Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Розробка та реалізація клієнт-серверного застосунку для синхронного редагування коду/документів

Текстова частина до курсової роботи
за спеціальністю «Інженерія програмного забезпечення» 121

Керівник курсової роботи
к.ф.-м.с., доцент, кандидат наук
Жежерун О. П.
_____
(підпис)
“____” _____ 2020 р.
Виконав студент 4-го курсу
Бойчук О.Р.
“____” _____ 2020 р.

Київ 2021

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем,
доцент, к.ф.- м.н.
_____ О.П. Жежерун
(підпис)
“____”_____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Бойчуку Олегу
4-го курсу факультету інформатики

ТЕМА: Розробка та реалізація клієнт-серверного застосунку для синхронного редагування коду/документів

Вихідні дані:

- Веб-застосунок для редагування та спільного доступу до коду в реальному часі

Зміст ТЧ до курсової роботи:

Вступ
1. Аналіз предметної області. Постановка завдання курсової роботи
2. Теоретичні відомості
3. Опис реалізації програмного продукту
Висновки
Список джерел
Додатки (за необхідністю)

Дата видачі “___” _____ 2020 р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Розробка та реалізація клієнт-серверного застосунку для синхронного редагування коду/документів.

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	Жовтень-листопад 2020р.	
2.	Вивчення та аналіз задачі	Листопад-грудень 2020р.	
3.	Розробка архітектури та загальної структури програми	Січень-лютий 2021р.	
4.	Створення веб-застосунку	Лютий-березень 2021р.	
5.	Написання текстової частини	Березень-квітень 2021р.	
6.	Перегляд курсової роботи науковим керівником	Квітень 2021р.	
7.	Створення презентації	Квітень 2021р.	
8.	Захист курсової роботи		

Студент Бойчук О.Р. _____

Керівник Жежерун О. П. _____

“_____” _____ 2020 р.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

IDE - Integrated Development Environment.

URL – Uniform Resource Locator.

MVC – Model-View-Controller.

API – Application Program Interface.

СКБД – Система Керування Базами Даних.

JSON – JavaScript Object Notation.

HTTP – Hypertext Transfer Protocol.

REST – Representational State Transfer.

PaaS – Platform as a service

АНОТАЦІЯ

Робота присвячена створенню веб-застосунку для спільного, синхронного редагування коду/документів.

Проект створено за допомогою клієнт-серверної архітектури. Серверна частина реалізована динамічною мовою програмування JavaScript за допомогою фреймворку Express.js. Клієнт реалізований з використанням фреймворку Angular. База даних PostgreSQL використана для збереження даних користувачів.

ВСТУП

У сучасному світі технологій веб-застосункам вже не достатньо користуватись тільки HTTP-запитами. Для комфортного користування веб-ресурсами більшість сучасних застосунків перейшли на оновлення інформації в реальному часі – соціальні мережі, пошта, онлайн магазини тощо. Це дає змогу полегшити життя користувачу, якому більше не потрібно кожного разу оновлювати сторінку, щоб наприклад, побачити, що йому прийшов новий лист.

Через це використання real-time технологій є неабияк актуальним і потрібним зараз, так як завдяки ньому прискорюється робота з застосунком та збільшується зручність використання. Ні для кого не секрет, що з ростом технологій користувачі стають все більш вибагливими та нетерплячими, тож якщо стоятиме вибір між сервісом який потрібно буде постійно оновлювати і автоматичним, користувач вибере останній. Тож важливо розібратись в підходах розробки real-time застосунків і звичайно ж вміти вибрати та правильно застосувати найефективніший з них для окремого завдання.

Метою роботи є створення веб-ресурсу, який дасть змогу в режимі реального часу редагувати файл декільком користувачам з можливістю спілкування між ними, збереження файлів у системі та завантаження. Я поставив наступні завдання роботи:

1) З’ясувати масштаби проблеми і необхідність створення саме такого веб-ресурсу.
2) Проаналізувати аналоги з детальним виокремленням недоліків та переваг, для того щоб в майбутньому використати ці знання у створенні свого додатку.
3) Окреслити цільову аудиторію.

Щоб досягнути цієї мети я використав наступне програмне забезпечення:

- Back-end: Node JS + Express.js

- Front-end: Angular 8
- PaaS: Heroku
- Database: PostgreSQL

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАННЯ КУРСОВОЇ РОБОТИ

1.1. Аналіз сучасного стану питання та обґрунтування теми

Онлайн-редактор коду – інструмент, який знаходиться на віддаленому сервері і доступ до якого можна з легкістю отримати за допомогою браузера. Деякі онлайн-редактори коду володіють функціоналом, який схожий на звичайні текстові редактори, а інші більше нагадують повноцінні IDE.

Для чого потрібні онлайн-редактори коду?

- Співпраця. Налагодити роботу з іншою людиною за допомогою десктопного IDE доволі непросте завдання, яке зазвичай потребує завантаження додаткових плагінів. Завдяки онлайн-редакторам, можна значно спростити цей процес, робота з ними є дуже схожою на роботу з Google Docs.
- Співбесіди. В сучасних умовах пандемії та переходу більшості компаній в онлайн, співбесіди нових працівників також мігрували в мережу, тож для того, щоб інтерв’юер зміг побачити як кандидат працює з кодом і підходить до вирішення задач онлайн-редактори є дуже корисними.
- Навчання. За допомогою онлайн-редакторів можна поділитися своїм кодом з колегами, студентами та однолітками, а потім навчити їх. Університети та коледжі по всьому світу можуть використовувати редактори коду щодня. Ви можете розміщувати вбудовані уривки коду у своєму блозі, документації, посібниках.

1.2. Огляд існуючих аналогів розробки

На даний момент існує багато веб-застосунків, які використовують функціонал у режимі реального часу. Розглянемо декілька найпопулярніших застосунків які дають можливість спільного редагування коду для декількох людей одночасно.

Playcode – це простий інструмент для швидкого прототипування і перегляду результатів написання коду(див. рис. 1). Він має класичний інтерфейс, який складається з трьох вікон: редактор коду, консоль і вікно перегляду результату. Застосунок має базову файлову структуру, але в ньому немає контролю версій і інших функцій IDE.

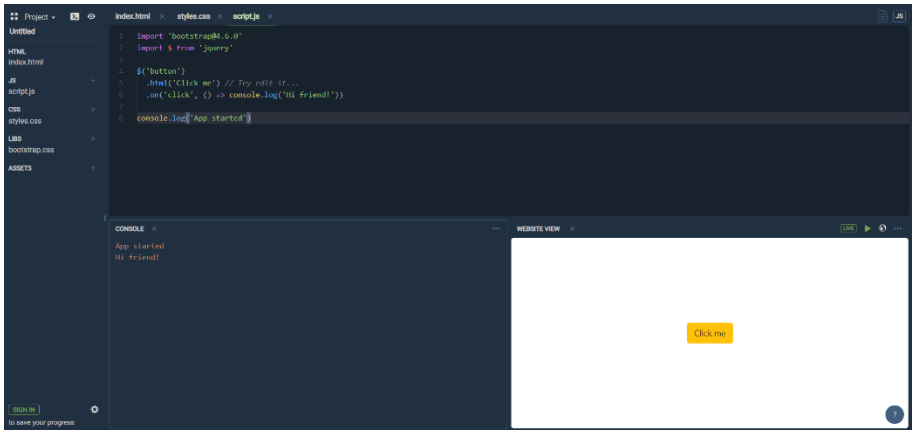

Рисунок 1Інтерфейс Playcode

З плюсів даного редактору можна виділити простий в дизайні і зручний у використанні інтерфейс, можливість вибрати стиль редактору та змінювати шрифт. З мінусів – Playcode підтримує тільки JavaScript, HTML и CSS.

Playcode підійде для тих користувачів, які займаються виключно веб-розробкою і яким достатньо мінімального набору функціоналу для роботи.

Codeshare – застосунок для спільного редагування та поширення коду або звичайного тексту(див. рис. 2). В нього простий та інтуїтивно зрозумілий інтерфейс.

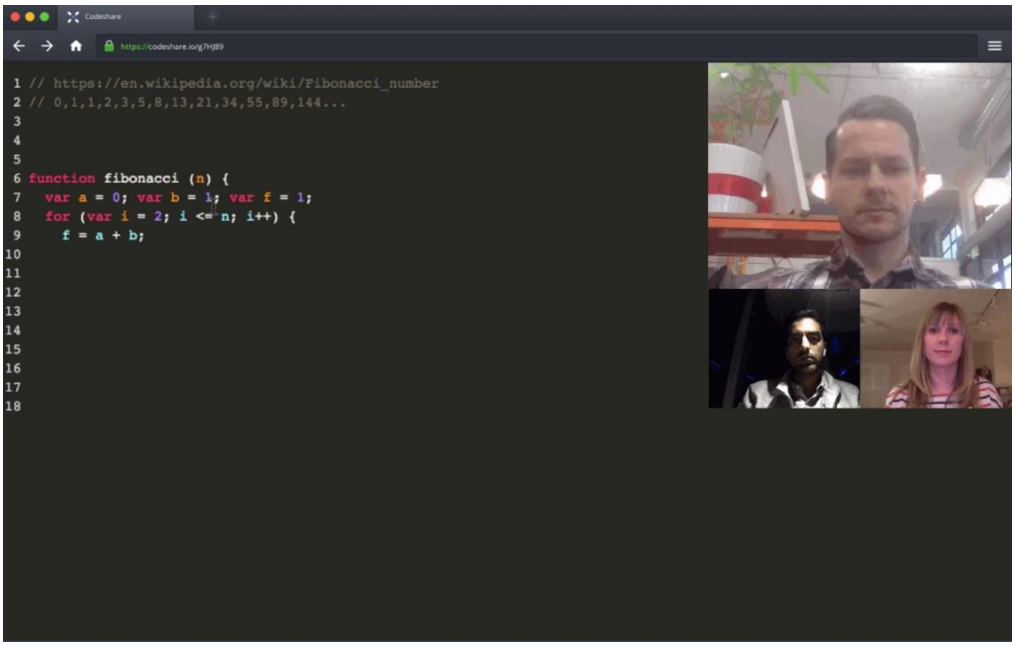

Рисунок 2 Інтерфейс Codeshare

Плюси: підтримка синтаксису більше ста мов програмування, можливість спілкування з іншими користувачами завдяки відео чату та можливість редагувати файли без реєстрації. Мінусами є неможливість зберігання файлів у системі та спілкування виключно з увімкненою відеокамерою.

Codeshare – ідеальний сервіс для проведення технічних співбесід який використовує мінімум інтернет трафіку.

CollabEdit – веб-застосунок для редагування коду або звичайного тексту(див. рис. 3). Використовується здебільшого для проведення лекцій чи співбесід. Є можливість користуватися без реєстрації та

автентифікації.

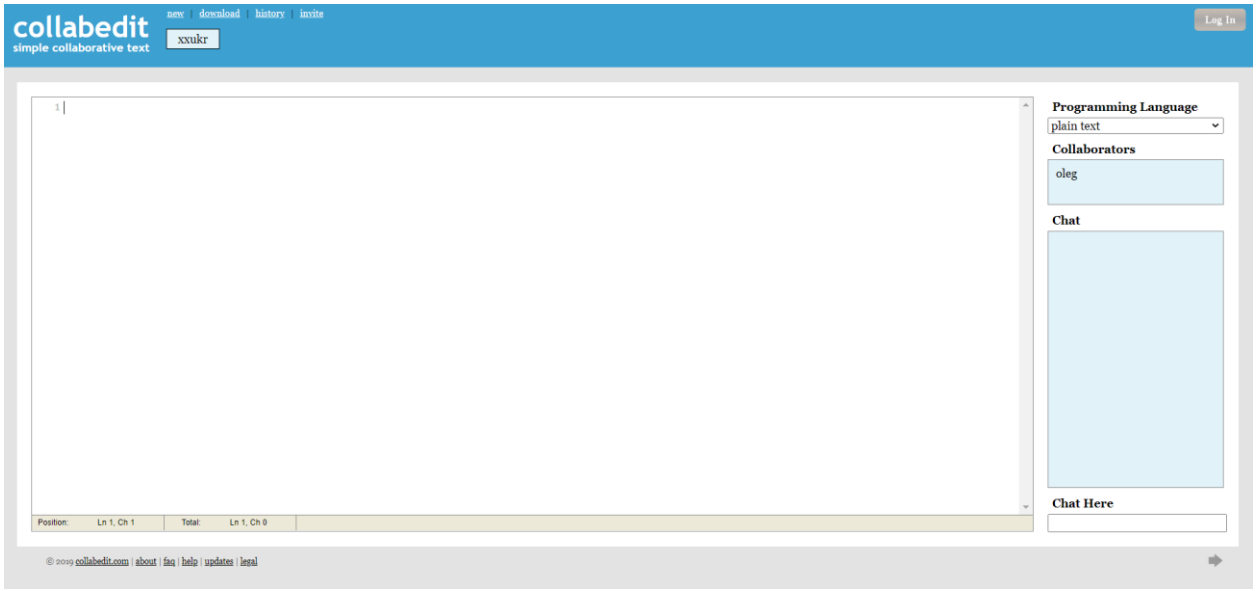

Рисунок 3 Інтерфейс Collabedit

Плюсом є простота інтерфейсу та наявність чату для спілкування. З недоліків можна виділити доволі велику затримку під час спільного редагування даних.

1.3. Постановка завдання

Створити веб-застосунок для спільного, синхронного редагування коду/документів.

Основні модулі системи:

1) Частина клієнта, яка відповідатиме наступним вимогам:
   a) Створення простого та зрозумілого інтерфейсу користувача.
   b) Написання функціоналу який би задовільнив потреби клієнта для взаємодії з сервером.
2) Частина сервера, яка реалізує взаємодії між користувачами застосунку. Вимоги:
   a) Написання функціоналу потрібного серверу для роботи з клієнтською частиною.
   b) Безпечна авторизація та автентифікація користувачів.
   c) Проектування СКБД.

Функціональні можливості системи:

1) Реєстрація та авторизація користувача.
2) Додавання, видалення та завантаження файлу.
3) Можливість поділитись з іншими користувачами доступом до редагування файлу.
4) Функціонал адміністратора – можливість власника файлу забороняти іншим користувачам редагувати чи завантажувати файл, або ж змінювати його тип.
5) Можливість синхронного редагування коду для декількох користувачів в режимі реального часу.
6) Створення чату для спілкування користувачів під час редагування файлу.

РОЗДІЛ 2
ТЕОРЕТИЧНІ ВІДОМОСТІ

2.1. Варіанти реалізації застосунку який оновлюватиме дані в режимі реального часу

2.1.1. Коротке опитування

Веб-застосунки спочатку були розроблені як звичайна модель клієнт-сервер, в якій клієнт ініціює HTTP-запит, що в свою чергу запитує певні дані з частини сервера. Наприклад, робота класичного веб-додатку з клієнт-серверною моделлю буде виглядати наступним чином(див. рис. 4):

- Клієнт надсилає HTTP-запит, запитуючи з сервера якісь дані.
- Сервер зчитує запит клієнта.
- Сервер відправляє відповідь клієнту.
- Клієнт обробляє дані з серверу.

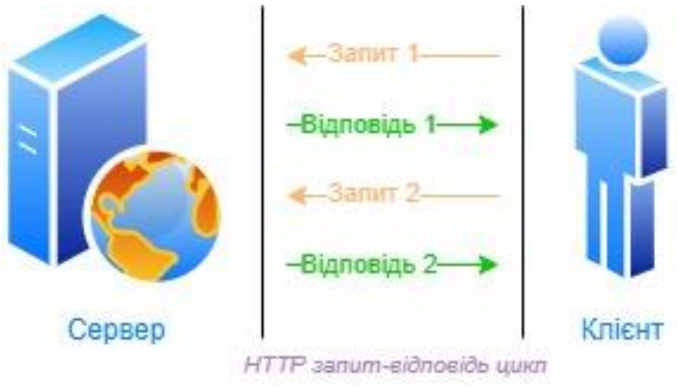

Рисунок 4 Polling схема

Описана вище модель спілкування клієнта з сервером також відома як опитування(англ. polling) – модель, коли після запиту клієнт відразу ж отримає відповідь від серверу. Розглянемо її як варіант для нашого застосунку. Для того, щоб реалізувати задумку з оновленням у реальному часі, нам доведеться надсилати запити на оновлення документу, щонайменше кожну секунду, а це в свою чергу приведе нас до створення величезного навантаження на сервер, адже йому доведеться опрацьовувати величезну кількість запитів, навіть тоді, коли не відбувається жодних змін в коді.

2.1.2. Довге опитування

Щоб подолати цей недолік, розробники веб-застосунків можуть використовувати підхід, який називається довге опитування(англ. long polling) – це, по суті, більш ефективна форма оригінальної методики опитування. Повторні запити до сервера витрачають ресурси даремно, так як кожне нове вхідне з'єднання повинно бути встановлено, заголовки HTTP повинні бути проаналізовані, запит на нові дані повинен бути виконаний, і відповідь (здебільшого без нових даних) повинна бути згенерована і доставлений на клієнт. Після цього, з'єднання має бути закрите, а всі ресурси очищені. Замість того щоб повторювати цей процес кілька разів для кожного клієнта до тих пір, поки не стануть доступні нові дані для даного клієнта, тривале опитування – це метод при якому сервер тримає клієнтське з'єднання відкритим якомога довше, доставляючи відповідь тільки після того, як дані стануть доступними або буде досягнутий поріг тайм-ауту. Коли клієнт отримує нову інформацію, він негайно надсилає інший запит, і операція повторюється.

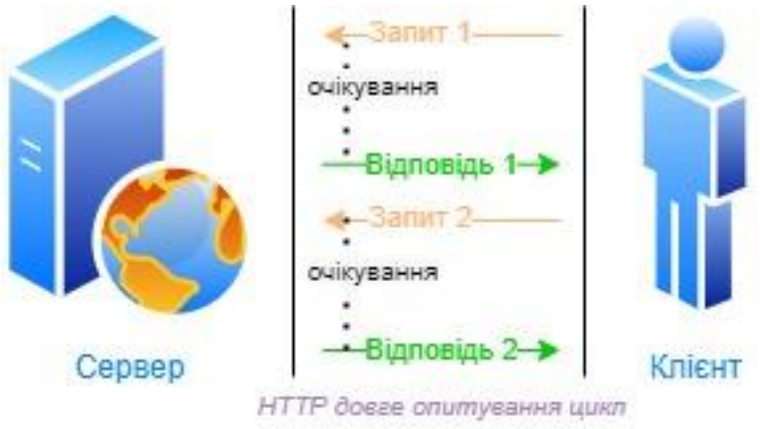

Рисунок 5 Long polling схема

Якщо розглядати цю модель для нашого випадку, то в неї є один великий недолік – при редагуванні коду система буде оновлювати зміни після кожного введеного/видаленого символу, що буде змушувати клієнт створювати новий запит, а це знову ж таки – велике навантаження на сервер. Отже, цей метод доречно використовувати, коли ви обмінюєтеся одним викликом з сервером, а сервер виконує деяку роботу у фоновому

режимі або ж коли ви більше не будете запитувати сервер на тій же сторінці.

2.1.3. Server Sent Events

Server-sent events(далі SSE) – це односторонній канал зв'язку, по якому події передаються тільки від сервера до клієнта. Надіслані сервером події дозволяють клієнтам браузера отримувати потік подій з сервера по HTTP-з'єднанню без опитування. SSE API міститься всередині інтерфейсу *EventSource*. Щоб відкрити з'єднання з сервером для початку запису подій, які він надсилає, необхідно створити новий об'єкт *EventSource*, який буде вказувати на URI скрипта, який створює події:

```
const newEvtSource = new EventSource("URI");
```

Як тільки ви створили екземпляр *EventSource*, ви можете почати отримувати повідомлення з сервера, додавши обробник події *message*:

```
newEvtSource.onmessage = function($event) {
   const newEl = document.createElement("div");
   const list = document.getElementById('elements-list');
   newEl.innerHTML = "message:" + $event.data;
   list.appendChild(newEl);
}
```

Цей код обробляє вхідні повідомлення (тобто повідомлення від сервера, на яких немає поля $event) і додає текст повідомлення в список в HTML-документі.

Отже, порівняно з попередніми методами, тут ми створюємо тільки одне з’єднання, що суттєво полегшує роботу для сервера, проте так як це односторонній канал зв’язку, то ми можемо надсилати дані тільки з сервера, а для роботи нашого застосунку потрібна підтримка двостороннього зв’язку.

2.1.4. WebSockets

2.1.4.1. Короткий огляд

Протокол WebSocket призначений для заміни існуючих технологій двостороннього зв'язку, що використовують HTTP як транспортний

рівень для отримання переваг від існуючої інфраструктури (проксі-сервери, фільтрація, аутентифікація). Такі технології були реалізовані як компроміс між ефективністю і надійністю, оскільки HTTP спочатку не призначався для використання для двостороннього зв'язку. Протокол WebSocket намагається вирішити завдання існуючих двосторонніх технологій HTTP в контексті існуючої інфраструктури HTTP; таким чином, він призначений для роботи через HTTP-порти 80 і 443, а також для підтримки HTTP-проксі і посередників, навіть якщо це передбачає деяку складність, специфічну для поточного середовища.

Протокол WebSocket забезпечує взаємодію між клієнтом і веб-сервером з меншими накладними витратами, забезпечуючи передачу даних з сервера і на сервер в режимі реального часу. WebSockets підтримує з'єднання відкритим, дозволяючи передавати повідомлення між клієнтом і сервером. Таким чином, між клієнтом і сервером може відбуватися двостороння безперервна розмова.

Рисунок 6 WebSoket схема

2.1.4.2. WebSocket handshake

WebSockets не використовують схему **http://** або **https://** (оскільки вони не дотримуються протоколу HTTP). Швидше за все, URI WebSocket використовують нову схему **ws:** (або **wss:** для безпечного WebSocket). Решта URI збігається з URI HTTP: хост, порт, шлях і будь-які параметри запиту.

```
"ws:" "//" host [ ":" port ] path [ "?" query ]
"wss:" "//" host [ ":" port ] path [ "?" query ]
```

З'єднання WebSocket можуть бути встановлені тільки з URI, які дотримуються цієї схеми. Тобто, якщо ви бачите URI зі схемою **ws://** (або **wss://**), то і клієнт, і сервер мають дотримуватись протоколу підключення WebSocket.

Клієнт, який підтримує WebSockets і бажає встановити з'єднання, має відправити HTTP-запит, який повинен містити кілька обов'язкових заголовків:

- *Connection: Upgrade*

  Заголовок *connection* визначає, чи залишається мережеве з'єднання відкритим після завершення поточної транзакції. Під час відкриття рукостискання WebSocket ми встановлюємо заголовок для оновлення, сигналізуючи, що хочемо зберегти з'єднання активним і використовувати його для не HTTP-запитів.

- *Upgrade: websocket*

  Заголовок *upgrade* використовується клієнтами, щоб попросити сервер переключитися на один з перерахованих протоколів в порядку спадання пріоритету. Ми вказуємо *websocket* тут, щоб сигналізувати, що клієнт хоче встановити WebSocket з'єднання.

- *Sec-WebSocket-Key: RhbwzJDM4OshzAxS3C+6Vg==*

  Sec-WebSocket-Key – це рандомно згенероване на стороні клієнта 16-байтове значення, яке закодоване в base64.

- *Sec-WebSocket-Version: 13*

  Єдина прийнята версія протоколу WebSocket – 13. Будь-яка інша версія, зазначена в цьому заголовку, недійсна.

Разом ці заголовки утворюють HTTP-запит GET від клієнта до **ws://** URI, як у наступному прикладі:

```
GET wss://duo-code.herokuapp.com/ HTTP/1.1
Host: duo-code.herokuapp.com
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Sec-WebSocket-Version: 13
Sec-WebSocket-Key: RhbwzJDM4OshzAxS3C+6Vg==
```

Як тільки клієнт відправляє початковий запит на відкриття з'єднання WebSocket, він чекає відповіді сервера. Відповідь сервера повинна містити HTTP-заголовки, що підтверджують успішне оновлення з'єднання:

- *Connection: Upgrade*

  Підтвердження, що з'єднання було оновлено.
- *Upgrade: websocket*

  Підтвердження, що з'єднання було оновлено.
- *Sec-WebSocket-Accept: sOWdyuVwum0RrqjAi9OsQnRBcdY=*

  Sec-WebSocket-Accept – це кодування **base64**, **SHA-1** хешоване значення. Це значення генерується шляхом об'єднання ключа клієнта *Sec-WebSocket-Key* і статичного значення **258eafa5-E914-47da-95ca-C5AB0DC85B11**, визначеного в **RFC 6455**.

Після того як клієнт отримує відповідь сервера, з'єднання WebSocket відкрито для початку передачі даних.

2.2. Express

Express – найпопулярніший веб-фреймворк для Node. Він також являється базовою бібліотекою для багатьох інших популярних веб-фреймворків Node.js. Express додає дві великі функції до Node.js HTTP-серверу:

- HTTP-сервер, який абстрагує більшу частину його складності. Наприклад, відправка одного png-файлу в RAW досить складна для Node.js (особливо в плані продуктивності); Express же в свою чергу скорочує його до одного рядка.
- Він надає можливість змінювати монолітну функцію обробника запитів на менші, які обробляють тільки певні біти і окремі фрагменти. Це дає змогу досягти модульності та спрощує процес відлагодження помилок.

У той час як сам *Express* досить мінімалістичний, розробники створили сумісні пакети проміжного програмного забезпечення для вирішення практично будь-якої проблеми з веб-розробкою. Існують бібліотеки для роботи з файлами cookie, сесіями, веб-секетами, параметрами URL, даними POST та багатьма іншими.

Теоретично Express можна використовувати для створення будь-якої веб-програми. Він може обробляти вхідні запити та відповідати на них, тому він може робити те, що ви можете робити в більшості інших фреймворків. Одна з переваг написання коду в Node.js це можливість спільного використання JavaScript-коду між браузером і сервером. Це корисно з точки зору коду, тому що ви можете буквально запускати один і той же код на клієнті і сервері. Це також дуже корисно з ментальної точки зору; вам не потрібно переводити свій розум в режим сервера, а потім перемикатися в режим клієнта—на якомусь рівні це одне і те ж. Це означає, що front-end розробник може писати back-end код без необхідності вивчати абсолютно нову мову і його парадигми, і навпаки.

Express часто використовується для створення односторінкових додатків (SPA). SPA – дуже важкі на інтерфейсі JavaScript, і вони зазвичай вимагають серверного компонента. Сервер зазвичай потрібно просто обслуговувати HTML, CSS і JavaScript, але часто існує і REST API. Express може робити і те, і інше досить добре. Він відмінно підходить для обслуговування HTML та інших файлів, а також для створення API.

2.3. Socket.IO

Socket.IO – це бібліотека, яка забезпечує двосторонній і почерговий зв'язок між браузером і сервером в режимі реального часу. Вона складається з:

- Node.js серверу;
- Клієнтської бібліотеки JavaScript (яка також може бути запущена з Node.js);

Socket.IO – це не реалізація WebSocket. Хоча Socket.IO дійсно використовує WebSocket як транспорт, коли це можливо, він додає додаткові метадані до кожного пакету. Ось чому клієнт WebSocket не зможе успішно підключитися до сервера Socket.IO, а клієнт Socket.IO також не зможе підключитися до звичайного сервера WebSocket.

Ось функції, що додані в Socket.IO і які відсутні в WebSockets:

- Надійність (резервний варіант для тривалого опитування HTTP у випадку, якщо з'єднання WebSocket не може бути встановлено);
- Автоматичне перепідключення;
- Буферизація пакетів;
- Підтвердження;
- Мовлення на всіх клієнтів або на підмножину клієнтів;

Остання функція може бути дуже корисною для нашого застосунку, так як вона дає змогу створювати необмежену кількість «кімнат» для спілкування між різними клієнтами.

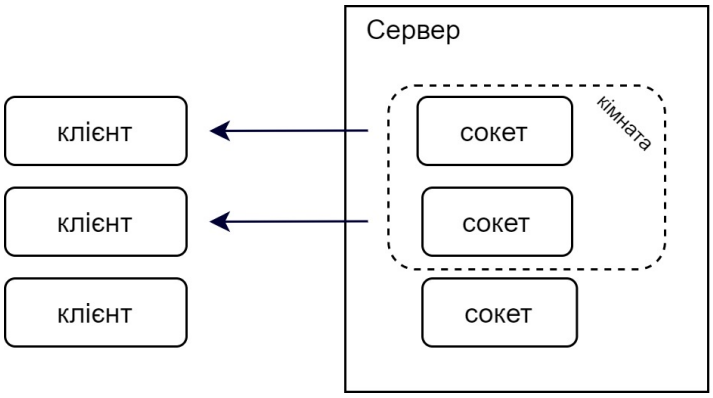

Рисунок 7 Схема роботи "кімнат" в Soket.IO

Кімната – це довільний канал, в який можуть входити і виходити сокети. Він може бути використаний для трансляції подій на підмножину клієнтів(див. рис. 7).

РОЗДІЛ 3
ОПИС РЕАЛІЗАЦІЇ ПРОГРАМНОГО ПРОДУКТУ

3.1. Обґрунтування алгоритму й структури програми

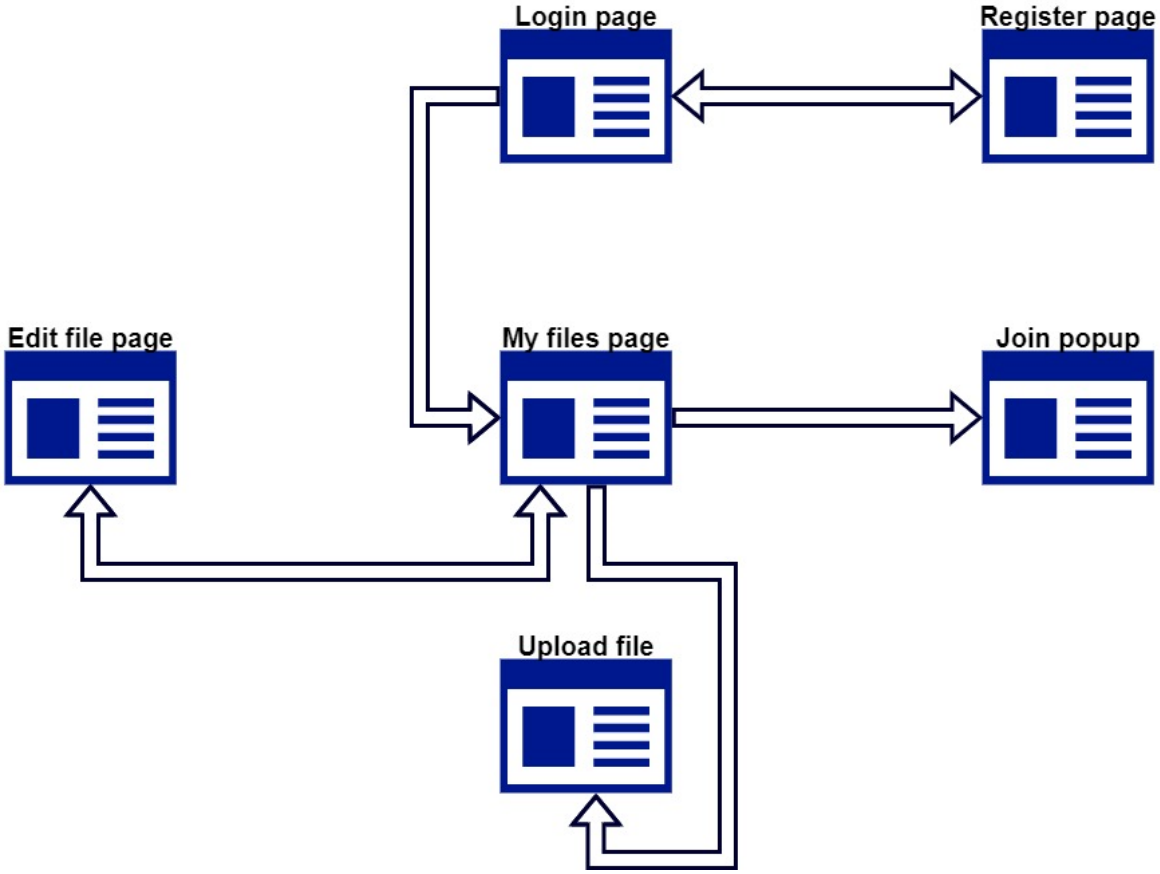

Рисунок 8 - Діаграма структури програми

На UML-діаграмі(див. рис. 8) зображена структуру застосунку та алгоритм роботи. Отже, дивлячись на неї, можемо розбити проект на два основі модулі:

- Аутентифікація;
- Робота з файлами;

3.2. Обґрунтування вибору засобів розробки

Частина клієнта реалізована за допомогою фреймворку Angular [10]. Він побудований на основі архітектури MVC.

MVC архітектура надає можливість ізолювання логіки застосунку від рівня клієнта. **M** — модель предметної області, **V** — відображення і **C** — функція обробник (контролер). Контролер приймає всі запити для застосунку і взаємодіє з моделлю, для того щоб підготувати дані, які необхідні для відображення, яке в свою чергу приймає дані, підготовлені контролером, і вимальовує кінцеву відповідь(див. рис. 9).

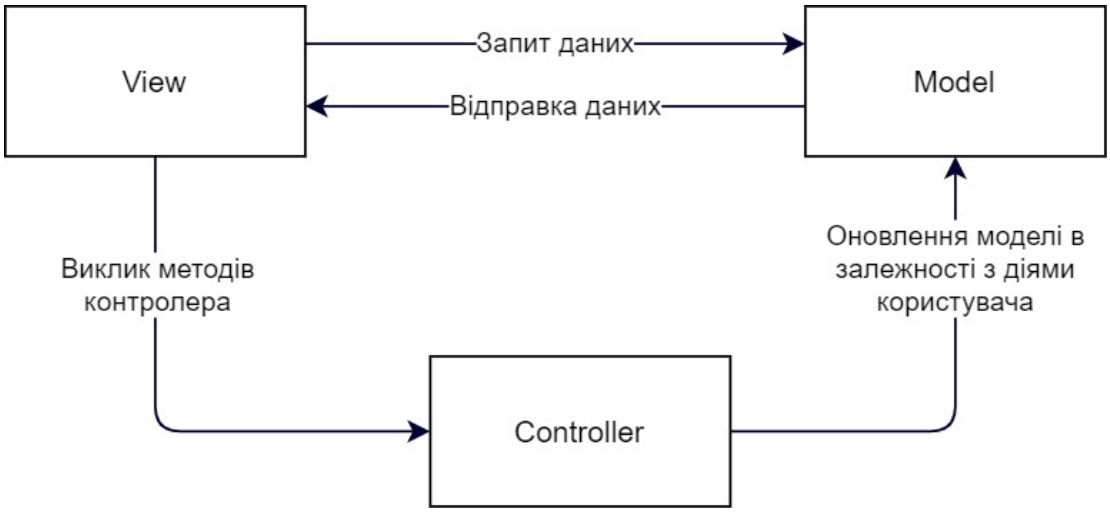

Рисунок 9 Схема взаємодії даних в MVC

Модулі є основними блоками побудови архітектури Angular. Модулі дозволяють Angular визначити контекст для компіляції шаблонів.

Сервер реалізований за допомогою платформи Node.js та фреймворку Express.js [9]. Він надіє нам один з найпростіших, і одночасно найпотужніший способі створення сервера.

Visual Studio Code – один з найпопулярніших редакторів коду, серед програмістів. Він швидкий, легкий і до того ж потужний. Microsoft розробила VSC як крос-платформний редактор коду для написання веб і хмарних додатків. [7].

Для розробки застосунку було використано СКБД PostgreSQL. PostgreSQL – система управління базами даних корпоративного класу з відкритим вихідним кодом. Вона підтримує як SQL, так і JSON для реляційних та нереляційних запитів. PostgreSQL підтримує розширені типи даних і функції оптимізації продуктивності, які доступні тільки в дорогих комерційних базах даних, таких як Oracle і SQL Server [8].

3.3. Опис файлів даних та інтерфейсу програми

При розробці програми було використано реляційну базу даних PostgreSQL. Для реалізації поставленого завдання розроблено наступну модель даних(див. рис. 10):

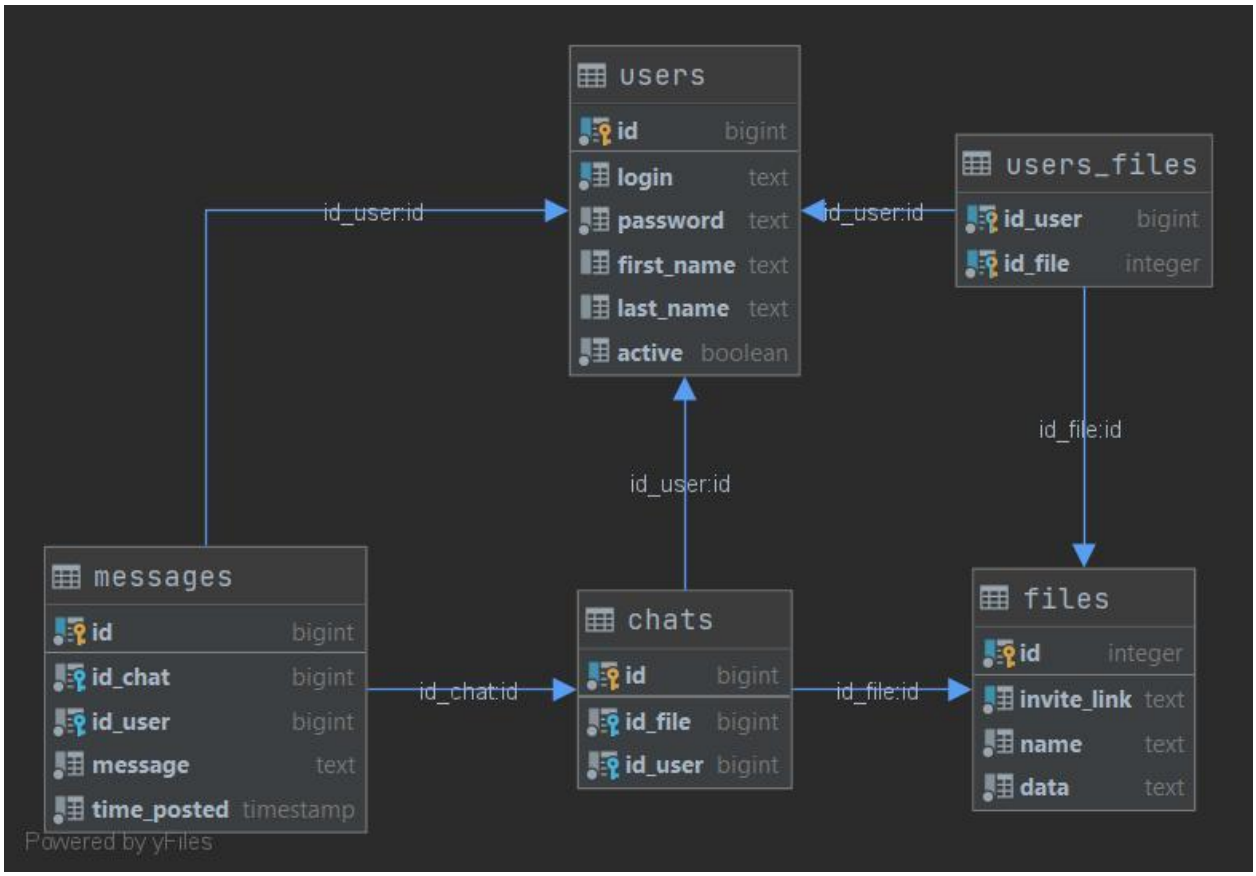

Рисунок 10 Схема бази даних

3.4. Тестування програми і результати її виконання

Веб-додаток розміщено на PaaS платформі Heroku, що дає змогу запустити проект в браузері перейшовши за наступним покликанням: https://duo-code.herokuapp.com/.

Після загрузки застосунку ми спочатку потрапляємо на вікно логіну(див. рис. 11):

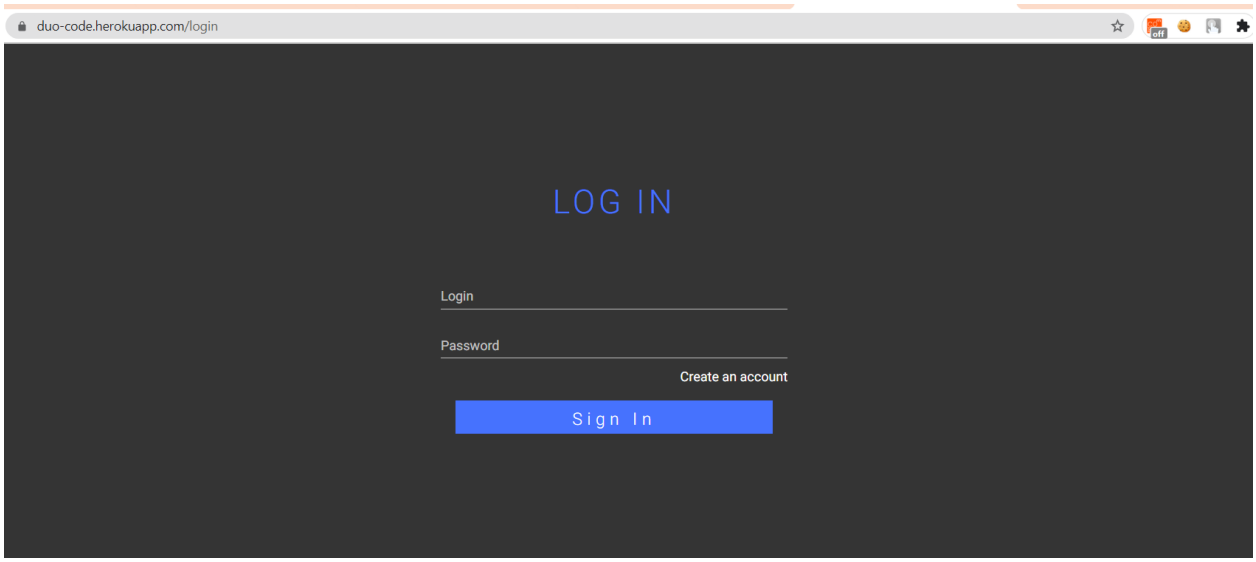

Рисунок 11 Сторінка логіну

Тут користувач може увійти в систему зі свого аккаунту або ж перейти на сторінку реєстрації і створити нового користувача(див. рис. 12).

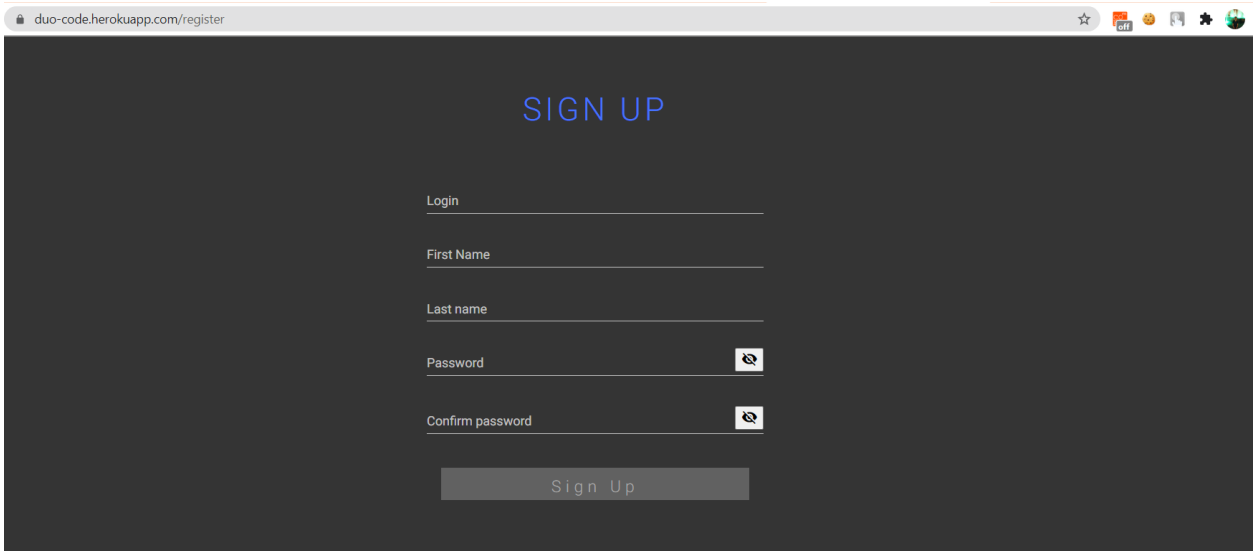

Рисунок 12 Сторінка реєстрації

В проекті використовується валідатор реактивних форм Angular для всіх полів, кнопок та інших елементів, де користувач вводить/змінює інформацію(див. рис. 13).

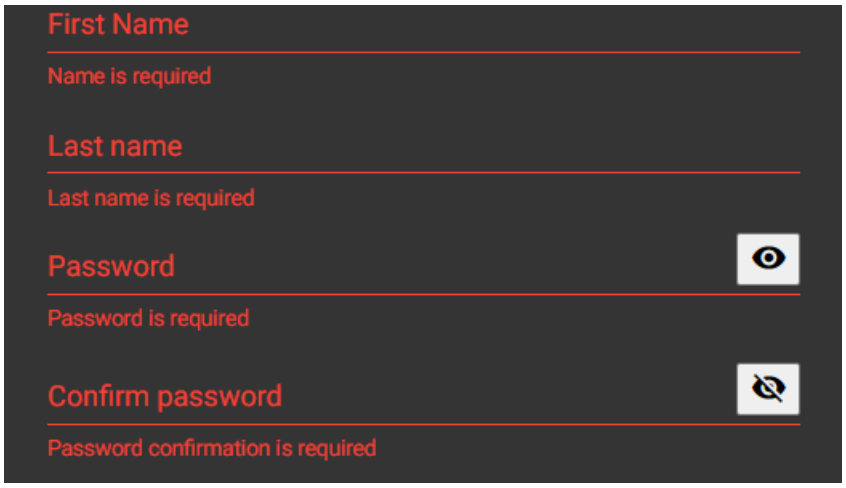

Рисунок 13 Валідація

Після успішного входу в систему користувач потрапляє на сторінку з усіма його файлами(див. рис. 14).

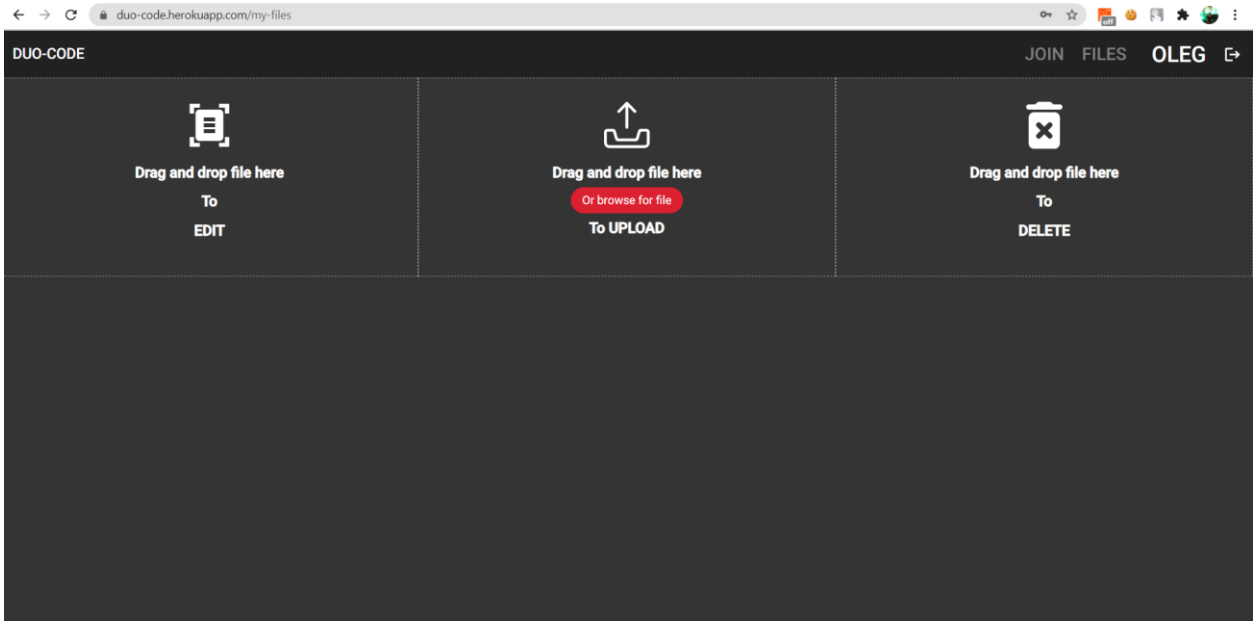

Рисунок 14 Сторінка з файлами користувача

Щоб почати роботу з файлами, спочатку його потрібно завантажити. Для цього потрібно в центральне поле перетягнути файл, яким користувач планує поділитись для спільного доступу(див. рис. 15) або ж натиснути на це поле і вибрати файл за допомогою провідника. Після чого почнеться процес завантаження файлу на сервер.

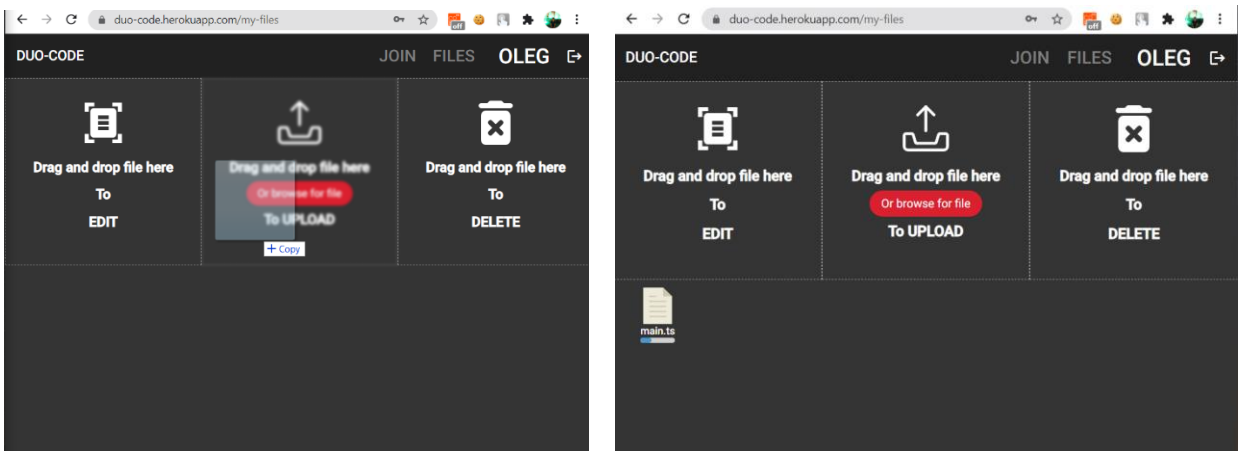

Рисунок 15 Завантаження документу

Тепер коли у користувача є файл, він може почати його редагувати, для цього необхідно просто перетягнути його на поле для редагування(див. рис. 16).

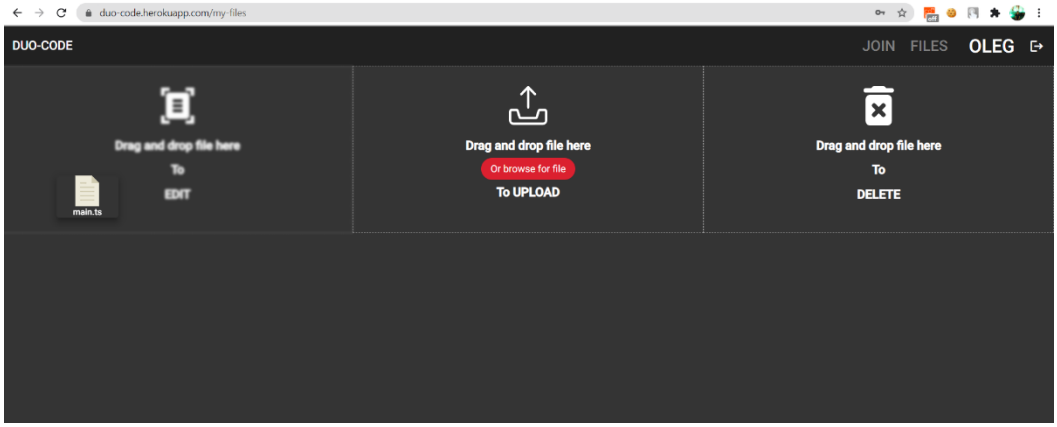

Рисунок 16 Відкриття документу

Після цього користувач потрапляє на сторінку редагування файлу(див. рис. 17).

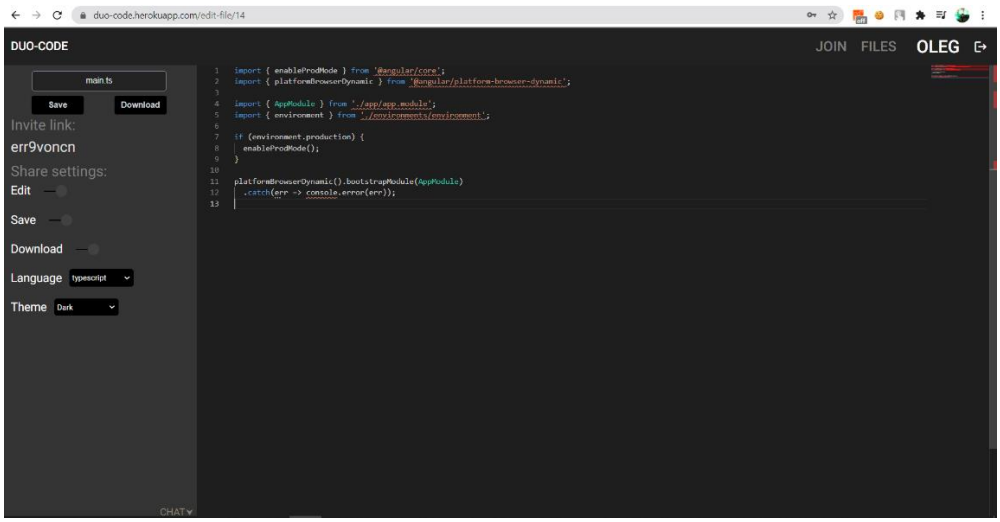

Рисунок 17 Сторінка редагування

Справа на сторінці більшу частину займає редактор для коду, в якому присутня міні-карта для навігації по документу та підтримка синтаксису більш як п’ятдесяти мов програмування. Користувач має можливість самостійно обирати синтаксис якої мови йому необхідно підключити. Також редактор підтримує три варіанти теми інтерфейсу: дві темні та світлу. В лівому верхньому кутку є можливість змінити ім’я файлу, зберегти його після редагування та завантажити. Для того, щоб надати доступ іншим користувачам для спільного редагування, потрібно скопіювати посилання для запрошення (зробити це можна просто натиснувши правою кнопкою мині на нього, посилання автоматично збережеться в буфер для обміну). Щоб приєднатись до редагування, інший користувач має спочатку увійти в систему, після чого в полі навігації натиснути кнопку «JOIN» і у спливаюче вікно вставити посилання, яким з ним поділився власник файлу(див. рис. 18).

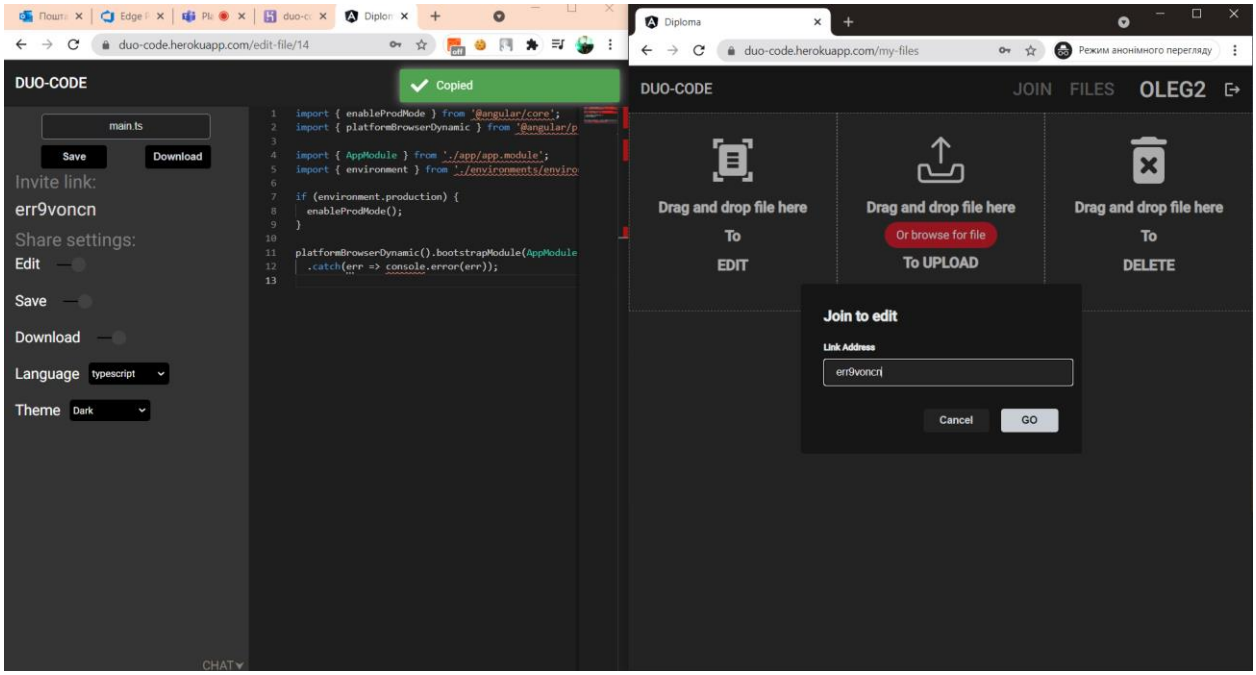

Рисунок 18 Приєднання користувача до спільного редагування

Після цього він потрапляє на сторінку редагування(див. рис. 19). Користувач який надав доступ отримує сповіщення в чат про те, що приєднався новий юзер. У них є можливість спілкуватись в чаті за допомогою текстових повідомлень. Чат можна приховати якщо він не

використовується, коли прийде нове повідомлення, біля іконки з’явиться сповіщення, яке відображає кількість нових повідомлень. Чат як і редагування реалізований за допомогою бібліотеки Soket.IO, яка використовує технологію WebSokets, тож сповіщення приходять без затримки по тому ж каналу даних.

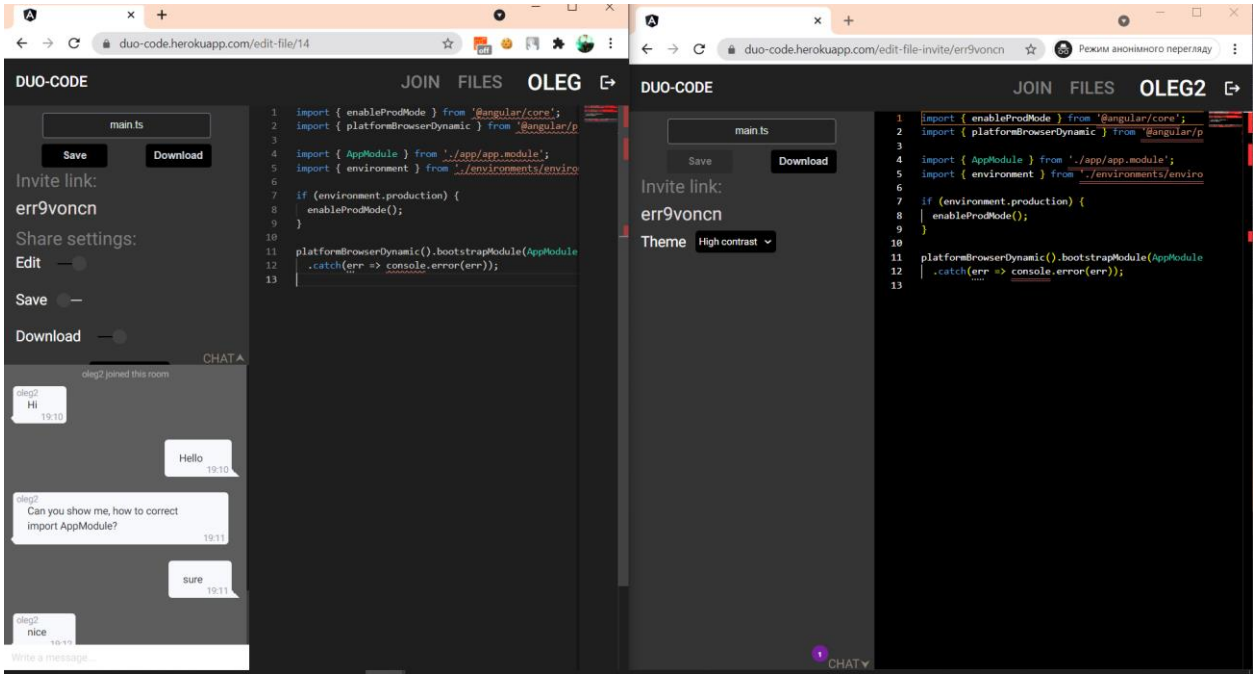

Рисунок 19 Редагування декількома користувачами

У власника файлу та у користувача який приєднався до редагування різні права. Власник може змінювати синтаксис, забороняти іншим користувачам редагування, завантаження та зберігання змін. Кожен юзер має можливість обирати зручну для себе тему перегляду файлу, також ділитись посиланням для приєднання інших людей.

Окрім підтримки синтаксису, редактор також підсвічує помилки які допустив користувач при редагуванні та автодоповнює код(див. рис. 20).

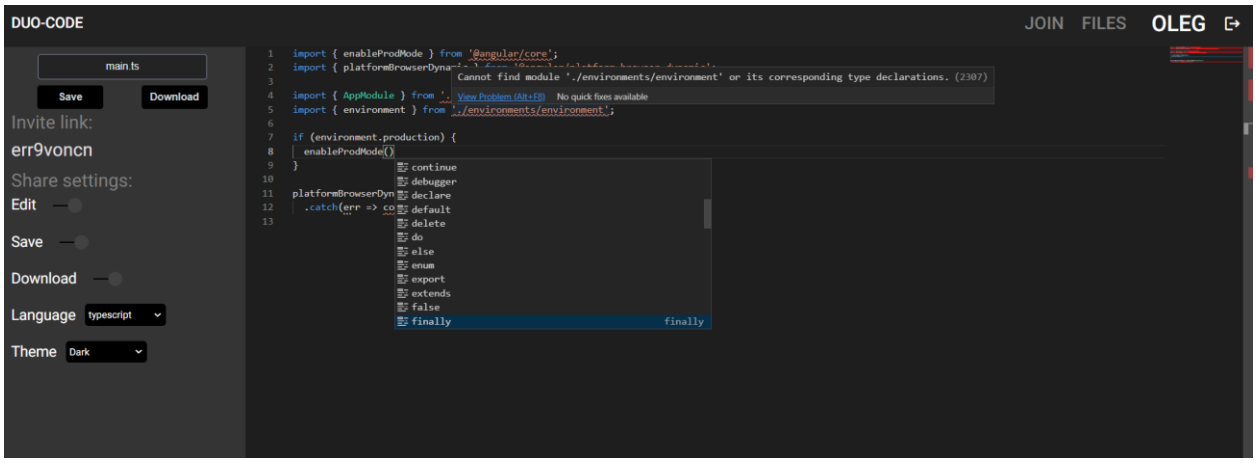

Рисунок 20 Автодоповнення

ВИСНОВКИ

Отже, мета проекту була досягнута, функціонал веб-додатоку співпадає з тими вимогами, які описані у завданні. Під час виконання цієї роботи я зміг детально вивчити технологію WebSokets та використати її на практиці. Також я зміг закріпити свої знання фреймворку Angular та більш детально розібратись у роботі PaaS платформи Heroku.

Цей застосунок слід використовувати людям, що займаються проведенням онлайн співбесід, лекторам та працівникам у сфері IT. Він допоможе швидко і з мінімальним трафіком поділитись кодом з іншими користувачами.

Розширити можливості системи можна додавши можливість спілкуватись за допомогою голосового або ж відео чату.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Playcode: https://playcode.io/
2. Codeshare: https://codeshare.io/
3. CollabEdit: http://collabedit.com/
4. VS Code: https://code.visualstudio.com/
5. Ian Fette & Alexey Melnikov, “The WebSocket Protocol”, RFC 6455, December 2011
6. PostgreSQL: https://www.postgresql.org/
7. Node JS: https://nodejs.org/uk/docs/
8. Express: https://expressjs.com/
9. Angular: https://habr.com/en/post/348818/#s1
10. WebSokets:https://sookocheff.com/post/networking/how-do-websockets-work/

ДОДАТОК А

Код клієнта – сторінка всіх файлів

```
import {Component, OnInit} from '@angular/core';
import {CdkDragDrop, moveItemInArray} from '@angular/cdk/drag-drop';
import {IFileModel, IFullFileModel} from '../../models/file.model';
import {FileService} from '../../services/file.service';
import {Router} from '@angular/router';
import {ToastrService} from 'ngx-toastr';

@Component({
 selector: 'app-my-files',
 templateUrl: './my-files.component.html',
 styleUrls: ['./my-files.component.scss']
})
export class MyFilesComponent implements OnInit {

 constructor(private fileService: FileService,
         private router: Router,
         private toastr: ToastrService) {
 }

 files: Array<IFullFileModel> = [];

 ngOnInit(): void {
  this.fileService.getAllUserFiles().subscribe(res => {
   res.forEach(el => {
    el.progress = 100;
   });
   this.files = res;
  });
 }

 onFileDropped($event): void {
  this.prepareFilesList($event);
 }

 fileBrowseHandler(files): void {
  this.prepareFilesList(files);
 }

 deleteFile(index: any): void {
  const id = this.files[index.previousIndex].id;
  this.fileService.deleteFile(id).subscribe(res => {
   this.files.splice(index.previousIndex, 1);
   this.toastr.success('File deleted', 'Done!');
  }, error => {
   this.toastr.error(error.error.message, 'ERROR!');
  });
 }

 editFile($event: CdkDragDrop<any>): void {
  this.router.navigate(['/edit-file', this.files[$event.previousIndex].id]);
 }

 drop(event: CdkDragDrop<string[]>): void {
  moveItemInArray(this.files, event.previousIndex, event.currentIndex);
```

```
  }

  uploadFilesSimulator(index: number): void {
   setTimeout(() => {
    if (index === this.files.length) {
     return;
    } else {
     const progressInterval = setInterval(() => {
      if (this.files[index].progress === 100) {
       clearInterval(progressInterval);
       this.uploadFilesSimulator(index + 1);
      } else {
       this.files[index].progress += 5;
      }
     }, 200);
    }
   }, 1000);
  }

  prepareFilesList(files: Array<any>): void {
   for (const item of files) {
    const reader = new FileReader();
    reader.onload = (evt) => {
     const newFile = {
      invite_link: Math.random().toString(36).substr(2, 9),
      name: item.name,
      data: evt.target.result,
     } as IFileModel;
     this.fileService.addFile(newFile).subscribe(res => {
      res[0].progress = 0;
      this.files.push(res[0]);
     }, error => {
      this.toastr.error('Can`t add this type of file', 'ERROR!');
     });
    };
    reader.readAsText(item);
   }
   this.uploadFilesSimulator(0);
  }

}
```

ДОДАТОК Б

Код сервера – файл-контролер

```
let express = require('express');
let router = express.Router();
const {Pool} = require('pg');
let bodyParser = require('body-parser');
router.use(bodyParser.urlencoded({extended: false}));
router.use(bodyParser.json());
let jwt = require('jsonwebtoken');
const userRequests = require('../../sql/queryes/file.js');

const pool = new Pool({
 connectionString: ',
 ssl: true,
});

router.route('/files')
 .get((req, res) => {
  let token = req.header('x-access-token');
  let id = jwt.decode(token).id;
  pool.query(userRequests.find_user_files, [id], (err, result) => {
   if (err) return res.status(500).send({message: 'Error on the server.'});
   res.status(200).json(result.rows)
  });
 });

router.route('/file')
 .post((req, res) => {
  let token = req.header('x-access-token');
  let id = jwt.decode(token).id;
  pool.query(userRequests.add_file_to_user, [req.body.name, req.body.data, req.body.invite_link], (err,
result) => {
   if (err) return res.status(500).send({message: 'Error on the server.'});
   let fileId = result.rows[0].id;
   pool.query(userRequests.add_reference_file_to_user, [id, fileId], (err, resul) => {
    if (err) return res.status(500).send({message: 'Error on the server.'});
    res.status(200).json(result.rows)
   });
  });
 })
 .put((req, res) => {
  pool.query(userRequests.change_file, [req.body.id, req.body.name, req.body.data], (err, result) => {
   if (err) return res.status(500).send({message: 'Error on the server.'});
   res.status(200).json(result.rows)
  });
 });

router.route('/file/:id_file')
 .get((req, res) => {
  let token = req.header('x-access-token');
  let id = jwt.decode(token).id;
  pool.query(userRequests.find_user_file, [id, req.params.id_file], (err, result) => {
   if (err) return res.status(500).send({message: 'Error on the server.'});
   if (!result.rows[0]) return res.status(403).send({message: 'Error: You don`t have access to this file.'});
   res.status(200).json(result.rows[0])
  });
```

```
 })
 .delete((req, res) => {
   pool.query(userRequests.delete_user_file, [req.params.id_file], (err, result) => {
     if (err) throw err;
     res.status(200).json(result.rows)
   });
 });

router.route('/file/link/:invite_link')
 .get((req, res) => {
   pool.query(userRequests.find_file_by_link, [req.params.invite_link], (err, result) => {
     if (err) return res.status(500).send({message: 'Error: Wrong link.'});
     if (!result.rows[0]) return res.status(403).send({message: 'Error: Wrong link.'});
     res.status(200).json(result.rows[0])
   });
 })

module.exports = router;
```