3.16– Перевірка на помилки через декоратори.

```
throw new UnauthorizedException('Access token is expired');
```

Рисунок 3.17 – Виведення помилки за некоректні дані в сервісі.

3.2 Інструкція користувача

3.2.1 Вікно авторизації та реєстрації

При відкриті програми вас вітає вікно авторизації.

Можливі дії:

- якщо ви зареєстровані, вам треба ввести логін та пароль та увійти до системи, натиснувши кнопку “Login” (див. рис. 3.18).

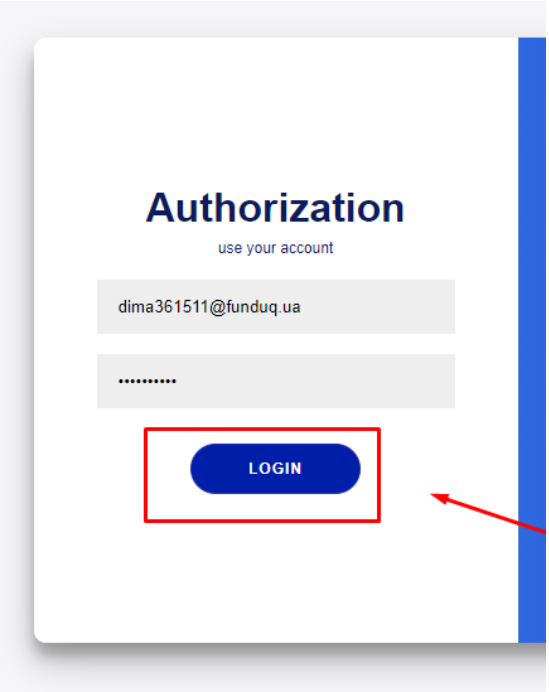

Рисунок 3.18 – Кнопка авторизації

- якщо ви не маєте акаунту ви повинні перейти на сторінку реєстрації після натискання кнопки “Registration” (див. рис. 3.19), ввести ім’я, пароль та пошту, потім натиснути кнопку “Register” (див. рис. 3.20).

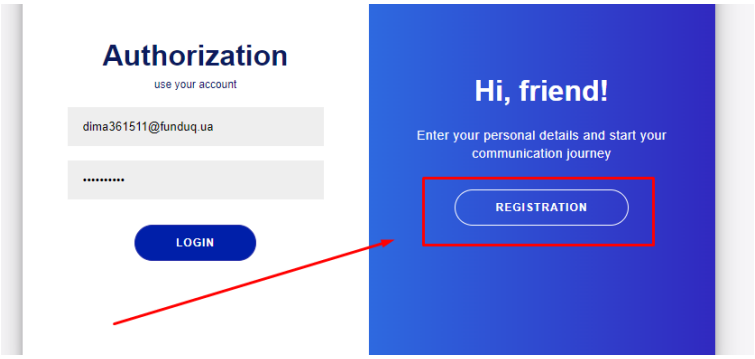

Рисунок 3.19 – Кнопка переходу на реєстрацію

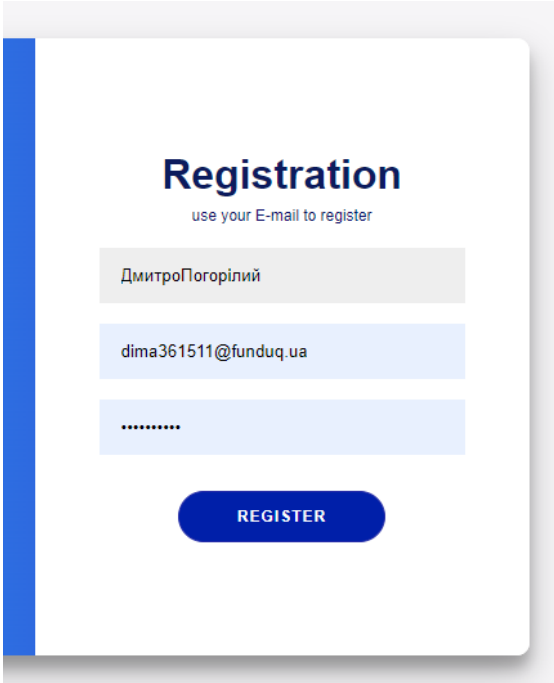

Рисунок 3.20 – Кнопка реєстрації

Після авторизації або реєстрації, ви переходите на головну сторінку вебсервісу.

3.2.2 Головна сторінка

Можливі дії:

- створення чату, потрібно натиснути на кнопку “+” (див. рис. 3.21) та ввести id користувача з яким ви би хотіли почати діалог.

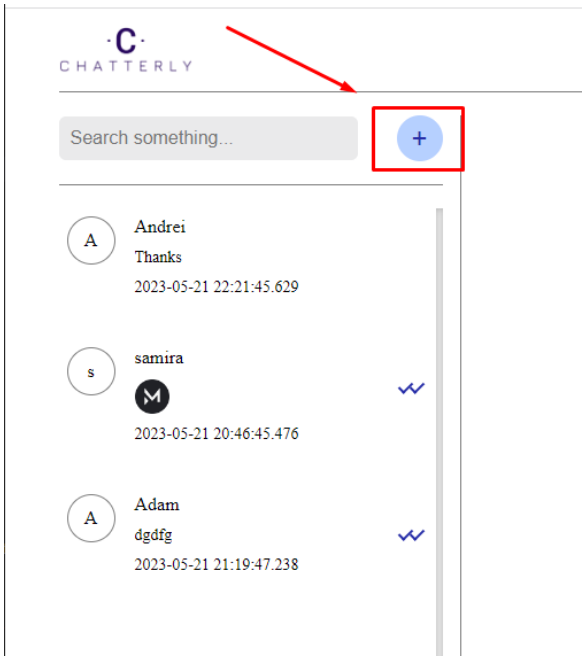

Рисунок 3.21 – Кнопка відкриття вікна для створення чату

- пошук чату по полю для пошуку чатів, після запису даних в це поле та натисканню кнопки “Enter”, відбувається пошук чатів де у користувача є такі літери в імені або в любому повідомленні (див. рис. 3.22, 3.23).

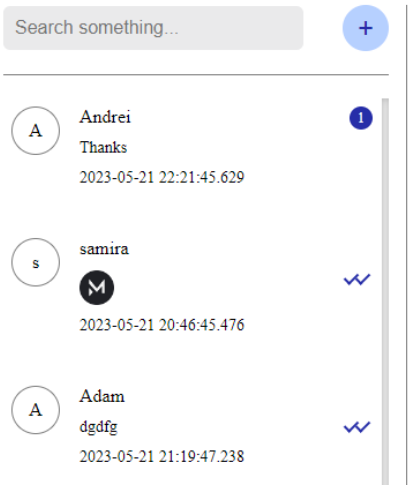

Рисунок 3.22 – Список чатів до початку пошуку

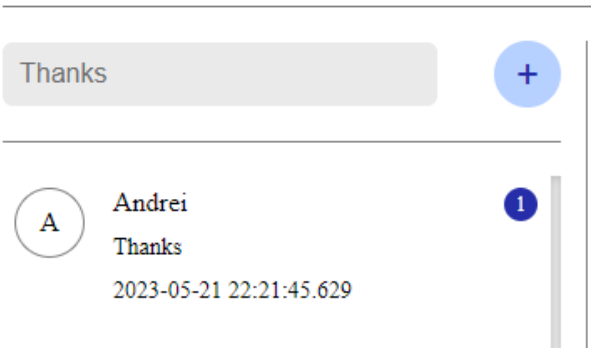

Рисунок 3.23 – Список чатів після пошуку по набору символів “Thanks”

- вивід чату з користувачем, треба натиснути на чат, з’являється список повідомлень (див. рис. 3.24).

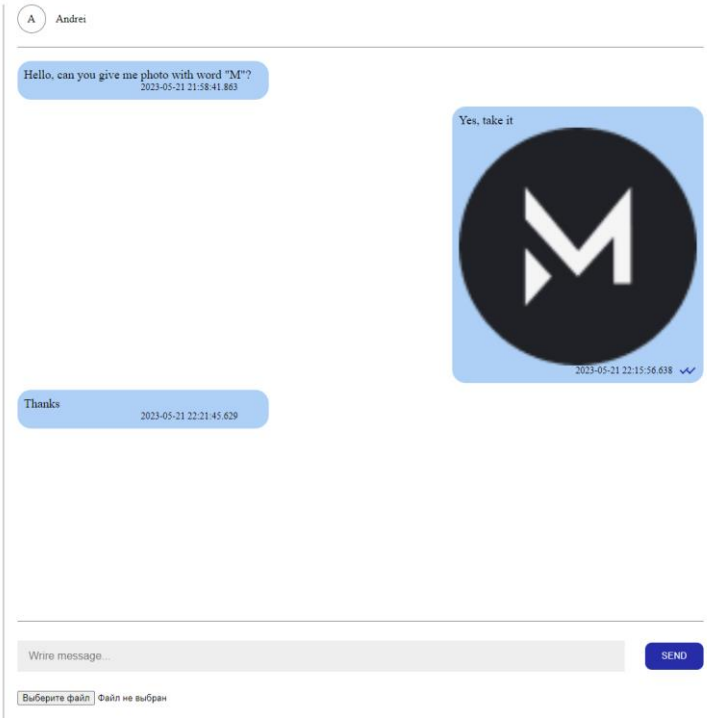

Рисунок 3.24 – Вивід чату з користувачем Andrei

- відправлення повідомлення користувачу, треба в нижній частині чата ввести текст (див. рис. 3.25), зображення (див. рис. 3.26) або і текст і зображення та натиснути кнопку “Send”.

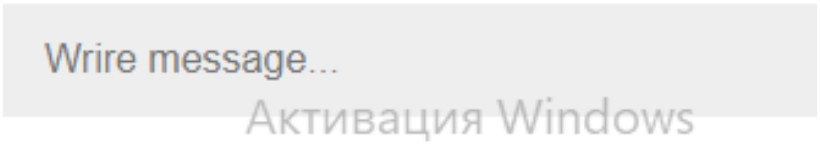

Рисунок 3.25– Поле для вводу тексту

Рисунок 3.26 – Поле для вводу зображення

- вихід з аккаунту, для виходу потрібно натиснути кнопку “Logout” в верхньому правому кутку (див. рис. 3.27).

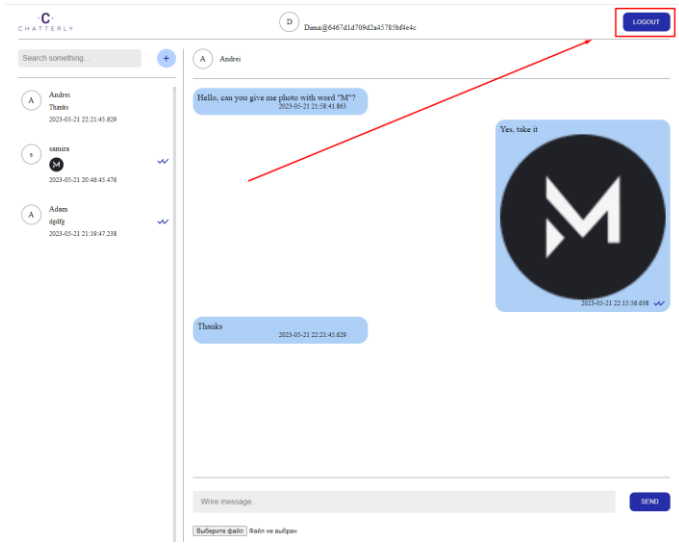

Рисунок 3.27 – Кнопка виходу з аккаунту

3.2.3 Вікно створення чату

В вікні створення чату потрібно ввести коректний id іншого користувача (див. рис. 3.28).

Якщо ви, або користувач вже створювали такий чат, вікно зникне і ні чого не відбудеться.

Якщо ви ввели не коректні дані то з’явиться помилка (див. рис. 3.30), яка буде казати вам, що такого користувача не існує.

Рисунок 3.28 – Введення id іншого користувача

Якщо ви ввели не коректні дані то з’явиться помилка (див. рис. 3.29), яка буде казати вам, що такого користувача не існує.

Рисунок 3.29 – Виведення помилки при некоректних данних

Також є можливість закрити вікно двома способами:

1) натиснути в пустому місці на екрані
2) натиснути на кнопку “Cancel”

3.3 Висновки до третього розділу

В результаті створення вебсервісу отримано багато нових практичних навичок. Освоєно багато нових технологій. Вивчено які саме продукти зараз найцінніші на ринку.

Створено вебсервіс, який можливо інтегрувати в в інші вебсервіси. Це можливо зробити завдяки модульній архітектурі серверної частини. Окремі модулі можливо приєднати до інших проєктів або інтегрувати весь вебсервіс в інший за допомогою мікро-сервісної архітектури.

При написанні серверної частини, для роботи з базою даних, вибір пав на технологію програмування, яка зв’язує базу даних з концепціями об’єктно орієнтованих мов програмування. Після написання прикладів на декількох з них, вибір пав на Prisma. Ця технологія не тільки робить всю роботу по запитах на сервер, але і створює дуже точну типізацію даних. На даний час, конкуренти знаходяться дуже далеко від її рівня.

Клієнтська та серверна частина задовольняють вимогам. При написані було використано багато новітніх технологій. Таким чином, завдання вирішені в повному обсязі.

Клієнтська частина працює швидко завдяки реактивному типу даних у фреймворку Vue.JS. За допомогою бібліотек по маршрутизації та оптимізації запитів на серверну частину, швидкість стає ще більшою.

Серверна частина має захист від сторонніх факторів. За допомогою технології під назвою «CORS» та ліміту на запити в хвилину, зловмисники не зможуть легко зупинити стабільну роботу вебсервісу та відправляти велику кількість некоректних даних

Серверна частина не вимикається при помилках. Наявність класа для перехоплення помилок є однією з функцій в фреймворку NestJS. Незважаючи на тип помилки та її код відповіді, цей клас перехопить її, зробить структурованою та легкою для читання на клієнтській частині та відправить

користувачу або запише в файл з помилками, який адміністратор потім прочитає та зробить висновки.

Використані новітні технології для шифрування даних. За допомогою токенів та криптографічного шифрування зловмисники не зможуть розшифрувати особисті дані користувача. Дані 16 разів перероблюються за стандартом RFC 7519.

Використана технологія для аутентифікації даних яка існує не більше декількох років. При кожному запиті на сервер, клієнтська частина відправляє токен користувача. З його допомогою сервер розуміє кто саме хочу до нього звернутись. Такий токен дуже важно підробити.

ВИСНОВОК

Завершуючи роботу, можна прийти до висновку, що робота з фреймворками та бібліотеками на мові програмування TypeScript та JavaScript – це дуже потужний інструмент. Завдяки цьому набору інструментів можна створювати веб-сервіси, які будуть швидкі, приємні кінцевому користувачу та мають укомплектовану та зрозумілу архітектуру.

Visual Studio Code, що є однією з найрозвиненіших в своїй категорії, дозволяє повноцінно реалізувати програмне забезпечення, забезпечує всі стандарти TypeScript , підтримує підключення різних бібліотек, створення індивідуальних налаштувань, та ще багато іншого.

У даній роботі були розглянуті особливості побудови та роботи з двома популярними фреймворками.. У результаті була створена програма, яка задовольняє умовам поставленої задачі. Повністю реалізуються можливості аутентифікації за допомогою токенів, введення діалогів в реальному часі, відправлення текстових повідомлень, медіа повідомлень та все разом.

Вебсервіс може бути конкурентом для схожих типів продуктів. Має можливості, які не мають вебсервіси інших невеликих компаній. З великими вебсервісами, такими як “Telegram” та “Facebook”, зараз конкурувати не зможе.

Створений програмний продукт кваліфікується за категорією «Система забезпечення віддаленого зв’язку».

Якщо відокремити серверну частину від клієнтської частини, то можливо провести детальний аналіз плюсів та недоліків вебсервісу.

Плюси клієнтської частини:

1) легка в підтримані;
2) фреймворк має велике майбутнє та вже має багато вдосконалень, які будуть викладені в ближчому майбутньому;
3) велика швидкість;
4) динамічна зміна даних;

5) в вебсервісу були враховані всі новітні тенденції на ринку вебсервісів які пропонують віддалений зв’язок;
6) підтримка багатьма браузерами.

Недоліки клієнтської частини:

1) є багато нереалізованих шляхів для оптимізації;
2) треба більший захист даних.

Пропозиції по покращенню клієнтської частини:

1) створення слайдера для галереї фотографій;
2) налаштування профілю;
3) видалення чатів та смс;
4) використання стікерів;
5) додавання більше одного файлу в повідомлення;
6) видалення профілю;
7) додавання акаунтів в список заблокованих осіб;
8) введення панелі для адміністраторів.

Плюси серверної частини:

1) шифрування даних;
2) легка для розуміння архітектура;
3) майже всі дані проходять перевірку;
4) швидке завантаження файлів;
5) сумісність з багатьма бібліотеками;
6) перехоплення всіх помилок.

Недолік у серверній частині пов’язаний з людським фактором, якщо у когось є інформація про ключі доступу, то він зможе призупинити нормальне функціонування програми.

Пропозиції по покращенню серверної частини:

1) відслідковувати статус користувача в системі;
2) більш детальні повідомлення для клієнтської частини WebSocket;

Модульна архітектура є дуже популярної у виробництві вебсервісів. Розробка може вестись окремо кожного модуля і потім лише поєднуватись за допомогою більш глобальних модулів, або імпортуючи один модуль в інший.

Фреймворк NestJS надає велике поле для уяви за допомогою модульної архітектури. Даний вебсервіс можливо інтегрувати в інші більші системи як мікро-сервіс. Наприклад, при розробці сервісу для банку, цей вебсеріс можливо використовувати як мікросервіс для спілкування між клієнтами та підтримкою банку. Або використовувати як мікро-сервіс для комунікації в сервісах для доставлення їжі.

Можливе розширення даного вебсервісу за рахунок різних мікро-сервісів.

Перший приклад - це комунікація між носіями різних мов. Для цього потрібно імпортувати мікро-сервіс під назвою «Автоматичний переклад».

Другий приклад – це чат з можливістю переведення криптовалюти між рахунками. Для цього потрібно імпортувати мікро-сервіс який буде відповідати за захист транзакцій та мікро-сервіс для зручного переведення різних типів фінансів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Документація по MongoDB. URL: https://www.mongodb.com/docs/

2. Документація по бібліотеці AXIOS. URL: https://axios-http.com/uk/docs/intro

3. Документація по Node.js. URL: https://nodejs.org/uk/docs/

4. Документація по NestJS. URL: https://docs.nestjs.com/

5. Головний сайт Json Web Tokens. URL: https://jwt.io/

6. Документація по Vue.js. URL: https://vuejs.org/guide/introduction.html

7. Документація по Vuex Store. URL: https://vuex.vuejs.org/guide/

8. Документація по Vue Router. URL: https://router.vuejs.org/guide/

9. Документація по Swagger. URL: https://swagger.io/solutions/api-documentation/

10. Документація по Prisma. URL: https://www.prisma.io/docs/concepts/overview/what-is-prisma

11. Документація по Express. URL: https://expressjs.com/ru/guide/routing.html

12. Документація по bcrypt. URL: https://www.npmjs.com/package/bcrypt

13. Документація по moment. URL: https://momentjs.com/docs/

14. Документація по Passport. URL: https://www.passportjs.org/docs/

15. Документація по Vite. URL: https://vitejs.dev/guide/

16. Документація по Atlas. URL: https://www.mongodb.com/atlas

17. Документація по Pinia. URL: https://pinia.vuejs.org/

18. Документація по Fastify. URL: https://www.fastify.io/

18. Документація по Socket.IO. URL: https://socket.io/docs/v4/

19. Документація по WebSocket. URL: https://github.com/theturtle32/WebSocket-Node/tree/cce6d468986dd356a52af5630fd8ed5726ba5b7a

19. Документація по Json Web Tokens для Node.js. URL: https://www.npmjs.com/package/jsonwebtoken

20. Документація по Swagger. URL: https://swagger.io/docs/

21. Документація по class-validator. URL: https://www.npmjs.com/package/class-validator

ДОДАТОК А

Додаток А.1: Код серверної частини

schema.prisma

```
//start page
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "mongodb"
  url      = env("DATABASE_URL")
}

model User {
  id           String          @id @default(auto()) @map("_id") @db.ObjectId
  email        String          @unique
  nickname     String
  password     String
  createdAt    DateTime        @default(now())
  access_tokens UserAccessToken[]
  UserChat     UserChat[]
  Message      Message[]
  UserMessage  UserMessage[]
}

model UserRefreshToken {
  id           String          @id @default(auto()) @map("_id") @db.ObjectId
  expires_at   DateTime        @db.Timestamp
```

```
  access_token_id String          @unique @db.ObjectId
   access_token       UserAccessToken  @relation(references:  [id],  fields:
[access_token_id], onDelete: Cascade, onUpdate: Cascade)
}

model UserAccessToken {
  id           String          @id @default(auto()) @map("_id") @db.ObjectId
  expires_at    DateTime         @db.Timestamp
  user         User            @relation(references: [id], fields: [user_id], onDelete:
Cascade, onUpdate: Cascade)
  user_id       String          @db.ObjectId
  refresh_token UserRefreshToken?
}

model Chat {
  id           String       @id @default(auto()) @map("_id") @db.ObjectId
  created_at    DateTime      @default(now())
  updated_at    DateTime      @default(now())
  messages      Message[]
  user_chats    UserChat[]
  user_messages UserMessage[]
}

model UserChat {
  id      String @id @default(auto()) @map("_id") @db.ObjectId
  user_id String @db.ObjectId
  chat_id String @db.ObjectId
   user      User     @relation(fields: [user_id], references: [id], onUpdate:
Cascade, onDelete: Cascade)
```

```
    chat      Chat     @relation(fields: [chat_id], references: [id], onUpdate:
Cascade, onDelete: Cascade)
  }

  model Message {
    id          String       @id @default(auto()) @map("_id") @db.ObjectId
    user_id     String       @db.ObjectId
    chat_id     String       @db.ObjectId
    text        String?
    files       String[]
    created_at  DateTime     @default(now())
    user        User         @relation(fields: [user_id], references: [id], onUpdate:
Cascade)
    chat        Chat         @relation(fields: [chat_id], references: [id], onUpdate:
Cascade, onDelete: Cascade)
    user_messages UserMessage[]
  }

  model UserMessage {
    id         String   @id @default(auto()) @map("_id") @db.ObjectId
    chat_id    String   @db.ObjectId
    user_id    String   @db.ObjectId
    message_id String   @db.ObjectId
    read       Boolean  @default(false)
    created_at DateTime @default(now())
    chat       Chat     @relation(fields: [chat_id], references: [id], onUpdate:
Cascade, onDelete: Cascade)
    user       User     @relation(fields: [user_id], references: [id], onUpdate:
Cascade, onDelete: Cascade)
```

```
    message    Message  @relation(fields: [message_id], references: [id], onUpdate: Cascade, onDelete: Cascade)
  }
  //end page
  login.dto.ts
  //start page
  import { ApiProperty } from '@nestjs/swagger';
  import { Transform } from 'class-transformer';
  import { IsDefined, IsEmail, IsString } from 'class-validator';

  export class LoginDto {
    @Transform(({ value }) =>
      typeof value === 'string' ? value.toLowerCase() : value,
    )
    @IsDefined({ message: 'Email is required' })
    @IsEmail({}, { message: 'Incorrect email address' })
    @ApiProperty({ example: 'adam@gmail.com' })
    email: string;
    @IsDefined({ message: 'Password is required' })
    @IsString({ message: 'Password must be a string' })
    @ApiProperty({ example: 'MySr0ngP@ssw0rd' })
    password: string;
  }
  //end page
  refresh.dto.ts
  //start page
  import { IsDefined, IsString } from 'class-validator';

  export class RefreshDto {
```

```
  @IsDefined()
  @IsString()
  token: string;
}
//end page
```

register.dto.ts

```
//start page
import { ApiProperty } from '@nestjs/swagger';
import { Transform } from 'class-transformer';
import { IsDefined, IsEmail, IsString, Length } from 'class-validator';
import { EntityExists } from '../../helpers/validation/entity-exists';
export class RegisterDto {
  @Transform(({ value }) =>
    typeof value === 'string' ? value.toLowerCase() : value,
  )
  @IsDefined({ message: 'Email is required.' })
  @IsEmail({}, { message: 'Please enter a valid email address.' })
  @Length(7, 100, {
    message: 'Email must be between 7 and 150 characters long.',
  })
  @EntityExists('user', 'email', false)
  @ApiProperty({ example: 'adam@gmail.com' })
  email: string;
  @IsDefined({ message: 'Password is required.' })
  @Length(8, 32, {
    message: 'Password must be between 8 and 32 characters long.',
  })
  @IsString({ message: 'Password must be a string.' })
  @ApiProperty({ example: 'MySr0ngP@ssw0rd' })
```

```
  password: string;
  @IsDefined({ message: 'Nickname is required.' })
  @IsString({ message: 'Nickname must be a string.' })
  @Length(2, 50, {
    message: 'Nickname must be between 2 and 50 characters long.',
  })
  @ApiProperty({ example: 'Adam' })
  nickname: string;
}
//end page
```

jwt-auth.guard.ts

```
//start page
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor() {
    super();
  }

  handleRequest(err, user) {
    if (err) {
      throw err;
    }
    if (user) return user;
    throw new UnauthorizedException('Invalid token');
  }
}
```

```
//end page
local-auth-guard.ts
//start page
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {
  handleRequest(err, user) {
    if (!user) {
      throw new UnauthorizedException('Incorrect email or password');
    }
    return user;
  }
}
//end page
payload.interface.ts
//start page
export interface IPayload {
  sub: string;
  jti: string;
  exp?: number;
}
//end page
request.interface.ts
//start page
export interface IRequest {
  user: IRequestUser;
}
```

```
export interface IRequestUser {
  id: string;
  token_id: string;
}
//end page
jwt.strategy.ts
//start page
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable, UnauthorizedException } from '@nestjs/common';
import 'dotenv/config';
import { AuthService } from '../auth.service';

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly authService: AuthService) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: process.env.JWT_SECRET_KEY,
    });
  }

  async validate(payload: any) {
    const user = await this.authService.validateAccessToken(payload);
    if (!user) {
      throw new UnauthorizedException('Invalid token');
    }
    return user;
```

```
  }
}
//end page
local-strategy.ts
//start page
import { Strategy } from 'passport-local';
import { PassportStrategy } from '@nestjs/passport';
import {
  Injectable,
  UnauthorizedException,
  UnprocessableEntityException,
} from '@nestjs/common';
import { AuthService } from '../auth.service';
import { plainToInstance } from 'class-transformer';
import { LoginDto } from '../dto/login.dto';
import { validateSync } from 'class-validator';

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super({
      usernameField: 'email',
      passwordField: 'password',
    });
  }

  //verified callback
  async validate(email: string, password: string): Promise<any> {
    const loginDto = plainToInstance(LoginDto, { email, password });
```

```
    const validationErrors = validateSync(loginDto);
    if (validationErrors.length !== 0) {
      throw new UnprocessableEntityException({
        errors: validationErrors,
      });
    }
    const user = await this.authService.validateUser(email, password);
    if (!user) {
      throw new UnauthorizedException('Incorrect email or password');
    }
    return user;
  }
}
//end page
```

auth.controller.ts

```
//start page
import {
  Body,
  Controller,
  Get,
  HttpCode,
  Post,
  Req,
  UseGuards,
} from '@nestjs/common';
import {
  ApiBody,
  ApiCreatedResponse,
  ApiForbiddenResponse,
```

```
  ApiOperation,
  ApiTags,
  ApiUnprocessableEntityResponse,
  ApiOkResponse,
  ApiBearerAuth,
  ApiUnauthorizedResponse,
} from '@nestjs/swagger';

import {
  ForbiddenResponse,
  UnauthorizedResponse,
  UnprocessableEntityResponse,
} from '../helpers/docs/errorSchema';
import { SuccessResponseDocs } from '../helpers/docs/successShema';
import { AuthService } from './auth.service';
import { LoginDto } from './dto/login.dto';
import { LoginEntity, TokensEntity } from './entities/login.entity';
import { LocalAuthGuard } from './guards/local-auth-guard';
import { IRequest } from './interfaces/request.interface';
import { RefreshDto } from './dto/refresh.dto';
import { JwtAuthGuard } from './guards/jwt-auth.guard';
import { RegisterEntity } from './entities/register';
import { RegisterDto } from './dto/register.dto';

@ApiTags('Auth')
@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}
```

```
// #region Swagger Decorators
@ApiUnprocessableEntityResponse(
  UnprocessableEntityResponse('first_name', 'First name is required'),
)
@ApiCreatedResponse({ type: RegisterEntity })
@ApiOperation({ summary: 'Register' })
// #endregion
@Post('register')
register(@Body() registerDto: RegisterDto): Promise<LoginEntity> {
  return this.authService.register(registerDto);
}

// #region Swagger Decorators
@ApiOperation({ summary: 'Login' })
@ApiUnprocessableEntityResponse(
  UnprocessableEntityResponse('password', 'Password is required'),
)
@ApiForbiddenResponse(ForbiddenResponse('Incorrect credentials'))
@ApiCreatedResponse({ type: LoginEntity })
@ApiBody({ type: LoginDto })
@HttpCode(200)
// #endregion
@UseGuards(LocalAuthGuard)
@Post('login')
login(@Req() { user }: IRequest) {
  return this.authService.login(user.id);
}

// #region Swagger Decorators
```

```
  @ApiOperation({ summary: 'Refresh token' })
  @ApiCreatedResponse({ type: TokensEntity })
  @ApiUnauthorizedResponse(UnauthorizedResponse)
  @ApiUnprocessableEntityResponse(
    UnprocessableEntityResponse('token', 'Token is required'),
  )
  @ApiBody({ type: RefreshDto })
  @HttpCode(200)
  // #endregion
  @Post('refresh')
  refresh(@Body() refreshDto: RefreshDto): Promise<TokensEntity> {
    return this.authService.validateRefreshToken(refreshDto.token);
  }

  // #region Swagger Decorators
  @ApiOperation({ summary: 'Logout' })
  @ApiOkResponse(SuccessResponseDocs)
  @ApiUnauthorizedResponse(UnauthorizedResponse)
  @ApiBearerAuth()
  // #endregion
  @UseGuards(JwtAuthGuard)
  @Get('/logout')
  logout(@Req() { user }: IRequest) {
    return this.authService.logout(user.token_id);
  }
}
//end page
```

auth.module.ts

```
//start page
```

```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { AuthController } from './auth.controller';
import { PrismaModule } from '../prisma/prisma.module';
import { ConfigService } from '@nestjs/config';
import { LocalStrategy } from './strategies/local-strategy';
import { FileHelper } from '../helpers/file-system/file-helper';
import { EntityExistsRule } from '../helpers/validation/entity-exists';
import { JwtModule } from '@nestjs/jwt';
import { JwtStrategy } from './strategies/jwt.strategy';

@Module({
 imports: [
  PrismaModule,
  JwtModule.registerAsync({
   useFactory: (config: ConfigService) => {
    return {
     secret: config.get('JWT_SECRET_KEY'),
     signOptions: {
      expiresIn: config.get<number>('JWT_EXPIRATION_TIME'),
     },
    };
   },
   inject: [ConfigService],
  }),
 ],
 controllers: [AuthController],
 providers: [
  AuthService,
```

```
    LocalStrategy,
    JwtStrategy,
    FileHelper,
    EntityExistsRule,
  ],
})
export class AuthModule {}
//end page
auth.service.ts
//start page
import {
  Injectable,
  UnauthorizedException,
  UnprocessableEntityException,
} from '@nestjs/common';
import * as bcrypt from 'bcrypt';
import { IPayload } from './interfaces/payload.interface';
import { ConfigService } from '@nestjs/config';
import * as moment from 'moment';
import { JwtService } from '@nestjs/jwt';
import { PrismaService } from '../prisma/prisma.service';
import { PrismaPromise } from '@prisma/client';
import { RegisterDto } from './dto/register.dto';

@Injectable()
export class AuthService {
  constructor(
    private readonly prisma: PrismaService,
    private readonly configService: ConfigService,
```

```
  private readonly jwtService: JwtService,
) {}

async register(data: RegisterDto) {
  data.password = this.hashPassword(data.password);
  const user = await this.prisma.user.create({
    data,
    select: {
      id: true,
    },
  });
  return await this.login(user.id);
}

async validateUser(email: string, password: string): Promise<any> {
  const user = await this.prisma.user.findUnique({
    where: { email },
    select: {
      id: true,
      email: true,
      password: true,
    },
  });
  if (user && bcrypt.compareSync(password, user.password)) {
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    const { password, ...result } = user;
    return {
      ...result,
    };
```

```
  }
  return null;
}

async validateRefreshToken(token: string) {
  let payload: IPayload;
  try {
    payload = this.jwtService.verify(token, {
      secret: this.configService.get('JWT_REFRESH_SECRET_KEY'),
    });
  } catch {
    throw new UnprocessableEntityException('Uncorrected token');
  }
  const refresh_token = await this.prisma.userRefreshToken.findUnique({
    where: { id: payload.jti },
    select: {
      id: true,
      access_token: { select: { user_id: true } },
      expires_at: true,
    },
  });
  if (!refresh_token || refresh_token.access_token.user_id !== payload.sub) {
    throw new UnauthorizedException('Invalid refresh token');
  }
  if (refresh_token.expires_at.valueOf() < Date.now().valueOf()) {
    await this.prisma.userRefreshToken.delete({
      where: { id: payload.jti },
    });
    throw new UnauthorizedException('Refresh token is expired');
```

```
  }
  return this.getNewTokens(payload.sub, payload.jti);
}

async validateAccessToken(payload: IPayload) {
  const { user_id, ...access_token } =
    await this.prisma.userAccessToken.findUniqueOrThrow({
      where: { id: payload.jti },
      select: {
        user_id: true,
        id: true,
        expires_at: true,
      },
    });
  if (access_token.expires_at.valueOf() < Date.now().valueOf()) {
    throw new UnauthorizedException('Access token is expired');
  }
  if (user_id !== payload.sub) {
    throw new UnauthorizedException('Invalid access token');
  }
  return {
    id: payload.sub,
    token_id: payload.jti,
  };
}

async login(user_id: string) {
  const user = await this.prisma.user.findFirstOrThrow({
    where: { id: user_id },
```

```
    select: {
      id: true,
      email: true,
      nickname: true,
    },
  });
  const tokens = await this.getNewTokens(user_id);
  return {
    user,
    ...tokens,
  };
}

async logout(token_id: string) {
  await this.prisma.userAccessToken.delete({
    where: { id: token_id },
  });
  return { status: 'success' };
}

async getNewTokens(user_id: string, access_id?: string) {
  const promiseArray: PrismaPromise<any>[] = [];
  const refresh_expires_in = +this.configService.get<string>(
    'JWT_REFRESH_EXPIRATION_TIME',
  );
  const access_expires_in = +this.configService.get<string>(
    'JWT_EXPIRATION_TIME',
  );
  promiseArray.push(
```

```
  this.prisma.userAccessToken.create({
    data: {
      user_id,
      expires_at: moment().add(refresh_expires_in, 'ms').toDate(),
      refresh_token: {
        create: {
          expires_at: moment().add(access_expires_in, 'ms').toDate(),
        },
      },
    },
    select: { id: true, refresh_token: { select: { id: true } } },
  }),
);
if (access_id) {
  promiseArray.push(
    this.prisma.userAccessToken.delete({
      where: { id: access_id },
    }),
  );
}
const [access] = await this.prisma.$transaction(promiseArray);
const payloadAccess: IPayload = {
  sub: user_id,
  jti: access.id,
};
const payloadRefresh: IPayload = {
  sub: user_id,
  jti: access.refresh_token.id,
};
```

```
      const refresh_token = this.jwtService.sign(payloadRefresh, {
                                                                    expiresIn:
this.configService.get<number>('JWT_REFRESH_EXPIRATION_TIME'),
        secret: this.configService.get('JWT_REFRESH_SECRET_KEY'),
      });
      const access_token = this.jwtService.sign(payloadAccess);
      return {
        access_token,
        refresh_token,
      };
    }

    hashPassword(password: string): string {
      return bcrypt.hashSync(password, this.configService.get('JWT_SALT'));
    }
  }
  //end page
  create-chat.dto.ts
  //start page
  import { ApiProperty } from '@nestjs/swagger';
  import { IsDefined, IsMongoId } from 'class-validator';

  import { EntityExists } from '../../helpers/validation/entity-exists';

  export class CreateChatDto {
    @EntityExists('user')
    @IsMongoId()
    @IsDefined({ message: 'id must be filled' })
    @ApiProperty({ example: '35353535345' })
```

```
  id: string;
}
//end page
```

get-chats-query.dto.ts

```
//start page
import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsOptional, IsString, MinLength } from 'class-validator';

import { PaginationQueryDto } from '../../helpers/dto/pagination-query.dto';

export class GetChatsQueryDto extends PaginationQueryDto {
  @MinLength(1)
  @IsString()
  @IsOptional()
  @ApiPropertyOptional({
    example: 'Lo',
    description: 'String for universal search',
  })
  universal?: string;
}
//end page
```

chats.controller.ts

```
//start page
import {
  Controller,
  Get,
  Post,
  Body,
  Patch,
```

```
  Param,
  Delete,
  UseGuards,
  Req,
  Query,
} from '@nestjs/common';
import { ChatsService } from './chats.service';
import {
  ApiBody,
  ApiCreatedResponse,
  ApiOperation,
  ApiOkResponse,
  ApiTags,
} from '@nestjs/swagger';
import { CreateChatDto } from './dto/create-chat.dto';
import { SuccessResponseWithIdDocs } from '../helpers/docs/successShema';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { IRequest } from '../auth/interfaces/request.interface';
import { GetChatsQueryDto } from './dto/get-chats-query.dto';
import { SearchEntity } from './entities/get-chats.entities';

@ApiTags('Chats')
@Controller('chats')
export class ChatsController {
  constructor(private readonly chatsService: ChatsService) {}

  // #region Swagger Decorators
  @ApiOperation({ summary: 'Create one chat between people' })
  @ApiBody({ type: CreateChatDto })
```

```
  @ApiCreatedResponse(SuccessResponseWithIdDocs)
  // #endregion
  @UseGuards(JwtAuthGuard)
  @Post()
  createChat(
    @Body() createChatDto: CreateChatDto,
    @Req() { user: { id } }: IRequest,
  ) {
    return this.chatsService.createChat(id, createChatDto);
  }

  // #region Swagger Decorators
  @ApiOperation({ summary: 'Get chats' })
  @ApiOkResponse({ type: SearchEntity })
  // #endregion
  @UseGuards(JwtAuthGuard)
  @Get()
  getChats(
    @Req() { user: { id } }: IRequest,
    @Query() getChatsQueryDto: GetChatsQueryDto,
  ): Promise<SearchEntity[]> {
    return this.chatsService.getChats(id, getChatsQueryDto);
  }
}
//end page
```

chats.module.ts

```
//start page
import { Module } from '@nestjs/common';
import { ChatsService } from './chats.service';
```

```
import { ChatsController } from './chats.controller';
import { PrismaModule } from 'src/prisma/prisma.module';
import { FileHelper } from 'src/helpers/file-system/file-helper';

@Module({
  imports: [PrismaModule],
  controllers: [ChatsController],
  providers: [ChatsService, FileHelper],
})
export class ChatsModule {}
//end page
chats.service.ts
//start page
import { Injectable } from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateChatDto } from './dto/create-chat.dto';
import { Prisma } from '@prisma/client';
import { GetChatsQueryDto } from './dto/get-chats-query.dto';
import { FileHelper } from 'src/helpers/file-system/file-helper';

@Injectable()
export class ChatsService {
  constructor(
    private readonly prisma: PrismaService,
    private readonly fileHelper: FileHelper,
  ) {}

  async createChat(user_id: string, { id }: CreateChatDto) {
    const ids = [user_id, id];
```

```
  //chat was created?
  const createdChat = await this.prisma.chat.findFirst({
    where: {
      user_chats: { every: { user_id: { in: ids } } },
    },
    select: {
      id: true,
    },
  });
  //return chat_id if chat was created
  if (createdChat) {
    return { status: 'success', id: createdChat.id };
  }
  //create chat
  const chat = await this.prisma.chat.create({
    data: {
      user_chats: {
        createMany: {
          data: ids.map((id) => ({
            user_id: id,
          })),
        },
      },
    },
    select: { id: true },
  });
  return { status: 'success', id: chat.id };
}
```

```
async getChats(
  user_id: string,
  { skip = 0, take = 10, universal }: GetChatsQueryDto,
) {
  //create where
  const where: Prisma.ChatWhereInput = {
    user_chats: { some: { user_id } },
  };
  if (universal) {
    where.OR = [
      {
        user_chats: { some: { user: { nickname: { contains: universal } } } },
      },
      {
        messages: {
          some: { text: { contains: universal } },
        },
      },
    ];
  }
  //get chats
  const chats = await this.prisma.chat.findMany({
    where,
    select: {
      id: true,
      user_chats: {
        where: { user_id: { not: user_id } },
        select: {
          user: {
```

```
        select: {
          nickname: true,
          email: true,
        },
      },
    },
  },
  user_messages: {
    where: { user_id, read: false },
    select: { id: true },
  },
  messages: {
    select: {
      user_id: true,
      text: true,
      files: true,
      created_at: true,
      user_messages: {
        orderBy: { created_at: 'desc' },
        select: { read: true, user_id: true },
        where: { user_id: { not: user_id } },
        take: 1,
      },
    },
    take: 1,
    orderBy: { created_at: 'desc' },
  },
},
orderBy: { created_at: 'desc' },
```

```
      skip,
      take,
    });
    return chats.map(({ messages, id, user_messages, user_chats }) => ({
      id,
      user: user_chats[0].user,
      messages:
        messages.length > 0
          ? {
              created_at: messages[0].created_at,
              text: messages[0].text,
              files: this.fileHelper.getFullPath(messages[0]?.files[0]),
              unread_count: user_messages.length,
              sent:
                user_id === messages[0].user_id &&
                messages[0].user_messages[0].read === false
                  ? true
                  : undefined,
              delivered:
                user_id === messages[0].user_id &&
                messages[0].user_messages[0].read === true
                  ? true
                  : undefined,
            }
          : null,
    }));
  }
}
//end page
```

errorSchema.ts

```
//start page
import { ApiResponseOptions } from '@nestjs/swagger';

const ErrorSchema: ApiResponseOptions = {
  schema: {
    properties: {
      message: { type: 'string' },
      error: { type: 'string' },
      errors: {
        properties: {
          fieldName: {
            properties: {
              constraints: { type: 'array', items: { type: 'string' } },
              children: {
                properties: {
                  fieldIndex: {
                    properties: {
                      constraints: { type: 'array', items: { type: 'string' } },
                      children: { type: 'array', items: { type: 'object' } },
                    },
                  },
                },
              },
            },
          },
        },
      },
      status: { type: 'number' },
```

```
      timestamp: { type: 'string' },
      path: { type: 'string' },
    },
  },
};
export const UnprocessableEntityResponse = (
  field = 'name',
  message = 'Name is required',
): ApiResponseOptions => ({
  schema: {
    ...ErrorSchema,
    example: {
      status: 422,
      message: 'Unprocessable entity',
      error: 'Invalid data',
      errors: {
        [field]: {
          constraints: [message],
          children: {},
        },
      },
      timestamp: '2022-09-09T07:53:01.147Z',
      path: '/example/path/from/request',
    },
  },
});
export const ForbiddenResponse = (
  message = 'You cannot do this',
): ApiResponseOptions => ({
```

```
  schema: {
    ...ErrorSchema,
    example: {
      message,
      error: 'Forbidden',
      status: 403,
      timestamp: '2022-09-09T07:35:53.654Z',
      path: '/example/path/from/request',
    },
  },
});

export const ForbiddenUserNotActive: ApiResponseOptions = {
  schema: {
    ...ErrorSchema,
    example: {
      message: 'User is not active',
      error: 'Forbidden',
      status: 403,
      timestamp: '2022-09-09T07:35:53.654Z',
      path: '/auth/login',
    },
  },
};
export const NotFoundResponse: ApiResponseOptions = {
  schema: {
    ...ErrorSchema,
    example: {
      message: 'Company not found',
```

```
      error: 'Not Found',
      status: 404,
      timestamp: '2022-09-09T07:19:13.422Z',
      path: '/companies/13562/departments/1',
    },
  },
};
export const UnauthorizedResponse: ApiResponseOptions = {
  schema: {
    ...ErrorSchema,
    example: {
      message: 'Unauthorized',
      error: 'Unauthorized',
      status: 401,
      timestamp: '2022-09-09T07:30:38.089Z',
      path: '/companies/13562/departments/1',
    },
  },
};
//end page
successShema.ts
//start page
import { ApiResponseOptions } from '@nestjs/swagger';

export const SuccessResponseDocs: ApiResponseOptions = {
  schema: {
    properties: {
      status: { type: 'string', example: 'success' },
    },
```

```
  },
};

export const SuccessResponseWithIdDocs: ApiResponseOptions = {
  schema: {
    properties: {
      status: { type: 'string', example: 'success' },
      id: { type: 'string', example: '1312313131313' },
    },
  },
};
//end page
file-helper.ts
//start page
import { Injectable, NotFoundException } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { mkdir, rm, stat, rename, rmdir } from 'fs/promises';
import { existsSync, mkdirSync, createWriteStream, createReadStream } from 'fs';
import { getExtension, getType } from 'mime';
import { pipeline } from 'stream';
import { promisify } from 'util';
import { StorageFolders } from './storage-folders.enum';

@Injectable()
export class FileHelper {
  private storage: string;
  private app_public = 'app\\public\\';
  constructor(private readonly configService: ConfigService) {
```

```
        this.storage = this.configService.get('STORAGE_FOLDER') +
'\\app\\public\\';
    if (!existsSync(this.storage)) mkdirSync(this.storage, { recursive: true });
    if (
      !existsSync(
        this.configService.get('STORAGE_FOLDER') + '\\' +
StorageFolders.TEMP,
      )
    ) {
      mkdirSync(
        this.configService.get('STORAGE_FOLDER') + '\\' +
StorageFolders.TEMP,
      );
    }
    if (!existsSync(this.configService.get('STORAGE_FOLDER') + '\\logs'))
      mkdirSync(this.configService.get('STORAGE_FOLDER') + '\\logs');
    if (!existsSync(this.storage + '\\chats'))
      mkdirSync(this.storage + '\\chats');
  }

  async save(fileBody, mimetype: string, path: string, randomName = false) {
    if (randomName)
      path += `${Date.now()}-${Math.round(Math.random() * 10000)
        .toString()
        .padStart(5, '0')}`;
    const destination =
      this.configService.get('STORAGE_FOLDER') +
      '\\' +
      path +
```

```
        '.' +
        getExtension(mimetype);
      const pump = await promisify(pipeline);
      await pump(fileBody, createWriteStream(destination));
      return destination;
    }

    async read(path) {
      if (existsSync(path)) {
        return createReadStream(path);
      } else {
                                                    throw                    new
NotFoundException('exceptions.NOT_FOUND|{"args":["FILE"]}');
      }
    }

    async createFolder(path: string) {
      if (!existsSync(this.storage + path)) {
        const new_folder = await mkdir(this.storage + path, { recursive: true });
      }
      return path;
    }

    async removeFolder(path) {
      await rm(this.storage + path, { recursive: true });
    }

    async moveTemp(oldPath: string, chat_id: string) {
      try {
```

```
    if (!oldPath.match(`\\\\${StorageFolders.TEMP}\\\\`)) {
      return oldPath;
    }
    const folder = 'chats\\' + chat_id;
    const newPath = oldPath.replace(
      StorageFolders.TEMP,
      this.app_public + folder,
    );
    await this.createFolder(folder);
    await rename(oldPath, newPath);
    return newPath;
  } catch (error) {
    throw error;
  }
}

async remove(path: string | string[]) {
  try {
    if (typeof path === 'string') {
      await rm(path);
    } else {
      await Promise.all(path.map((filePath) => rm(filePath)));
    }
  } catch {}
}

getFullPath(path: string) {
  if (!path) {
    return null;
```

```
    }
    console.log(path);
    path = path.replace('storage\\app\\public\\', '');
    return process.env.APP_URL + '\\' + path;
  }

  log(type: 'app' | 'db', log: object) {
    const logFile = createWriteStream(
      `${this.configService.get('STORAGE_FOLDER')}\\logs\\${new Date()
        .toISOString()
        .slice(0, 10)}_${type}.log`,
      { flags: 'a' },
    );
    logFile.write(
      JSON.stringify({ time: new Date().toISOString(), ...log }) + '\n',
    );
  }
}
//end page
entity-exists.ts
//start page
import {
  ValidationArguments,
  ValidationOptions,
  registerDecorator,
  ValidatorConstraint,
  ValidatorConstraintInterface,
} from 'class-validator';
import { Injectable } from '@nestjs/common';
```

```
import { PrismaService } from '../../prisma/prisma.service';

@ValidatorConstraint({ name: 'EntityExists', async: true })
@Injectable()
export class EntityExistsRule implements ValidatorConstraintInterface {
 constructor(private readonly prisma: PrismaService) {}

 async validate(value: any, args: ValidationArguments) {
  const [entity, column, shouldExist] = args.constraints;
  const where: {
   [field: string]: any;
  } = {};
  if (!value) return true;
  if (Array.isArray(value)) {
   if (!value.every((element) => typeof element === 'string')) return false;
   where.id = { in: value };
  } else {
   if (column === 'id' && typeof value !== 'string') return false;
   where[column] = value;
  }
  try {
   const result = await this.prisma[entity].findMany({
    select: { id: true },
    where,
   });
   if (shouldExist) {
    if (Array.isArray(value)) {
     return result.length === [...new Set(value)].length;
    }
```

```
        return !!result.length;
      } else {
        if (Array.isArray(value)) {
          return result.length === 0;
        }
        return !result.length;
      }
    } catch (error) {
      throw error;
    }
  }
  defaultMessage(args: ValidationArguments) {
    if (args.constraints[2]) {
      return `This ${args.property} does not exists!`;
    }
    return `This ${args.property} already exists!`;
  }
}

type PrismaEntity = Exclude<
  keyof PrismaService,
  `$${string}` | 'onModuleInit' | 'enableShutdownHooks'
>;
type PrismaEntityKey<T extends PrismaEntity> = Exclude<
  keyof Parameters<PrismaService[T]['create']>[0]['select'],
  '_count'
>;

type EntityExistsOverload = {
```

```
  (entity: PrismaEntity): PropertyDecorator;
  (
    entity: PrismaEntity,
    validationOptions: ValidationOptions,
  ): PropertyDecorator;
  <TEntity extends PrismaEntity>(
    entity: TEntity,
    column: PrismaEntityKey<TEntity>,
  ): PropertyDecorator;
  <TEntity extends PrismaEntity>(
    entity: TEntity,
    column: PrismaEntityKey<TEntity>,
    validationOptions: ValidationOptions,
  ): PropertyDecorator;
  <TEntity extends PrismaEntity>(
    entity: TEntity,
    column: PrismaEntityKey<TEntity>,
    shouldExist: boolean,
  ): PropertyDecorator;
  <TEntity extends PrismaEntity>(
    entity: TEntity,
    column: PrismaEntityKey<TEntity>,
    shouldExist: boolean,
    validationOptions: ValidationOptions,
  ): PropertyDecorator;
};
export const EntityExists: EntityExistsOverload = (
  entity: PrismaEntity,
  second?: string | ValidationOptions,
```

```
  third?: boolean | ValidationOptions,
  forth?: ValidationOptions,
) => {
  const column = typeof second === 'string' ? second : 'id';
  const shouldExist = typeof third === 'boolean' ? third : true;
  const validationOptions =
    typeof second !== 'string'
      ? second
      : typeof third !== 'boolean'
      ? third
      : forth;
  return function (object: any, propertyName: string) {
    registerDecorator({
      name: 'EntityExists',
      target: object.constructor,
      propertyName: propertyName,
      constraints: [entity, column, shouldExist],
      options: validationOptions,
      validator: EntityExistsRule,
    });
  };
};
//end page
all-exceptions.filter.ts
//start page
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
```

```
  HttpException,
  HttpStatus,
  UnprocessableEntityException,
} from '@nestjs/common';
import { HttpAdapterHost } from '@nestjs/core';

import { AppLogger } from './app.logger';
@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(
    private readonly httpAdapterHost: HttpAdapterHost,
    private readonly logger: AppLogger,
  ) {}

  formatErrors(errors) {
    const object = {};
    for (const err of errors) {
      object[err.property] = {};
      if (err.children && err.children.length) {
        object[err.property].children = this.formatErrors(err.children);
      }
      if (err.constraints) {
        if (typeof err.constraints === 'object') {
          object[err.property].constraints = Object.values(err.constraints);
        } else {
          object[err.property].constraints = err.constraints;
        }
      }
    }
```

```
    return object;
  }

  catch(exception: unknown, host: ArgumentsHost): void {
    console.log(exception);
    this.logger.error(JSON.stringify(exception));
    const { httpAdapter } = this.httpAdapterHost;
    const ctx = host.switchToHttp();
    let httpStatus: number;
    let responseBody: any = {};
    if (exception instanceof UnprocessableEntityException) {
      httpStatus = exception.getStatus();
      responseBody = exception.getResponse();
      const errors = responseBody.errors || [];
      responseBody.errors = this.formatErrors(errors);
      responseBody.error = 'Unprocessable Entity';
      responseBody.message = exception.message || 'Invalid data';
    } else if (exception instanceof HttpException) {
      httpStatus = exception.getStatus();
      responseBody.message = exception.message;
    } else {
      httpStatus = HttpStatus.INTERNAL_SERVER_ERROR;
    }
    responseBody.status = httpStatus;
    responseBody.timestamp = new Date().toISOString();
    responseBody.path = httpAdapter.getRequestUrl(ctx.getRequest());
    httpAdapter.reply(ctx.getResponse(), responseBody, httpStatus);
  }
}
```

```
//end page
create-message.dto.ts
//start page
import { ApiPropertyOptional } from '@nestjs/swagger';
import { IsOptional, IsString, MinLength } from 'class-validator';

export class CreateMessageDto {
  @MinLength(1)
  @IsString()
  @IsOptional()
  @ApiPropertyOptional({ example: 'Hello' })
  text?: string;
  @IsOptional()
  @ApiPropertyOptional({ type: 'string', format: 'binary' })
  files?;
}
//end page
chat.gateway.ts
//start page
import * as jwt from 'jsonwebtoken';
import {
  OnGatewayConnection,
  OnGatewayDisconnect,
  OnGatewayInit,
  WebSocketGateway,
  WebSocketServer,
} from '@nestjs/websockets';
import { Server, Socket } from 'socket.io';
import { IPayload } from '../auth/interfaces/payload.interface';
```

```
@WebSocketGateway({
  namespace: 'messages',
  cors: {
    origin: 'http://localhost:8080',
    methods: ['GET', 'POST'],
  },
})
export class ChatGateway
        implements OnGatewayInit, OnGatewayConnection,
OnGatewayDisconnect
{
  @WebSocketServer()
  server: Omit<Server, 'adapter'> & {
    adapter: {
      rooms: Map<string, Set<string>>;
      sids: Map<string, Set<string>>;
    };
  };

  //send message from sender to other user room
  sendMessage(id: string, data) {
    this.server.to(id).emit('msgToClient', data);
  }

  //server starter logic
  afterInit(server: Server) {
    console.log('Server with sockets started');
  }
```

```
//socket disconnect
handleDisconnect(client: Socket) {
  console.log(`Client disconnected: ${client.id}`);
}

//socket connect
async handleConnection(client: Socket, ...args: any[]) {
  let user: IPayload;
  try {
    user = jwt.verify(
      client.handshake.headers?.authorization.split(' ')[1],
      process.env.JWT_SECRET_KEY,
    ) as IPayload;
    if (!user || typeof user?.sub !== 'string') {
      console.log('Invalid token');
      client.emit('error', { message: 'Invalid token' });
      client.disconnect();
    }
  } catch (error) {
    console.log(error);
    client.emit('error', { message: 'Invalid token' });
    client.disconnect();
    return;
  }
  await client.join(user.sub);
  //fields for info
  const room_id = [...client.rooms][1];
  console.log(`Client ${user.sub} connected to room: ${room_id}`);
```

```
  }
}
//end page
```

messages.controller.ts

```
//start page
import {
  Controller,
  Post,
  Body,
  Param,
  UseInterceptors,
  Req,
  UploadedFiles,
  UseGuards,
  Get,
  Query,
  ParseFilePipe,
  MaxFileSizeValidator,
} from '@nestjs/common';
import { MessagesService } from './messages.service';
import {
  ApiBearerAuth,
  ApiBody,
  ApiConsumes,
  ApiCreatedResponse,
  ApiOkResponse,
  ApiOperation,
  ApiTags,
} from '@nestjs/swagger';
```

```
import { SuccessResponseDocs } from '../helpers/docs/successShema';
import { FilesInterceptor } from '@nestjs/platform-express';
import { diskStorage } from 'multer';
import { extname } from 'path';
import { ChatIdDto } from './dto/chat-id.dto';
import { IRequest } from '../auth/interfaces/request.interface';
import { CreateMessageDto } from './dto/create-message.dto';
import { JwtAuthGuard } from '../auth/guards/jwt-auth.guard';
import { GetMessagesEntity } from './entities/get-messages.entities';
import { PaginationQueryDto } from '../helpers/dto/pagination-query.dto';
import { randomBytes } from 'crypto';

@ApiTags('Messages')
@Controller('messages')
export class MessagesController {
  constructor(private readonly messagesService: MessagesService) {}

  // #region Swagger Decorators
  @ApiOperation({ summary: 'Create message' })
  @ApiBody({ type: CreateMessageDto })
  @ApiCreatedResponse(SuccessResponseDocs)
  @ApiConsumes('multipart/form-data')
  @ApiBearerAuth()
  // #endregion
  @UseGuards(JwtAuthGuard)
  @UseInterceptors(
    FilesInterceptor('files', undefined, {
      storage: diskStorage({
        destination: './storage/temp',
```

```
      filename: (req, file, cb) => {
        cb(
          null,
          `${randomBytes(30).toString('hex')}${extname(file.originalname)}`,
        );
      },
    }),
  }),
)
@Post(':chat_id')
createMessage(
  @Param() { chat_id }: ChatIdDto,
  @Req() { user }: IRequest,
  @Body() createMessageDto: CreateMessageDto,
  @UploadedFiles(
    new ParseFilePipe({
      fileIsRequired: false,
      validators: [new MaxFileSizeValidator({ maxSize: 10000000 })],
    }),
  )
  files: Array<Express.Multer.File>,
) {
  return this.messagesService.sendMessage(
    user.id,
    chat_id,
    createMessageDto,
    files,
  );
}
```

```
  // #region Swagger Decorators
  @ApiOperation({ summary: 'Get messages from chat' })
  @ApiOkResponse({ type: GetMessagesEntity })
  @ApiBearerAuth()
  // #endregion
  @UseGuards(JwtAuthGuard)
  @Get(':chat_id')
  getMessages(
    @Param() { chat_id }: ChatIdDto,
    @Req() { user }: IRequest,
    @Query() dto: PaginationQueryDto,
  ): Promise<GetMessagesEntity> {
    return this.messagesService.getMessages(user.id, chat_id, dto);
  }
}
//end page
```

messages.service.ts

```
//start page
import {
  HttpException,
  Injectable,
  UnprocessableEntityException,
} from '@nestjs/common';
import { PrismaService } from '../prisma/prisma.service';
import { CreateMessageDto } from './dto/create-message.dto';
import { FileHelper } from '../helpers/file-system/file-helper';
import { Prisma } from '@prisma/client';
import { PaginationQueryDto } from '../helpers/dto/pagination-query.dto';
```

```
import { ChatGateway } from 'src/messages/chat.gateway';

@Injectable()
export class MessagesService {
 constructor(
  private readonly prisma: PrismaService,
  private readonly fileHelper: FileHelper,
  private readonly chatGateway: ChatGateway,
 ) {}

 async saveFiles(files: Array<{ path: string }>, chat_id: string) {
  try {
   return await Promise.all(
    files.map(
     async (file) => await this.fileHelper.moveTemp(file.path, chat_id),
    ),
   );
  } catch {
   throw new HttpException('Files can`t be saved', 500);
  }
 }

 async updateChatUpdatedAt(chat_id: string) {
  await this.prisma.chat.update({
   where: { id: chat_id },
   data: { updated_at: new Date() },
  });
 }
```

```
async sendMessage(
  user_id: string,
  chat_id: string,
  dto: CreateMessageDto,
  files: Array<Express.Multer.File>,
) {
  if (!files?.length && !dto.text) {
    throw new UnprocessableEntityException('Message is empty');
  }
  const filesArray: string[] = [];
  if (files?.length) {
    filesArray.push(...(await this.saveFiles(files, chat_id)));
  }
  const userChats = await this.prisma.userChat.findMany({
    where: { chat_id, user_id: { not: user_id } },
    select: { user_id: true },
  });
  const data: Prisma.MessageUncheckedCreateInput = {
    user_id,
    chat_id,
    text: dto.text,
    files: filesArray,
    user_messages: {
      createMany: {
        data: userChats.map(({ user_id }) => ({
          chat_id,
          user_id,
        })),
      },
```

```
  },
};
const messageSended = await this.prisma.message.create({
  data,
  select: {
    user_messages: { select: { user_id: true } },
    chat_id: true,
    created_at: true,
    files: true,
    text: true,
  },
});
if (messageSended) {
  const { user_messages, ...result } = messageSended;
  await this.updateChatUpdatedAt(chat_id);
  this.chatGateway.sendMessage(
    messageSended.user_messages[0].user_id,
    result,
  );
} else {
  if (filesArray?.length) {
    Promise.all(
      filesArray.map(async (file) => await this.fileHelper.remove(file)),
    );
  }
}
return {
  status: 'success',
};
```

```
}

async getMessages(
  user_id: string,
  chat_id: string,
  { skip = 0, take = 100 }: PaginationQueryDto,
) {
  //create orderBy
  const orderBy: Prisma.MessageOrderByWithRelationInput = {
    created_at: 'desc',
  };
  //create where
  const where: Prisma.MessageWhereInput = {
    chat_id,
  };
  //create select
  const select = {
    id: true,
    text: true,
    files: true,
    created_at: true,
    user_id: true,
    user_messages: {
      select: { read: true, user_id: true },
      where: { user_id: { not: user_id } },
    },
  } satisfies Prisma.MessageSelect;
  //get messages
  const messages = await this.prisma.message.findMany({
```

```
  where,
  orderBy,
  select,
  skip,
  take,
});
const result = {
  user_id,
  messages: messages.map(
    ({ user_messages, user_id: user_id_m, files, ...message }) => ({
      user_id: user_id_m,
      ...message,
      files: files.map((file) => this.fileHelper.getFullPath(file)),
      sent:
        user_id === user_id_m && !user_messages[0].read ? true : undefined,
      delivered:
        user_id === user_id_m && user_messages[0].read ? true : undefined,
    }),
  ),
};
console.log(result.messages[0]);
await this.prisma.userMessage.updateMany({
  where: {
    message_id: { in: messages.map(({ id }) => id) },
    user_id,
  },
  data: { read: true },
});
return result;
```

```
  }
}
//end page
```

З повним кодом, можливо ознайомитись за посиланням: https://gitlab.com/dimitriy1/bacalavr

Додаток А.2: Код клієнтської частини

З кодом можливо ознайомитись за посиланням: https://gitlab.com/SamiraRadzhabova/bacalavr_front