Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра програмного забезпечення

Бакалаврська дипломна робота

на тему: «Веб-додаток для онлайн-консультацій пацієнтів з лікарями»

Виконав: студент 4 курсу групи 1ПІ-18б
спеціальності
121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Заболотний Н.С.
(прізвище та ініціали)

Керівник: д.т.н., проф. каф. ПЗ Ліщинська Л.Б.
(прізвище та ініціали)

Рецензент: к.т.н., доц. каф. ЛОТ Кожем'яко А.В.
(прізвище та ініціали)

Допущено до захисту
Зав. кафедри _____Романюк О.Н.
«____» _____ 2022 р.

ВНТУ – 2022

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти перший бакалаврський
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
_____Романюк О. Н.
25 березня 2022 р.

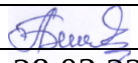
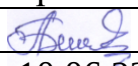
З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Заболотному Назарію Станіславовичу

1. Тема роботи – «Веб-додаток для онлайн-консультацій пацієнтів з лікарями»
Керівник роботи: Ліщинська Людмила Броніславівна, д-р т.н., проф. каф. ПЗ, затверджені наказом вищого навчального закладу від 24 березня 2022 р. № 66
2. Строк подання студентом роботи 15 червня 2022 р.

3. Вихідні дані до роботи: середовище розробки WebStorm, мова розробки JavaScript (React.js для графічного інтерфейсу та Node.js для серверної частини).
4. Зміст розрахунково-пояснювальної записки: вступ; аналіз стану проблеми; порівняльний аналіз аналогів; постановка задачі; розробка алгоритму роботи модуля відеозв'язку; розробка алгоритму роботи модуля текстових повідомлень; розробка структури інтерфейсу програмного додатку; тестування веб-додатку; розробка інструкції користувача; висновки; перелік посилань; додатки; графічна частина.
5. Перелік графічного матеріалу: зображення аналогів; блок-схеми алгоритмів роботи додатку; графічний інтерфейс додатку; тестування додатку.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Ліщинська Л.Б., д-р т.н., проф. каф. ПЗ		
		28.03.22	10.06.22

7. Дата видачі завдання 25 березня 2022 р

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської дипломної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз задачі, обґрунтування актуальності розробки та постановка задачі	26.03.2022-16.04.2022	Вик.
2	Розробка алгоритмів роботи веб-додатку	17.04.2022 – 01.05.2022	Вик.
3	Розробка коду веб-додатку для онлайн-консультацій пацієнтів з лікарями.	02.05.2022 – 25.05.2022	Вик.
4	Тестування роботи додатку	26.05.2022 – 02.06.2022	Вик.
5	Оформлення матеріалів до захисту БДР	03.06.2022 – 10.06.2022	Вик.

Студент _____ **Заболотний Н.С.**

(підпис) (прізвище та ініціали)

Керівник бакалаврської дипломної роботи _____ **Ліщинська Л.Б.**

(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Бакалаврська дипломна робота складається з 53 сторінок формату А4, містить 44 рисунки, 6 таблиць, список використаних джерел містить 19 найменувань.

У бакалаврській дипломній роботі розроблено веб-додаток, який дозволяє лікарям проводити онлайн-консультації зі своїми пацієнтами за допомогою відеозв’язку та текстових повідомлень.

Розроблено модуль відеозв’язку з використанням мови програмування JavaScript, бібліотеки WebRTC та середовища розробки WebStorm.

Розроблено модуль текстових повідомлень з використанням мови програмування JavaScript, бібліотеки Socket.io та середовища розробки WebStorm.

Розроблено програмний додаток, що використовує модулі відеозв’язку та текстових повідомлень. При розробці використано фреймворк React, платформу Node.js та середовища розробки WebStorm.

Отримані в бакалаврській дипломній роботі результати можна використати для вдосконалення існуючих веб-додатків, що використовують технологію WebRTC.

Ключові слова: веб-додаток, телемедицина, JavaScript, React, NestJS, WebRTC, Socket.io, MUI.

ABSTRACT

The bachelor's thesis consists of 53 A4 pages, contains 44 figures, 6 tables, the list of used sources contains 19 titles.

The bachelor's thesis has developed a web application that allows doctors to consult online with their patients via video and text messaging.

A video communication module has been developed using the JavaScript programming language, the WebRTC library and the WebStorm development environment.

A text messaging module has been developed using the JavaScript programming language, the Socket.io library, and the WebStorm development environment.

Developed a software application that uses video and text messaging modules.

The React framework, the Node.js platform and the WebStorm development environment were used in the development.

The results obtained in the bachelor's thesis can be used to improve existing web applications that use WebRTC technology.

Keywords: web application, telemedicine, JavaScript, React, NestJS, WebRTC, Socket.io, MUI.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. Телемедицина – це вид сучасної медицини, при якому медичний фахівець має можливість дистанційно вирішувати проблеми, що хвилюють пацієнта та обирають онлайн медичні рекомендації з діагностики, що використовують спостереження за гаджетами з аудіо або відеозв'язком [1].

Сьогодні у всьому світі можливі види телемедичних послуг:

- телемедичне консультування;
- телемоніторинг;
- телескринінг;
- телеприсутність;
- телеасистування;

Найбільш поширеним є телемедичне консультування, або простіше кажучи – онлайн консультації на відстані, що є важливою частиною повсякденної лікувально-діагностичної роботи лікарів, тобто відбувається процес відео чи аудіо консультації з пацієнтом, що забезпечує наближення кваліфікованої допомоги, дозволяє швидко приймати та передавати клінічні рішення, підвищує якість та доступність медичної допомоги. Такий віддалений відео прийом лікаря допоможе заощадити не тільки час, але й гроші: вартість прийому онлайн дешевша, ніж індивідуальна консультація наживо. Все просто – використання онлайн зв'язку. Якщо у лікарів виникнуть додаткові питання, вони порадяться зі своїми колегами за допомогою аудіозв'язку. Сімейний лікар може надіслати лікареві вузької спеціалізації результати аналізів з використанням Інтернету, а ті, у свою чергу, можуть встановити діагноз. Медики вже проводять відео консультації один з одним через різноманітні мессенджери, відправляючи фотографії, якщо, наприклад, йдеться про дерматит, псоріаз, екзему, пролежнів тощо.

Значні досягнення в області технологій різко змінили те, як і де ведеться бізнес, коли більше людей працюють віддалено, розвиток миттєвого зв’язку

відповідає вимогам. Для розвитку та глобалізації сучасного та інноваційного бізнесу необхідно будувати та підтримувати якісні відносини з партнерами, постачальниками, внутрішніми командами, інвесторами та клієнтами. Відеоконференції підвищують продуктивність, заощаджують час, зменшують витрати на відрядження та загалом сприяють співпраці. Перевагою відеозв'язку є можливість сприяти всім цим перевагам, не вимагаючи постійних подорожей для спілкування віч-на-віч.

Тема дослідження набуває особливої актуальності у зв'язку з пандемією Covid-19 у світі. Так як розроблений модуль може використовуватись окремо, його можна використати у додатку будь-якого призначення. У випадку з телемедициною, модуль надає можливість дистанційного спілкування, що унеможливлює розповсюдження певних хвороб.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалась згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою дослідження є підвищення ефективності взаємодії і віддаленого консультування лікаря з пацієнтами.

Основними задачами роботи є:

- обґрунтування вибору методу розробки і постановка задачі;
- розробка методу, структури та алгоритмів роботи веб-додатку для онлайн консультацій;
- розробка основних модулів додатку;
- проведення тестування додатку.

Об'єкт дослідження – процеси взаємодії і віддаленого консультування лікаря з пацієнтами.

Предмет дослідження – методи і засоби взаємодії і віддаленого консультування лікаря з пацієнтами.

Методи дослідження. У процесі досліджень використовувались методи дослідження: методи веб-комунікацій; теорія нереляційних баз даних; методи однорангових з’єдання; методи тестування та валідації програмних продуктів.

Наукова новизна отриманих результатів. Подальшого розвитку отримала технологія однорангового з’єднання користувачів без посередників, яка на відміну від існуючих, використовує технологію WebRTC та дозволяє оптимізувати швидкість і захищеність з’єдання за допомогою використання react-хуків.

Практична цінність отриманих результатів. Практична цінність одержаних результатів полягає в тому, що на основі отриманих у бакалаврській дипломній роботі теоретичних положень запропоновано алгоритми та розроблено веб-додаток з інтегрованими модулями відеозв’язку і текстових повідомлень, що може використовуватись лікарями для проведення онлайн-консультацій.

Особистий внесок здобувача. Автор особисто отримав усі наукові результати, викладені у бакалаврській дипломній роботі. У науковій роботі, опублікованій у співавторстві [2], автору належать такі результати: розробка модуля для відеозв’язку; розробка модуля для текстових повідомлень у реальному часі.

Апробація матеріалів бакалаврської дипломної роботи. Основні положення бакалаврської дипломної роботи доповідалися та обговорювалися на науково-технічній конференції підрозділів Вінницького національного технічного університету (Вінниця, ВНТУ, 2022р.).

Публікації. Основні результати дослідженні опубліковані у тезах доповіді на науково-технічній конференції підрозділів Вінницького національного те

1 ОБҐРУНТУВАННЯ ВИБОРУ МЕТОДУ РОЗРОБКИ ТА ПОСТАНОВКА ЗАДАЧ ДОСЛІДЖЕННЯ

1.1 Аналіз стану сучасної телемедицини

Те, що сьогодні називається телемедициною, почалося в 1950-х роках, коли кілька лікарняних систем та університетських медичних центрів почали намагатися знайти способи обміну інформацією та зображеннями по телефону. В один із перших успіхів два медичні центри в Пенсільванії змогли передати радіологічні зображення по телефону.

Спочатку телемедицина використовувалася здебільшого для з'єднання лікарів, які працюють з пацієнтом в одному місці, зі спеціалістами в іншому місці. Це було дуже корисно для сільського або важкодоступного населення, де спеціалісти недоступні. Протягом наступних кількох десятиліть обладнання, необхідне для проведення дистанційних відвідувань, залишалося дорогим і складним, тому використання підходу, незважаючи на зростання, було обмеженим.

Піднесення епохи Інтернету принесло з собою глибокі зміни в практику телемедицини. Поширення розумних пристроїв, здатних передавати відео високої якості, відкрило можливість надання дистанційної медичної допомоги пацієнтам у їхніх домівках, на робочих місцях або в допоміжних установах як альтернативу особистим відвідуванням як первинної, так і спеціалізованої допомоги.

Хоча терміни телездоров'я та телемедицина часто використовуються як синоніми, між ними є відмінність. Термін телездоров'я включає широкий спектр технологій та послуг для надання допомоги пацієнтам та покращення системи надання медичної допомоги в цілому. Телездоров'я відрізняється від телемедицини, оскільки воно відноситься до більш широкого кола дистанційних медичних послуг, ніж телемедицина. У той час як телемедицина відноситься саме до віддалених клінічних послуг та може стосуватися віддалених доклінічних послуг, таких як навчання постачальників послуг, адміністративні зустрічі та безперервна медична освіта, на додаток до клінічних послуг. За даними Всесвітньої

організації охорони здоров’я, телездоров’я включає в себе «функції спостереження, зміцнення здоров’я та охорони здоров’я».

Телемедицина передбачає використання електронних комунікацій та програмного забезпечення для надання клінічних послуг пацієнтам без особистого відвідування. Технологія телемедицини часто використовується для подальших відвідувань, лікування хронічних захворювань, лікування ліків, консультацій спеціалістів та низки інших клінічних послуг, які можна надавати дистанційно через безпечні відео- та аудіо-з’єднання [3].

1.2 Порівняльний аналіз аналогів

Першим із запропонованих аналогів є додаток «Teladoc» (див. рисунок 1.1).

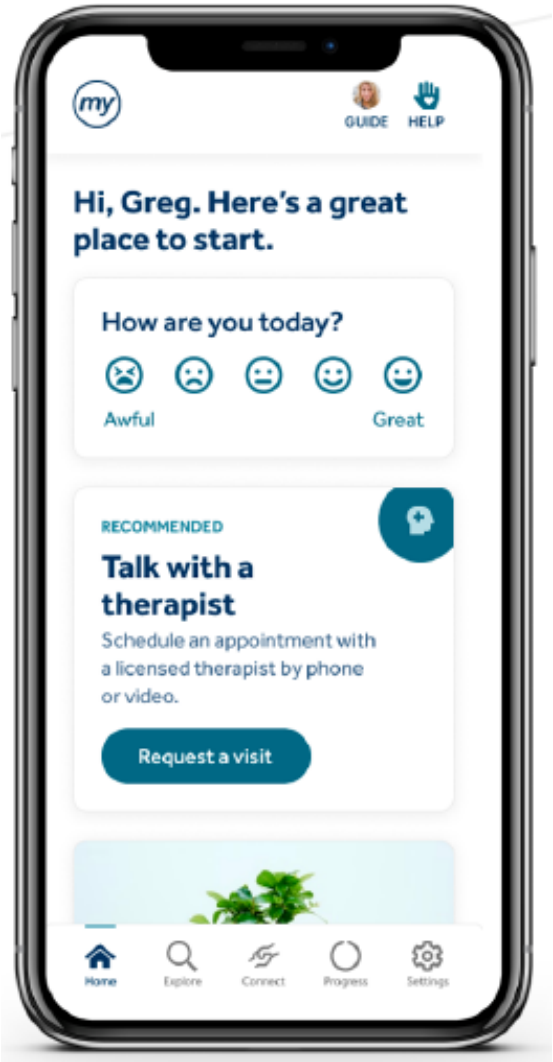


Рисунок 1.1 – Інтерфейс програми «Teladoc»

Додаток потрібно встановлювати на мобільний пристрій з операційною системою Android або IOS, що унеможливлює її використання на пристроях під керуванням операційних систем Windows, Linux та MacOS. Також відсутність локалізації ускладнює використання додатку без знання англійської мови.

Наступним агалогом є додаток «LikarOnline». На відміну від попереднього, додаток має українську локалізацію та можливість його використання за допомогою пристроїв з будь-якою операційною системою. Проте, інтерфейс додатку є застарілим та не досить зручним. Також програма не має можливості текстового спілкування.

Інтерфейс додатку «LikarOnline» представлено на рисунку 1.2.

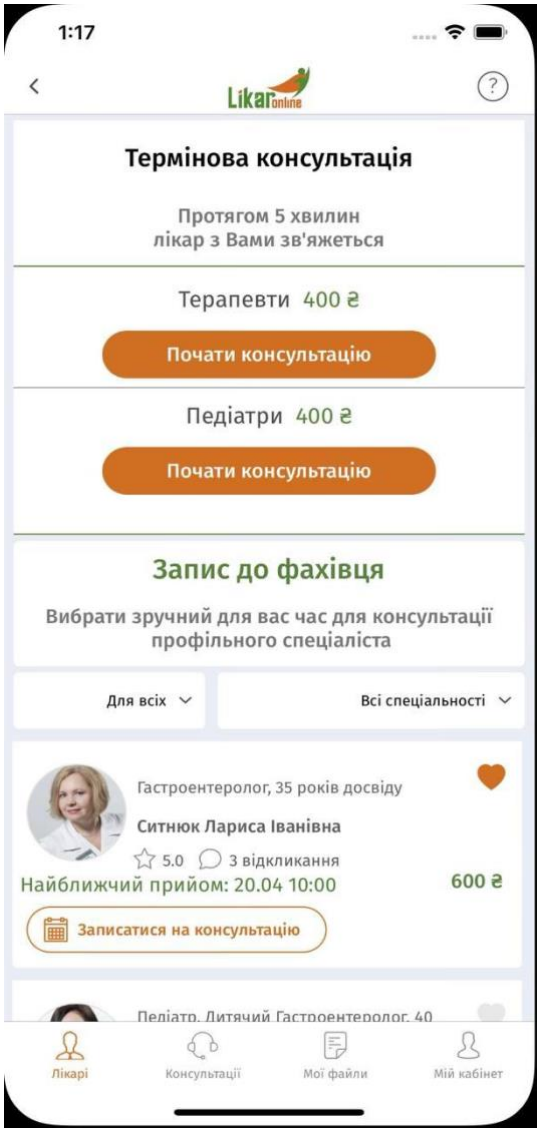


Рисунок 1.2 – Інтерфейс програми «LikarOnline»

Досить цікавим аналогом є інтернет-портал «Telemed24», що знаходиться за адресою https://telemed24.ua. Цей комплекс програм є набагато складнішим, ніж вищеописані, тому що має можливість взаємодії з іншими послугами телемедицини, такими як: DoctorOnline, Medikit та Medinet.

Інтерфейс інтернет-порталу «Telemed24» зображено на рисунку 1.3.

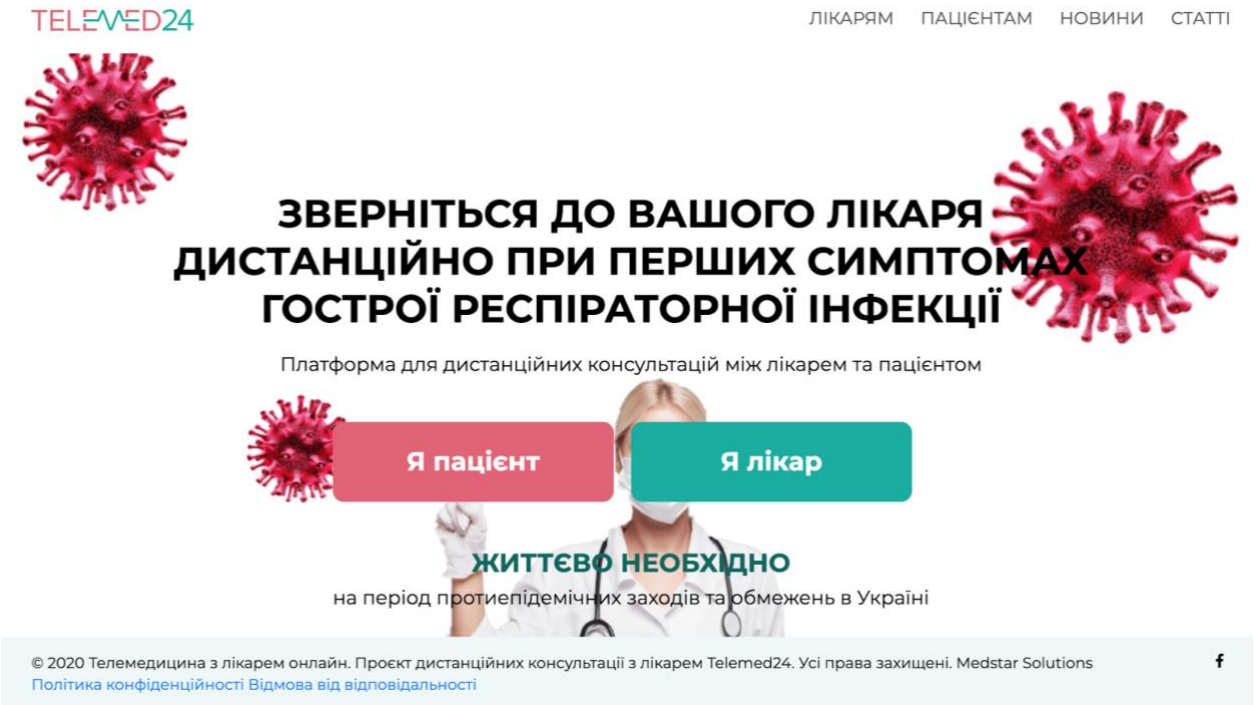


Рисунок 1.3 – Інтерфейс програми «Telemed24»

Останнім з вибраних аналогів є веб-додаток «PlushCare». PlushCare – телемедична компанія, яка розширилась, щоб запропонувати послуги з психічного здоров'я. Пропонує послуги розмовної терапії з ліцензованими терапевтами, а також послуги з лікування за допомогою ліків. Лікарі PlushCare забезпечують лікування медикаментами під керівництвом психіатрів, які працюють у компанії.

До переваг можна віднести такі аспекти: сучасний веб-сайт, можливість вибору лікаря, можливість щотижневі консультації входять у підписку, сесії прийому налаштовані на 45 хвилин.

Недоліками є: відсутність конференцій, неможливо повернути гроші за невикористані консультації, відсутня можливість текстового спілкування.

Інтерфейс веб-додатку «PlushCare» зображено на рисунку 1.4.

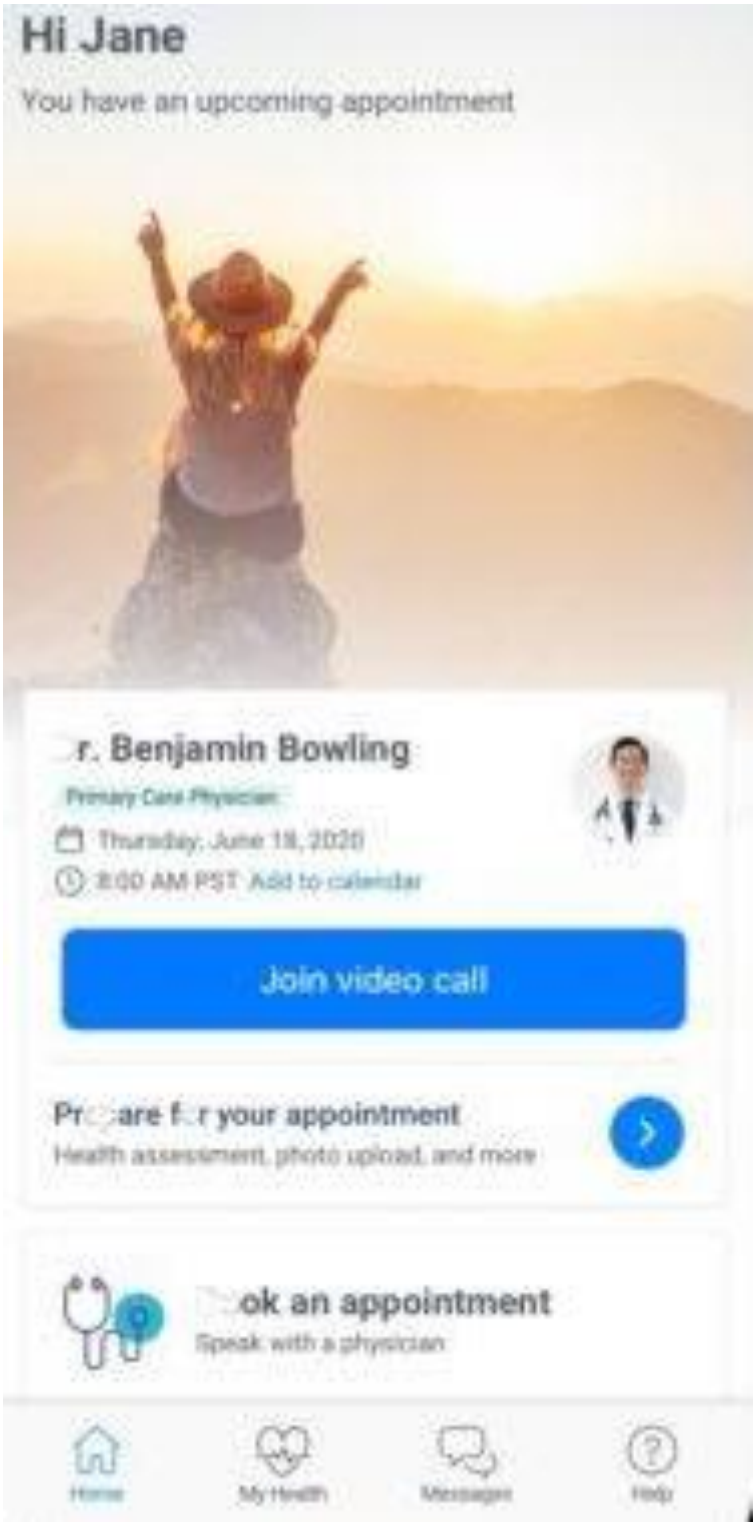


Рисунок 1.4 – Інтерфейс додатку «PlushCare»

Для порівняння аналогів та визначення актуальності розробки власного додатку створено таблицю 1.1.

Таблиця 1.1 – Порівняння аналогів з додатком «ЗДОРОВ’Я»

Параметр\Назва додатку	Teladoc	LikarOnline	Telemed24	ЗДОРОВ’Я	PlushCare
Локалізація	-	+	+	+	-
Багатоплатформенність	-	+	+	+	+
Можливість створити конференцію	-	-	-	+	-
Можливість текстового спілкування	+	-	+	+	-
Наявність відеозв’язку	+	+	+	+	+
Сумарний коефіцієнт	2	3	4	5	2

Проаналізувавши вищенаведені дані, розроблюваний додаток «ЗДОРОВ’Я» має вищий сумарний коефіцієнт відносно аналогів: Teladoc на 60%, LikarOnline на 40%, Telemed24 на 20% та PlushCare на 60%.

Отже, проаналізувавши усі описані програмні додатки, виявлено, що розробка власного додатку є актуальним питанням.

1.3 Постановка задачі розробки

Після аналізу стану додатків для телемедицини та порівняння існуючих аналогів було визначено завдання, які необхідно виконати для розробки програмного продукту:

- створити українську локалізацію;
- розробити схеми інтерфейсу;
- проаналізувати моделі комунікацій;

- створити модель бази даних;
- розробити модуль для текстових повідомлень у реальному часі;
- розробити модуль для відеозв’язку;
- розробити захист з’єднання;
- пароль користувача повинен зберігатись зашифрованим;
- розробити можливість реєстрації/авторизації користувача;
- додати можливість проведення конференцій;
- додати можливість закріпити пацієнта за певним лікарем;
- провести тестування програмного продукту;
- додаток повинен мати файл конфігурації середовища;
- додаток повинен обробляти помилки та сповіщати користувача у разі їх виникнення;
- інтерфейс повинен бути сучасним;
- час затримки передачі відеозображення не повинен перевищувати 3 секунди;
- веб-додаток повинен бути масштабованим.

1.4 Висновки

У першому розділі розглянуто стан питання сучасної телемедицини.

Проаналізовано існуючі аналоги та проведено їх порівняння з розроблюваним додатком. У результаті порівняння доведено доцільність розробки власного програмного додатку.

Сформовано основні задачі, які необхідно виконати для розробки програмного додатку.

2 РОЗРОБКА МЕТОДУ, СТРУКТУРИ ТА АЛГОРИТМІВ ВЕБ-ДОДАТКУ ДЛЯ ОНЛАЙН-КОНСУЛЬТАЦІЙ ПАЦІЄНТІВ З ЛІКАРЯМИ

2.1 Аналіз даних та варіативні моделі комунікацій

Будь-який додаток базується на роботі з даними. Обмін даними між окремими модулями відбувається постійно. Для зручної та стабільної роботи таких додатків використовуються бази даних.

База даних – це організована сукупність структурованої інформації або даних, які зазвичай зберігаються в електронній формі в комп’ютерній системі. База даних зазвичай контролюється системою управління базами даних (СУБД). Разом дані, СУБД та додатки, які з ними пов’язані, називають системою баз даних, часто скороченою до просто бази даних [4].

Дані в найпоширеніших типах баз даних, які діють сьогодні, зазвичай моделюються в рядках і стовпцях у серії таблиць, щоб зробити обробку та запити даних ефективними. Дані можна легко отримувати, керувати, змінювати, оновлювати, контролювати та організовувати. Більшість баз даних використовує мову структурованих запитів (SQL) для запису та запиту даних.

У розробленому програмному додатку моделей зв’язку є встановлення безпосереднього відеозв’язку між користувачами та обмін текстовими повідомленнями у режимі реального часу.

Модель бази даних містить такі сутності:

- користувач;
- консультація;
- підключення;
- повідомлення.

Сутність користувача містить такі поля: електронна адреса, роль, ім’я, прізвище та дата народження. Ці дані збираються за допомогою форм у графічному інтерфейсі користувача.

Кожна сутність консультації містить дані про пацієнта, лікаря, тривалість з’єдання та заключення, що виносить лікар після опитування. Всі дані, крім заключення, заповнюються автоматично після завершення консультаційного виклику.

Сутність підключення містить ідентифікаційний номер підключення та ідентифікаційний номер користувача. Використовується для ідентифікації користувачів у мережі підключення.

Остання сутність – повідомлення. Вона зберігає інформацію про отримувача, інформацію про відправника, дату та час відправки та текст повідомлення.

Діаграму бази даних зображено на рисунку 2.1.

Рисунок 2.1 – Діаграма бази даних

Подальшого розвитку отримав метод peer 2 peer з’єдання, що базується на технології WebRTC. Його модель зображено на рисунку 2.2.

Однорангова мережа (peer to peer) – це інфраструктура інформаційних технологій (IT), що дозволяє двом (або більше) комп’ютерним системам підключатися та спільно використовувати ресурси без необхідності окремого сервера або серверного програмного забезпечення. Налаштувати мережу P2P можливо шляхом фізичного з’єднання комп’ютерів у пов’язану систему або створення віртуальної мережі. Ви також можете налаштувати комп’ютери як клієнти та сервери їхньої мережі [5].

Мережа P2P відрізняється від мережі клієнт-сервер, яка традиційно використовується в мережах. Мережа клієнт-сервер – це з'єднання між клієнтським комп'ютером і комп'ютером-сервером для забезпечення клієнта ресурсами сервера.

У мережі P2P кожен пристрій вважається одноранговим – таким чином, «рівноправним» – з функціями, які сприяють роботі мережі. Кожен комп’ютер є одночасно клієнтом і сервером, і вони спільно використовують ресурси з іншими мережевими комп’ютерами.

Ключові переваги однорангової мережі:

- Легкий обмін файлами: розширена мережа P2P може швидко обмінюватися файлами на великі відстані. Доступ до файлів можна отримати в будь-який час;
- Зменшення витрат: не потрібно інвестувати в окремий комп’ютер для сервера під час налаштування мережі P2P. Для цього не потрібна мережева операційна система або штатний системний адміністратор;
- Адаптивність: мережа P2P легко розширюється, щоб додавати нових клієнтів. Ця перевага робить ці мережі більш гнучкими, ніж мережі клієнт-сервер;
- Надійність: на відміну від мережі клієнт-сервер, яка може вийти з ладу, якщо центральний сервер виходить з ладу, мережа P2P залишиться функціональною, навіть якщо центральний сервер вийде з ладу. Якщо

один комп’ютер виходить з ладу, інші продовжують працювати в звичайному режимі. Це також запобігає вузьким місцям, оскільки трафік розподіляється між кількома комп’ютерами;

- Висока продуктивність: у той час як мережа клієнт-сервер працює менш ефективно, коли до мережі приєднується більше клієнтів, мережа P2P може покращити свою продуктивність, коли до неї приєднується більше клієнтів. Це тому, що кожен клієнт у мережі P2P також є сервером, який надає ресурси мережі;
- Ефективність: нові мережі P2P дозволяють співпрацювати між пристроями, які мають різні ресурси, що може бути корисним для всієї мережі.

Рисунок 2.2 – Модель peer to peer з’єднання

WebRTC (Web Real-Time Communication) – це технологія, яка дозволяє веб-додаткам захоплювати та передавати аудіо та/або відео медіа, а також обмінюватися довільними даними між браузерами без потреби посередника [6]. Схему роботи технології зображено на рисунку 2.3. Набір стандартів WebRTC дає змогу обмінюватися даними та проводити телеконференції одноранговi, не вимагаючи від користувача встановлення плагінів або будь-якого іншого стороннього програмного забезпечення. WebRTC складається з кількох взаємопов’язаних API та протоколів, які працюють разом для досягнення цієї мети.

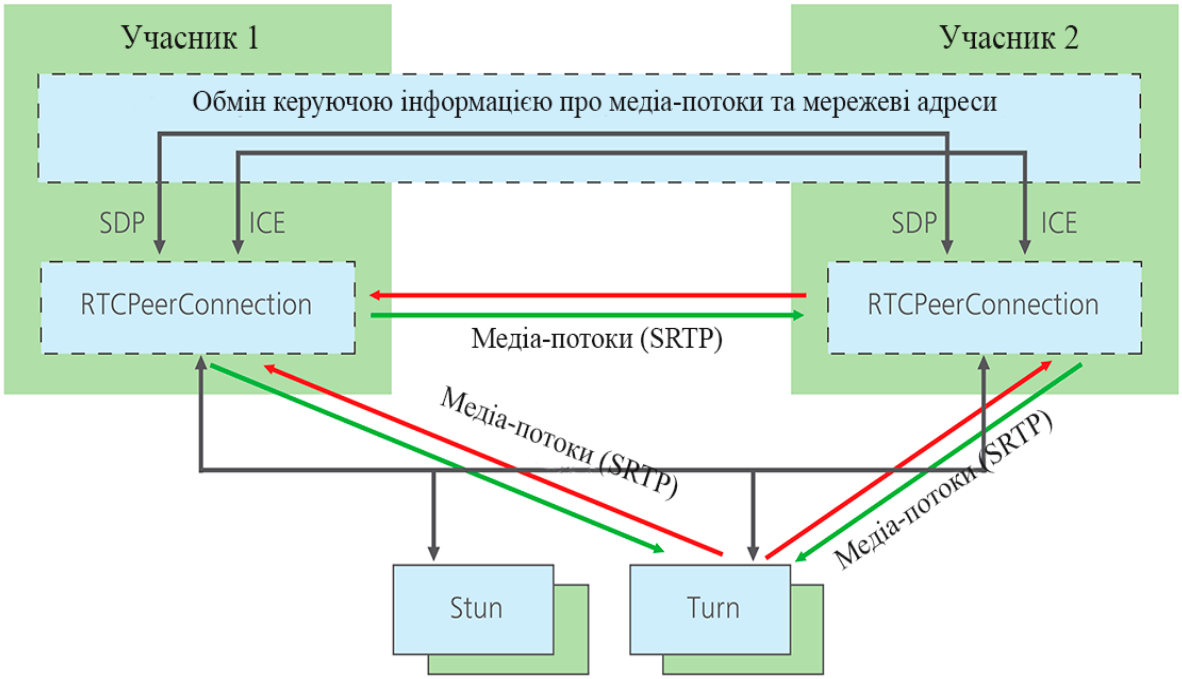


Рисунок 2.3 – Схема роботи технології WebRTC

У даному підрозділі створено діаграму бази даних, описано однорангову мережу, створено модель однорангового з'єдання та створено схему роботи технології WebRTC.

2.2 Розробка алгоритмів модуля відеозв'язку

Алгоритм – це набір інструкцій для вирішення проблеми або виконання завдання. Одним із поширених прикладів алгоритму є рецепт, який складається з конкретних інструкцій для приготування страви. Кожен комп'ютеризований пристрій використовує алгоритми для виконання своїх функцій у формі програмних або апаратних програм [7].

Однією з основних функцій додатку «ЗДОРОВ'Я» є можливість здійснювати аудіо- та відеодзвінки. Для реалізації цієї функції використано технологію WebRTC.

Для того, щоб розробити модуль відеозв'язку, потрібно розробити загальний алгоритм його роботи. Загальний алгоритм складається з таких кроків: встановлення з'єднання з сервером за допомогою технології WebSocket, перевірка

з’єднання, обмін адресами користувачів, встановлення безпосереднього зв’язку між браузерами користувачів, вивід зображення.

WebSocket – це постійне з'єднання між клієнтом і сервером. WebSockets забезпечують двонаправлений канал зв'язку, який працює через HTTP через єдине з'єднання TCP/IP [8]. По суті, протокол WebSocket полегшує передачу повідомлень між клієнтом і сервером. WebSockets, з іншого боку, дозволяють надсилати дані на основі повідомлень, подібно до UDP, але з надійністю TCP. WebSocket використовує HTTP як початковий транспортний механізм, але підтримує з'єднання TCP після отримання відповіді HTTP, щоб його можна було використовувати для надсилання повідомлень між клієнтом і сервером. WebSockets дозволяють нам створювати програми в режимі реального часу без використання тривалого опитування.

Блок-схему алгоритму загальної роботи модуля відеозв’язку зображено на рисунку 2.4.

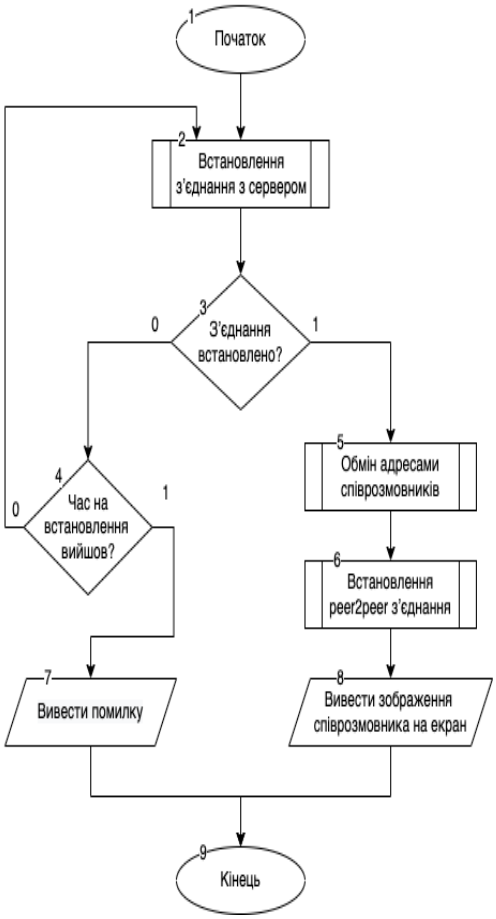


Рисунок 2.4 – Блок-схема алгоритму загальної роботи модуля відеозв’язку

Спочатку модуль відеозв’язку повинен встановити зв’язок з сервером за допомогою протоколу HTTP. Після чого, клієнтська частина додатку отримує адресу браузера співрозмовника. Отримавши всі необхідні дані, за допомогою протоколу WebSocket надсилається сповіщення про виклик. Сервер очікує відповідь користувача, та обробляє її. Якщо відповідь позитивна – встановлюється peer2peer з’єднання між браузерами, в результаті чого відеопотоки користувачів виводяться на екран.

2.3 Розробка алгоритмів модуля текстових повідомлень

Також важливою функцією додатку є модуль текстових повідомлень.

Для реалізації цієї функції розглянуто такі методи: REST API, Google Firebase cloud messaging та Socket.IO.

REST – це набір архітектурних обмежень, а не протокол чи стандарт. Розробники API можуть реалізувати REST різними способами.

API – це набір визначень і протоколів для створення та інтеграції прикладного програмного забезпечення. Інколи його називають контрактом між постачальником інформації та користувачем інформації, який встановлює вміст, який вимагається від споживача (виклик) і контент, який вимагається виробником (відповідь). Наприклад, у дизайні API для служби погоди можна було б вказати, що користувач надає поштовий індекс, а виробник відповідає відповіддю з двох частин, перша – висока температура, а друга – низька [9].

Firebase Cloud Messaging (FCM) – це багатоплатформне хмарне рішення для повідомлень та сповіщень для Android, iOS та веб-додатків. Firebase Cloud Messaging дозволяє розробникам додатків сторонніх розробників надсилати сповіщення або повідомлення з серверів, розміщених на FCM, користувачам платформи або кінцевим користувачам. На платформі користувачі можуть інтегрувати та комбінувати різні функції Firebase як у веб-додатках, так і в мобільних [10].

Socket.IO – бібліотека JavaScript, що базується на веб-сокетах. Вона

використовує веб-сокети, коли вони доступні, або такі технології, як Flash Socket, AJAX Long Polling, AJAX Multipart Stream, коли веб-сокети недоступні. На відміну від веб-сокетів, Socket.IO дозволяє відправляти повідомлення всім підключеним клієнтам. Наприклад, потрібно повідомити всіх учасників чату про підключення нового користувача. За допомогою бібліотеки Socket.IO легко можна це реалізувати за допомогою однієї операції. При використанні веб-сокетів для реалізації подібних завдань потрібно мати список підключених клієнтів та відправляти повідомлення по одному [11].

У веб-сокетах складно використовувати проксування і балансувальники нагрузки, Socket.IO підтримує ці технології з коробки. Також бібліотека підтримує автоматичне перепідключення при розриві з'єднання.

Для порівняння методів створення модуля текстових повідомлень та визначення технології, що найкраще підходить для його реалізації, створено таблицю 2.1.

Таблиця 2.1 – Порівняння методів реалізації модуля текстових повідомлень

Параметр\Назва методу	REST API	FCM	Socket.IO
Продуктивність	-	+	+
Можливість відправляти повідомлення декільком користувачам одночасно	-	+	+
Простота впровадження	+	-	+
Можливість отримувати сповіщення	-	+	+
Можливість передачі медіафайлів	+	-	+
Сумарний коефіцієнт	2	3	5

У результаті проведеного порівняння, обрано метод з використанням бібліотеки Socket.IO, так як він має вищий сумарний коефіцієнт відносно інших розглянутих методів: REST API на 60%, та FCM на 20%.

Перед розробкою, важливо створити загальний алгоритм роботи модуля. Адже тільки тоді буде повністю описано всі випадки, що пришвидшить розробку. Саме тому прийнято рішення створити та описати загальний алгоритм роботи модуля текстових повідомлень. Блок-схему алгоритму загальної роботи модуля текстових повідомлень зображено на рисунку 2.5.

Рисунок 2.5 – Блок-схема алгоритму загальної роботи модуля текстових повідомлень

Першим етапом роботи модуля є встановлення з’єднання за допомогою протоколу WebSocket. Потім виконується його перевірка. Якщо з’єдання успішно встановлено, користувачі потрапляють в окрему кімнату. Далі вмикається слухач змін стану повідомлень у кімнаті, що реалізований за шаблоном проектування «Спостерігач». При зміні стану повідомлень, усі користувачі, що знаходяться у кімнаті, отримують новий список повідомлень. Останнім етапом є перемалювання списку повідомлень у вікні чату.

Спостерігач – це поведінковий патерн проектування, який створює механізм підписки, що дозволяє одним об'єктам стежити та реагувати на події, що відбуваються в інших об'єктах [12]. Шаблон визначає постачальника (також відомого як суб’єкт або спостережуваний) і нуль, один або кілька спостерігачів. Спостерігачі реєструються у постачальника, і щоразу, коли відбувається зміна попередньо визначеної умови, події або стану, постачальник автоматично сповіщає всіх спостерігачів, викликаючи один із їхніх методів.

2.4 Розробка структури графічного інтерфейсу програмного додатку

Графічний інтерфейс користувача – це інтерфейс, за допомогою якого користувач взаємодіє з електронними пристроями, такими як комп’ютери та смартфони, за допомогою значків, меню та інших візуальних елементів керування. Графічні інтерфейси відображають інформацію та відповідні елементи керування користувача за допомогою графіки, на відміну від текстових інтерфейсів, де дані та команди містяться виключно в тексті [13].

Веб-інтерфейс дозволяє користувачеві взаємодіяти з контентом або програмним забезпеченням, запущеним на віддаленому сервері, за допомогою веб-браузера. Веб-сторінка завантажується з веб-сервера, і користувач може взаємодіяти з цим контентом у веб-браузері, який виступає у ролі клієнта. Існує два типи веб-інтрефейсів. Тип Server-side дозволяє зберігати вміст та генерувати веб-сторінки на віддаленому сервері, тоді як тип client-side генерує-веб сторінку

безпосередньо на машині користувача та забезпечує зручний доступ до контенту за допомогою веб-браузера.

Графічний інтерфейс додатка складається з модуля відеозв’язку, модуля текстових повідомлень, модуля реєстрації, модуля авторизації та модуля керування.

Схему інтерфейсу модуля відеозв’язку зображено на рисунку 2.6.

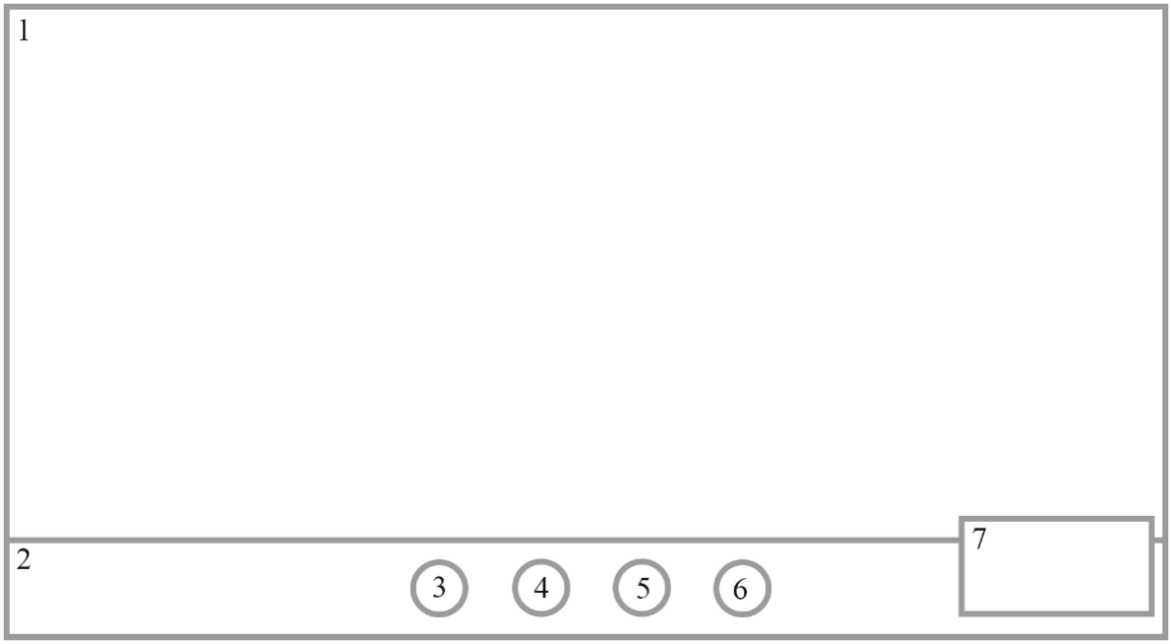


Рисунок 2.6 – Схема інтерфейсу модуля відеозв’язку

Основні елементи інтерфейсу модуля відеозв’язку:

1. Зображення з відеокамери співрозмовника.
2. Панель керування дзвінком.
3. Кнопка перемикання стану відеокамери.
4. Кнопка перемикання стану мікрофона.
5. Кнопка відкриття чату.
6. Кнопка завершення дзвінка.
7. Зображення з відеокамери користувача.

Схему інтерфейсу модуля текстових повідомлень зображено на рисунку 2.7.

Схему інтерфейсу модуля реєстрації зображено на рисунку 2.8.

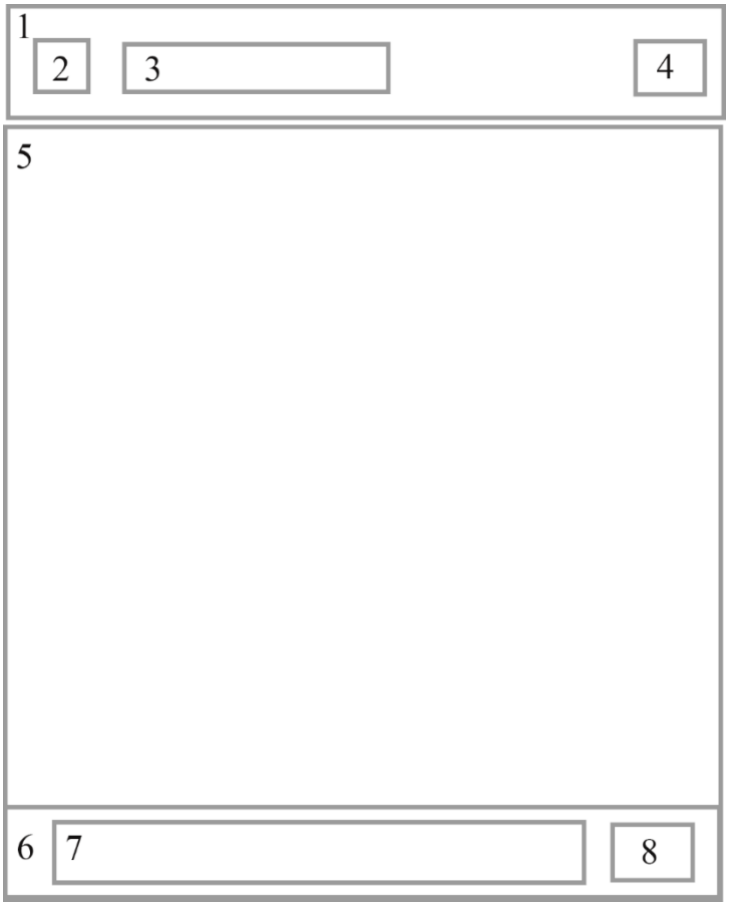


Рисунок 2.7 – Схема інтерфейсу модуля текстових повідомлень

Основні елементи інтерфейсу модуля реєстрації:

1. Робоча область;
2. Поле вводу ім’я;
3. Поле вводу прізвища;
4. Поле вводу дати народження;
5. Поле вводу електронної адреси;
6. Поле вводу пароля;
7. Поле повторного вводу пароля;
8. Область вибору ролі користувача;
9. Кнопка реєстрації;
10. Кнопка переходу до сторінки авторизації.
11. Схему інтерфейсу модуля авторизації зображено на рисунку 2.9.

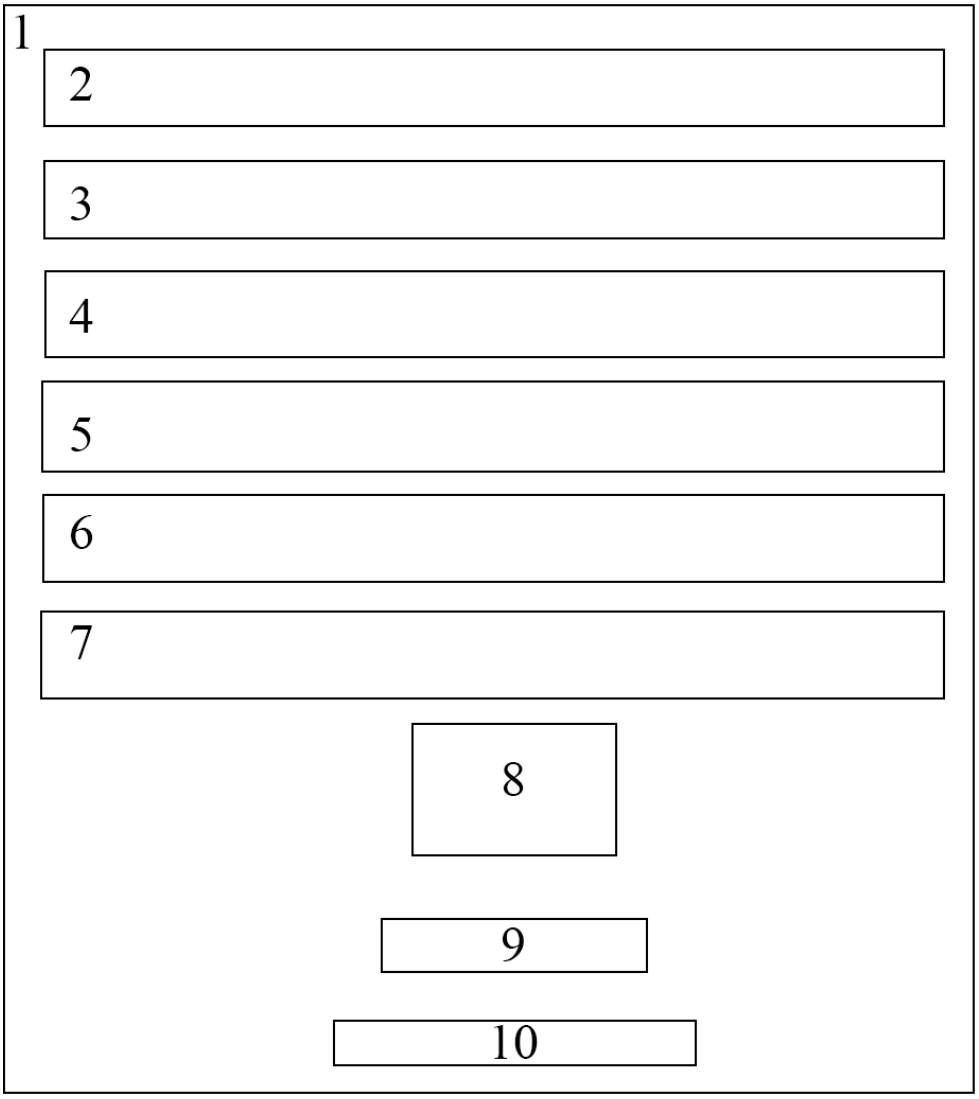


Рисунок 2.8 – Схема інтерфейсу модуля реєстрації

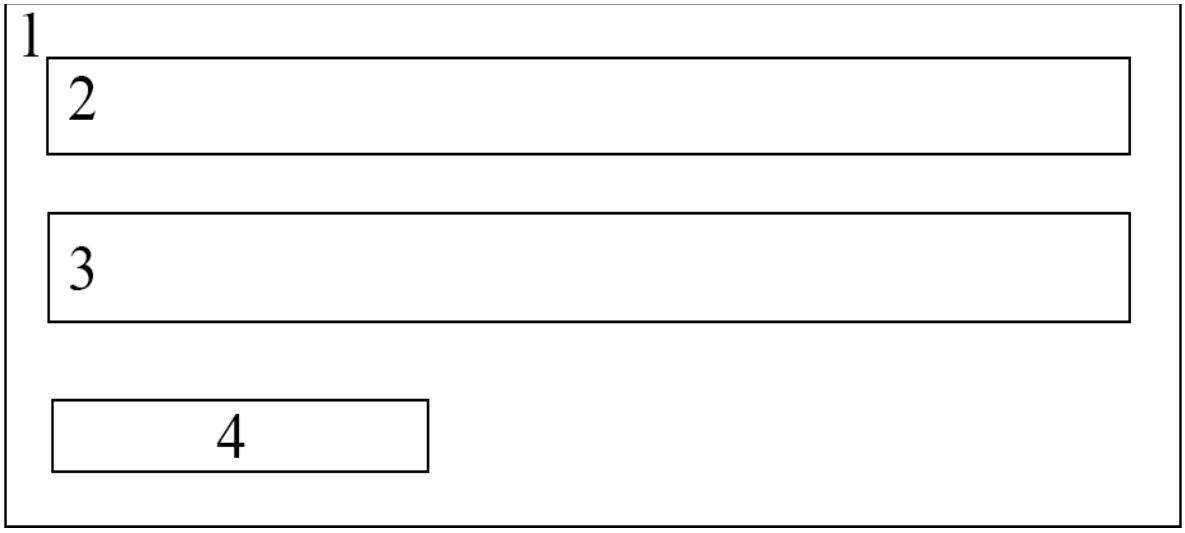


Рисунок 2.9 – Схема інтерфейсу модуля авторизації

Основні елементи інтерфейсу модуля авторизації:

1. Робоча область;
2. Поле вводу електронної адреси;
3. Поле вводу пароля;
4. Кнопка авторизації;

Схему інтерфейсу модуля керування зображено на рисунку 2.10.

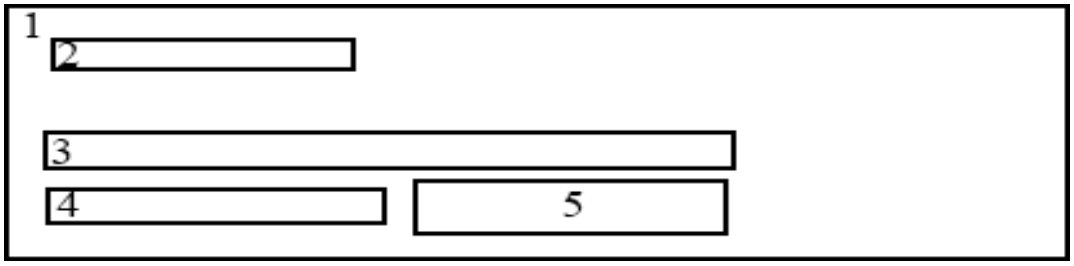


Рисунок 2.10 – Схема інтерфейсу модуля керування

Основні елементи інтерфейсу модуля керування:

1. Робоча область;
2. Дата поточного дня;
3. Час до наступної консультації;
4. Інформація про пацієнта/лікаря;
5. Кнопка створення кімнати.

2.5 Висновки

У другому розділі розроблено та описано алгоритми загальної роботи модулів текстових повідомлень та відеозв’язку.

Розроблено стратегію забезпечення послідовного виконання кроків алгоритму, з обробкою помилок та можливостю продовження роботи програми, за умови неуспішного виконання попереднього кроку алгоритму.

Розроблено схеми інтерфейсів модулів відеозв’язку та текстових повідомлень.

3 РОЗРОБКА ОСНОВНИХ МОДУЛІВ ДОДАТКУ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації програмного додатку

JavaScript – це динамічна скриптова мова програмування, яка підтримує конструювання об’єктів на основі прототипів. Основний синтаксис схожий як на Java, так і на C++, щоб зменшити кількість нових понять, необхідних для вивчення мови. Мова може функціонувати як процедурна, так і як об'єктно-орієнтована. Об’єкти створюються програмно, шляхом приєднання методів і властивостей до порожніх об’єктів під час виконання, на відміну від визначень синтаксичних класів, поширених у компільованих мовах, таких як C++ і Java. Після створення об’єкта його можна використовувати як прототип для створення подібних об’єктів [14].

Java – універсальна, заснована на класах, об'єктно-орієнтована мова програмування; обчислювальна платформа для розробки додатків. Тому Java безпечна та надійна. Широко використовується для розробки Java-додатків для ноутбуків, центрів обробки даних, ігрових консолей, наукових суперкомп'ютерів, мобільних телефонів тощо. Скомпільований код (байт-код) працює на більшості операційних систем (ОС), включаючи Windows, Linux і MacOS. За синтаксисом Java дуже схожа до мов програмування C і C++.

Python – це інтерпретована, об’єктно-орієнтована мова програмування високого рівня з динамічною семантикою. Його високорівневі вбудовані структури даних у поєднанні з динамічною типізацією роблять його дуже привабливим для швидкої розробки додатків, а також для використання в якості скриптової мови або проміжним рівнем для з’єднання існуючих компонентів разом. Простий синтаксис Python, який легко вивчати, підкреслює читабельність і, отже, знижує витрати на обслуговування програми. Python підтримує модулі та пакети, що сприяє модульності програм і повторному використанню коду [15].

PHP – це скриптова мова з відкритим вихідним кодом, інтерпретована й об'єктно-орієнтована, яка може виконуватися на стороні сервера. PHP добре

підходить для веб-розробки. Тому він використовується для розробки веб-додатків (додаток, який виконується на сервері і створює динамічну сторінку). Скрипт PHP виконується набагато швидше, ніж скрипти, написані іншими мовами, такими як JSP і ASP. PHP використовує власну пам'ять, тому навантаження на сервер і час завантаження автоматично зменшуються, що призводить до більш швидкої обробки та кращої продуктивності.

Проаналізувавши мови програмування, було складено таблицю 3.1, яка демонструє відмінності між JavaScript, Java, Python та PHP.

Таблиця 3.1 – Функціональні характеристики мов програмування

Функціональна характеристика/Мова	JavaScript	Java	Python	PHP
Швидкість	1	0	0	1
Універсальність	1	0	1	1
Багатоплатформенність	1	1	1	0
Різноманіття бібліотек та фреймворків	1	1	0,5	0
Функціональність	1	0	1	0
Об’єкто-орієнтованість	1	1	0	1
Сумарний коефіцієнт	6	3	3,5	3

Проаналізувавши вищенаведені дані, обрано мову програмування JavaScript, так як вона краща за інші: Java на 50%, Python на 42% та PHP на 50%.

Отже, проаналізувавши відмінні ознаки мов програмування JavaScript, Java, Python та PHP, було прийнято рішення, що JavaScript найбільше підходить для виконання поставленої задачі.

Для вибору середовища розробки створення програмного додатку проаналізовано такі середовища розробки: WebStorm, VSCode, Sublime Text 3 та Brackets.

WebStorm – це інтегроване середовище розробки для кодування в JavaScript і пов'язаних з ним технологій, включаючи TypeScript, React, Vue, Angular, Node.js, HTML і таблиці стилів [16]. Розроблене на основі платформи Intellij IDEA.

Інтерфейс середовища розробки WebStorm зображено на рисунку 3.1.

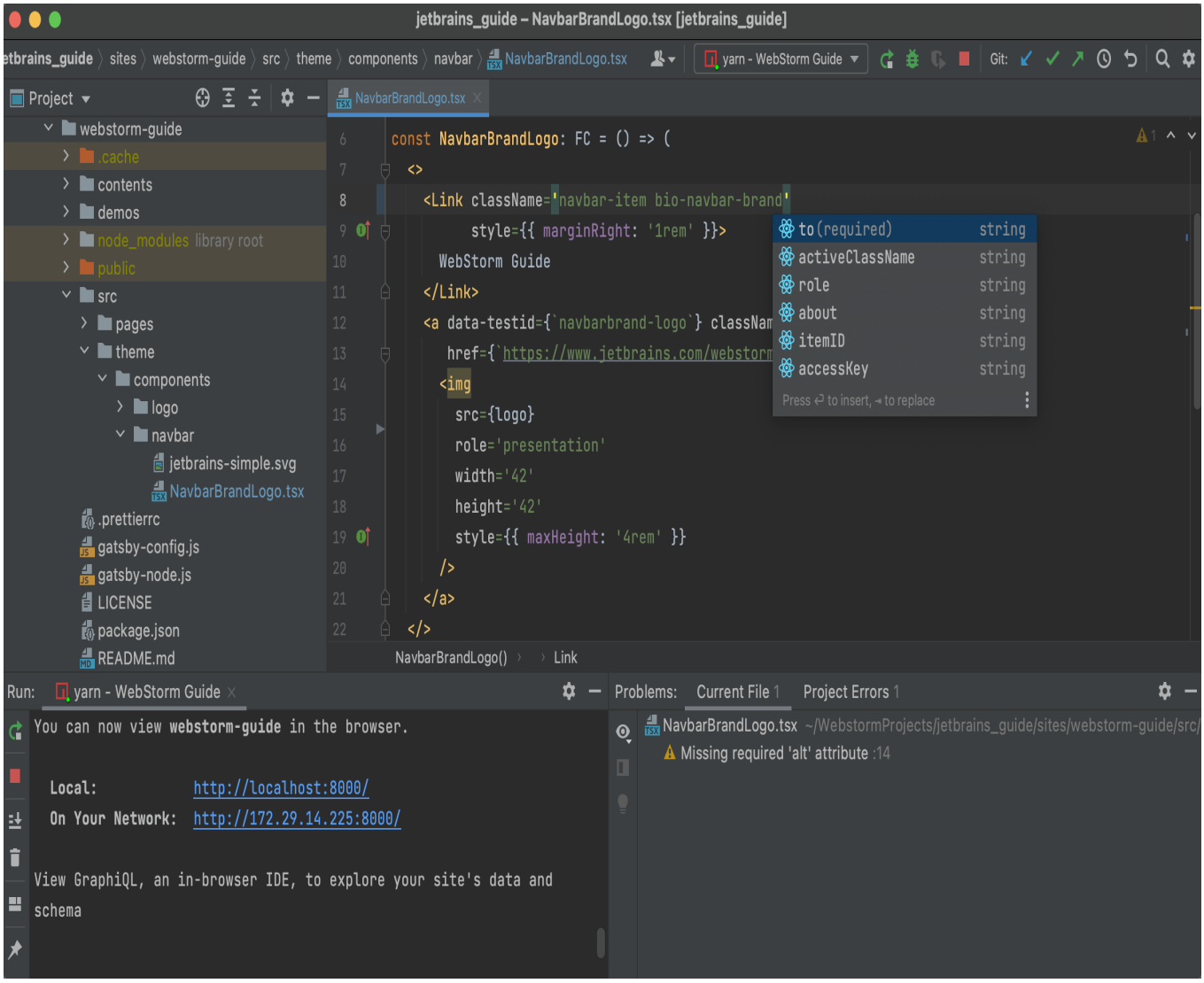


Рисунок 3.1 – Інтерфейс середовища розробки WebStorm

WebStorm забезпечує автодоповнення, аналіз коду на ходу, навігацію за кодом, рефакторинг, відладку та інтеграцію з системами контролю версій. Важливою перевагою інтегрованої середи розробки WebStorm є робота з проектами (в тому числі, рефакторинг коду JavaScript, що знаходиться в різних файлах і папках проекту, а також вставленого в HTML). Підтримується множинна вкладеність (коли в HTML документі вставлений скрипт на Javascript, у який вставлений інший код HTML, всередині якого вставлений Javascript) – в таких конструкціях підтримується коректний рефакторинг. Наявна безкоштовна підписка для студентів.

Visual Studio Code – редактор вихідного коду, розроблений Microsoft для Windows, Linux і macOS.

Інтерфейс середовища розробки VSCode зображено на рисунку 3.2.

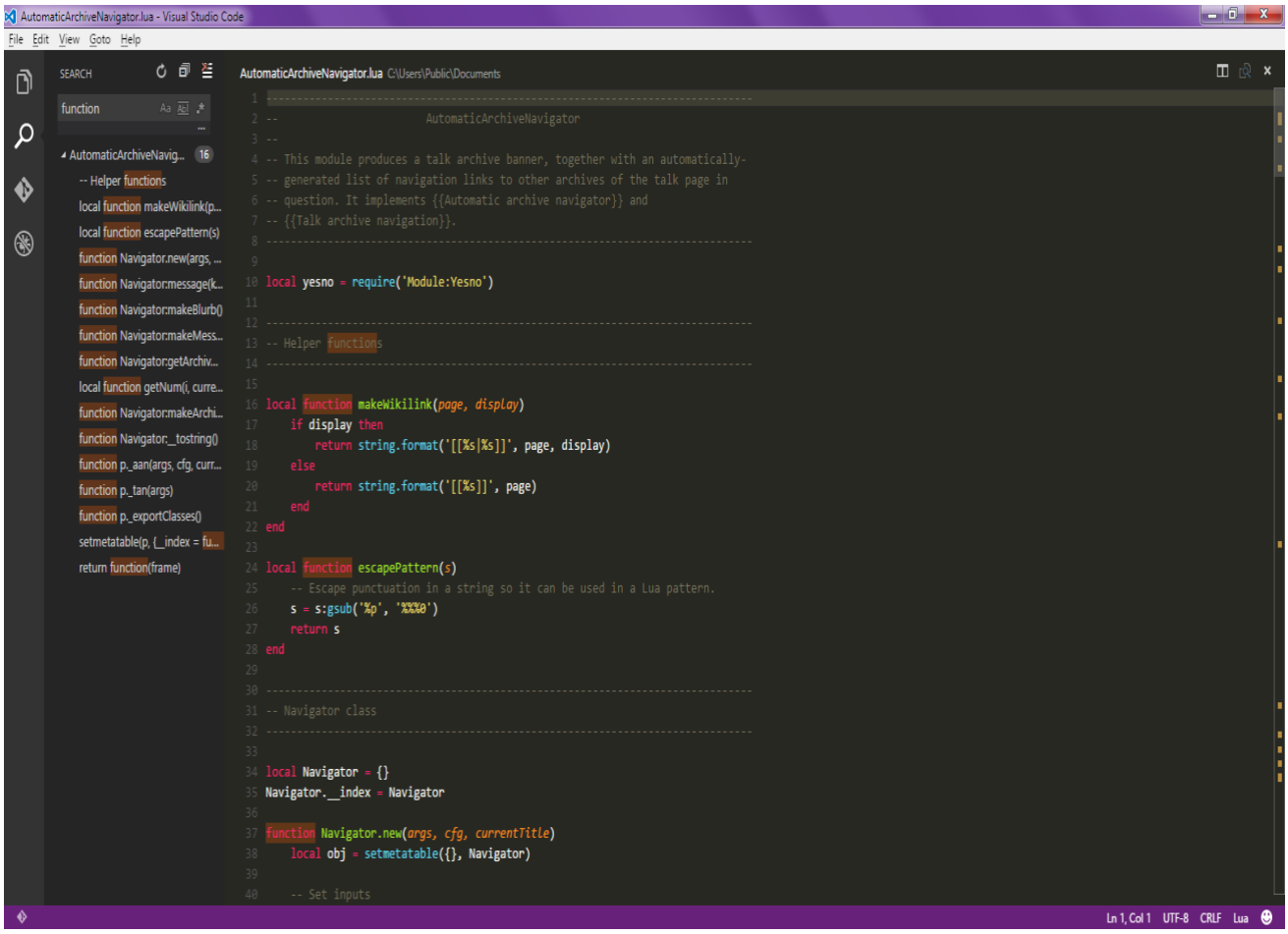


Рисунок 3.2 – Інтерфейс середовища розробки VSCode

Середовище розробки VSCode позиціонується як «легкий» редактор коду для багатоплатформної розробки веб- та хмарних додатків. Включає в себе відладчик, інструменти для роботи з Git, підсвітку синтаксису, IntelliSense та засоби для рефакторингу. Є широкі можливості для налаштування: користувацькі теми, комбінації клавіш і файли конфігурації. Розповсюджується безкоштовно, розробляється як програмне забезпечення з відкритим вихідним кодом.

Brackets – це редактор вихідного коду, зосереджений на веб-розробці. Інтерфейс середовища розробки Brackets зображено на рисунку 3.3.

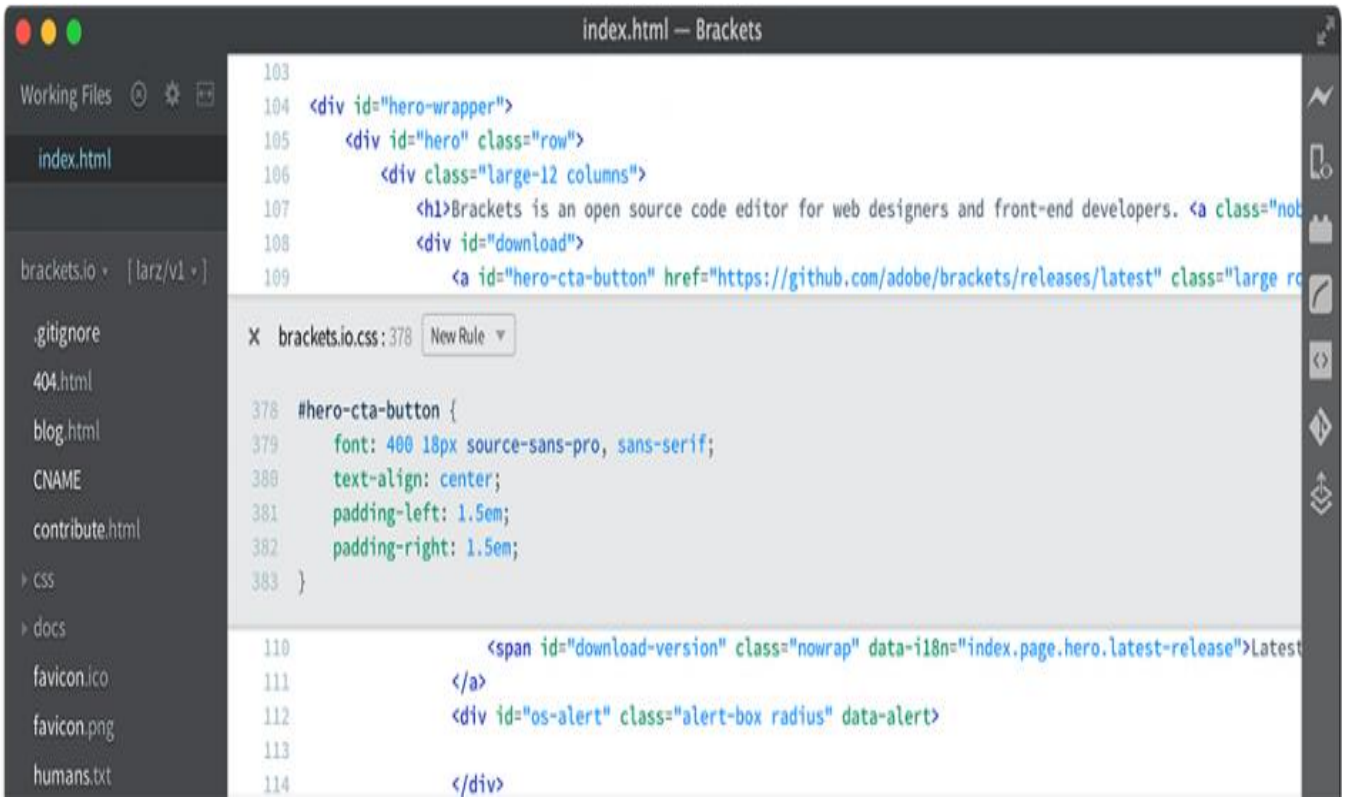


Рисунок 3.3 – Інтерфейс середовища розробки Brackets

Редактор створений корпорацією Adobe Inc., має безкоштовне поширення з відкритим кодом, ліцензоване за ліцензією MIT, і наразі воно підтримується на GitHub розробниками з відкритим кодом. Він написаний на JavaScript, HTML і CSS. Brackets є багатоплатформним, доступним для macOS, Windows і більшості дистрибутивів Linux. Основною метою Brackets є функціональні можливості редагування HTML, CSS та JavaScript.

Sublime Text 3 – швидкий багатоплатформний текстковий редактор, що використовується для розробки програмного забезпечення.

Інтерфейс середовища розробки Sublime Text 3 зображено на рисунку 3.4.

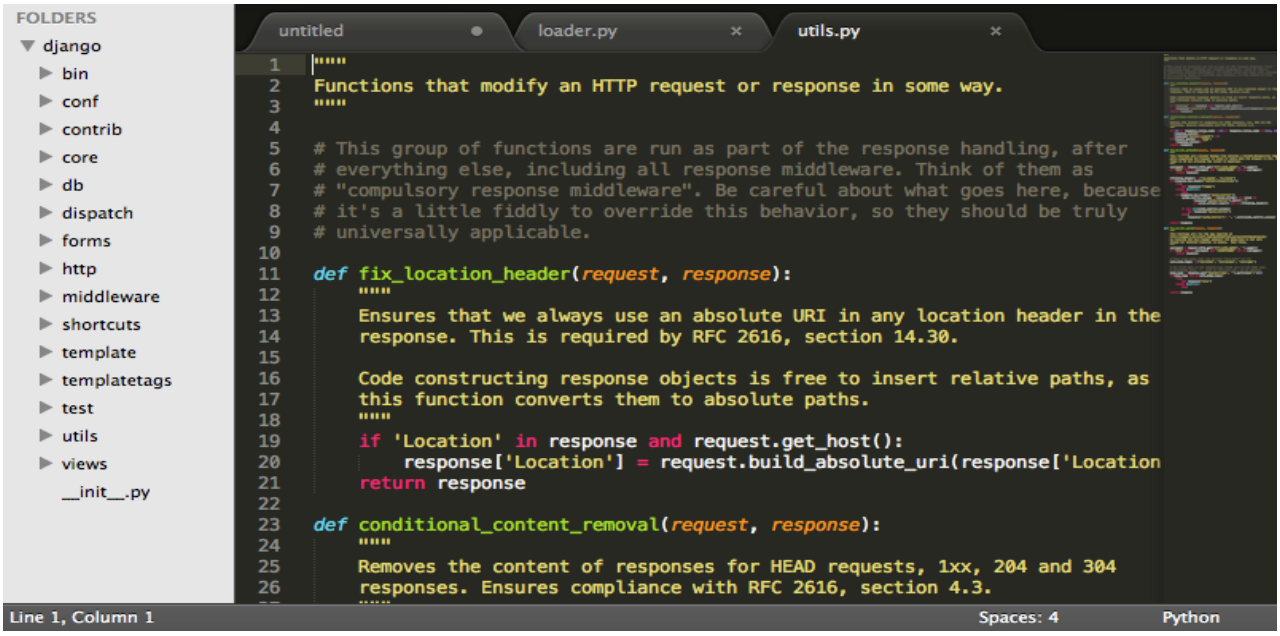


Рисунок 3.4 – Інтерфейс середовища розробки Sublime Text 3

Середовище підримує плагіни, розроблені з використанням мови програмування Python. Дозволяє працювати майже з усіма мовами програмування. Поширюється на платній основі, проте є пробна підписка.

Для порівняння середовищ програмування створено таблицю 3.2.

У результаті порівняння виявлено, що середовище програмування WebStorm краще за: Sublime Text 3 на 28%, Brackets на 53% та VSCode на 16%.

Таблиця 3.2 – Характеристики середовищ програмування

Функціональна характеристика/Мова	WebStorm	Sublime Text 3	Brackets	VSCode
Безкоштовна ліцензія	0,5	0	1	1
Виправлення помилок	1	1	1	1

Продовження таблиці 3.2.

Функціональна характеристика/Мова	WebStorm	Sublime Text 3	Brackets	VSCode
Багатоплатформенність	1	1	1	1
Швидка навігація	1	0	0	0
Множинна вкладеність	1	0	0	0
Система контролю версій	1	1	0	1
Налаштування	1	1	0	1
Історія змін	1	0	0	0
Підтримка різних мов програмування	0	1	0	1
Сумарний коефіцієнт	7,5	5	3	6

Отже, враховуючи всі переваги та особливості, прийнято рішення, що середовище розробки WebStorm найбільше підходить для виконання поставленої задачі.

3.2 Розробка модуля відеозв’язку

Модуль відеозв’язку розміщується у файлі useStartPeerSession.ts, і використовується для того щоб встановити зв’язок між користувачами.

Для зручнішого підключення нових користувачів до кімнати, створено клас PeerConnectionSession, що має такі методи:

- addPeerConnection;
- removePeerConnection;
- callUser;
- joinRoom;
- onCallMade;
- onAnswerMade;
- clearConnections.

Код класу PeerConnectionSession зображено на рисунку 3.5.

```
import io from 'socket.io-client'

const { RTCPeerConnection, RTCSessionDescription } = window

function capitalizeFirstLetter(str: any) {
  return str.charAt(0).toUpperCase() + str.slice(1)
}

class PeerConnectionSession {
  _onConnected: any
  _onDisconnected: any
  _room: any
  peerConnections: any = {};
  senders: any[] = [];
  listeners: any = {};
  socket: any;

  constructor(socket: any) {
    this.socket = socket
    this.onCallMade()
  }
```

Рисунок 3.5 – Код класу PeerConnectionSession

Метод addPeerConnection використовується для створення нового peer2peer з’єдання. Приймає такі аргументи: id, stream та callback.

Аргумент id використовується для ідентифікації підключеного користувача, та поміщається у масив підключень.

Аргумент stream містить посилання на відео- та аудіодоріжки, що транслюються за допомогою браузера іншого користувача.

Аргумент callback – це функція, що передається з фукнціональної компоненти та виконується при успішному підключенні. Код методу addPeerConnection зображено на рисунку 3.6.

```
addPeerConnection(id: any, stream: any, callback: (stream: any) => void) {
  this.peerConnections[id] = new RTCPeerConnection( configuration: {
    iceServers: [{ urls: 'stun:stun.l.google.com:19302' }],
  })

  stream.getTracks().forEach((track: MediaStreamTrack) => {
    this.senders.push(this.peerConnections[id].addTrack(track, stream))
  })

  this.listeners[id] = (event: any) => {
    const fn = this[('_on' + capitalizeFirstLetter(this.peerConnections[id].connectionState))
    fn && fn(event, id)
  }

  this.peerConnections[id].addEventListener('connectionstatechange', this.listeners[id])

  this.peerConnections[id].ontrack = function ({ streams: [stream] }: { streams: any }) {
    console.log({ id, stream })
    callback(stream)
  }

  console.log(this.peerConnections)
}
```

Рисунок 3.6 – Код методу addPeerConnection

Метод removePeerConnection приймає аргумент id, що означає ідентифікаційний номер підключення, та використовується для розриву з'єднання. Його реалізацію зображено на рис 3.7.

```
removePeerConnection(id: any) {
  this.peerConnections[id].removeEventListener('connectionstatechange', this.listeners[id])
  delete this.peerConnections[id]
  delete this.listeners[id]
}
```

Рисунок 3.7 – Реалізація методу removePeerConnection

Метод callUser приймає аргумент to, що означає ідентифікаційний номер користувача, з яким проводиться спроба встановити з'єдання. Використовується для сповіщення користувача про вхідний виклик. Код метода callUser зображено на рисунку 3.8.

```
async callUser(to: any) {
  if (this.peerConnections[to].iceConnectionState === 'new') {
    const offer = await this.peerConnections[to].createOffer()
    await this.peerConnections[to].setLocalDescription(new RTCSessionDescription(offer))

    this.socket.emit('call-user', { offer, to })
  }
}
```

Рисунок 3.8 – Код метода callUser

Метод joinRoom приймає аргумент room, що означає ідентифікаційний номер кімнати, до якої користувач доєднується. Використовується для додавання

користувача до кімнати та для сповіщення інших користувачів про нового користувача. Його реалізацію зображено на рисунку 3.9.

```
joinRoom(room: any) {
  this._room = room
  this.socket.emit('joinRoom', room)
}
```

Рисунок 3.9 – Реалізацію метода joinRoom

Метод onCallMade використовується для сповіщення сервера про успішне створення вихідного виклику, після чого сповіщає співрозмовника про вхідний виклик. Його код зображено на рисунку 3.10.

```
onCallMade() {
  this.socket.on('call-made', async (data: any) => {
    await this.peerConnections[data.socket].setRemoteDescription(new RTCSessionDescription(data.offer))
    const answer = await this.peerConnections[data.socket].createAnswer()
    await this.peerConnections[data.socket].setLocalDescription(new RTCSessionDescription(answer))

    this.socket.emit('make-answer', {
      answer,
      to: data.socket,
    })
  })
}
```

Рисунок 3.10 – Код метода onCallMade

Метод onAnswerMade приймає аргумент callback, що являє собою функцію, яку потрібно потрібно виконати у разі якщо співрозмовник позитивно відповість на вхідний виклик. Використовується для сповіщення сервера про успішне

з’єдання обох користувачів з сервером, після чого передає клієнтській частині додатку адресу співрозмовника, що надає змогу з’єднати два браузера між собою без посередників. Реалізацію метода зображено на рисунку 3.11.

```
onAnswerMade(callback: (socket: any) => void) {
  this.socket.on('answer-made', async (data: any) => {
    await this.peerConnections[data.socket].setRemoteDescription(new RTCSessionDescription(data.answer))
    callback(data.socket)
  })
}
```

Рисунок 3.11 – Реалізація метода onAnswerMade

Метод clearConnections використовується для очистки даних, що накопичувались протягом дзвінка та закриває з’єднання зі співрозмовником за протоколом WebSocket. Його код зображено на рисунку 3.12.

```
clearConnections() {
  this.socket.close()
  this.senders = []
  Object.keys(this.peerConnections).forEach(this.removePeerConnection.bind(this))
}
```

Рисунок 3.12 – Код метода clearConnections

У даному підрозділі розглянуто та описано реалізацію спостерігачів, що виконують основні функції модуля відеозв’язку.

3.3 Розробка модуля текстових повідомлень

Модуль текстових повідомлень знаходиться у файлі useChat.ts та використовується для обміну повідомлення між користувачами у реальному часі.

Спочатку розглянемо серверну частину модуля. Вона представлена у вигляді набору спостерігачів.

Спостерігач IS_USER використовується для ідентифікації користувача. Його реалізацію зображено на рисунку 3.13.

```
socket.on( events.IS_USER, ( nickname, cb ) => {
    methods.isUser( users, nickname ) ? cb({ isUser: true, user: null }) :
        cb({ isUser: false, user: methods.createUser( nickname, socket.id )})
})
```

Рисунок 3.13 – Реалізація спостерігача IS_USER

Спостерігач NEW_USER відслідковує створення нових підключень та сповіщає клієнтську частину про приєднання нового користувача до кімнати.

Його реалізацію зображено на рисунку 3.14.

```
socket.on( events.NEW_USER, user => {
    users = methods.addUsers( users, user )
    socket.user = user
    io.emit( events.NEW_USER, { newUsers: users })
})
```

Рисунок 3.14 – Реалізація спостерігача NEW_USER

Спостерігач LOGOUT відслідковує статус користувача та сповіщає клієнтську частину про те, що співрозмовник покинув чат. Його реалізацію зображено на рисунку 3.15.

```
socket.on( events.LOGOUT, () => {
    users = methods.delUser( users, socket.user.nickname )
    io.emit( events.LOGOUT, { newUsers: users, outUser: socket.user.nickname}  )
})
```

Рисунок 3.15 – Реалізація спостерігача LOGOUT

Спостерігач MESSAGE_SEND відслідковує відправку нового повідомлення та сповіщає про це клієнтську частину. Код метода зображено на рисунку 3.16.

```
socket.on( events.MESSAGE_SEND, ({ channel, msg }) => {
    let message = methods.createMessage( msg, socket.user.nickname )
    io.emit( events.MESSAGE_SEND, ({ channel, message }))
})
```

Рисунок 3.16 – Код спостерігача MESSAGE_SEND

Спостерігач TYPING надсилає повідомлення про те, що користувач друкує повідомлення в цей момент. Його код зображено на рисунку 3.17.

```
socket.on( events.TYPING, ({ channel, isTyping }) => {
    socket.user && io.emit( events.TYPING, { channel, isTyping, sender: socket.user.nickname })
})
```

Рисунок 3.17 – Код спостерігача TYPING

Спостерігач CHECK_CHANNEL відповідає за перевірку існування текстових повідомлень між користувачами. Якщо користувачі раніше не використовували модуль текстових повідомлень, то для них створиться та налаштується з'єднання. Його реалізацію зображено на рисунку 3.18.

```
socket.on( events.CHECK_CHANNEL, ({ channelName, channelDescription }, cb ) => {
    if( methods.isChannel( channelName, chatsList ) ){
        cb( true )
    } else {
        let newChat = methods.createChat({ name: channelName, description: channelDescription })
        chatsList.push( channelName )
        chats.push( newChat )
        io.emit( events.CREATE_CHANNEL, newChat )
        cb( false )
    }
})
```

Рисунок 3.18 – Реалізація спостерігача CHECK_CHANNEL

На клієнтській стороні присутні майже всі спостерігачі, що й на стороні сервера. При ініціалізації сторінки вмикається перелік спостерігачів, що відслідковують всі зміни у модулі текстових повідомлень. Перелік спостерігачів, що вмикаються при ініціалізації модуля, зображено на рисунку 3.19.

```
componentDidMount(){
  let { socket } = this.props
  socket.emit(events.INIT_CHATS, this.initChats )
  socket.on( events.MESSAGE_SEND, this.addMessage )
  socket.on( events.TYPING, this.addTyping )
  socket.on( events.P_MESSAGE_SEND, this.addPMessage )
  socket.on( events.P_TYPING, this.addPTyping)
  socket.on( events.CREATE_CHANNEL, this.updateChats )
}
```

Рисунок 3.19 – Перелік спостерігачів на стороні клієнта

У даному підрозділі розглянуто та описано реалізацію спостерігачів, що виконують основні функції модуля текстових повідомлень.

3.4 Висновки

У третьому розділі проаналізовано мови програмування та середовища програмування, що використовуються для веб-розробки.

Прийнято рішення використовувати WebStorm у якості середовища розробки, адже за результатом порівняння він найкраще підходить для виконання поставленої задачі.

Після проведення аналізу мов програмування, обрану мову JavaScript, так як вона має усі необхідні інструменти для зручної розробки раніше визначених модулів.

Розроблено та описано модулі текстових повідомлень та відеозв’язку.

4 ТЕСТУВАННЯ ОСНОВНИХ МОДУЛІВ ДОДАТКУ

4.1 Тестування веб-додатку

Веб-тестування або тестування веб-додатків – це практика програмного забезпечення, яка забезпечує якість шляхом тестування того, що функціональність даного веб-додатка працює за призначенням або відповідно до вимог [17]. Веб-тестування дозволяє знаходити помилки в будь-який момент, до випуску або щодня.

Функціональне тестування – це процес, який включає в себе кілька параметрів тестування, таких як користувальницький інтерфейс, API, тестування бази даних, тестування безпеки, клієнтське та серверне тестування та основні функції веб-сайту. Функціональне тестування дуже зручне і дозволяє користувачам виконувати як ручне, так і автоматичне тестування. Це виконується для перевірки функціональності кожної функції на веб-сайті.

Статичне тестування – це тип тестування програмного забезпечення, при якому програмне забезпечення тестується без виконання коду. Для виявлення помилок проводяться ручні або автоматизовані перевірки коду, документів вимог та оформлених документів. Основною метою статичного тестування є покращення якості програмного забезпечення шляхом виявлення помилок на ранніх етапах процесу розробки програмного забезпечення.

Під динамічним тестуванням виконується код. Він перевіряє функціональну поведінку програмної системи, використання пам’яті/процесора та загальну продуктивність системи. Звідси назва «Динамічний». Основна мета цього тестування – підтвердити, що програмний продукт працює відповідно до вимог бізнесу. Це тестування також називається технікою виконання або перевірочним тестуванням.

Розглянемо ще два підходи до тестування ПЗ: тестування чорної скрині та тестування білої скрині.

Тестування білої скрині – це тестування конструкції. У ході проведення тестування перевіряється внутрішнє кодування та інфраструктура програмного забезпечення, зосереджене на перевірці попередньо визначених вхідних даних щодо очікуваних і бажаних результатів [18]. Цей метод заснований на внутрішній роботі програми і базується на тестуванні внутрішньої структури. У цьому типі тестування необхідні навички програмування для розробки тестових випадків. Основна мета тестування білої скрині – зосередитися на потоці вхідних і вихідних даних за допомогою ПЗ.

Тестування чорної скрині – це метод тестування програмного забезпечення, при якому перевіряються функціональні можливості програмних додатків без знання внутрішньої структури коду, деталей реалізації та внутрішніх шляхів [19]. Цей метод в основному зосереджується на вхідних та вихідних даних програмних додатків і повністю базується на вимогах та специфікаціях програмного забезпечення. Метод також відомий як поведінковий тест.

Коли користувач вперше заходить у веб-додаток, він одразу ж бачить головну сторінку. Тому дуже важливо, щоб ця сторінка працювала та відображалась правильно.

Головну сторінку веб-додатку «ЗДОРОВ’Я» зображено на рисунку 4.1.

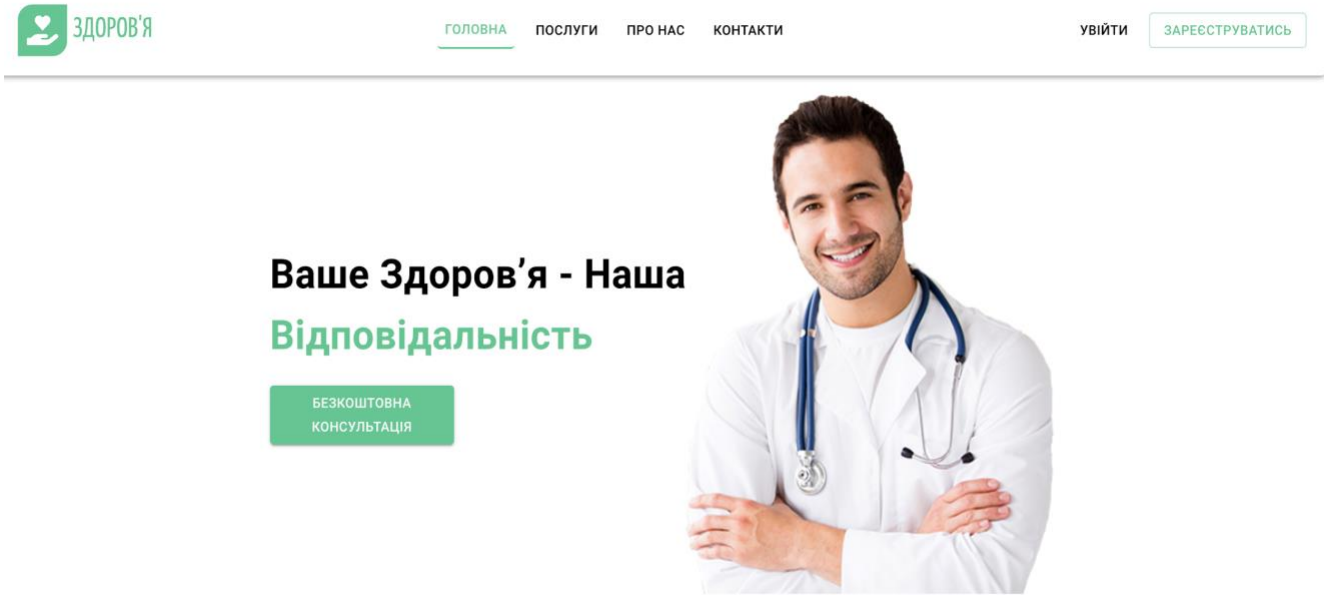


Рисунок 4.1 – Головна сторінка

Форма реєстрації — це веб-сторінка, спливаюче вікно або модальна сторінка, де користувачі вводять інформацію, необхідну для доступу до служб цього веб-додатку. Зібрана інформація визначається видом діяльності сервісу та послугами, які він пропонує. Більшість форм реєстрації вимагають імені, адреси електронної пошти, імені користувача та пароля.

Форму реєстрації зображено на рисунку 4.2.

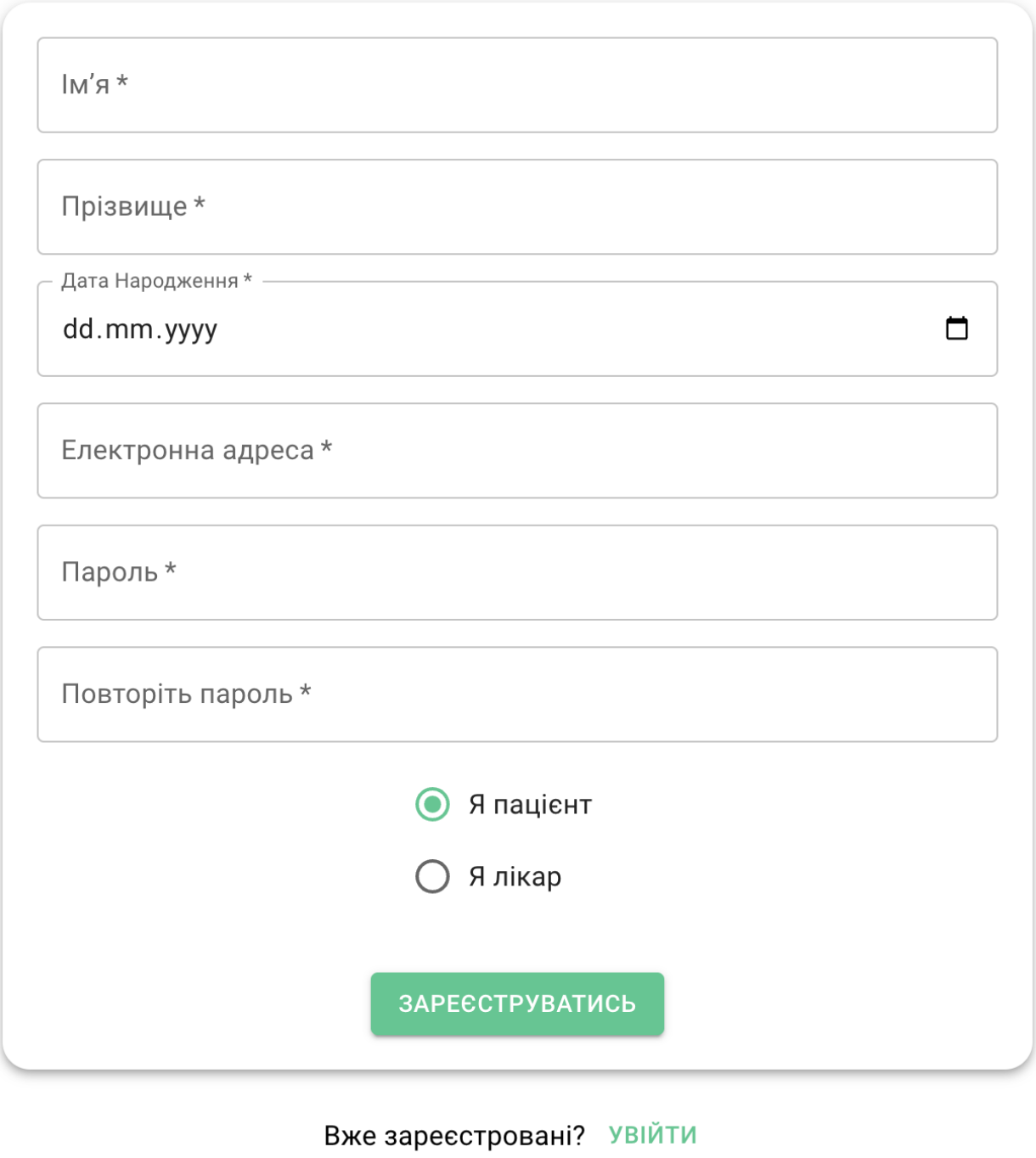


Рисунок 4.2 – Форма реєстрації користувача

Важливо перевірити форму, надіслану користувачем, оскільки вона може мати невідповідні значення. Отже, для автентифікації користувача необхідна перевірка.

JavaScript забезпечує можливість перевірки форми на стороні клієнта, тому обробка даних буде швидшою, ніж перевірка на стороні сервера. Більшість веб-розробників віддають перевагу перевірці форми саме цьому типу валідації даних.

За допомогою JS можна перевірити ім’я, пароль, електронну пошту, дату, номери мобільних телефонів та інші поля.

Приклад валідації форми зображено на рисунку 4.3.

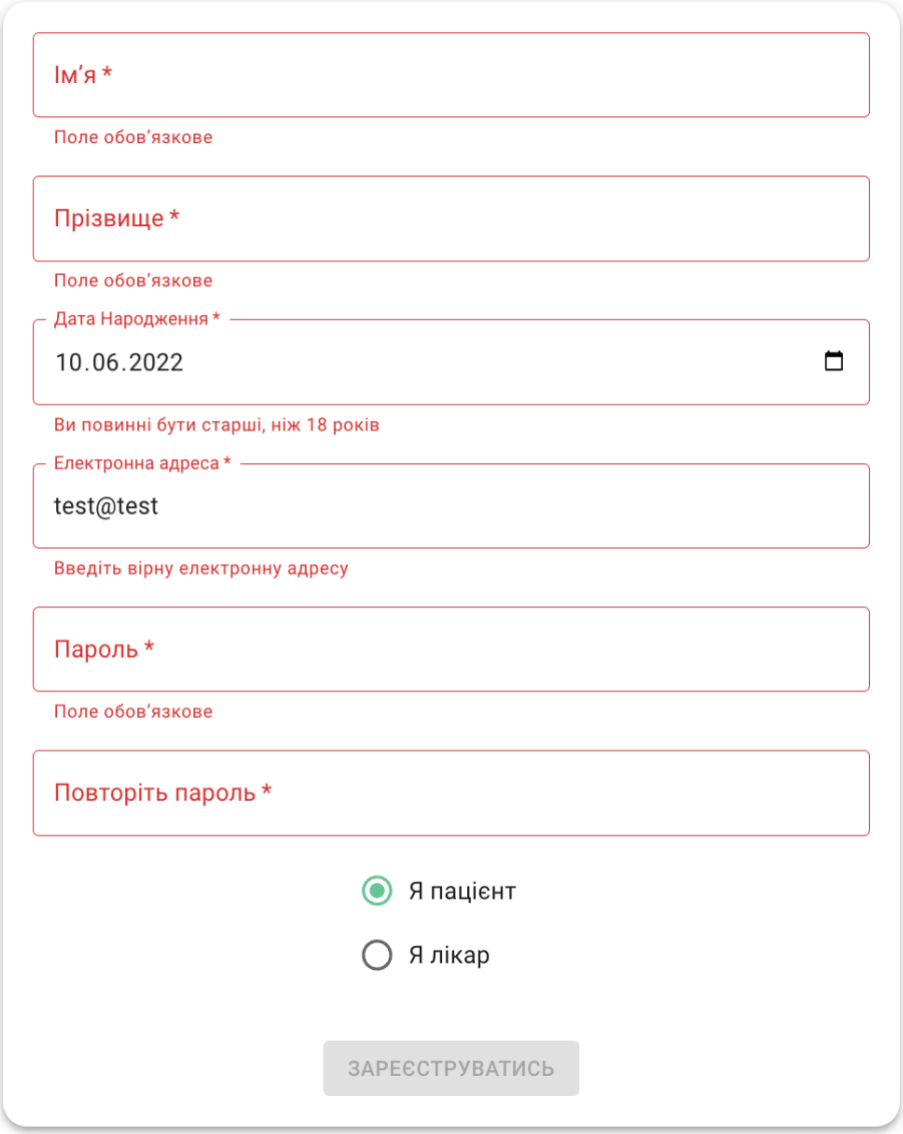


Рисунок 4.3 – Валідація форми реєстрації користувача

Форма авторизації користувача використовується для введення облікових даних для доступу до ресурсів веб-додатку. Форма входу містить поля для імені користувача та пароля. Після надсилання форми входу її базовий код перевіряє автентичність облікових даних, надаючи користувачеві доступ до ресурсів веб-додатку.

Форму авторизації користувача зображено на рисунку 4.4.

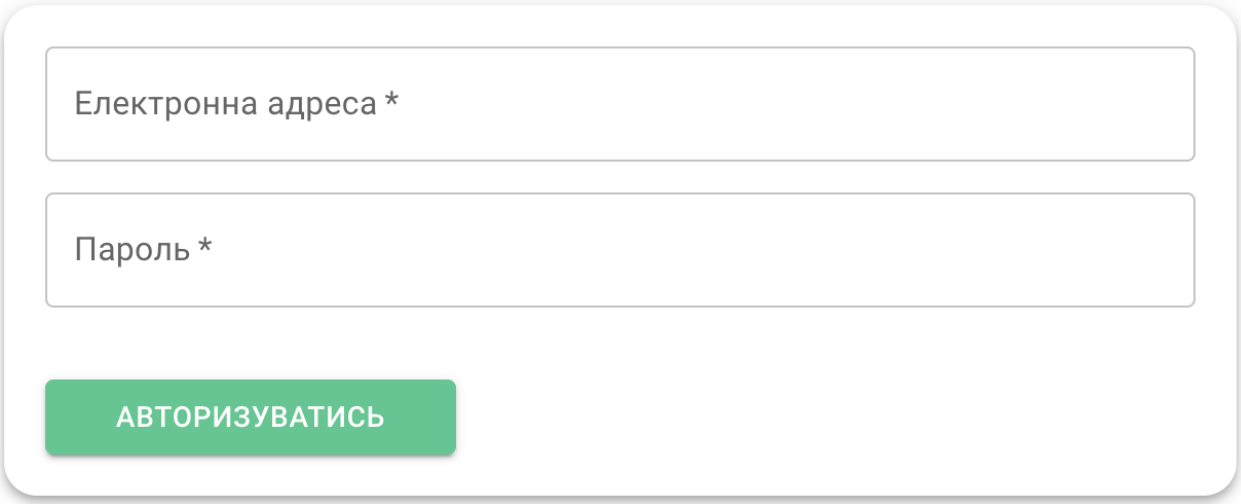


Рисунок 4.4 – Форма авторизації користувача

Валідацію форми авторизації користувача зображено на рисунку 4.5.

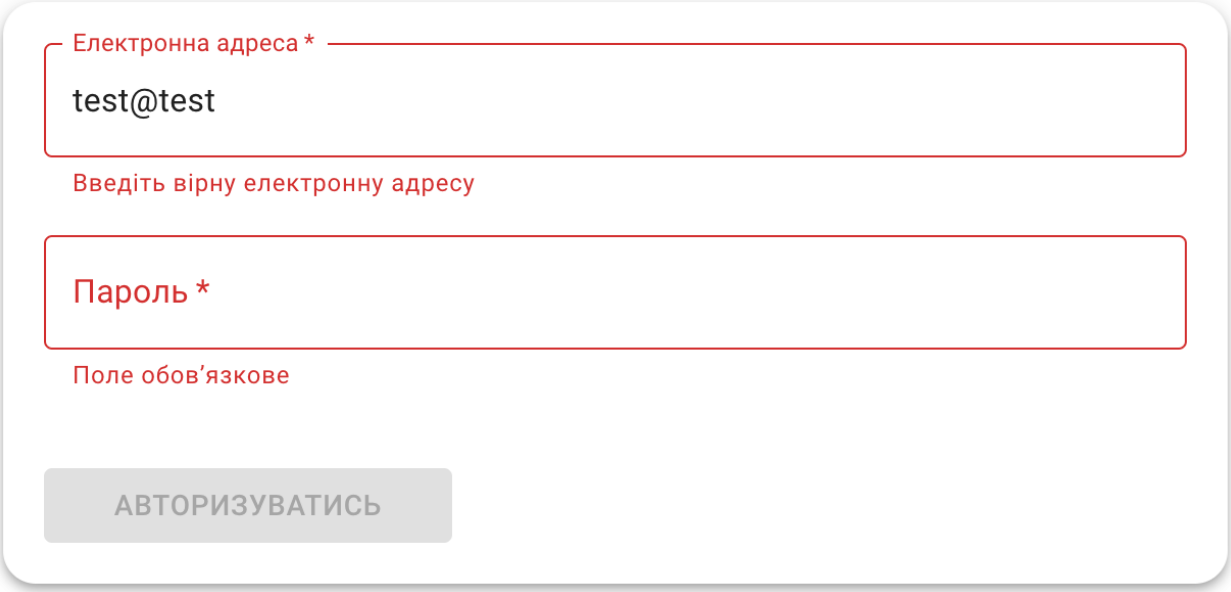


Рисунок 4.5 – Валідація форми авторизації користувача

Потрібно повідомити користувача, якщо авторизація невдала. Для цього використовуються сповіщення.

Сповіщення – це повідомлення, що з'являється, коли програма хоче, щоб ви звернули увагу. Сповіщення – це спосіб сповістити вас про те, що сталося щось нове, щоб не пропустити нічого, що могло б бути вартим вашої уваги та з'явилося незалежно від того, використовуєте ви програму чи ні.

Приклад сповіщення зображено на рисунку 4.6.

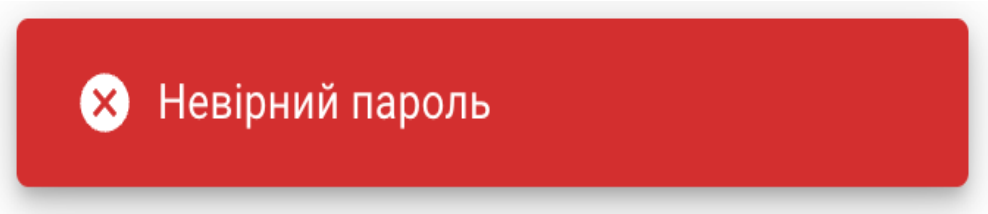


Рисунок 4.6 – Сповіщення про невірний пароль

Після успішної авторизації користувач потрапляє на сторінку керування. Панель керування містить інформацію про наступну консультацію та дозволяє створити приватну кімнату для проведення консультації з певним пацієнтом.

Приклад панелі керування лікаря зображено на рисунку 4.7.

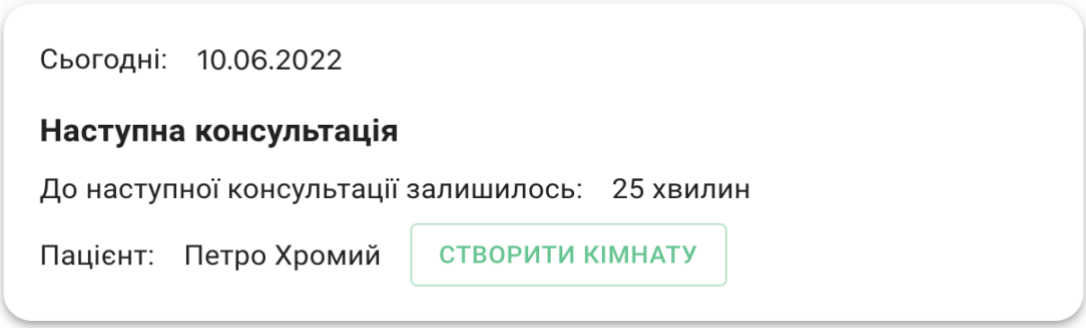


Рисунок 4.7 – Панель керування лікаря

Також розроблено панель керування для пацієнта, де він може дізнатись про свою найближчу заплановану консультацію та приєднатись до створеної лікарем кімнати.

Приклад панелі керування пацієнта зображено на рисунку 4.8.

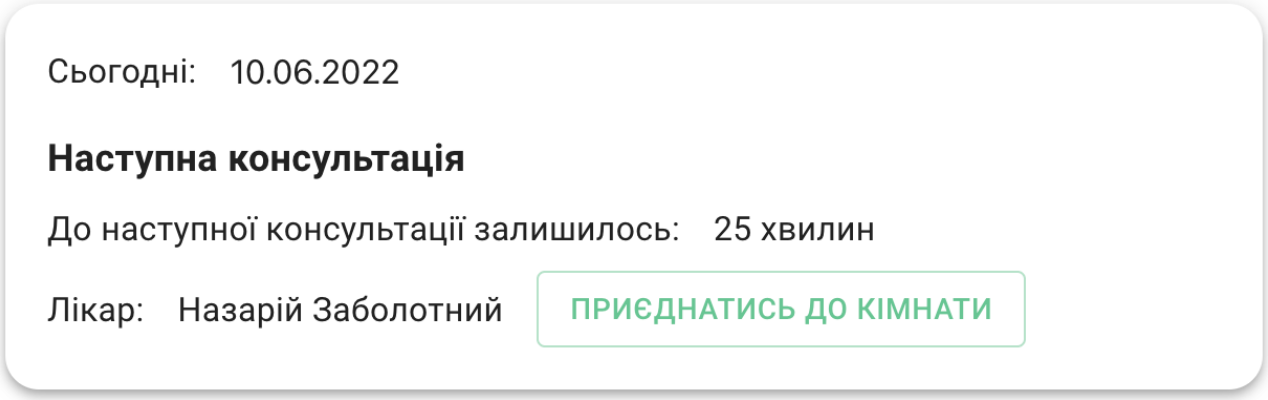


Рисунок 4.8 – Панель керування пацієнта

Після створення кімнати, лікар потрапляє у приватний канал відеовз’язку, де очікує на пацієнта. Там він має змогу побачити себе, вимкнути камеру, вимкнути мікрофон, відкрити та переглянути чат та закрити доступ до кімнати.

Приклад стану встановлення зв’язку зображено на рисунку 4.9.

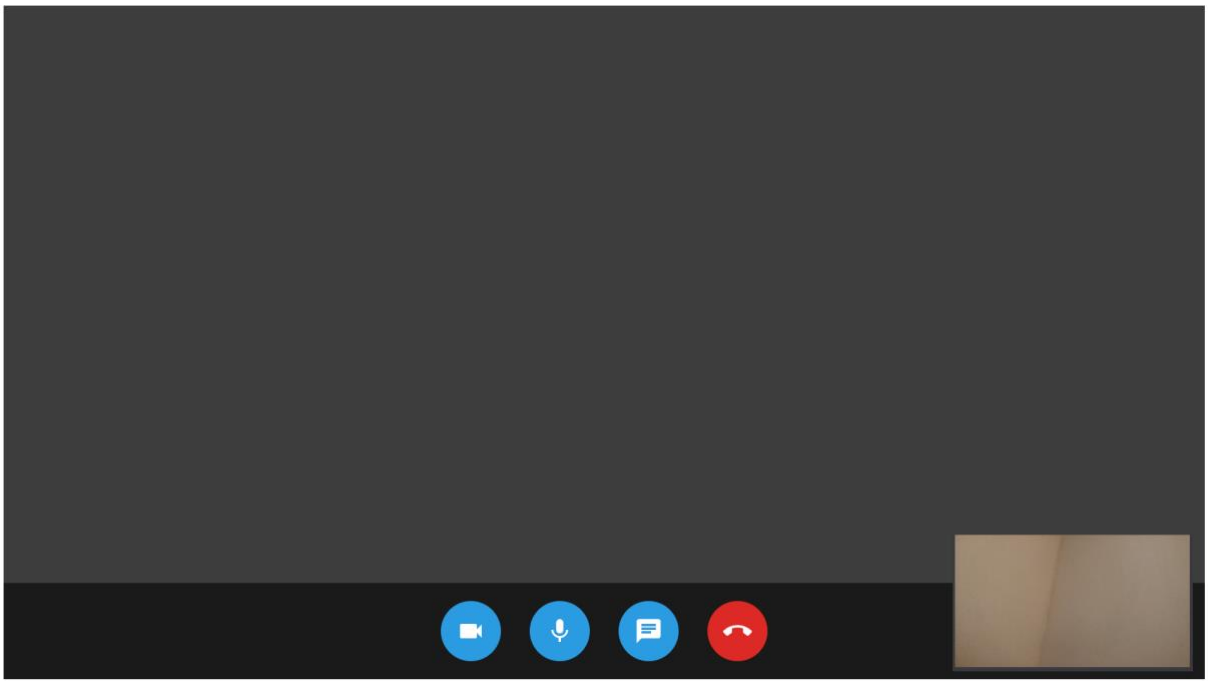


Рисунок 4.9 –Модуль відеозв’язку під час встановлення з’єдання

Коли зв’язок успішно встановлено, модуль відеозв’язку відображає відео співрозмовника, відео користувача та панель керування дзвінком.

Приклад роботи модуля відеозв’язку зображено на рисунку 4.10.

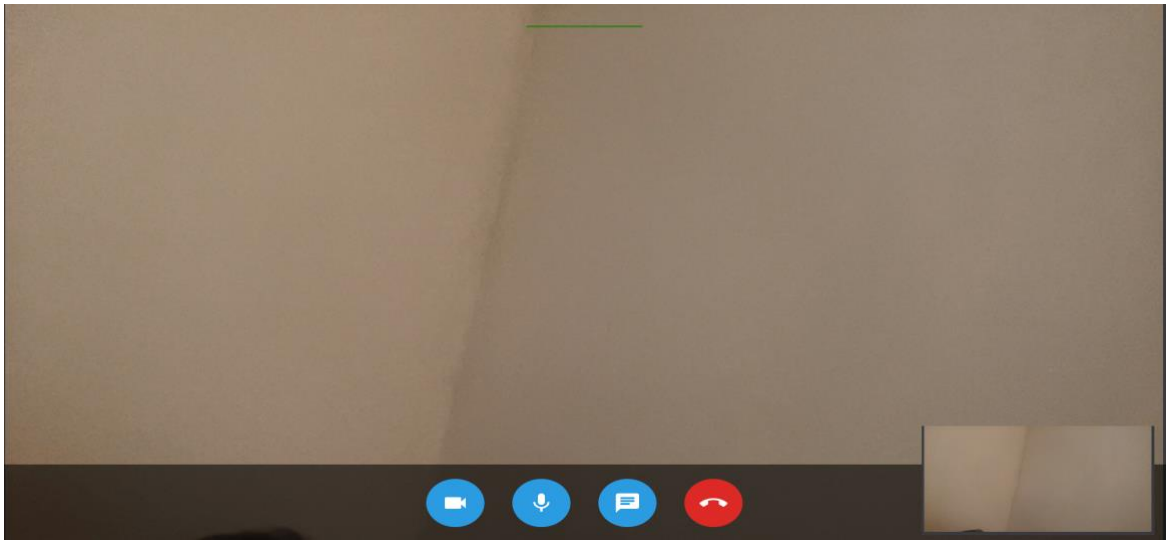


Рисунок 4.10 – Робота модуля відеозв’язку

Кнопка «Відеокамера» використовується для зміни стану відеокамери. Коли користувач натискає на внопку перемикання відеокамери, іконка повинна змінитись, а камера – змінити стан на протилежний.

Стан модуля відеозв’язку з кимкненою камерою зображено на рисунку 4.11.

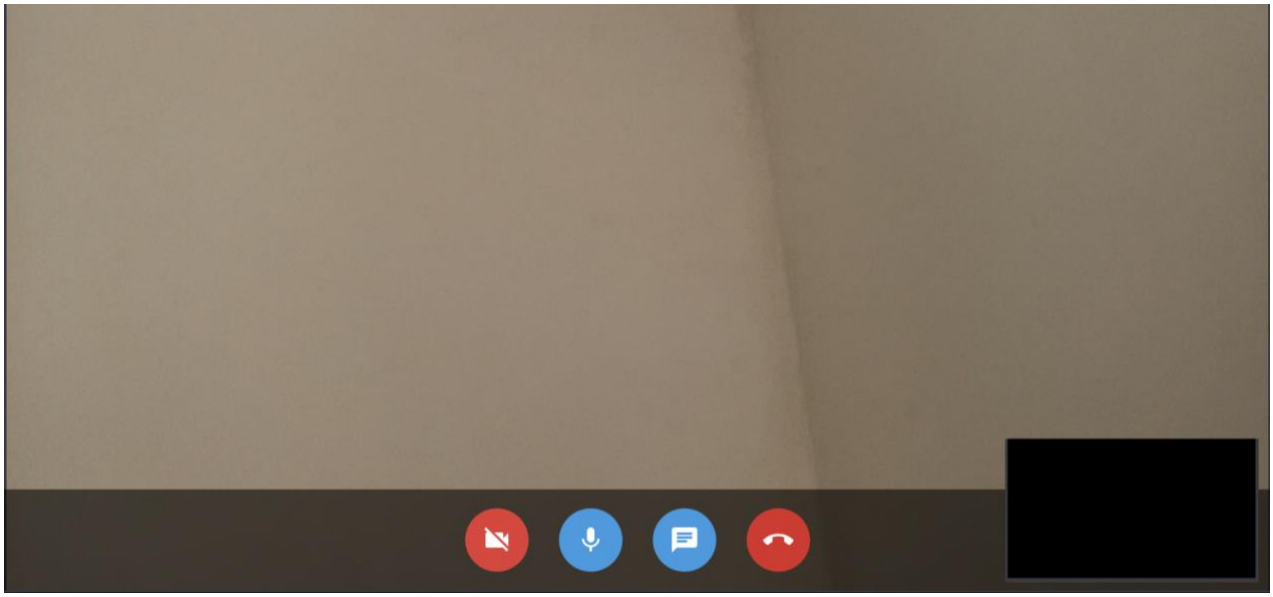


Рисунок 4.11 – Стан вимкненої камери

Кнопка мікрофона використовується для перемикання стану мікрофона. Коли користувач натискає на внопку перемикання стану мікрофона, іконка повинна змінитись, а мікрофон – змінити стан на протилежний.

За допомогою кнопки чату можна відобразити вікно модуля текстових повідомлень зі співрозмовником.

Кнопка «Завершити виклик» відповідає за розрив з’єднання та направляє користувача на сторінку керування.

Приклад вікна модуля текстових повідомлень зображено на рисунку 4.12.

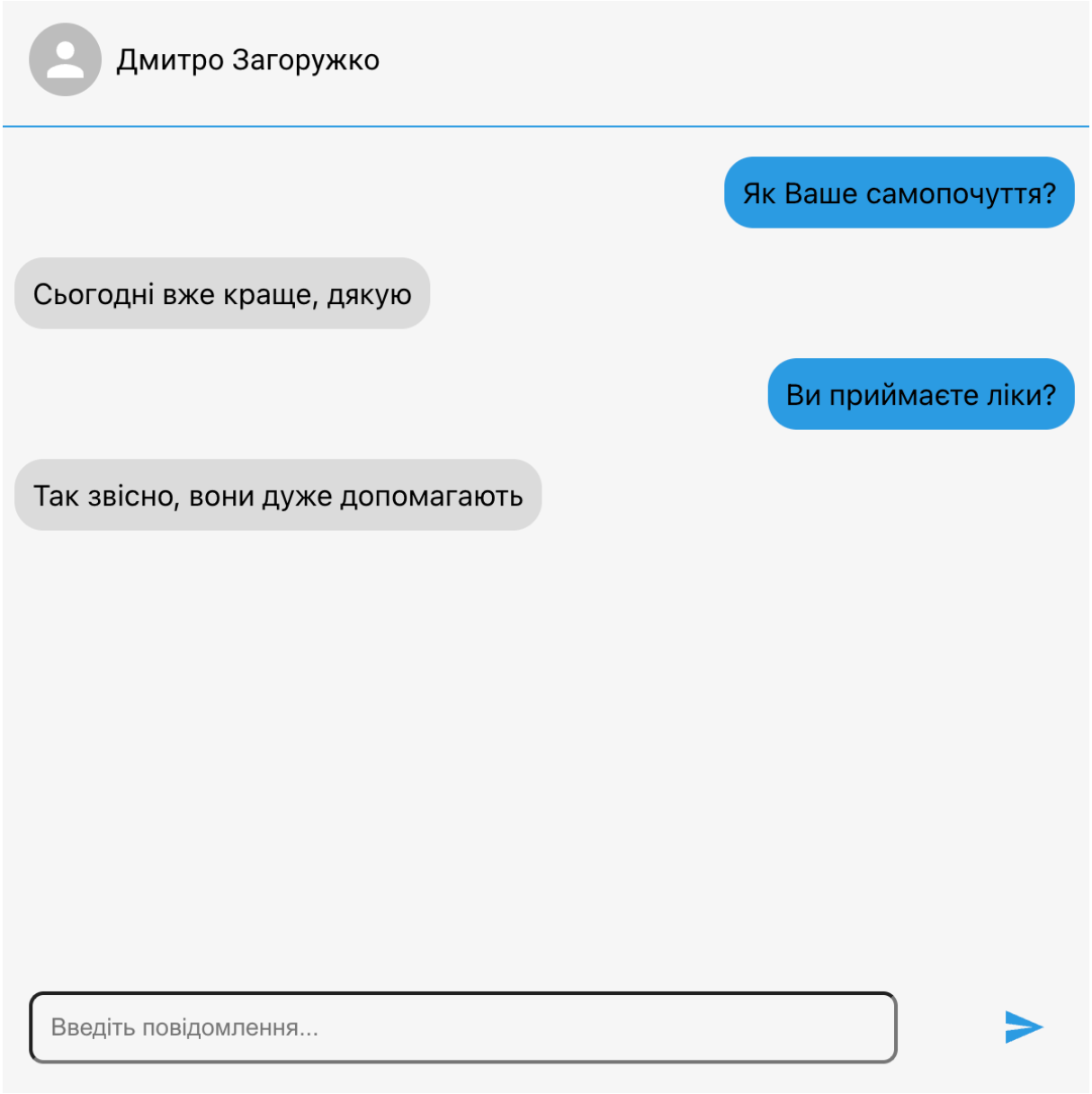


Рисунок 4.12 – Вікно модуля текстових повідомлень

Отже, у підрозділі описано та протестовано основні модулі додатку. Розроблений програмний додаток працює правильно. Повідомлення про помилку виводяться у вигляді сповіщення. Валідація полів вводу працює коректно. Модулі відеозв’язку та текстових повідомлень працюють вірно.

Тестування програмного продукту показало повну відповідність поставленому технічному завданню.

4.2 Розробка інструкції користувача

Інструкція користувача передбачає визначення технічних вимог для запуску програмного продукту та описує процес запуску серверної та клієнтської частин додатку.

Опис мінімальної та рекомендованої конфігурацій машини, що запускає код можна знайти в таблицях 4.1 та 4.2.

Таблиця 4.1 – Мінімальна конфігурація:

Тип процесора	32-розрядний (x86), 64-розрядний (x64) або ARM процесор з тактовою частотою 1 ГГц
Об’єм оперативної пам’яті	2 ГБ
Місце на жорсткому диску	600 МБ
Операційна система	Ubuntu 16.04/ Windows 7/ MacOS High Sierra

Таблиця 4.2 – Рекомендована конфігурація:

Тип процесора	32-розрядний (x86), 64-розрядний (x64) або ARM процесор з тактовою частотою 2.4 ГГц
Об’єм оперативної пам’яті	4 ГБ
Розмір жорсткого диску	1 ГБ
Операційна система	Ubuntu 21.04/ Windows 10/ MacOS Monterey

Для запуску програмного продукту необхідно встановити Docker та менеджер пакетів NPM. За допомогою сервісу Docker потрібно завантажити та запустити контейнер з образом системи черг повідомлень Redis.

Redis – це швидка база даних і кеш із відкритим вихідним кодом під ліцензією BSD, написана на C і оптимізована для швидкості. Його часто називають сервером структури даних, оскільки його основні типи даних подібні до тих, що зустрічаються в мовах програмування, таких як рядки, списки, словники (або хеші), набори та відсортовані набори. Він також надає багато інших структур даних і функцій для приблизного підрахунку, геолокації та обробки потоків.

З баз даних NoSQL різні структури даних Redis найбільш близькі до нативних структур даних, які програмісти найчастіше використовують у програмах та алгоритмах. Така простота використання робить його ідеальним для швидкої розробки та швидких додатків, оскільки основні структури даних легко розподіляються між процесами та службами.

За замовчуванням Redis зберігає дані в пам’яті з періодичним збереженням диска за замовчуванням. Оскільки Redis зберігає дані на диску, він може служити як класична база даних для багатьох випадків використання, або як кеш. Коли Redis буде заповнено, клієнт отримує помилку, але його можна налаштувати як кеш для видалення старих і менш важливих даних у міру надходження нових даних. В обох випадках розмір доступної пам’яті є основним обмеженням для її використання.

NPM (Node Package Manager) – це менеджер пакетів для платформи Node JavaScript. Npm відомий як найбільший у світі реєстр програмного забезпечення. Розробники з відкритим кодом у всьому світі використовують npm для публікації та обміну вихідним кодом.

Наступним кроком потрібно запустити контейнер Redis за допомогою графічного інтерфейсу сервісу Docker.

Далі потрібно зайти у папку «server» та виконати команди «npm install»l та «npm start». Виконавши ці команди, буде запущено серверну частину додатку.

Потім потрібно зайти у папку «client» та виконати вищенаведені команди.

Після запуску системи користувач повинен авторизуватися, після чого він буде перенаправлений на сторінку керування. В результаті він отримає інформацію про найближчу заплановану консультацію. На даній сторінці він зможе створити кімнату для зв’язку з пацієнтом/лікарем. Після подання заявки користувача буде переведено на сторінку, де знаходяться модулі відуозв’язку та текстових повідомлень.

4.3 Висновки

Важливим етапом розробки додатку є тестування, так як воно дозволяє знайти та вчасно виправити всі недоліки розробленого додатку, що значно покращить оцінку користувачів.

У даному розділі розглянуто різні стратегії тестування та визначено, що для розробленого додатку найоптимальнішим є тестування «чорного ящика». Даний метод допоможе протестувати програмний продукт так, як його використовує користувач, тому протестує всі можливі ситуації, що виникнуть при експлуатації програмного продукту. Також даний метод є достатньо швидким, адже не вимагає аналізу стану програмного продукту, використання ресурсів програми, аналізу пам’яті тощо.

При проведенні тестування отримана повна відповідність вхідних даних і вихідних результатів. Помилок при роботі програми не виявлено, і тому вона підтверджує нормальний режим роботи додатку. Також було розроблено інструкцію користувача, в якій описано вхідні дані, необхідні для коректної роботи програмного продукту та дії, що користувач повинен виконати при використанні продукту.

ВИСНОВКИ

У бакалаврській дипломній роботі розроблено веб-додаток для онлайн-консультацій пацієнтів з лікарями.

Виконано аналіз основних аналогів, виявлено їх переваги та недоліки. На основі отриманих результатів прийнято рішення про створення власного додатку, що вирішує проблеми, що присутні в аналогах.

Також в результаті проведення варіантного аналізу, обрано мову програмування JavaScript та середовище розробки – WebStorm.

Вирішено такі задачі:

- обґрунтувано вибір методу розробки і встановлено задачу;
- розроблено метод, структуру та алгоритми роботи веб-додатку для онлайн консультацій;
- розроблено основні модулі додатку;
- проведено тестування додатку.

Тестування додатку довело повну працездатність розробленого програмного продукту та відповідність поставленому технічному завданню.

Розроблено інструкцію користувача.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Що таке телемедицина. *Ehealth* : веб-сайт. URL: https://ehealth.gov.ua/2021/11/15/shho-take-telemedytsyna/ (дата звернення: 03.06.2022).
2. Заболотний Н.С. Розробка веб-додатку «ЗДОРОВ'Я» для онлайн-консультацій пацієнтів з лікарями. Матеріали науково-технічної конференції підрозділів Вінницького національного технічного університету (НТКП ВНТУ). Вінниця, 2022. *Conferences.vntu* : веб-сайт. URL: https://conferences.vntu.edu.ua/index.php/all-fitki/all-fitki-2022/paper/view/15576 (дата звернення: 03.06.2022).
3. Що таке телездоров'я. *Catalyst* : веб-сайт. URL: https://catalyst.nejm.org/doi/full/10.1056/CAT.18.0268 (дата звернення: 03.06.2022).
4. What is database. *Javatpoint* : веб-сайт. URL: https://www.javatpoint.com/what-is-database. (дата звернення: 04.06.2022).
5. Налаштування однорангової мережі. Instructables : веб-сайт. URL: https://www.instructables.com/Peer-to-Peer-Network-Sharing (дата звернення: 04.06.2022).
6. Основи WebRTC. Freeconference: веб-сайт. URL: https://www.freeconference.com/uk/blog/the-basics-what-you-need-to-know-about-webrtc/ (дата звернення: 04.06.2022).
7. Алгоритм та його властивості. Step: веб-сайт. URL: https://step.org.ua/konspekt/algoritm/tema1 (дата звернення: 04.06.2022).
8. Основи налаштування протоколу WebSocket. Javascript: веб-сайт. URL: https://javascript.info/websocket (дата звернення: 04.06.2022).
9. Що таке API. Marketer: веб-сайт. URL: https://marketer.ua/ua/shho-take-api/ (дата звернення: 05.06.2022).

10. Blokdyk G. Firebase The Ultimate Step-By-Step Guide: навч. посіб. Берлін : Emereo Pty Limited, 2020. 302 с.
11. Cadenhead T. Socket.io Cookbook: довідник. Нью-Йорк: Apress, 2015. 184 ст.
12. Johnson R. Design Patterns: Elements of Reusable Object-Oriented Software : навч. посіб. Лос-Анджелес: Addison-Wesley professional, 1994. 416 ст.
13. Графічний інтерфейс користувача. Wikiwand: веб-сайт. URL: https://www.wikiwand.com/uk/%D0%93%D1%80%D0%B0%D1%84%D1%96%D1%87%D0%BD%D0%B8%D0%B9_%D1%96%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D1%83%D0%B2%D0%B0%D1%87%D0%B0 (дата звернення: 07.06.2022).
14. Фленаган. Д. JavaScript. Повне керівництво. 7-е видання / Нью-Йорк – Диалектика-Вільямс, 2021. 722 ст.
15. Цікаві факти про Python. Itschool: веб-сайт. URL: http://www.itschool.vn.ua/interesting-python/ (дата звернення: 08.06.2022).
16. WebStorm. Wikipedia: веб-сайт. URL: https://uk.wikipedia.org/wiki/WebStorm (дата звернення: 08.06.2022).
17. Повне керівництво по тестування веб-додатків. URL: https://uk.myservername.com/web-application-testing-complete-guide (дата звернення: 09.06.2022).
18. What is white box testing. URL: https://www.guru99.com/white-box-testing.html (дата звернення: 10.06.2022).
19. Black Box Testing. URL: https://www.imperva.com/learn/application-security/black-box-testing (дата звернення: 10.06.2022).

ДОДАТКИ

Додаток А – Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ

д.т.н., проф.

_____О. Н. Романюк

25 березня 2022 р.

Технічне завдання

на бакалаврську дипломну роботу «Веб-додаток для онлайн-консультацій пацієнтів з лікарями»

за спеціальністю

121 – Інженерія програмного забезпечення

Керівник бакалаврської дипломної роботи:

_____ д.т.н., проф. Ліщинська Л.Б

"____"_____2022 р.

Виконав:

_____ ст. гр. 1ПІ-18б Н.С. Заболотний

"____"_____2022 р.

Вінниця – 2022 року

1. Найменування та галузь застосування

Бакалаврська дипломна робота: «Веб-додаток для онлайн-консультацій пацієнтів з лікарями».

Галузь застосування – медичні інтернет-портали, платформи для телемедицини.

2. Підстава для розробки.

Завдання на роботу, яке затверджене на засіданні кафедри програмного забезпечення – протокол № 12 від « 07 » лютого 2022 р.

3. Мета та призначення розробки.

Метою роботи є удосконалення існуючого метода однорангового зв'язку двох користувачів за допомогою технології WebRTC та створення власного безкоштовного додатку, що дозволить українським лікарям проводити онлайн-консультації зі своїми пацієнтами, використовувати сучасні технології.

Призначення роботи – кінцева реалізація веб-додатку з інтегрованими модулями відеозв'язку та текстових повідомлень, що може використовуватись лікарями для проведення онлайн-консультацій

4. Вихідні дані для проведення НДР

1. Blokdyk G. Firebase The Ultimate Step-By-Step Guide: навч. посіб. Берлін : Emereo Pty Limited, 2020. 302 с.
2. Cadenhead T. Socket.io Cookbook: довідник. Нью-Йорк: Apress, 2015. 184 ст.
3. Johnson R. Design Patterns: Elements of Reusable Object-Oriented Software : навч. посіб. Лос-Анджелес: Addison-Wesley professional, 1994. 416 ст.
4. Фленаган. Д. JavaScript. Повне керівництво. 7-е видання / Нью-Йорк – Диалектика-Вільямс, 2021. 722 ст.

5. Технічні вимоги

Вхідні дані – особисті дані користувача, текст повідомлень; зображення з відеокамери користувача; звук з пристроїв запису користувача.

6. Конструктивні вимоги.

Конструкція пристрою повинна відповідати естетичним та ергономічним вимогам, повинна бути зручною в обслуговуванні та керуванні.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до бакалаврської дипломної роботи;

– технічне завдання;

– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

Додаток Б – Лістинг програми

Chat.tsx

```
import { makeStyles } from '@mui/styles'
import { Avatar, Box, IconButton, Typography } from '@mui/material'
import React, { useEffect, useState } from 'react'
import SendIcon from '@mui/icons-material/Send'

export const Chat = ({ socket }: any) => {
  const classes = useStyles()
  const [message, setMessage] = useState('')
  const [messageList, setMessageList] = useState<{ author: string, text:
string }[]>([])

  useEffect(() => {
    socket.emit('get-messages')
    socket.on('update-messages', ({ messages }: { messages: { author: string,
text: string }[] }) => {
      setMessageList(messages)
    })
  }, [])

  const handleChangeMessage = (e: any) => {
    setMessage(e.target.value)
  }

  const handleSendMessage = (e: any) => {
    socket.emit('send-message', { author: 'Назарій Заболотнийііі', text:
message })
    setMessage('')
  }

  return (
    <Box className={classes.chatContainer}>
      <Box className={classes.chatHeader}>
        <Avatar />
        <Typography>Vasya Pupkin</Typography>
      </Box>
      <Box className={classes.messagesContainer}>
        {/* eslint-disable-next-line react/jsx-key */}
        {messageList.map(({ author, text }) => <Message author={author}
text={text} />)}
      </Box>
      <Box className={classes.chatControls}>
        <input
          className={classes.messageInput}
          type="text"
          placeholder="Введіть повідомлення..."
          value={message}
          onChange={handleChangeMessage}
        />
        <IconButton onClick={handleSendMessage}>
          <SendIcon style={{ color: '#2a9be2' }} />
        </IconButton>
      </Box>
    </Box>
  )
}

const Message = ({ author, text }: { author: string, text: string }) => {
```

```
  const classes = useMessageStyles()
  const user = 'Назарій Заболотний'
  const isMy = author === user

  return (
    <Box className={`${classes.message} ${isMy ? classes.myMessage :
classes.participantMessage}`}>
      {text}
    </Box>
  )
}

const useMessageStyles = makeStyles({
  message: {
    padding: 10,
    borderRadius: 16,
    maxWidth: '60%',
    margin: '8px',
  },
  myMessage: {
    alignSelf: 'flex-end',
    backgroundColor: '#2a9be2',
  },
  participantMessage: {
    alignSelf: 'flex-start',
    backgroundColor: 'rgba(0,0,0,0.11)',
  },
})

const useStyles = makeStyles({
  chatContainer: {
    display: 'flex',
    flexDirection: 'column',
    width: 600,
    height: 600,
    backgroundColor: '#80808012',
    boxSizing: 'border-box',
  },
  chatHeader: {
    display: 'flex',
    alignItems: 'center',
    gap: 8,
    borderBottom: '1px solid #2a9be2',
    padding: 16,
  },
  messagesContainer: {
    overflow: 'auto',
    flex: 1,
    padding: '8px 0',
    display: 'flex',
    flexDirection: 'column',
  },
  chatControls: {
    display: 'flex',
    justifyContent: 'space-between',
    alignItems: 'center',
    padding: 16,
  },
  messageInput: {
```

```
    padding: 10,
    borderRadius: 8,
    backgroundColor: 'transparent',
    width: '80%',
  },
})
```

peerConnectionSession.ts

```
import io from 'socket.io-client'

const { RTCPeerConnection, RTCSessionDescription } = window

function capitalizeFirstLetter(str: any) {
  return str.charAt(0).toUpperCase() + str.slice(1)
}

class PeerConnectionSession {
  _onConnected: any
  _onDisconnected: any
  _room: any
  peerConnections: any = {};
  senders: any[] = [];
  listeners: any = {};
  socket: any;

  constructor(socket: any) {
    this.socket = socket
    this.onCallMade()
  }

  addPeerConnection(id: any, stream: any, callback: (stream: any) => void) {
    this.peerConnections[id] = new RTCPeerConnection({
      iceServers: [{ urls: 'stun:stun.l.google.com:19302' }],
    })

    stream.getTracks().forEach((track: MediaStreamTrack) => {
      this.senders.push(this.peerConnections[id].addTrack(track, stream))
    })

    this.listeners[id] = (event: any) => {
      const fn = this[('_on' +
capitalizeFirstLetter(this.peerConnections[id].connectionState)) as keyof
PeerConnectionSession]
      fn && fn(event, id)
    }

    this.peerConnections[id].addEventListener('connectionstatechange',
this.listeners[id])

    this.peerConnections[id].ontrack = function ({ streams: [stream] }: {
streams: any }) {
      console.log({ id, stream })
      callback(stream)
    }

    console.log(this.peerConnections)
  }
```

```
  removePeerConnection(id: any) {
    this.peerConnections[id].removeEventListener('connectionstatechange',
this.listeners[id])
    delete this.peerConnections[id]
    delete this.listeners[id]
  }

  isAlreadyCalling = false;

  async callUser(to: any) {
    if (this.peerConnections[to].iceConnectionState === 'new') {
      const offer = await this.peerConnections[to].createOffer()
      await this.peerConnections[to].setLocalDescription(new
RTCSessionDescription(offer))

      this.socket.emit('call-user', { offer, to })
    }
  }

  onConnected(callback: () => void) {
    this._onConnected = callback
  }

  onDisconnected(callback: () => void) {
    this._onDisconnected = callback
  }

  joinRoom(room: any) {
    this._room = room
    this.socket.emit('joinRoom', room)
  }

  onCallMade() {
    this.socket.on('call-made', async (data: any) => {
      await this.peerConnections[data.socket].setRemoteDescription(new
RTCSessionDescription(data.offer))
      const answer = await this.peerConnections[data.socket].createAnswer()
      await this.peerConnections[data.socket].setLocalDescription(new
RTCSessionDescription(answer))

      this.socket.emit('make-answer', {
        answer,
        to: data.socket,
      })
    })
  }

  onAddUser(callback: (user: any) => void) {
    this.socket.on(`${this._room}-add-user`, async ({ user }: { user: any })
=> {
      callback(user)
    })
  }

  onRemoveUser(callback: (socketId: string) => void) {
    this.socket.on(`${this._room}-remove-user`, ({ socketId }: { socketId:
string }) => {
      callback(socketId)
    })
```

```
  }

  onUpdateUserList(callback: (users: any, current: any) => void) {
    this.socket.on(`${this._room}-update-user-list`, ({ users, current }: {
users: any, current: any }) => {
      callback(users, current)
    })
  }

  onAnswerMade(callback: (socket: any) => void) {
    this.socket.on('answer-made', async (data: any) => {
      await this.peerConnections[data.socket].setRemoteDescription(new
RTCSessionDescription(data.answer))
      callback(data.socket)
    })
  }

  clearConnections() {
    this.socket.close()
    this.senders = []

Object.keys(this.peerConnections).forEach(this.removePeerConnection.bind(this)
)
  }
}

export const createPeerConnectionContext = () => {
  const socket = io(`${process.env.REACT_APP_SOCKET_URL}`)

  return new PeerConnectionSession(socket)
}
```

Room.tsx

```
import React, { useRef, useState } from 'react'
import { useNavigate, useParams } from 'react-router-dom'
import { LocalVideo, RemoteVideo } from '../../components'
import { useCalculateVideoLayout, useCreateMediaStream, useStartPeerSession }
from '../../hooks'
import { ContentWrapper } from '../../ui/templates/content-wrapper'
import { Box, Collapse, IconButton } from '@mui/material'
import { makeStyles } from '@mui/styles'
import MicIcon from '@mui/icons-material/Mic'
import MicOffIcon from '@mui/icons-material/MicOff'
import VideocamIcon from '@mui/icons-material/Videocam'
import VideocaOffIcon from '@mui/icons-material/VideocamOff'
import ChatIcon from '@mui/icons-material/Chat'
import CallEndIcon from '@mui/icons-material/CallEnd'
import { paths } from '../../constants/paths'
import { UserTypes } from '../Login/type'
import { Chat } from '../../components/Chat'

export const Room = () => {
  const classes = useStyles()
  const { room } = useParams()
  const galleryRef = useRef<any>()
  const localVideoRef = useRef()
  const mainRef = useRef<any>()
  const [isVideoEnabled, setIsVideoEnabled] = useState(true)
```

```
  const [isAudioEnabled, setIsAudioEnabled] = useState(true)
  const [isChatOpen, setIsChatOpen] = useState(false)
  const navigate = useNavigate()

  const userMediaStream = useCreateMediaStream(localVideoRef)
  const { connectedUsers, peerVideoConnection } = useStartPeerSession(
    room,
    userMediaStream,
    localVideoRef,
    isVideoEnabled,
    isAudioEnabled,
  )

  useCalculateVideoLayout(galleryRef, connectedUsers.length + 1)

  const toggleVideo = () => setIsVideoEnabled((prev) => !prev)
  const toggleAudio = () => setIsAudioEnabled((prev) => !prev)
  const toggleChat = () => setIsChatOpen((prev) => !prev)

  const goToDashboard = () => navigate(paths.dashboard(UserTypes.DOCTOR))

  return (
    <ContentWrapper>
      <Box className={classes.remoteVideoContainer}>
        {connectedUsers.map((user) => (
          <RemoteVideo key={user} id={user} autoPlay playsInline
width="1000px" height="560px" />
        ))}
        <Box className={classes.controlsContainer}>
          <Box className={classes.controls}>
            <ControlButton
              isEnabled={isVideoEnabled}
              onClick={toggleVideo}
              EnabledIcon={VideocamIcon}
              DisabledIcon={VideocaOffIcon}
            />
            <ControlButton
              isEnabled={isAudioEnabled}
              onClick={toggleAudio}
              EnabledIcon={MicIcon}
              DisabledIcon={MicOffIcon}
            />
            <ControlButton EnabledIcon={ChatIcon} onClick={toggleChat} />
            <IconButton className={classes.endCallButton}
onClick={goToDashboard}>
              <CallEndIcon />
            </IconButton>
          </Box>
        </Box>
        <Box className={classes.localVideoContainer}>
          <LocalVideo ref={localVideoRef} autoPlay playsInline muted
width="200px" height="110px" />
        </Box>
      </Box>
      <Collapse in={isChatOpen}>
        <Chat socket={peerVideoConnection.socket} />
      </Collapse>
    </ContentWrapper>
  )
```

```
}

const ControlButton = ({ onClick, EnabledIcon, DisabledIcon, isEnabled = true
}: any ) => {
  const classes = useStyles()

  return (
    <IconButton className={isEnabled ? classes.defaultControlButton :
classes.endCallButton} onClick={onClick}>
      {isEnabled ? <EnabledIcon /> : <DisabledIcon />}
    </IconButton>
  )
}

const useStyles = makeStyles({
  remoteVideoContainer: {
    width: '1000px',
    height: '560px',
    position: 'relative',
  },
  localVideoContainer: {
    width: '200px',
    height: '110px',
    position: 'absolute',
    bottom: 10,
    right: 10,
  },
  controlsContainer: {
    width: '100%',
    height: 80,
    backgroundColor: 'rgba(0,0,0,0.57)',
    display: 'flex',
    justifyContent: 'center',
    alignItems: 'center',
    position: 'absolute',
    bottom: 0,
  },
  controls: {
    display: 'flex',
    gap: 24,
    width: 'max-content',
  },
  defaultControlButton: {
    '&:hover': {
      backgroundColor: '#2aace2',
    },
    backgroundColor: '#2a9be2',
    color: 'white',
    width: 50,
    height: 50,
  },
  endCallButton: {
    '&:hover': {
      backgroundColor: '#e53a36',
    },
    backgroundColor: '#dd2a26',
    color: 'white',
    width: 50,
    height: 50,
```

```
  },
})
```

Dashboard.tsx

```
import React, { useEffect, useLayoutEffect, useState } from 'react'
import { Box, Button, Paper, Typography } from '@mui/material'
import { makeStyles } from '@mui/styles'
import { ContentWrapper } from '../../ui/templates/content-wrapper'
import { useDispatch } from 'react-redux'
import { UserTypes } from '../Login/type'
import { useNavigate, useParams } from 'react-router-dom'
import moment from 'moment'
import { paths } from '../../constants/paths'

export const DashboardPage = () => {
  const classes = useStyles()
  const [remainingTime, setRemainingTime] = useState(38)

  const navigate = useNavigate()
  const dispatch = useDispatch()
  const { userType } = useParams<{userType: UserTypes}>()

  const users = {
    [UserTypes.DOCTOR]: 'Назарій Заболотний',
    [UserTypes.PATIENT]: 'Дмитро Загоружко',
  }

  useEffect(() => {
    const timeReducer = setInterval(() => {
      setRemainingTime((prev) => prev - 1)
    }, 60000) // 1 minute

    return () => {
      clearInterval(timeReducer)
    }
  }, [])

  const goToTheRoom = () => {
    navigate(paths.room('ee31dgvff32'))
  }

  const renderUserContent = () => (
    <Box className={classes.rowContainer} marginTop="8px">
      <Typography>
        {userType === UserTypes.DOCTOR ? 'Пацієнт:':'Лікар:'}
      </Typography>
      <Typography>
        {userType === UserTypes.DOCTOR ? users[UserTypes.PATIENT]:
users[UserTypes.DOCTOR]}
      </Typography>
      <Button color={'primary'} variant="outlined" onClick={goToTheRoom}>
        {userType === UserTypes.DOCTOR ? 'Створити кімнату':'Приєднатись до
кімнати'}
      </Button>
    </Box>
  )
```

```
  return (
    <ContentWrapper>
      <Paper elevation={4} className={classes.paper}>
        <Box className={classes.rowContainer}>
          <Typography>
            Сьогодні:
          </Typography>
          {moment().format('DD.MM.YYYY')}
        </Box>
        <Typography className={classes.sectionTitle}>Наступна
консультація</Typography>
        <Box className={classes.rowContainer}>
          <Typography>
            До наступної консультації залишилось:
          </Typography>
          <Typography>
            {remainingTime} хвилин
          </Typography>
        </Box>
        {renderUserContent()}
      </Paper>
    </ContentWrapper>
  )
}

const useStyles = makeStyles({
  paper: {
    padding: 20,
    boxSizing: 'border-box',
    borderRadius: 16,
    width: '100%',
    maxWidth: '600px',
    height: '100%',
    maxHeight: '500px',
    justifyItems: 'center',
  },
  rowContainer: {
    display: 'flex',
    alignItems: 'center',
    gap: 16,
  },
  sectionTitle: {
    fontWeight: 600,
    margin: '16px 0 8px',
    fontSize: 18,
  },
})
```

useStartPeerSession.ts

```
import { useEffect, useMemo, useState } from 'react'
import { createPeerConnectionContext } from '../utils/PeerConnectionSession'

export const useStartPeerSession = (room: any, userMediaStream: any,
localVideoRef: any, isVideoEnabled: boolean, isAudioEnabled: boolean) => {
  const peerVideoConnection = useMemo(() => createPeerConnectionContext(), [])

  const [displayMediaStream, setDisplayMediaStream] = useState<any>(null)
  const [connectedUsers, setConnectedUsers] = useState<any[]>([])
```

```
  useEffect(() => {
    if (userMediaStream) {
      peerVideoConnection.joinRoom(room)
      peerVideoConnection.onAddUser((user) => {
        setConnectedUsers((users: any) => [...users, user])
        peerVideoConnection.addPeerConnection(`${user}`, userMediaStream,
(_stream) => {
          const userContainer = document.getElementById(user) as any
          if(!userContainer) return
          userContainer.srcObject = _stream
        })
        peerVideoConnection.callUser(user)
      })

      peerVideoConnection.onRemoveUser((socketId) => {
        setConnectedUsers((users) => users.filter((user) => user !==
socketId))
        peerVideoConnection.removePeerConnection(socketId)
      })

      peerVideoConnection.onUpdateUserList(async (users) => {
        setConnectedUsers(users)
        for (const user of users) {
          peerVideoConnection.addPeerConnection(`${user}`, userMediaStream,
(_stream) => {
            const userContainer = document.getElementById(user) as any
            if(!userContainer) return
            userContainer.srcObject = _stream
          })
        }
      })

      peerVideoConnection.onAnswerMade((socket) =>
peerVideoConnection.callUser(socket))
    }

    return () => {
      if (userMediaStream) {
        peerVideoConnection.clearConnections()
        userMediaStream?.getTracks()?.forEach((track: MediaStreamTrack) =>
track.stop())
      }
    }
  }, [peerVideoConnection, room, userMediaStream])

  const cancelScreenSharing = async () => {
    const senders = await peerVideoConnection.senders.filter((sender) =>
sender.track.kind === 'video')

    if (senders) {
      senders.forEach((sender) =>
        sender.replaceTrack(userMediaStream.getTracks().find((track:
MediaStreamTrack) => track.kind === 'video')),
      )
    }

    localVideoRef.current.srcObject = userMediaStream
    displayMediaStream?.getTracks().forEach((track: MediaStreamTrack) =>
```

```
track.stop())
    setDisplayMediaStream(null)
  }

  const shareScreen = async () => {
    const stream = displayMediaStream || (await
navigator.mediaDevices.getDisplayMedia())

    const senders = await peerVideoConnection.senders.filter((sender) =>
sender.track.kind === 'video')

    if (senders) {
      senders.forEach((sender) => sender.replaceTrack(stream.getTracks()[0]))
    }

    stream.getVideoTracks()[0].addEventListener('ended', () => {
      cancelScreenSharing()
    })

    localVideoRef.current.srcObject = stream

    setDisplayMediaStream(stream)
  }

  const initMedia = async () => {
    const senders = await peerVideoConnection.senders
    const audios = senders.filter((sender, index) => sender?.track?.kind ===
'audio')
    const videos = senders.filter((sender, index) => sender?.track?.kind ===
'video')

    audios.forEach((audio) => audio.track.enabled = isAudioEnabled)
    videos.forEach((video) => video.track.enabled = isVideoEnabled)
  }

  initMedia()

  return {
    connectedUsers,
    peerVideoConnection,
    shareScreen,
    cancelScreenSharing,
    isScreenShared: !!displayMediaStream,
  }
}
```

useCalculateVoiceVolume.ts

```
import { useEffect } from 'react'

export const useCalculateVoiceVolume = (stream: any, id: any) => {
  useEffect(() => {
    if (!stream) return
    const audioCtx = new (window.AudioContext || window?.webkitAudioContext)()
    const analyser = audioCtx.createAnalyser()
    const biquadFilter = audioCtx.createBiquadFilter()
    const gainNode = audioCtx.createGain()
    const distortion = audioCtx.createWaveShaper()
```

```
    let drawVisual: any
    analyser.minDecibels = -90
    analyser.maxDecibels = -10
    analyser.smoothingTimeConstant = 0.85

    const canvas = document.getElementById(`canvas-${id}`) as
HTMLCanvasElement

    const canvasCtx = canvas?.getContext('2d')

    try {
      const source = audioCtx.createMediaStreamSource(stream)
      source.connect(distortion)
      distortion.connect(biquadFilter)
      biquadFilter.connect(gainNode)
      gainNode.connect(analyser)
      analyser.connect(audioCtx.destination)

      distortion.oversample = '4x'
      biquadFilter.gain.setTargetAtTime(0, audioCtx.currentTime, 0)
      const visualize = () => {
        const WIDTH = canvas.width
        const HEIGHT = canvas.height

        analyser.fftSize = 2048
        const bufferLength = analyser.fftSize

        const dataArray = new Uint8Array(bufferLength)

        if(!canvasCtx) return

        canvasCtx.clearRect(0, 0, WIDTH, HEIGHT)

        const draw = function () {
          canvasCtx.clearRect(0, 0, WIDTH, HEIGHT)

          drawVisual = requestAnimationFrame(draw)

          analyser.getByteTimeDomainData(dataArray)

          canvasCtx.fillStyle = 'transparent'
          canvasCtx.fillRect(0, 0, WIDTH, HEIGHT)

          canvasCtx.lineWidth = 1.5
          canvasCtx.strokeStyle = 'green'

          canvasCtx.beginPath()

          const sliceWidth = (WIDTH * 1.0) / bufferLength
          let x = 0

          for (let i = 0; i < bufferLength; i++) {
            const v = dataArray[i] / 128.0
            const y = (v * HEIGHT) / 2

            if (i === 0) {
              canvasCtx.moveTo(x, y)
            } else {
```

```
              canvasCtx.lineTo(x, y)
            }

            x += sliceWidth
          }

          canvasCtx.lineTo(canvas.width, canvas.height / 2)
          canvasCtx.stroke()
        }

        draw()
      }

      visualize()
    } catch (err) {
      console.log(err)
    }

    return () => {
      cancelAnimationFrame(drawVisual)
    }
  }, [stream, id])
}
```

SignUp.tsx

```
import React, { useState } from 'react'
import { Box, Checkbox, FormControl, FormControlLabel, Paper, Radio,
RadioGroup, Typography } from '@mui/material'
import { makeStyles } from '@mui/styles'
import { ContentWrapper } from '../../ui/templates/content-wrapper'
import { useFormik } from 'formik'
import { SignupForm } from './type'
import { FormField, Button } from '../../components/mui'
import { paths } from '../../constants/paths'
import { useNavigate } from 'react-router-dom'
import { validationSchema } from './validationSchema'
import moment from 'moment'


export const SignupPage = () => {
  const classes = useStyles()
  const navigate = useNavigate()

  const onSubmit = (values: SignupForm) => {
    console.log(values)
  }

  const formik = useFormik<SignupForm>({
    initialValues: {
      email: '',
      password: '',
      rePassword: '',
      firstName: '',
      lastName: '',
      birthDate: moment().subtract(18, 'years').toDate(),
      isPatient: true,
    },
```

```
    validationSchema,
    onSubmit,
  })

  const setIsPatient = (e: any, value: string) => {
    formik.setFieldValue('isPatient', JSON.parse(value))
  }

  const goToLogin = () => navigate(paths.login())

  return (
    <ContentWrapper>
      <Paper elevation={4} className={classes.paper}>
        <FormField
          label="Ім’я"
          formik={formik}
          name="firstName"
        />
        <FormField
          label="Прізвище"
          formik={formik}
          name="lastName"
        />
        <FormField
          label="Дата Народження"
          formik={formik}
          name="birthDate"
          type="date"
          InputLabelProps={{ shrink: true }}
        />
        <FormField
          label="Електронна адреса"
          formik={formik}
          name="email"
        />
        <FormField
          label="Пароль"
          formik={formik}
          name="password"
          type={'password'}
        />
        <FormField
          label="Повторіть пароль"
          formik={formik}
          name="rePassword"
          type={'password'}
          noHelperText
        />
        <RadioGroup
          value={formik.values.isPatient}
          onChange={setIsPatient}
        >
          <FormControlLabel value={true} control={<Radio />} label="Я пацієнт"
/>
          <FormControlLabel value={false} control={<Radio />} label="Я лікар"
/>
        </RadioGroup>
        <Button disabled={!formik.isValid} variant={'contained'} style={{
marginTop: 20 }} type="submit">Зареєструватись</Button>
```

```
      </Paper>
      <Box display="flex" gap='5px' alignItems="center" marginTop="20px">
        <Typography>Вже зареєстровані?</Typography>
        <Button onClick={goToLogin}>Увійти</Button>
      </Box>
    </ContentWrapper>
  )
}

const useStyles = makeStyles({
  paper: {
    padding: 20,
    boxSizing: 'border-box',
    borderRadius: 16,
    display: 'grid',
    gridTemplateColumns: '1fr',
    gap: 15,
    width: '100%',
    maxWidth: '600px',
    height: '100%',
    // maxHeight: '500px',
    justifyItems: 'center',
  },
})
```

ChatPage.tsx

```
import React, { Component } from 'react'
import events from '../../events'
import { Grid } from 'semantic-ui-react'
import Sidebar from './Sidebar';
import MessageHeader from './MessageHeader'
import MessagesBody from './MessagesBody'
import MessageInput from './MessageInput'


export class ChatPage extends Component {

  state = {
    chats: [],
    activeChannel: null
  }

  componentDidMount(){
    let { socket } = this.props
    socket.emit(events.INIT_CHATS, this.initChats )
    socket.on( events.MESSAGE_SEND, this.addMessage )
    socket.on( events.TYPING, this.addTyping )
    socket.on( events.P_MESSAGE_SEND, this.addPMessage )
    socket.on( events.P_TYPING, this.addPTyping)
    socket.on( events.CREATE_CHANNEL, this.updateChats )
  }

  initChats =  _chats  => this.updateChats( _chats, true )

  updateChats = ( _chats, init=false ) => {
    let { chats } = this.state
```

```
    let newChats = init ? [ ..._chats ] : [ ...chats, _chats ]
    this.setState({ chats: newChats, activeChannel: init ? _chats[0] :
this.state.activeChannel })
  }

  addTyping = ({ channel, isTyping, sender }) => {
    let { user } = this.props
    let { chats } = this.state
    if( sender === user.nickname ) return
    chats.map( chat => {
      if( chat.name === channel ){
        if( isTyping && !chat.typingUser.includes( sender )){
          chat.typingUser.push( sender )
        } else if( !isTyping && chat.typingUser.includes( sender )){
          chat.typingUser = chat.typingUser.filter( u => u !== sender )
        }
      }
      return null
    })
    this.setState({ chats })
  }

  addPTyping = ({ channel, isTyping }) => {
    console.log( channel, isTyping )
    let { pChats } = this.props
    pChats.map( pChat => {
      if( pChat.name === channel ){
        pChat.isTyping = isTyping
      }
      return null
    })
    this.setState({ pChats })
  }

  addMessage = ({ channel, message }) => {
    let { activeChannel, chats } = this.state

    chats.map( chat => {
      if( chat.name === channel ) {
        chat.messages.push( message )
        if ( activeChannel.name !== channel ) chat.msgCount ++
      }
      return null
    })
    this.setState({ chats })
  }

  addPMessage = ({ channel, message }) => {
    let { activeChannel } = this.state
    let { pChats } = this.props

    pChats.map( pChat => {
      if( pChat.name === channel ) {
        pChat.messages.push( message )
        if( activeChannel.name !== channel ) pChat.msgCount ++
      }
      return null
    })
    this.setState({ pChats })
```

```
  }

  sendMsg = msg => {
    let { socket, users  } = this.props
    let { activeChannel } = this.state
    if( activeChannel.type ) {
      let receiver = users[ activeChannel.name ]
      socket.emit( events.P_MESSAGE_SEND, { receiver, msg })
    } else {
      socket.emit( events.MESSAGE_SEND, { channel: activeChannel.name, msg })
    }

  }

  sendTyping = isTyping => {
    let { socket, users } = this.props
    let { activeChannel } = this.state
    if( activeChannel.type ){
      let receiver = users[ activeChannel.name ]
      socket.emit( events.P_TYPING, { receiver: receiver.socketId, isTyping })
    }
    socket.emit( events.TYPING, { channel: activeChannel.name, isTyping })
  }

  setActiveChannel = name => {
    let newActive = this.state.chats.filter( chat => chat.name === name )
    newActive[0].msgCount = 0
    this.setState({ activeChannel: newActive[0] })
  }

  setActivePChannel = name => {
    let newActive = this.props.pChats.filter( pChat => pChat.name === name )
    newActive[0].msgCount = 0
    this.setState({ activeChannel: newActive[0] })
  }

  render() {
    let { user, users, pChats, logout, socket } = this.props
    let { activeChannel, chats } = this.state
    return (
        <Grid style={{ height: '100vh', margin: '0px'}}>
    <Grid.Column computer={4} tablet={ 4 } mobile={6} style={{ background:
'#4c3c4c', height: '100%'}}>
    <Sidebar
        user = { user }
    users = { users }
    chats = { chats }
    socket = { socket }
    activeChannel = { activeChannel }
    logout = { logout }
    setActivePChannel = { this.setActivePChannel }
    setActiveChannel = { this.setActiveChannel }
    pChats = { pChats }
    />
    </Grid.Column>
    <Grid.Column computer={12} tablet={ 12 } mobile={10} style={{ background:
'#eee', height: '100%'}}>
    {
      activeChannel && (
```

```
          <React.Fragment>
              <MessageHeader activeChannel= { activeChannel } />
    <MessagesBody
      messages = { activeChannel.messages }
      user={ user }
      typingUser = { activeChannel.typingUser } />
    <MessageInput
      sendMsg = { this.sendMsg }
      sendTyping = { this.sendTyping } />
    </React.Fragment>
    )
    }
    </Grid.Column>
    </Grid>
  )
  }
}

export default ChatPage
```

message.gateway.ts

```
import {
  OnGatewayDisconnect,
  OnGatewayInit,
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer,
} from '@nestjs/websockets';
import { Socket } from 'socket.io';

import { Server } from 'ws';
import { Logger } from '@nestjs/common';

@WebSocketGateway({ namespace: 'chat' })
export class MessageGateway implements OnGatewayInit, OnGatewayDisconnect {
  @WebSocketServer() server: Server;

  private activeSockets: { room: string; id: string }[] = [];
  private messages: { author: string; text: string }[] = [];

  private logger: Logger = new Logger('MessageGateway');

  @SubscribeMessage('joinRoom')
  public joinRoom(client: Socket, room: string): void {
    /*
    client.join(room);
    client.emit('joinedRoom', room);
    */

    const existingSocket = this.activeSockets?.find(
      (socket) => socket.room === room && socket.id === client.id,
    );

    if (!existingSocket) {
      this.activeSockets = [...this.activeSockets, { id: client.id, room }];
      client.emit(`${room}-update-user-list`, {
        users: this.activeSockets
          .filter((socket) => socket.room === room && socket.id !== client.id)
```

```
        .map((existingSocket) => existingSocket.id),
      current: client.id,
    });

    client.broadcast.emit(`${room}-add-user`, {
      user: client.id,
    });
  }

  return this.logger.log(`Client ${client.id} joined ${room}`);
}

@SubscribeMessage('call-user')
public callUser(client: Socket, data: any): void {
  client.to(data.to).emit('call-made', {
    offer: data.offer,
    socket: client.id,
  });
}

@SubscribeMessage('get-messages')
public getMessages(client: Socket): void {
  client.emit('update-messages', {
    messages: this.messages,
  });
}

@SubscribeMessage('send-message')
public sendMessage(
  client: Socket,
  message: { author: string; text: string },
): void {
  this.messages.push(message);
  client.emit('update-messages', {
    messages: this.messages,
  });
}

@SubscribeMessage('make-answer')
public makeAnswer(client: Socket, data: any): void {
  client.to(data.to).emit('answer-made', {
    socket: client.id,
    answer: data.answer,
  });
}

@SubscribeMessage('reject-call')
public rejectCall(client: Socket, data: any): void {
  client.to(data.from).emit('call-rejected', {
    socket: client.id,
  });
}

public afterInit(server: Server): void {
  this.logger.log('Init');
}

public handleDisconnect(client: Socket): void {
  const existingSocket = this.activeSockets.find(
```

```
      (socket) => socket.id === client.id,
    );

    if (!existingSocket) return;

    this.activeSockets = this.activeSockets.filter(
      (socket) => socket.id !== client.id,
    );

    client.broadcast.emit(`${existingSocket.room}-remove-user`, {
      socketId: client.id,
    });

    this.logger.log(`Client disconnected: ${client.id}`);
  }
}
```

main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { NestExpressApplication } from '@nestjs/platform-express';
import { IoAdapter } from '@nestjs/platform-socket.io';
import * as redisIoAdapter from 'socket.io-redis';
import * as cors from 'cors';
import * as helmet from 'helmet';

export class RedisIoAdapter extends IoAdapter {
  createIOServer(port: number): any {
    const server = super.createIOServer(port);
    const redisAdapter = redisIoAdapter({
      host: process.env.REDIS_HOST,
      port: process.env.REDIS_PORT,
      auth_pass: process.env.REDIS_PASSWORD,
    });

    redisAdapter.prototype.on('error', function (err) {
      console.error('adapter error: ', err);
    });

    server.adapter(redisAdapter);
    return server;
  }
}

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule, {
    cors: true,
  });

  const whitelist = ['https://react-socket-io-webrtc-client.herokuapp.com/'];

  const corsOptionsDelegate = function (req, callback) {
    let corsOptions;
    if (whitelist.indexOf(req.header('Origin')) !== -1) {
      corsOptions = { origin: true }; // reflect (enable) the requested origin
in the CORS response
    } else {
```

```
      corsOptions = { origin: false }; // disable CORS for this request
    }
    callback(null, corsOptions); // callback expects two parameters: error and
options
  };

  app.use(cors(corsOptionsDelegate));
  app.use(helmet());
  app.useWebSocketAdapter(new RedisIoAdapter(app));

  const PORT = `${process.env.PORT}`;
  await app.startAllMicroservices();
  await app.listen(PORT || 3000);
}

bootstrap();
```

Додаток В – Протокол перевірки роботи на плагіат

ПРОТОКОЛ
ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ
НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Веб-додаток для онлайн-консультацій пацієнтів з лікарями

Тип роботи: БДР

Підрозділ : кафедра програмного забезпечення, ФІТКІ

Науковий керівник: Ліщинська Л.Б.

Оригінальність	89.9
Схожість	10.1

Аналіз звіту подібності

■ **Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.**

 Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.

 Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Особа, відповідальна за перевірку _____ Черноволик Г. О.

Ознайомлені з повним звітом подібності, який був згенерований системою Unicheck

Автор роботи _____ Заболотний Н.С.

Керівник роботи _____ Ліщинська Л.Б.

Додаток Г – Графічна частина

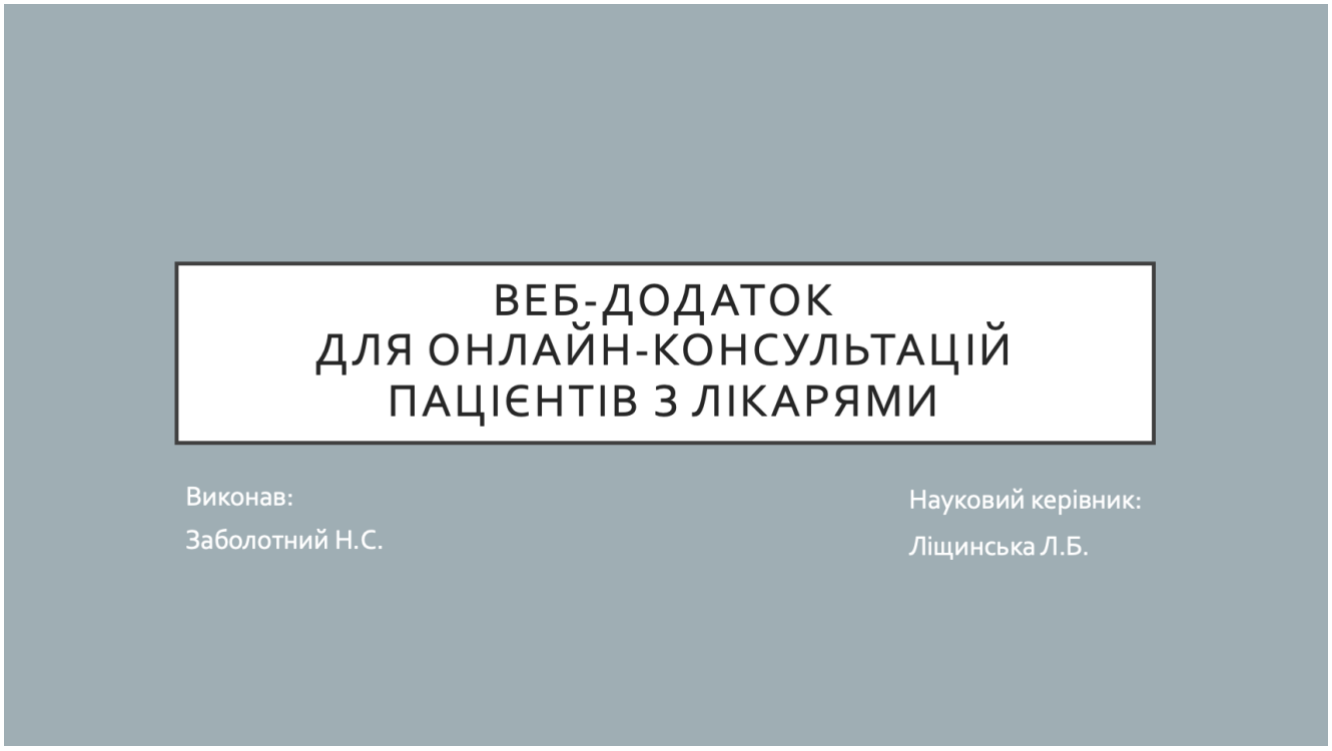


Рисунок Г.1 – Назва роботи

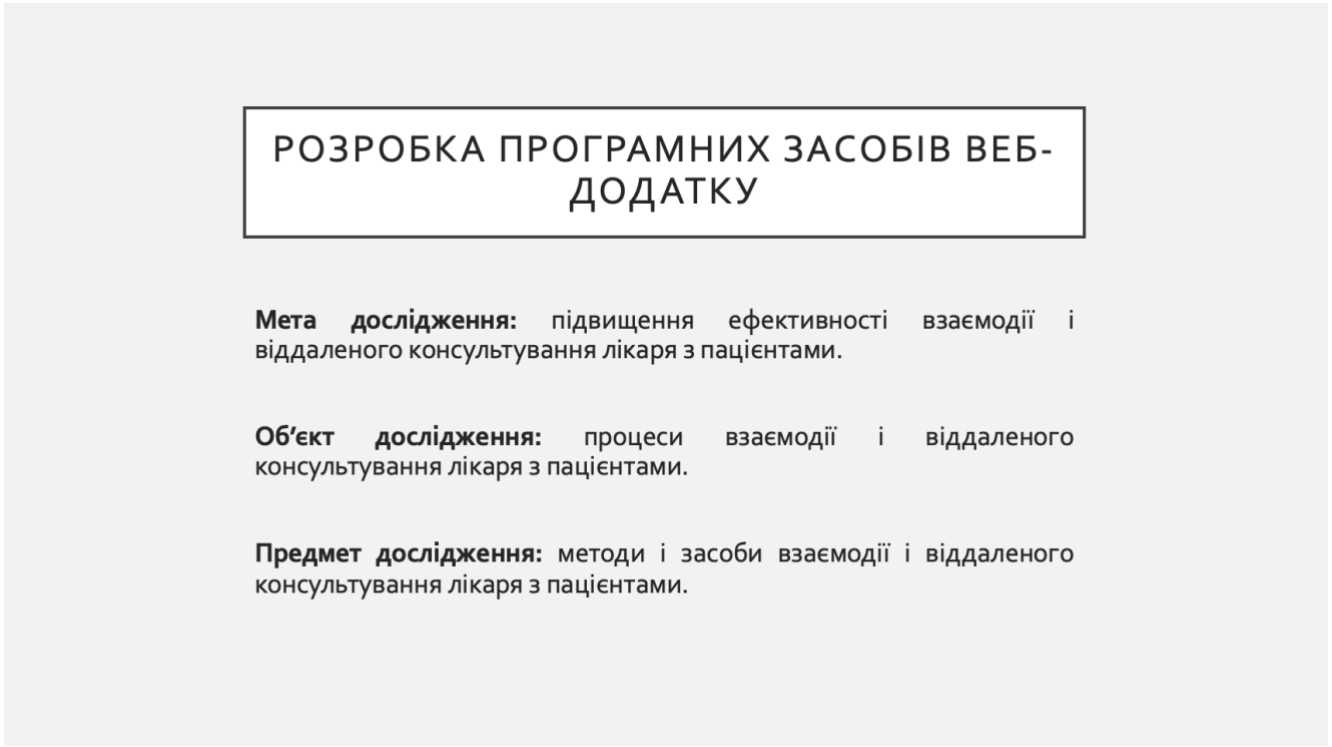


Рисунок Г.2 – Мета, об’єкт і предмет дослідження

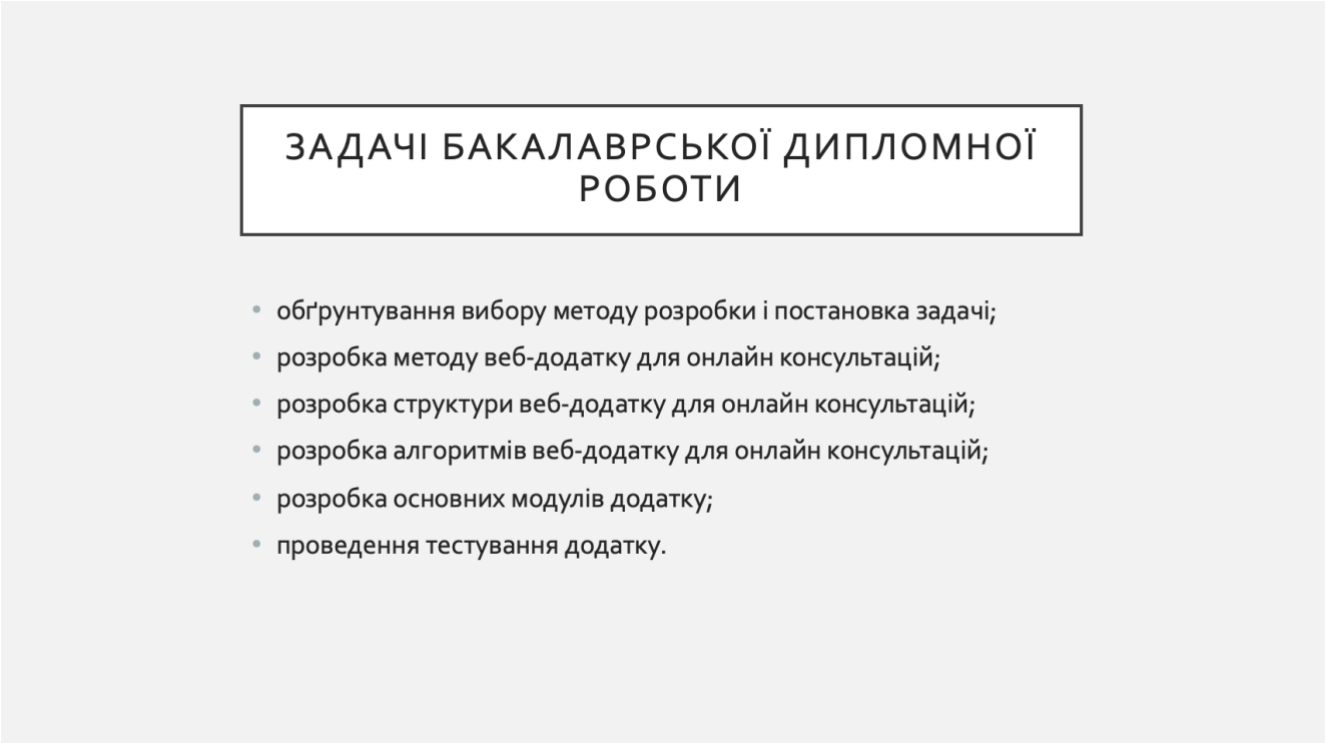


Рисунок Г.3 – Задачі БДР

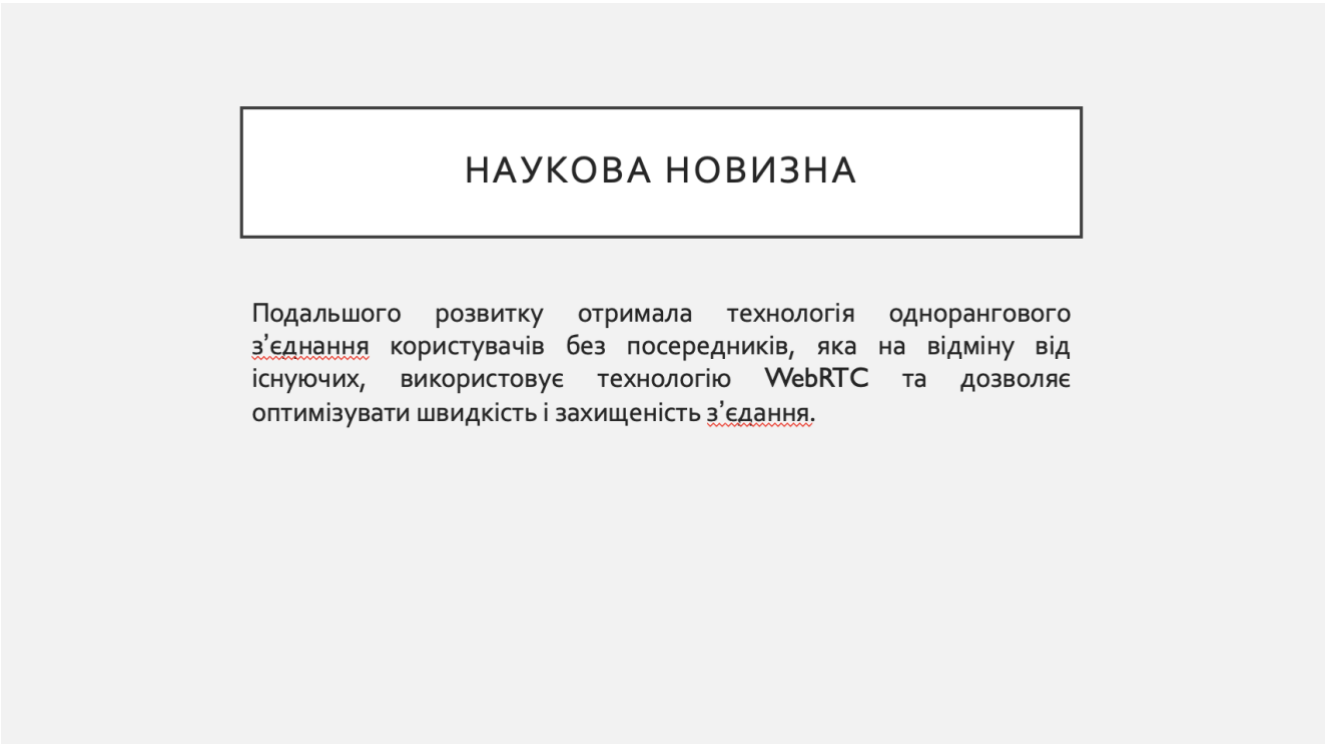


Рисунок Г.4 – Наукова новизна

Рисунок Г.5 – Практична цінність

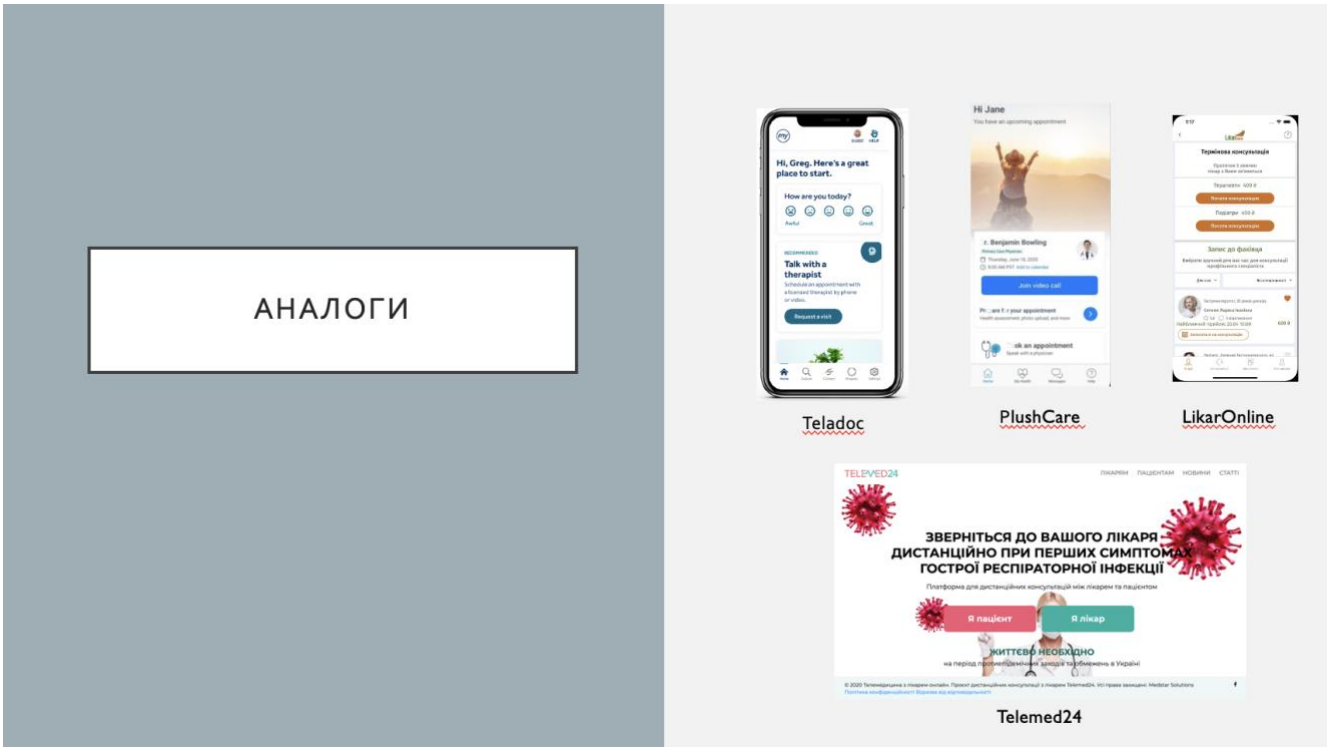


Рисунок Г.6 – Аналоги

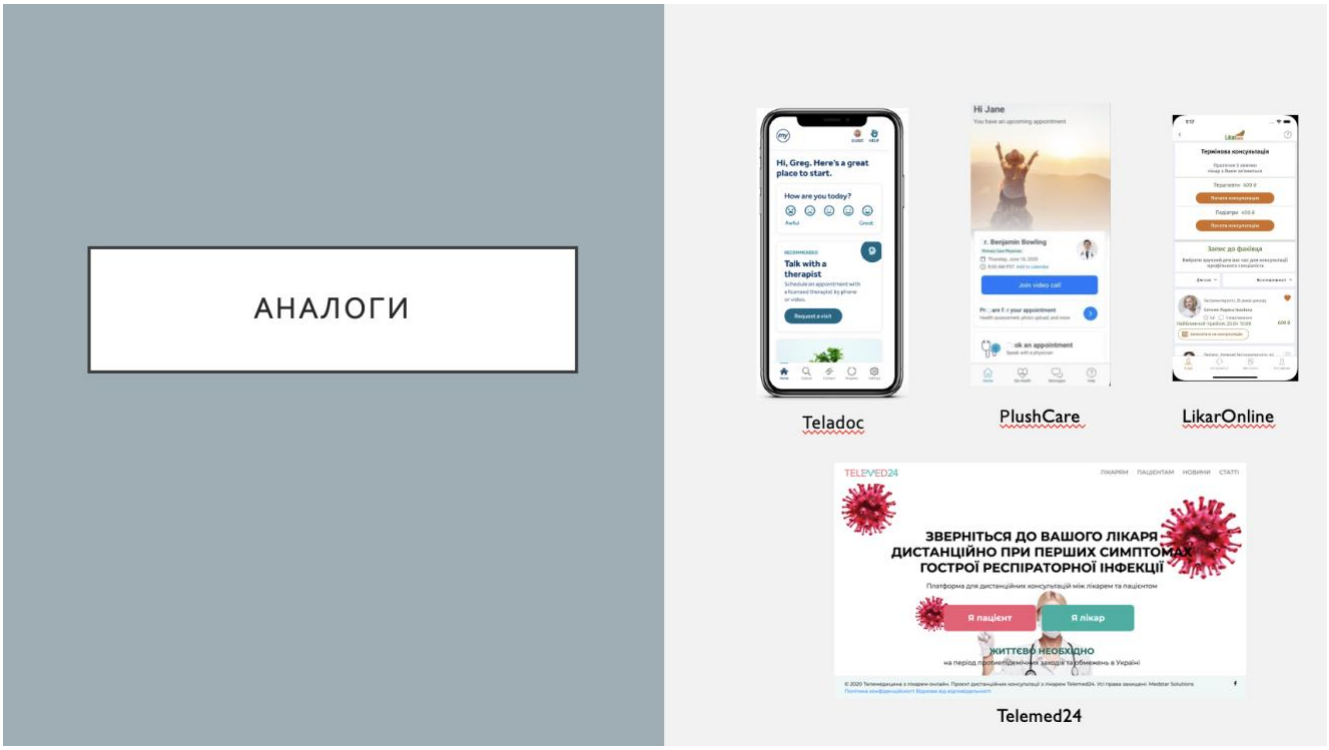


Рисунок Г.7 – Порівняльний аналіз аналогів

Рисунок Г.8 – Блок-схема алгоритму загальної роботи модуля відеозв'язку

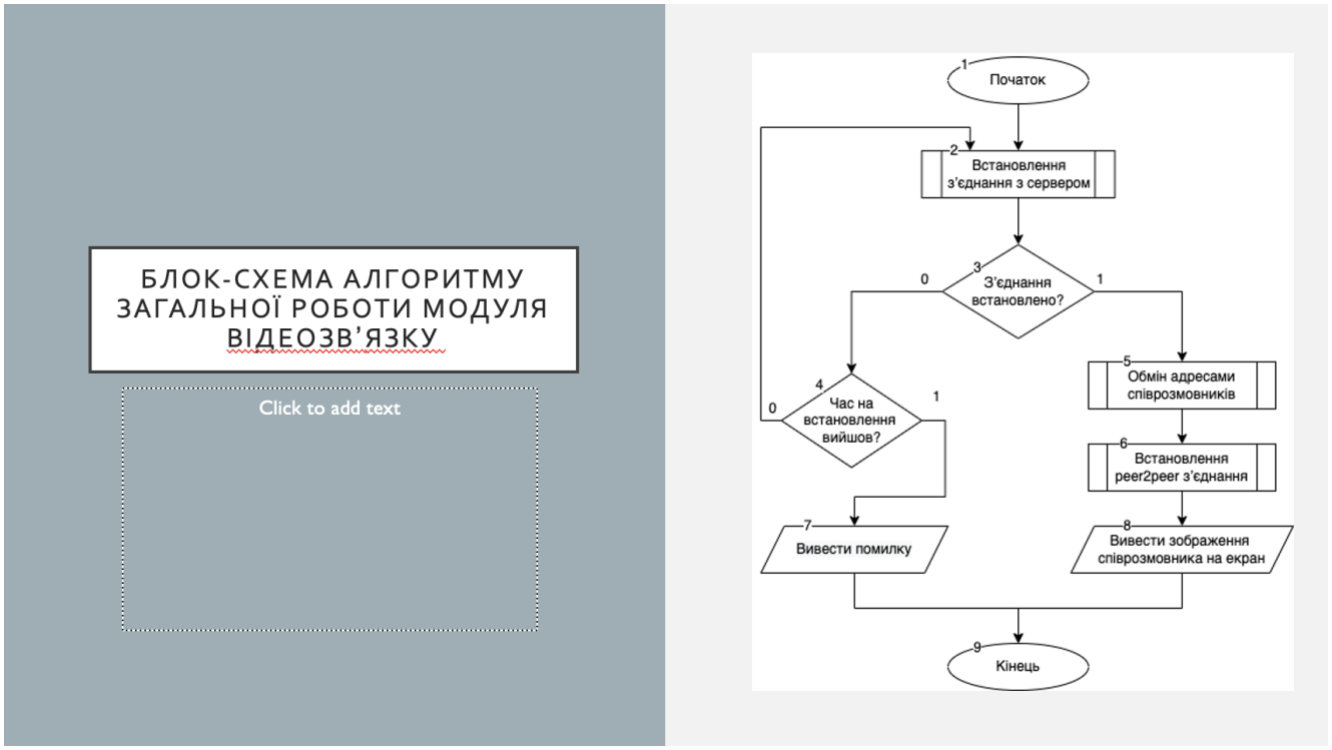


Рисунок Г.9 – Блок-схема алгоритму загальної роботи модуля текстових повідомлень

Рисунок Г.10 – Тестування веб-додатку

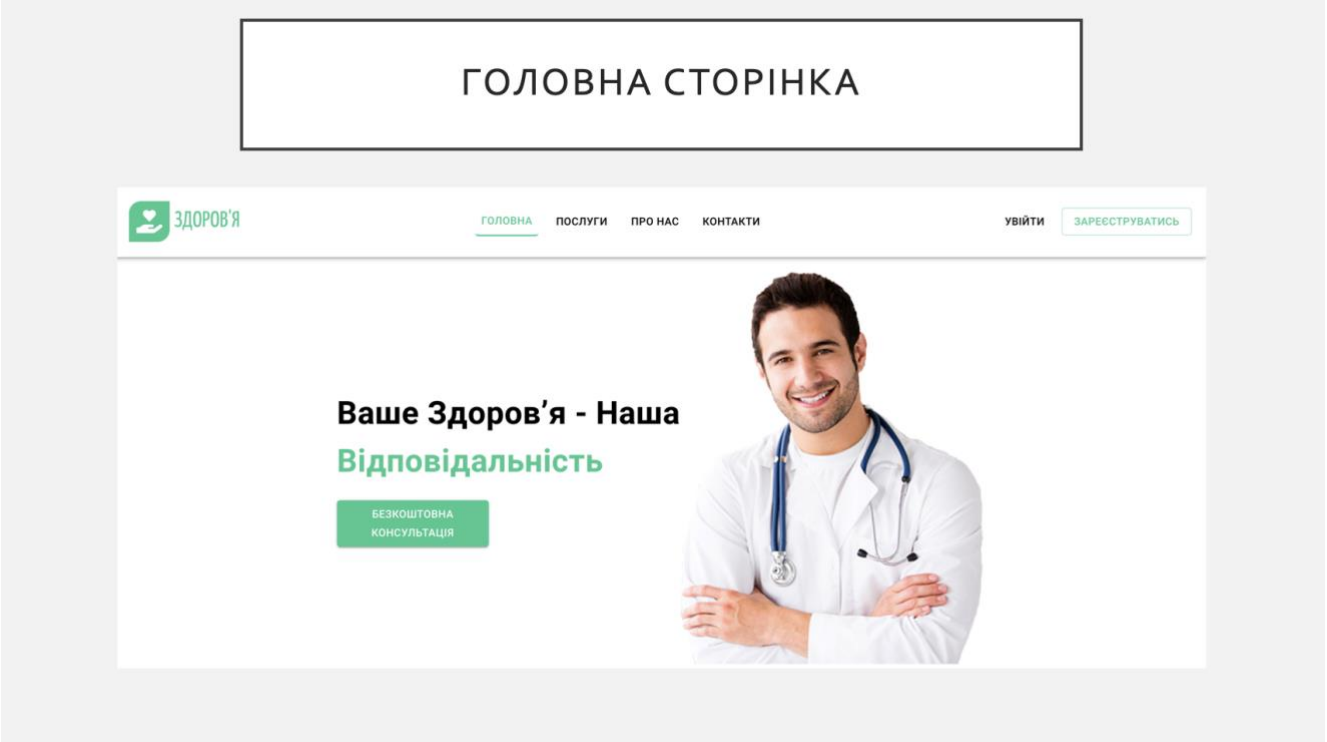


Рисунок Г.11 – Головна сторінка

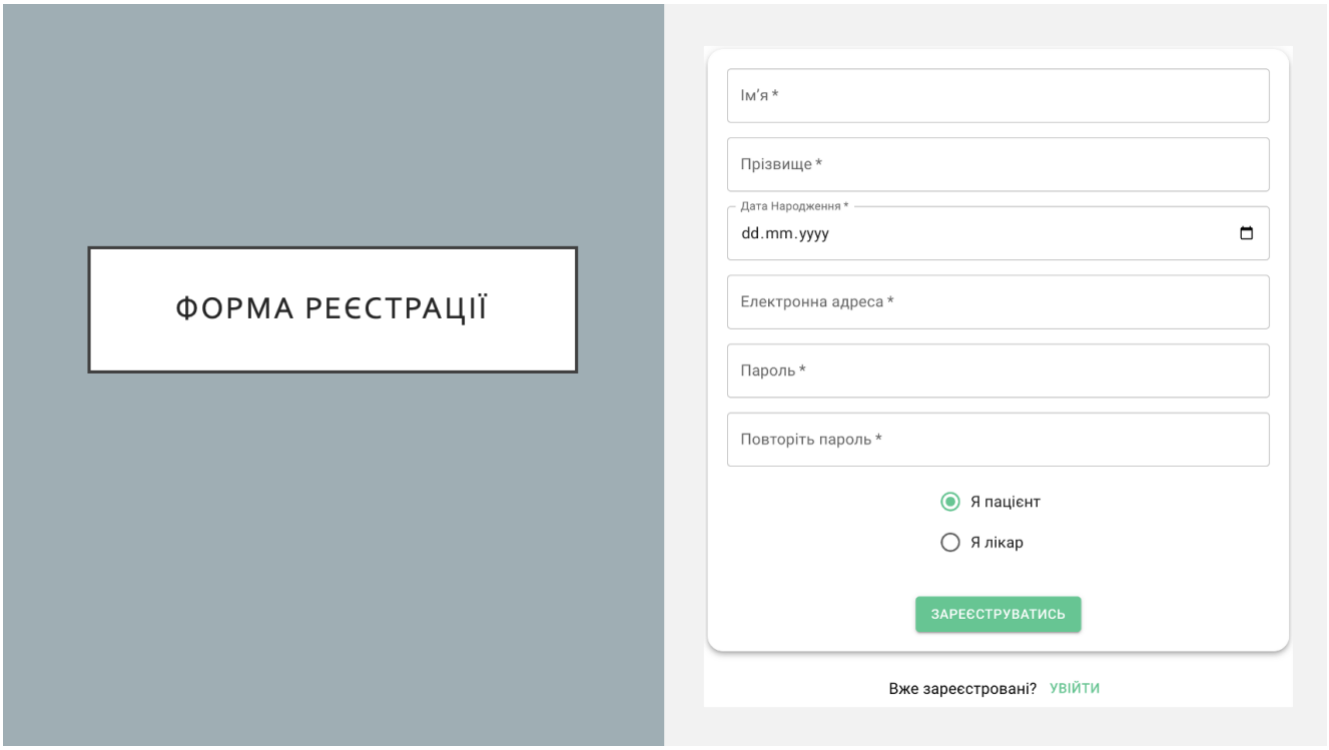


Рисунок Г.12 – Форма реєстрації

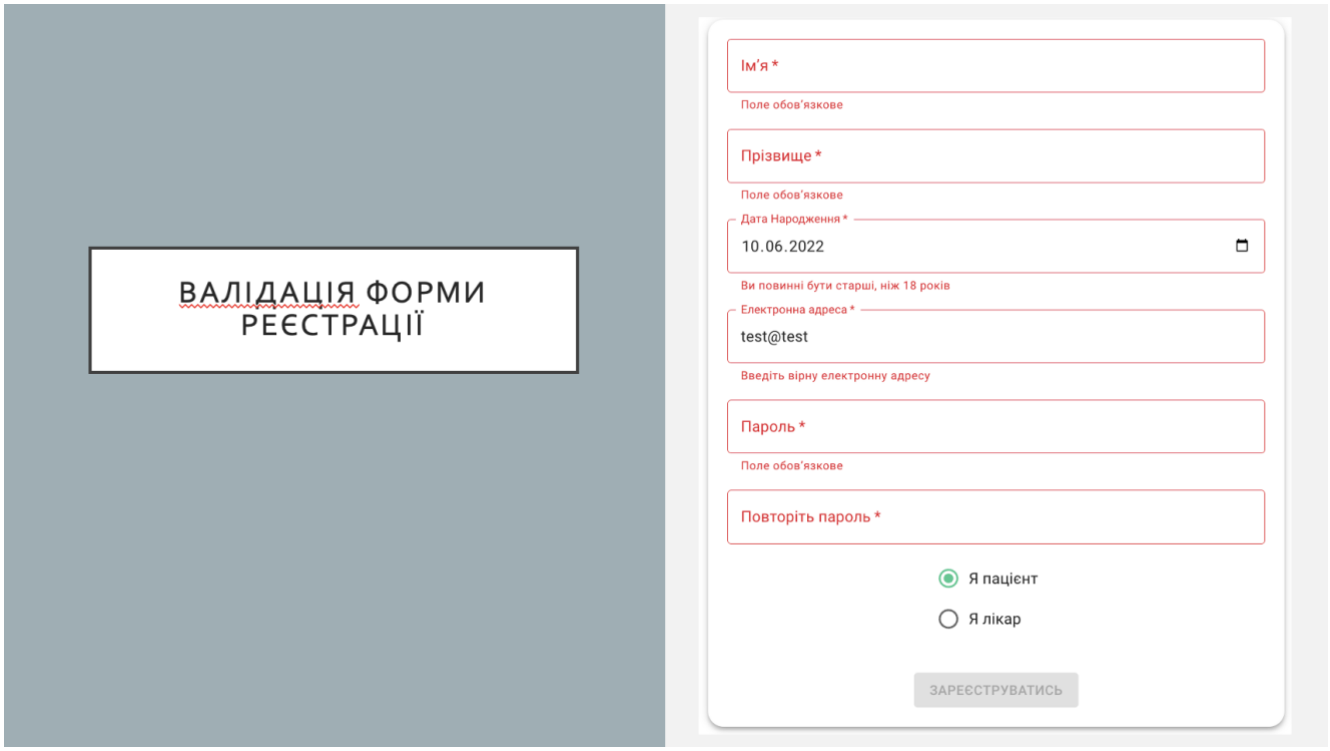


Рисунок Г.13 – Валідація форми реєстрації

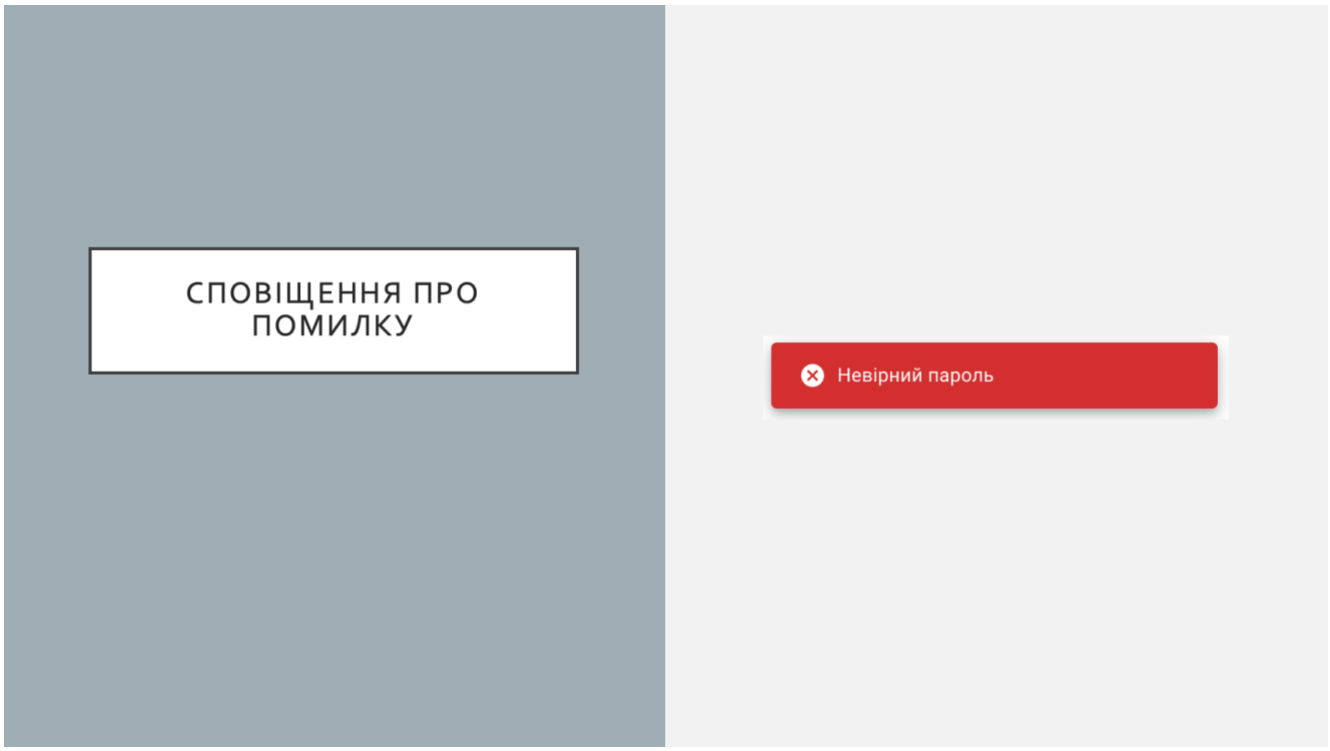


Рисунок Г.14 – Сповіщення про помилку

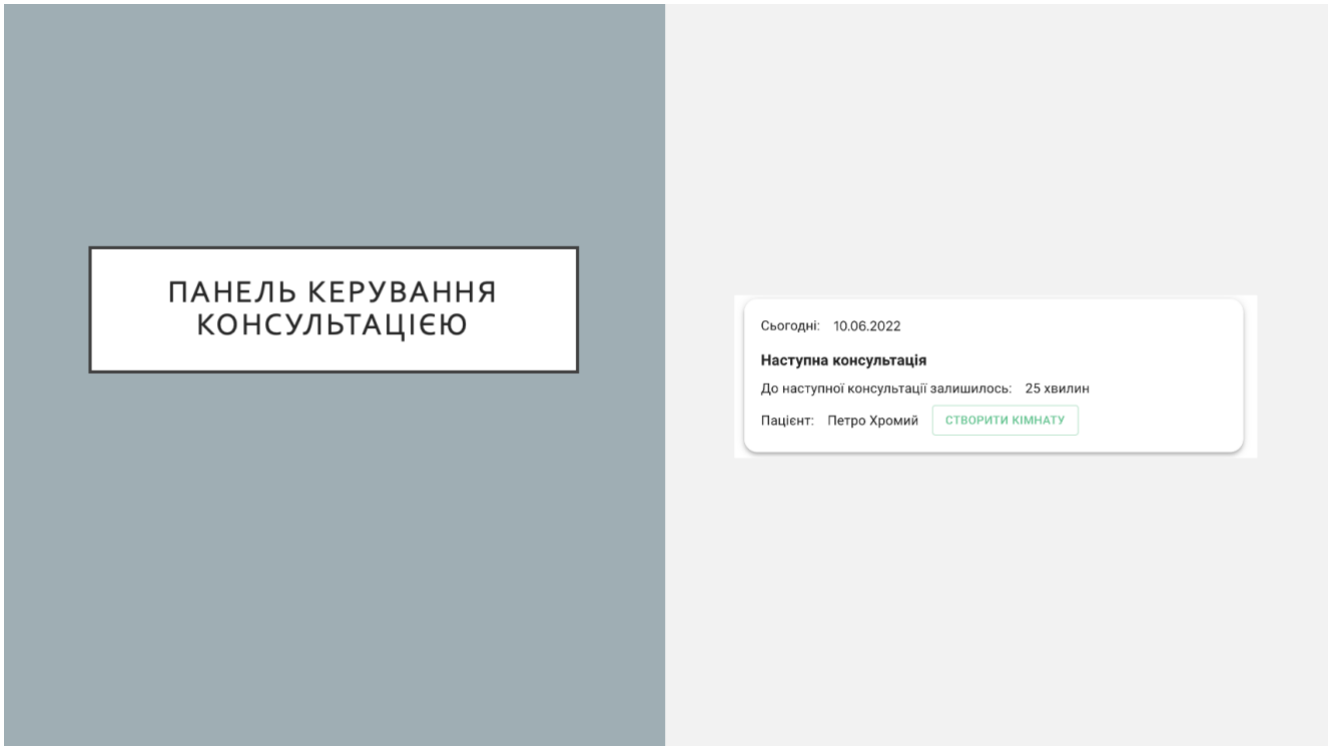


Рисунок Г.15 – Панель керування консультацією

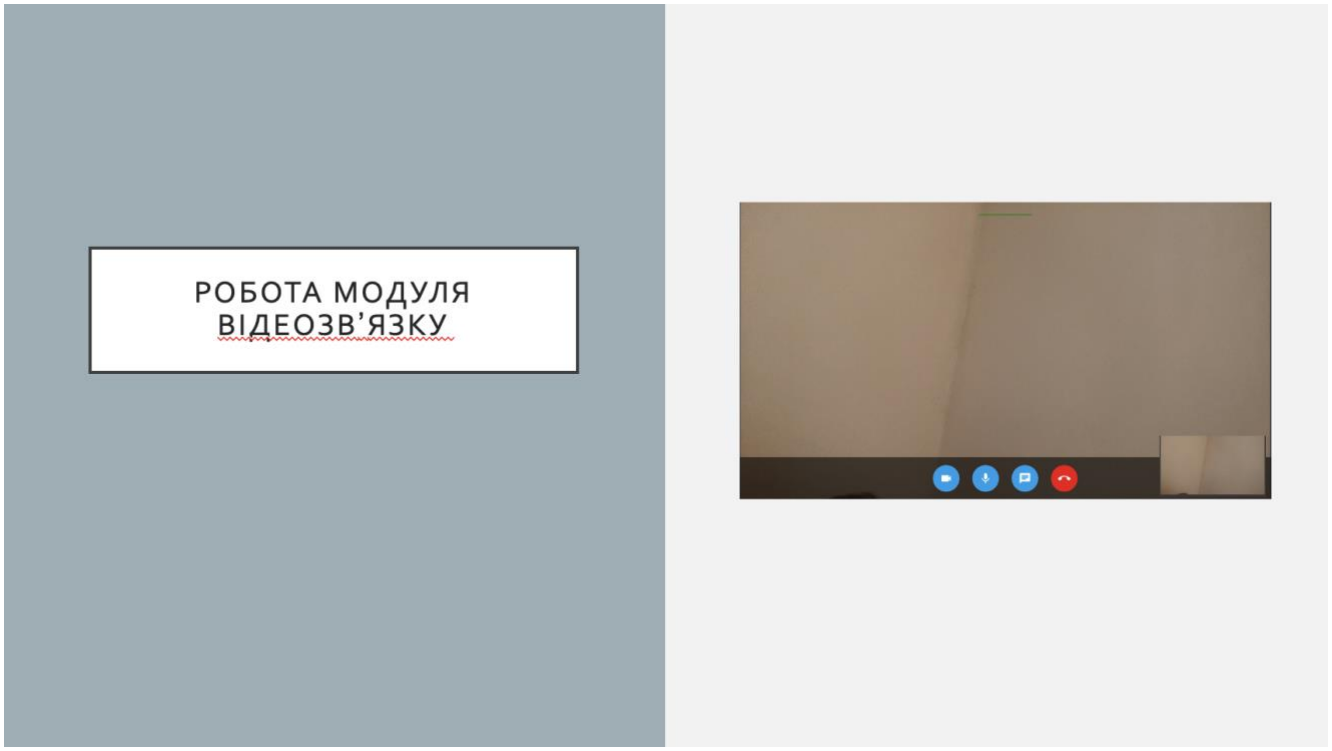


Рисунок Г.16 – Робота модуля відеозв'язку

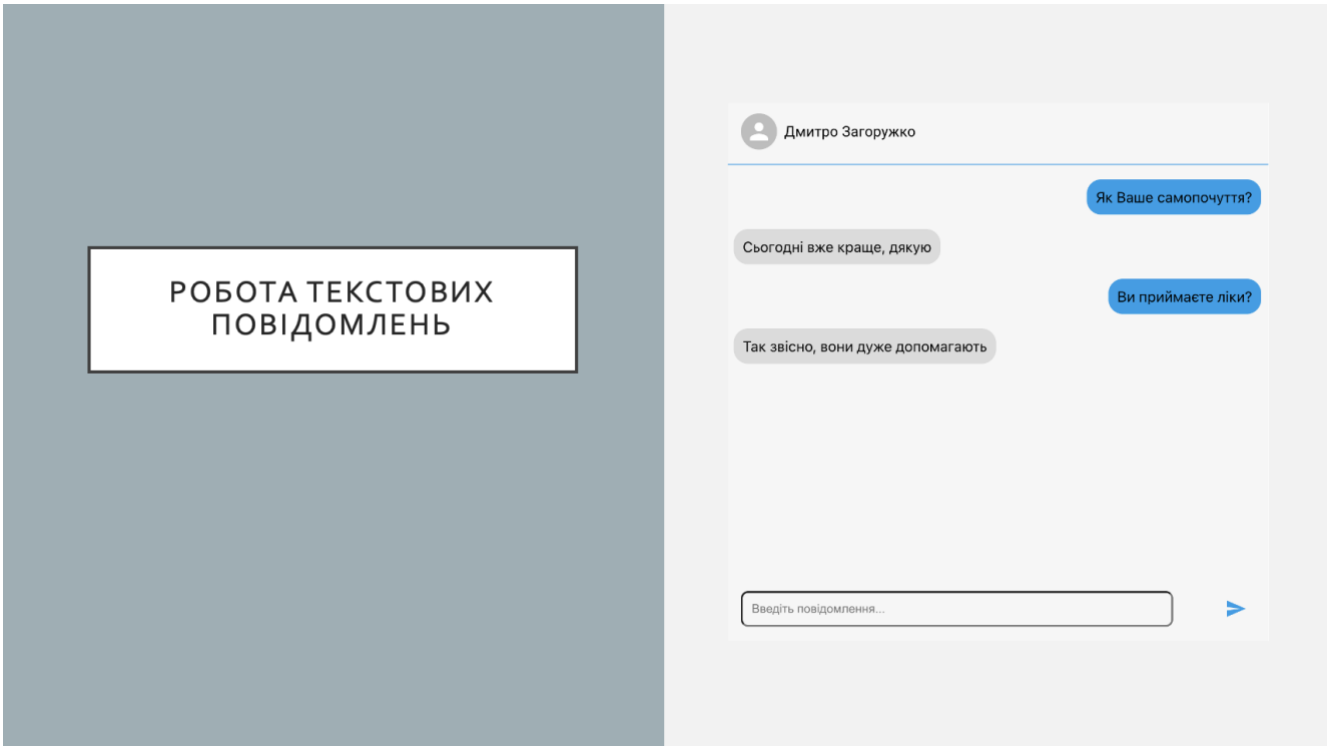


Рисунок Г.17 – Робота модуля текстових повідомлень

Рисунок Г.18 – Публікації

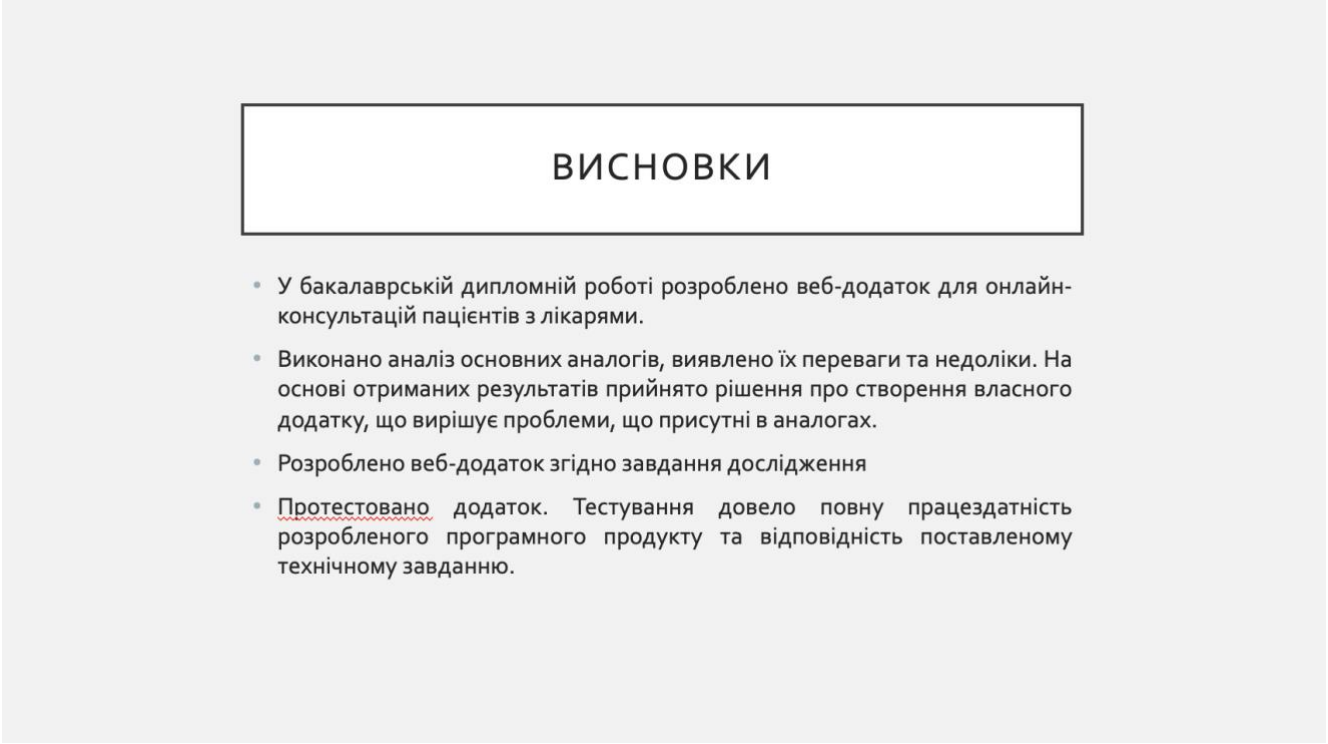


Рисунок Г.19 – Висновки