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

Real-time веб-застосунок для спільного доступу до документів та їх редагування

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

Керівник курсової роботи

к.ф.-м.н., ст.викл. Гречко А. В.

_____
(підпис)
“____” _____ 2020 р.

Виконала студентка 4-го курсу
Тищенко Д.А.
“____” _____ 2020 р.

Київ 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мережних технологій,

проф.,д.ф.-м.н.

_____ Г.І. Малашонок

(підпис)

“____”_____ 2019 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студентці _____Тищенко Дар’ї_____

__4-го__ курсу факультету інформатики

ТЕМА: Real-time веб-застосунок для спільного доступу до документів та їх редагування

Вихідні дані:

- Веб-застосунок для редагування та спільного доступу до документів 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 “___” _____ 2019 р.

Керівник _____ Завдання отримано _____

Календарний план виконання курсової роботи

Тема: Real-time веб-застосунок для спільного доступу до документів та їх редагування

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	Жовтень-листопад 2019р.	
2.	Огляд літератури за темою роботи	Листопад-грудень 2019р.	
3.	Оволодіння навичками роботи з SignalR	Лютий 2020р.	
4.	Створення веб-застосунку	Лютий-березень 2020р.	
5.	Написання текстової частини	Лютий-березень 2020р.	
6.	Перегляд курсової роботи науковим керівником	Березень-квітень 2020р.	
7.	Створення презентації	Квітень 2020р.	
8.	Захист курсової роботи	19.04.2020	

Студент Тищенко Д.А _____

Керівник Гречко А.В. _____

“_____”_____ 2020 р.

Зміст

Перелік умовних позначень 6
Анотація 7
Вступ 8
1. Аналіз предметної області. Постановка завдання курсової роботи 10
1.1. Огляд існуючих аналогів розробки 10
1.2. Постановка завдання 13
2. Теоретичні відомості 14
2.1. Обмеження моделі спілкування «запит-відповідь» 14
2.2. Способи реалізації real-time сервісу 15
2.2.1. Опитування 15
2.2.2. Довге опитування 15
2.2.3. Специфікація server-sent events 16
2.2.4. Веб-сокети 17
2.2.4.1. Загальний огляд 17
2.2.4.2. Рукостискання веб-сокетів 18
2.2.4.3. Типи повідомлень 19
2.3. SignalR 20
2.3.1. Загальний огляд 20
2.3.2. Транспортна система 21
2.3.3. Hubs 22
2.4. Операційне перетворення 23
3. Опис реалізації програмного продукту 26
3.1. Обґрунтування вибору засобів розробки 26
3.2. Опис розробки програми 27
3.3. Створення об'єктів і розробка головної програми 28
3.4. Опис файлів даних та інтерфейсу програми 31
3.5. Тестування програми і результати її виконання 38
Висновок 45
Список використаної літератури 46
ДОДАТОК А. Клієнтський код – компонент документа 48

ДОДАТОК Б. Серверний код – концентратор ManageYourDocsHub..... 52

Перелік умовних позначень

SSE – server-sent events (події, відправлені сервером).

AJAX – Asynchronous JavaScript And XML.

URL – Uniform Resource Locator.

JSON – JavaScript Object Notation.

GUID – Globally Unique Identifier.

RPC – Remote Procedure Call.

OT – Operational Transformation (операційне перетворення).

Анотація

Робота присвячена розробці веб-сервісу для спільного доступу до документів та їх редагування в режимі реального часу.

У даній роботі докладно розглянуто концепції і методи створення real-time застосунку, зокрема часте опитування, довге опитування, server-sent events та WebSockets, а також описано алгоритм операційного перетворення OT.

Проект базується на клієнт-серверній архітектурі. Клієнт реалізований з використанням JavaScript-бібліотеки react.js, а сервер на мові програмування C# з використанням фреймворку ASP.NET Core. Для спільного та одночасного редагування документів використано Signalr.

Вступ

У наш час веб-функціональність в режимі реального часу майже завжди необхідна при розробці веб-застосувань і вже недостатньо створити клієнт-серверний застосунок, в якому спілкування відбувається лише за допомогою звичайних HTTP-запитів. Майже всі сучасні застосунки використовують оновлення інформації у режимі реального часу – месенджери, онлайн аукціони, пошта тощо. Користувачам вже не потрібно постійно оновлювати сторінку, щоб побачити, що у нього, наприклад, з'явилося нове повідомлення.

Саме тому використання технологій real-time дуже актуальне та потрібне зараз, адже це пришвидшує роботу із сервісом та робить її більш зручною. Зараз при розробці свого продукту варто враховувати, що люди відвикли довго чекати та зайвий раз натискати на якусь кнопку і виберуть застосунок, у якому оновлення даних на сторінці буде відбуватися автоматично. І зрозуміло, що важливо знати, які існують підходи до розробки real-time застосувань та вміти вибрати серед них найефективніший для конкретної задачі. Як приклад, таким застосунком може бути одночасне редагування тексту декількома користувачами сервісу.

Метою даної роботи є створення зручного та швидкого веб-сервісу для збереження документів та їх спільного редагування користувачами у режимі реального часу з додатковими можливостями поширення цих документів серед інших користувачів системи, налаштування редагування документів та збереження змін у базі даних.

Сервіс базуватиметься на клієнт-серверній архітектурі. Його розробка здійснюватиметься за допомогою мови програмування C# та бібліотеки react.js. Для спілкування клієнта з сервером використовуватиметься Fetch API, а для реалізації функціоналу у режимі реального часу – бібліотека SignalR.

Дана робота складається зі вступу, трьох розділів, висновку, списку використаних джерел та додатків.

У першому розділі розглянуто сучасні методи, засоби та підходи до розробки real-time застосунків, проаналізовано існуючі системи-аналоги та сформульовано постановку завдання.

У другому розділі докладно описано теоретичні основи концепції real-time, розглянуто її способи реалізації у веб-застосуваннях та алгоритм ОТ.

Третій розділ присвячено реалізації програмної частини. Також коротко описано та обґрунтовано обрані технології, мови програмування та середовище розробки. Окрім цього, описано процес розробки застосування та наведено результати його тестування.

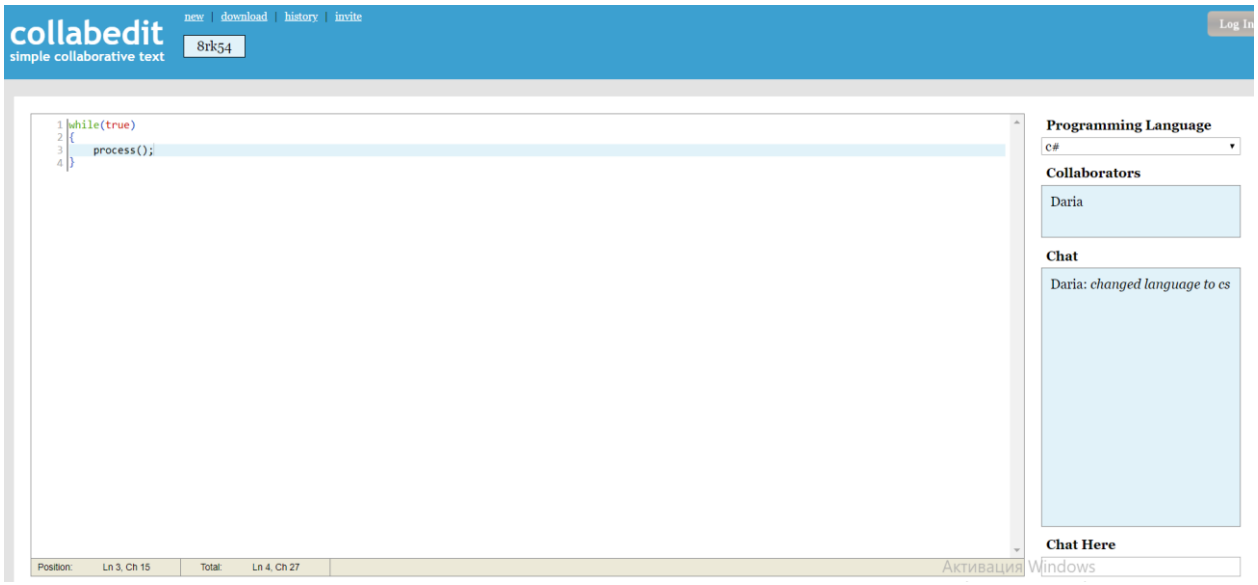
1. Аналіз предметної області. Постановка завдання курсової роботи

1.1. Огляд існуючих аналогів розробки

Наразі багато застосунків використовують функціонал у режимі реального часу – месенджери (Telegram, Viber), соціальні мережі (Facebook, Instagram), електронні пошти (Gmail) та програми для спільного редагування тексту, програмного коду тощо. Далі буде розглянуто декілька таких веб-застосувань.

CollabEdit – онлайн редактор програмного коду або звичайного тексту [1]. В основному призначений для проведення віддалених співбесід, лекцій чи семінарів. Не вимагає авторизації та реєстрації, але документи неавторизованого користувача будуть доступні для читання та модифікації усім, хто має ідентифікатор документу.

З особливостей даної системи – підсвічування синтаксису для обраної мови програмування. З недоліків – доволі помітна затримка при оновленні даних. Інтерфейс користувача виглядає наступним чином:



Google Docs [2] – чи не найпопулярніший застосунок для спільної роботи з документами. Це безкоштовний мережевий пакет, який надає не тільки можливість редагувати документ, а і залишати коментарі та переглядати його

історію змін. Також користувач одразу бачить, хто ще присутній на сторінці та які зміни він вносить.

Окрім цього даний застосунок має ряд функціональних можливостей, яких немає у його аналогів:

- Експорт документу у один з набору форматів файлів – .doc, .odt, PDF, HTML чи epub.
- Гнучка конфігурація рівнів доступу – можна відкрити документ для усіх, хто має посилання на нього, або для конкретного набору користувачів, деяким з яких дозволити лише читання, а також заборонити експорт, друк та копіювання. Також можна управляти можливістю інших користувачів поширювати доступ до документу.
- Можливості стильового оформлення та розмітки охоплюють функціонал настільного застосунку Microsoft Word та подібних йому.

Confluence – застосунок від розробників Atlasian, призначений для організації та керування роботи у команді [3]. Документи у даному продукті є частиною великої системи з документування певних задач та відслідковування їх прогресу. Тут вони відображають певне глобальне завдання, що стоїть перед командою.

Функціональні особливості редактора *Confluence* задовольняють специфіку його призначення: окрім звичайного тексту, документ може містити інші типи медіа-даних: задачі (містять текст та відмітку про статус виконання), згадування виконавців задачі (при додаванні згадування про певного користувача, він отримує сповіщення про нього), різні інформаційні панелі (текст та категорія інформації) тощо. Нижче зображений приклад такого документу з різними типами медіа-даних:

Course work

Some description

- [ ] Write theoretical part. @Daria IN PROGRESS сб, 4 апр.
- [ ] Develop an real-time app. @Daria IN PROGRESS 4 апр. 2020 г.

Very important info.

Some conclusion.

1.2. Постановка завдання

Необхідно розробити веб-сервіс для роботи з документами в режимі реального часу, який забезпечує такі функціональні можливості:

1) Реєстрація та авторизація користувача.
2) Створення нового документа або завантаження вже існуючого.
3) Пошук серед документів за назвою.
4) Можливість поділитися своїм документом із іншим користувачем за його поштою.
5) Редагування тексту і назви документа та збереження відповідних змін.
6) Зміни, внесені одним користувачем, мають відображатися для інших у режимі реального часу з урахуванням змін для позиції курсора.

2. Теоретичні відомості

2.1. Обмеження моделі спілкування «запит-відповідь»

Традиційно веб-застосунок працює за наступною схемою:

- Браузер (клієнт) ініціює HTTP-запит до сервера (як приклад, просить HTML-сторінку).
- Сервер оброблює запит і відправляє назад відповідь.
- Клієнт отримує відповідь та оброблює її.

Очевидно, що така схема не підходить для застосування, що працює в режимі реального часу, адже за такого підходу для сервера немає можливості відправляти повідомлення клієнту, якщо на ньому відбулися якісь зміни (з’явилося нове повідомлення). Але існує декілька низькорівневих технік, які роблять це можливим та які використовує ASP.NET Core SignalR.

Також варто згадати, що таке AJAX. Це техніка, яка дозволяє браузеру відправляти запит на сервер, використовуючи JavaScript, і отримана відповідь може бути відображена без оновлення сторінки. Це невелика частинка вирішення проблеми, яка допомагає зробити застосунок у режимі реального часу, – тепер ми можемо одразу додавати зміни на сторінку та не перезавантажувати її. Але також потрібно ще повідомити браузер, коли відбудуться зміни на сервері.

2.2. Способи реалізації real-time сервісу

2.2.1. Опитування

По суті, спілкування за протоколом HTTP – це процес, коли клієнт просить щось із сервера. Для даного застосунку необхідно, щоб браузер автоматично оновлював зміст документа, щоб відобразити зміни користувачу. Тобто виходить, що коли щось змінюється на сервері, браузер має про це знати.

Перше, що спадає на думку, найпростіший спосіб, – опитування (англ. polling). Він полягає в тому, щоб клієнт періодично запитував сервер, чи відбулися якісь зміни. Для кожного такого опитування створюється HTTP-запит і сервер або повертає оновлення або каже, що нічого не змінилося.

Уявимо, що ми питаємо про оновлення кожну хвилину, у нас буде дуже багато запитів через певний проміжок часу і зайве навантаження на сервер. Також у цього способу є ще проблема – затримка між оновленнями та отриманням їх на клієнті. Сервер буде повідомляти не тоді, коли щось зміниться, а тоді коли прийде черговий запит із браузера. [4]

2.2.2. Довге опитування

Щоб зекономити на ресурсах, покращенням попереднього способу, є довге опитування (англ. long polling). Цей підхід відрізняється тільки одним – сервер довше відповідає.

Клієнт відправляє запит і сервер не відповідає одразу, а залишає з’єднання відкритим. У кожного такого запиту є час, протягом якого серверу треба відповісти (timeout). Тому є тільки два випадки, коли клієнт отримає відповідь: якщо є оновлення і сервер відповість одразу та якщо з’єднання пора розірвати. Після чого браузер відправить ще один запит.

За такого підходу ми уникаємо частого посилання запитів і відповідно зменшуємо навантаження на сервер та збільшується ймовірність того, що клієнт одразу отримає повідомлення про оновлення.

Недоліками даного підходу є те, що при отриманні відповіді, запит переривається і створюється новий. Наприклад, клієнт відправляє запит, щоб дізнатися чи відбулися якісь зміни у документі. Якщо система зберігає зміни після кожного введеного символу, то клієнт буде дуже часто отримувати повідомлення і відправляти наступний запит. А кожний такий запит – це створення заголовків, за потреби додавання аутентифікації і так далі. Тому виходить, що довге опитування підходить в основному для сервісів, у яких рідко відбуваються оновлення. [4]

2.2.3. Специфікація server-sent events

Є ще один спосіб реалізації застосунку в режимі реального часу – створення сервером HTTP-підключення з клієнтом за допомогою server-sent events. Ця специфікація описує вбудований клас *EventSource*, який дозволяє підтримувати з'єднання із сервером та отримувати події (потік тексту в кодуванні UTF-8). Цей клас має подію *onMessage* для обробки вхідних повідомлень та події для обробки помилок, які можуть прийти.

Щоб це реалізувати, клієнт має створити об'єкт *EventSource* та вказати URL, з яким буде підтримувати з'єднання:

```
const evtSource = new EventSource("someURL");
```

Після цього можна додати прослуховування повідомлень із сервера за допомогою вищезгаданого обробника подій *onMessage*.

Сервер має відповісти зі статусом 200 та використовуючи MIME-тип *text/event-stream.* Після чого підтримує з'єднання відкритим та відправляє повідомлення в особливому форматі. Кожне таке повідомлення відправляється як блок тексту, який закінчується декількома символами

нового рядка. Але на практиці більш складні повідомлення відправляють у форматі JSON.

Отже, виходить, що всі події проходять тільки через одне з'єднання, що набагато краще за попередні способи. Але, на жаль, дані можна відправляти тільки в одну сторону (із сервера) та підтримує тільки текстові повідомлення.

Незважаючи на згадані недоліки, розробляти застосунок з використанням специфікації SSE простіше ніж з веб-сокетами та вона підтримує автоматичне перепідключення у разі втрати з'єднання. [5-6]

2.2.4. Веб-сокети

2.2.4.1. Загальний огляд

Протокол веб-сокет (WebSocket) надає можливість обмінюватися даними між сервером і браузером в два напрямки через один TCP сокет, який залишається відкритим, поки потік повідомлень не буде виконаний. Дані передаються як «пакети» без додаткових HTTP-запитів. Даний протокол має дві частини: рукостискання та передача даних. [7]

WebSocket — це ідеальна транспортна система для реалізації застосунку в реальному часі і тому бібліотека SignalR в основному їх і використовує. Веб-сокети забезпечують найбільш ефективне використання пам'яті сервера, надає повний дуплексний обмін даними між клієнтом і сервером та має найменшу затримку. Але також WebSocket має доволі строгі вимоги до серверу і якщо вони не виконуються, то SignalR для створення з'єднання буде намагатися використати інші транспортні системи. [8]

Веб-сокети — це новітня технологія, яка добре підходить для застосунків, які потребують постійний обмін даними в реальному часі – ігри, онлайн-майданчики для торгівлі тощо.

2.2.4.2. Рукостискання веб-сокетів

Кожний веб-сокет починає своє життя з відправки на сервер HTTP get-запиту, який просить оновити сокет. Якщо сервер підтримує WebSocket і погоджується, то далі спілкування починається за протоколом WebSocket, а не HTTP.

Клієнтське рукостискання – це запит на оновлення, який містить приблизно такі заголовки:

```
GET /example HTTP/1.1
Host: server.com
Origin: client.com
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

У даному запиті він просить оновитися до веб-сокета. Також він надсилає відкритий ключ – *Sec-WebSocket-Key* у форматі base64.

Варто відмітити, що якщо сервер прийняв запит на оновлення, то версія, яку вказав клієнт, підходить. В іншому випадку відповідь буде вказувати, які версії він підтримує, і тоді клієнт спробує ще раз з однією із них. Приклад такої відповіді:

```
HTTP/1.1 400 Bad Request
…
Sec-WebSocket-Version: 13, 8
```

Якщо сервер підтримує веб-сокети і версія відходить, він відповідає з кодом статусу 101 і перемикає протокол. Також він повертає ключ у *Sec-WebSocket-Accept* – хеш, вирахуваний з ключа, який відправив клієнт.

```
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: example
```

Хеш вираховується наступним чином [7]:

- До ключа клієнта додається константа – GUID, який має значення «258EAFA5-E914-47DA-95CA-C5AB0DC85B11».
- Далі вираховується sha1 від рядка, який ми тримали у попередньому кроці. Результат має бути у вигляді двійкового рядку із 20 символів.
- Кодуємо хеш у форматі base64.

Навіщо? Веб-запити мають тенденцію кешуватися у браузері і на сервері, тому це потрібно для того, щоб відповідь не була закешованою.

Коли ж і клієнт і сервер відправили свої рукостискання і вони були успішними, починається частина передачі даних – двосторонній зв'язок, по якому кожна зі сторін може надсилати дані незалежно від іншої. [7]

2.2.4.3. Типи повідомлень

Існує декілька типів повідомлень веб-сокетів: текстові дані (UTF-8), двійкові та контрольні повідомлення. Останній тип призначений для сигналізації на рівні протоколу, а не для передачі даних, – ping/pong комбінація, яка може використовуватися для перевірки чи з’єднання досі живе, або повідомлення що з’єднання має бути закритим – close.

Кожне повідомлення складається з одного або декількох кадрів (frames). Кожний такий фрейм одного повідомлення має однаковий тип даних. Всі кадри двійкові, вони мають біти у заголовку, щоб вказати, чи є це останнім фреймом у повідомленні чи ні, і який тип повідомлення це є. Також є біт заголовка, який вказує чи замасковано чи ні це повідомлення. І є біт,

щоб встановити довжину корисної нагрузки, а вона сама займає іншу частину фрейма [7].

Зазвичай ми не помічаємо ці фрейми, ми отримуємо тільки все повідомлення у події, а кадри будуть опрацьовані фоново. Але важливо знати про них, тому що якщо ми будемо відлагоджувати веб-сокет у браузері, ми побачимо саме їх.

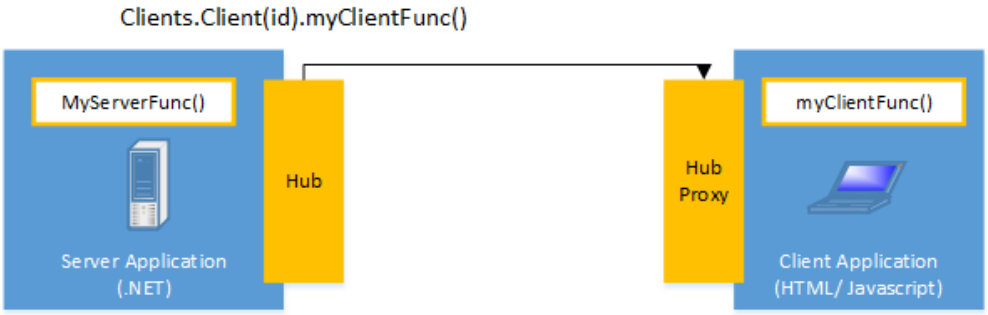
2.3. SignalR

2.3.1. Загальний огляд

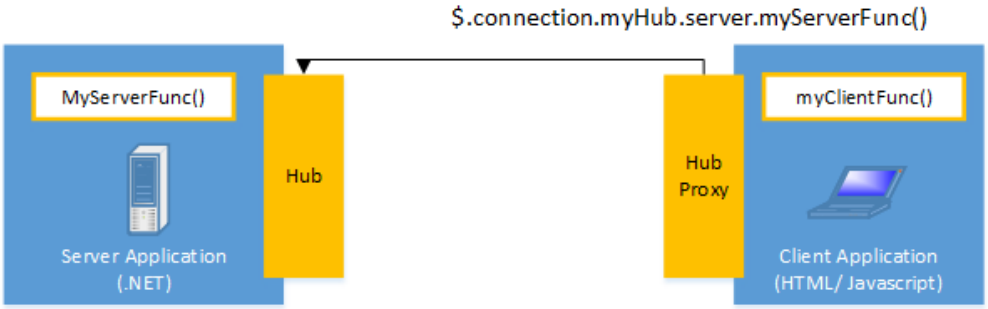
SignalR використовує техніки створення real-time застосунку, описані у попередньому розділі, та обгортає їх у відкриту бібліотеку. Часте опитування, довге опитування, server-sent events і веб-сокети називаються транспортною системою (transports) у даному фреймворку. Він лежить над всіма цими транспортами нижчого рівня. Це дозволяє розробнику зосередитися на бізнес-проблемах, а не на низькорівневих речах.

Дана бібліотека складається із двох частин: одна працює на сервері (у нашому випадку в ASP.NET Core застосунку), а інша – на клієнті. Клієнтом може бути не тільки браузер, а і десктопний застосунок – SignalR пропонує різні клієнтські бібліотеки.

SignalR опрацьовує з'єднання автоматично і надає можливість відправляти повідомлення всім під'єднаним клієнтам із групи одночасно, або якомусь конкретному. Наступна діаграма демонструє виклики клієнтом методу на сервері та сервером на клієнті [8]:



Server invocation of client method
myClientFunc()



Client invocation of server method
MyServerFunc()

2.3.2. Транспортна система

Окрім частого опитування, SignalR підтримує всі вище згадані способи. Так як не кожний браузер зможе підтримувати WebSockets і SSE, то SignalR може домовитися щодо транспортної системи.

Підключення SignalR починається через протокол HTTP, але так як використання веб-сокетів є найефективнішим способом, то буде спроба оновитися до WebSocket. Але якщо вони не підтримуються або сервером або клієнтом, то SignalR спробує використати SSE, але якщо і вони не підтримуються, то тільки тоді він звернеться до довгого опитування. Коли з'єднання буде встановлено, SignalR почне відправляти повідомлення, щоб перевірити чи все нормально, якщо ж ні, то кине помилку.

Так як SignalR є абстракцією над транспортною системою, він дозволяє працювати однаково, незалежно від того, яка система в результаті буде використовуватися.

Але не обов'язково використовувати поведінку SignalR за замовчуванням. Якщо відомі можливості клієнта і сервера, то можна одразу вказати, яку транспортну систему ми хочемо використовувати і за бажанням додати резервні варіанти. [8]

2.3.3. Hubs

API SignalR має дві моделі взаємодії – постійні підключення та концентратори (hubs).

З'єднанням у даному випадку є кінцева точка (endpoint) для відправки отримувачам повідомлень. Persistent Connection API дає прямий доступ до протоколу комунікацій низького рівня, які використовує SignalR.

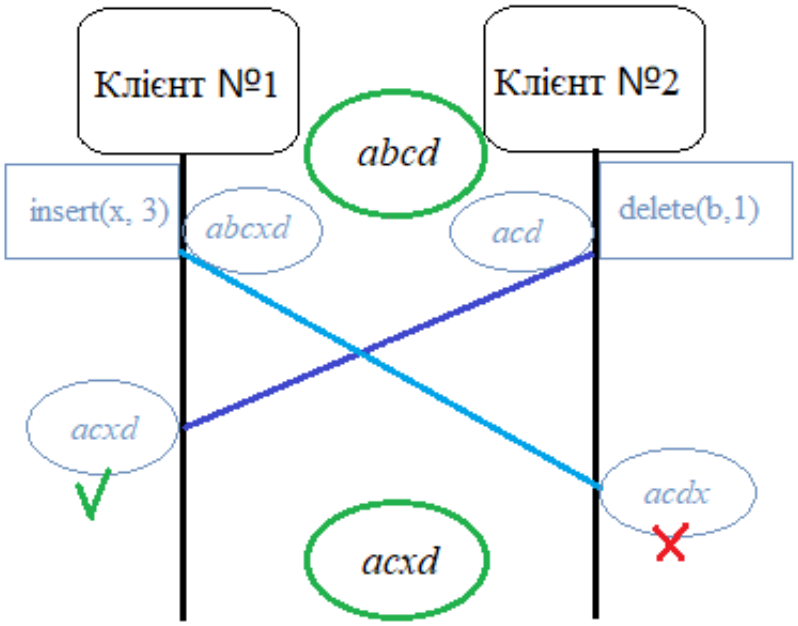
Концентратор (англ. hub) – це серверний клас, який побудований на основі Connection API та дозволяє і клієнту і серверу легко викликати методи на пряму так, наче вони знаходяться локально. Він відправляє повідомлення клієнту та приймає їх від нього, використовуючи RPC. Також використання концентраторів дозволяє передавати у методи строго типізовані параметри.

Коли сервер викликає метод клієнта, через активну транспортну систему відправляється пакет, який має ім'я даного методу та параметри, які передають. Якщо відправляється об'єкт, то він сереалізується використовуючи JSON. Після цього клієнт намагається знайти такий метод з такими ж параметрами у себе, і якщо йому це вдається, метод виконується. [8]

2.4. Операційне перетворення

Керування одночасним редагуванням документа багатьма користувачами доволі складна задача. Може виникнути чимало ситуацій, за яких текст в результаті буде мати відмінності у різних користувачів та на сервері, – так званих колізій.

Наступна діаграма демонструє, як ми можемо зіштовхнутися із вище згаданою проблемою. На сервері та всіх під'єднаних клієнтів вхідними даними є однаковий текст – "*abcd*".



Першою зміною, яка прийшла на сервер було видалення символу b в індексі 1. Клієнт №2 та сервер застосували її. А в цей час Клієнт №1, вставив символ x на позицію з індексом 3. Очевидно, що для другого клієнта це вже буде вставка символу не між "c" і "d", а після "d". Кожний в результаті мав би отримати текст "*acxd*". Але насправді перший клієнт отримає правильний текст, а сервер та другий користувач – "acdx". Зрозуміло, що тут виникає проблема з версіями документу. Її можна вирішити за допомогою операційних перетворень (OT).

ОТ – це алгоритм трансформації операцій таким чином, щоб вони в результаті застосування приводили документи, які відрізняються, в однаковий стан. Кожну зміну документа будемо вважати операцією, яку ми можемо застосувати до нього і після чого отримаємо новий стан документа. А для обробки таких паралельних операцій використовується функція трансформації, яка приймає дві операції, які були застосовані до одного стану документа і на різних клієнтах. В результаті ми отримаємо нову операцію, яку можна застосувати після другої і яка враховує зміни, що передбачені першою [9]. На практиці це буде виглядати наступним чином:

Початок на клієнті -> abcd Локально: insert(‘x’,3) -> abcxd Із серверу: delete(‘b’,1) -> acxd

Початок на сервері -> abcd Локально: delete(‘b’,1) -> acd Із клієнта: insert(‘x’,3) -> трансформується в нову операцію -> insert(‘x’,2) -> acxd

Зазвичай використовують наступне зображення роботи на двох документах одразу:

Тут ліва лінія відповідає операції клієнта, а права – зміні на сервері. Коли лінії клієнта і сервера проходять через одну точку, це значить, що на даний момент часу версія документа у обох синхронізувалася. [10]

Коли до сервера підключено декілька клієнтів, утворюється пара з кожного такого клієнта і сервера, яка має свій власний простір стану документа. Недоліками цього є те, що на сервері має бути місце для кожного під’єднаного клієнта і також ускладнюється трансформація операцій, адже потрібно її застосовувати між всіма цими просторами.

Google модифікував алгоритм ОТ, вимагаючи від клієнта чекати підтвердження його операції від сервера, перед тим як він зможе зробити наступні зміни. Коли сервер визнає операцію, то це означає, що він її трансформував, застосував до своєї копії та повідомив усіх під’єднаних користувачів. А поки клієнт чекає підтвердження, він кешує свої локальні операції і відправляє їх пізніше.

Отже, як видно, операційні перетворення допомагають вирішити проблему одночасного редагування документа без його блокування.

3. Опис реалізації програмного продукту

3.1.Обґрунтування вибору засобів розробки

Для клієнтської частини обрана JavaScript-бібліотека react.js. Вона спрощує створення інтерактивних інтерфейсів та вирішує проблему часткового оновлення сторінки. React.js не покладається на DOM браузера, він підтримує віртуальний DOM, що дозволяє даній бібліотеці вирішувати, як найефективніше оновлювати вміст сторінки.

Основною архітектурною одиницею застосунку, створеного з використанням react.js, є компоненти. Розробник може описати як виглядають різні частини інтерфейсу – створити невеликі компоненти, у яких буде свій внутрішній стан та які можна повторно використати, – а потім за допомогою їх композиції отримати більш складний інтерфейс. Це робить код більш простим для читання і розуміння, передбачуваним та його стає легше налагоджувати. [11]

Для спілкування клієнта із сервером обрано Fetch API. Він дозволяє робити запити схожі на XMLHttpRequest, але відміністю є те, що він використовує Promises і є зручнішим та зрозумілішим. Для того, щоб стоворити запит і отримати дані, використовується метод GlobalFetch.fetch(), який уже реалізований у багатьох інтерфейсах, наприклад у Window. [12]

Серверна частина написана мові програмування C# з використанням фреймворку ASP.NET Core версії 2.1. Даний фреймворк об’єднує у собі ASP.NET MVC і ASP.NET Web Api у єдину модель. Він легко інтегрується з популярними платформами, такими як Angular, React і Blazor, та є кросплатформеним – виконується на Windows, macOS і Linux. [13]

Для реалізації функціоналу в режимі реального часу була використана відкрита бібліотека SignalR. Розглянуті у другому розділі підходи вийшли за рамки звичайної моделі взаємодії «запит-відповідь», у кожного з них є свої переваги та недоліки, тому вибір, який саме спосіб використати залежить від

потреб користувача застосунку і від можливостей клієнта та сервера. Гарним рішенням є суміщення всіх цих технік для роботи у різних випадках, але це складно реалізувати і підтримувати. Тому розробниками Microsoft була створена бібліотека SignalR, яка вже надає це рішення, а також допомагає розробнику зосередитися на проектуванні застосування і вирішенні інших задач.

SignalR має дві частини – клієнтську та серверну. Дана бібліотека пропонує доволі просте API для створення віддалених викликів процедур «клієнт-сервер» та дозволяє керувати з'єднаннями та їх групувати. Вона влаштована так, що при відкритті клієнтського застосунку, відбувається перевірка, які транспортні системи підтримують браузер і сервер, і після чого з'єднання відбувається найбільш оптимальним способом. Однією із таких проблем є редагування документів у реальному часі. Це доволі складна задача, адже необхідно налаштувати систему так, щоб при оновленні тексту всі інші користувачі якнайшвидше дізнавалися про нього та не виникало ситуацій, за яких у них будуть відображатися різні версії документа. [8]

Microsoft Visual Studio [14] – повнофункціональне інтегроване середовище розробки від Microsoft, яке дозволяє розробляти консольні застосунки, програми із графічним інтерфейсом та різні веб-застосування. Даний продукт спрощує процес розробки та відлагодження застосунку, а також надає можливість розробнику використовувати рекомендації IntelliSense для пришвидшення його роботи. Окрім цього надає інструменти для роботи із системою контролю версії Git та для інтеграції із хмарною платформою Azure.

3.2.Опис розробки програми

Дана програма складається з двох окремих частин – сервера та клієнта. ASP.NET Core застосування виступає у ролі сервера, а клієнтом є react.js-застосунок.

Спілкування між клієнтом і сервером для отримання певних даних відбувається за протоколом HTTP з використанням Fetch API. Браузер відправляє запити серверу, наприклад, просячи дістати всі документи користувача, і коли повертається відповідь, змінює дані, які відображаються на екрані. А для реалізації функціоналу в режимі реального часу використано спілкування за протоколом WebSocket. Коли на сервер приходять якісь оновлення від користувача, він повідомляє про них всіх інших з його групи, а клієнтський застосунок відслідковує цю подію та оновлює вміст сторінки.

Клієнтський застосунок написаний за архітектурними принципами react.js[15]. Програма розбита на окремі компоненти, які мають свій внутрішній стан та в результаті за допомогою композиції утворюють більш складний інтерфейс. Коли відбувається якась зміна всередині такого компонента, React ефективно оновлює його та ще раз рендерить.

Серверний застосунок працює як Web API. Для обробки запитів від клієнтів він використовує контролери – класи, які наслідують *ControllerBase*. Кожний такий контролер містить методи, які є обробниками endpoint-ів.

3.3. Створення об'єктів і розробка головної програми

Так як клієнтська частина розроблена за принципами react.js, вона розділена на компоненти, кожний з яких відповідає за певну частину того, що відображається на екрані. Основним файлом є index.js, який містить головний компонент застосунку, який у свою чергу обгортає всі інші, а також метод, який рендерить цей компонент у вказаний контейнер у DOM-і:

```
ReactDOM.render(
  <App />,
  document.getElementById('root'));
```

Головним файлом серверної частини є *Program.cs*, з якого починається її виконання. Тут налаштовується та запускається веб-хост, а також ініціалізується база даних.

```
        public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
            WebHost.CreateDefaultBuilder(args)
                   .UseStartup<Startup>();

        public static void Main(string[] args)
        {
            var host = CreateWebHostBuilder(args).Build();

            using (var scope = host.Services.CreateScope())
            {
                var services = scope.ServiceProvider;

                try
                {
                     var context =
                            services.GetRequiredService<ManageYourDocsContext>();
                    SampleData.Initialize(context);
                }
                catch (Exception ex)
                {
                    var logger = services.GetRequiredService<ILogger<Program>>();
                    logger.LogError(ex, "An error occurred seeding the DB.");
                }
            }
            host.Run();
       }
```

Також важливим файлом є *Startup.cs*, який визначає клас *Startup* та описує логіку обробки запитів. Тут відбувається конфігурація застосунку та налаштування сервісів, які будуть надалі використовуватися. Даний клас визначає методи *Configure()* та *ConfigureServices()*. У даному застосунку в останньому методі реєструються сервіси SignalR, аутентифікації та DBContext:

```
public void ConfigureServices(IServiceCollection services)
{
            services.AddCors();

            string connection =
Configuration.GetConnectionString("DefaultConnection");
            services.AddDbContext<ManageYourDocsContext>(options =>
options.UseSqlServer(connection));

            services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
                .AddJwtBearer(options =>
                {
                    ...
                });
            ...
            services.AddSignalR();
            services.AddSpaStaticFiles(configuration =>
            {
                configuration.RootPath = "ClientApp/build";
            });
}
```

Взагалі архітектура даного застосування побудована на основі багаторівневого шаблону проектування [16]. Можна виділити чотири рівні, кожний з яких виконує певну роль. Клієнтський застосунок визначає рівень представлення – він відповідальний за обробку всього інтерфейсу користувача та за спілкування браузера із сервером.

Серверна програма складається із двох рівнів. Рівень бізнес-логіки відповідає за обробку запитів від клієнта та реалізує логіку застосунку. Він складається із контролерів, які обробляють запити за протоколом HTTP, сервісів та концентратору *ManageYourDocsHub*. У даній програмі реалізовано три контролери: для авторизації та реєстрації, для запитів, що стосуються користувачів та окремо для документів. Клас *ManageYourDocsHub* відповідає за організацію функціоналу в режимі реального часу та наслідує базовий клас *Hub* із бібліотеки SignalR. Він визначає методи, які відпрацьовують при під’єднанні користувача до групи (коли він відкриває документ) та його від’єднанні. Також тут визначенні методи для отримання повідомлень від клієнта про зміну тексту документа або його назви. Клас *DocumentService* реалізує логіку підтримки локального стану документів. Рівень доступу до даних складається із моделей даних, які є кодовим представленням бази,

класу контексту, який налаштовує базу, та із класів, які визначають методи для роботи із даними. Його відповідальність – це зв'язок із базою даних.

Останнім рівнем є база даних.

3.4. Опис файлів даних та інтерфейсу програми

База даних створювалася за підходом Code First за допомогою EF Core[17]. Спочатку були створенні всі необхідні моделі, а далі був налаштований контекст даних, який відповідає за створення схем у базі даних та зв'язок із нею. Було створено три моделі. Першою є модель користувача системи, яка визначається наступним чином:

```
        [Key]
        public Guid Guid { get; set; }
        [Required]
        public string Name { get; set; }
        [Required]
        public string Surname { get; set; }
        [Required]
        public string Password { get; private set; }
        [EmailAddress]
        [Required]
        public string Email { get; set; }

        public ICollection<Document> Documents { get; set; }
        public ICollection<SharedDocument> SharedDocuments { get; set; }
```

Друга – модель документа:

```
        [Key]
        public Guid Guid { get; set; }

        [Required]
        [StringLength(100)]
        public string Title { get; set; }

        [Required]
        public string Extension { get; set; }
        [Required]
        public DateTime CreationDateTime { get; set; }
        [Required]
        public DateTime LastChangedDateTime { get; set; }
        public string Content { get; set; }

        [Required]
        public Guid OwnerGuid { get; private set; }
        public virtual User Owner { get; set; }
```

Третя модель відображає зв'язуючу реляцію між користувачем і документом, яким з ним поділилися:

```
        [Key]
        public Guid UserGuid { get; set; }
        public User User { get; set; }

        [Key]
        public Guid DocumentGuid { get; set; }
        public Document Document { get; set; }
```

Все спілкування з базою даних відбувається через контекст – клас *ManageYourDocsContext*, який наслідує базовий клас EF Core *DbContext*. Він визначає набір сутностей, які будуть зберігатися у базі даних:

```
        public DbSet<User> Users { get; set; }
        public DbSet<Document> Documents { get; set; }
        public DbSet<SharedDocument> SharedDocuments { get; set; }
```

Як вже було сказано, для спілкування клієнта із сервером було використано Fetch API. Приклад такого запиту для поширення документа виглядає наступним чином:

```
return fetch(`/api/document/${props.doc.guid}/share`, {
            method: "POST",
            headers: {
                'Content-Type': 'application/json',
                Accept: 'application/json',
                'Authorization': `Bearer ${token}`
            },
            body: JSON.stringify(email),
})
```

Також у застосунку передбачена обробка помилок та виведення відповідних повідомлень користувачу. Наприклад, коли користувач хоче поділитися документом з кимось іншим, спочатку йде перевірка, чи введена пошта, якщо ж ні, то запит не відправляється. В іншому випадку на сервер приходить запит і він перевіряє, чи існує у базі даних користувач із вказаною поштою, чи користувач не ділиться документом із самим собою та чи у іншого користувача уже є доступ до нього. В іншому випадку створюється

відповідний запис у базі даних та сервер повертає відповідь зі кодом статусу 200:

```
        [HttpPost("{id}/share")]
        public ActionResult ShareDocument([FromRoute] Guid id, [FromBody] string
userEmail)
        {
            var ownerId = Guid.Parse(HttpContext.User.Identity.Name);
            var userToShare = _userRepository.GetSingle(u => u.Email == userEmail);

            if (userToShare == null)
                return BadRequest(new { message = "Такого користувача не знайдено!"
});

            if (userToShare.Guid == ownerId)
                return BadRequest(new { message = "Ви не можете поділитися з собою
своїм документом :)" });

            var document = _docRepository.GetSingle(d => d.Guid == id, d =>
d.SharedDocuments);

            var existingShare = document.SharedDocuments.ToList().Find(sd =>
sd.UserGuid == userToShare.Guid);
            if (existingShare == null)
            {
                _shareRepository.Add(new SharedDocument
                {
                    UserGuid = userToShare.Guid,
                    DocumentGuid = id
                });
                _shareRepository.Commit();

                return new OkResult();
            }
            return new ConflictResult();
       }
```

Коли клієнт отримує відповідь, він перевіряє статус коду та відображає користувачу відповідні повідомлення:

```
        .then(response => {
                if (response.ok) {
                    props.setMessage({
                        text: `Тепер ${email} може переглядати та редагувати ваш
документ.`,
                        severity: 'success'
                    });
                } else if (response.status == 409) {
                    props.setMessage({
                        text: `Ви вже поділилися цим документом з ${email}.`,
                        severity: 'info'
                    });
                } else if (response.status == 400) {
                    response.json()
                        .then(data => {
                            props.setMessage({
                                text: data.message,
                                severity: 'error'
                            });
                        })
                }
        })
```

Для реалізації функціоналу в режимі реального часу на сервері було створено додатковий сервіс, який зберігає ідентифікатор з'єднання кожного користувача та його групу, а також поточний стан відповідного документу. Коли відбуваються зміни у тексті документа, вони застосовуються до локальної версії на сервері. Оновлення документу у базі даних відбувається автоматично, коли останній користувач із групи від'єднується від неї, або коли він натискає на кнопку збереження.

Коли користувач відкриває документ, з клієнта відправляється запит на отримання його інформації. Спочатку дістається локальна версія документу, і якщо цей документ не був раніше відкритий іншим користувачем, відповідно немає його локальної версії, то він завантажується із бази даних і створюється його локальна копія на сервері. Для уникнення створення декількох копій одного документа у ситуації, коли декілька користувачів одночасно відкривають його, було використано блокування доступу до списку відкритих документів. Але так як у випадку, коли документ уже відкритий, блокування буде зайвою тратою ресурсів, використано блокування з подвійною перевіркою (double checked locking) [18]:

```
        [HttpGet("{id}")]
        public ActionResult GetDocument([FromRoute] Guid id)
        {
            var doc = _documentService.GetDocument(id);
            if (doc == null)
            {
                lock (_documentService.SyncDocuments)
                {
                    doc = _documentService.GetDocument(id);
                    if (doc == null)
                    {
                        doc = _docRepository.GetSingle(id);
                        _documentService.AddDocument(doc);
                    }

                }
            }
            return new JsonResult(doc);
        }
```

Також при відкритті документа створюється об'єкт *HubConnectionBuilder* із клієнтської бібліотеки SignalR – @microsoft/signalr:

```
    useEffect(() => {
        props.getDocument(params.docId)
            .then(data => {
                setDoc(data);
                setTitle(data.title)
            });

        const connection = new signalR.HubConnectionBuilder()
            .withUrl("/hub")
            .build();

        setHubConnection(connection);
    }, []);
```

Для нього вказується URL, який був раніше прописаний на сервері у методі класу Startup Configure():

```
app.UseSignalR(routes =>
    {
                routes.MapHub<ManageYourDocsHub>("/hub");
    });
```

Після того як з'єднання було створено та встановлено, викликається метод із сервера JoinGroup(string group), де у даному випадку назвою групи є ідентифікатор документа. Також прив'язуються обробники повідомлень із сервера для отримання змін у тексті та оновлення назви документа:

```
hubConnection
    .start()
    .then(() => hubConnection.invoke("JoinGroup", doc.guid)
        .catch(err => console.error(err)))
    .catch(err => console.log('Error while establishing connection :('));

hubConnection.on('ReceiveOperation', (operation) => receive(operation));
hubConnection.on('ReceiveNewTitle', (title) => receiveTitle(title));
```

Коли користувач змінює назву документа, на сервері викликається метод *ChangeTitle(string title)*, який приймає нову назву. У даному випадку оновлення розсилається всім під'єднаним клієнтам, зберігаються зміни локально на сервері та одразу у базі даних:

```
        public async Task ChangeTitle(string title)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                var task = Clients.OthersInGroup(group).SendAsync("ReceiveNewTitle",
title);
                var doc = _documentService.ChangeTitle(group, title);
                _docRepository.Update(doc);
                _docRepository.Commit();

                await task;
            }
       }
```

Оновлення тексту документа відбувається складніше. На клієнті відслідковуються його зміни та при кожному оновленні викликається метод *ChangeText(OperationModel operation)* на сервері. Кожна така зміна вважається операцією вставки або видалення. Для цього була створена додаткова модель, яка містить відповідну інформацію:

```
    public class OperationModel
    {
        [JsonConverter(typeof(StringEnumConverter))]
        public Operations Operation { get; set; }
        public int StartIndex { get; set; }
        public int EndIndex { get; set; }
        public string Text { get; set; }
    }

    public enum Operations
    {
        Insert,
        Delete
    }
```

Спочатку відбувається повідомлення всіх інших клієнтів із групи про оновлення. Далі зміна застосовується до локальної версії документа на сервері без збереження у базі даних. Тут використано блокування локального документа для того, щоб уникнути ситуації, коли одразу декілька користувачів вносять зміни та виникає колізія:

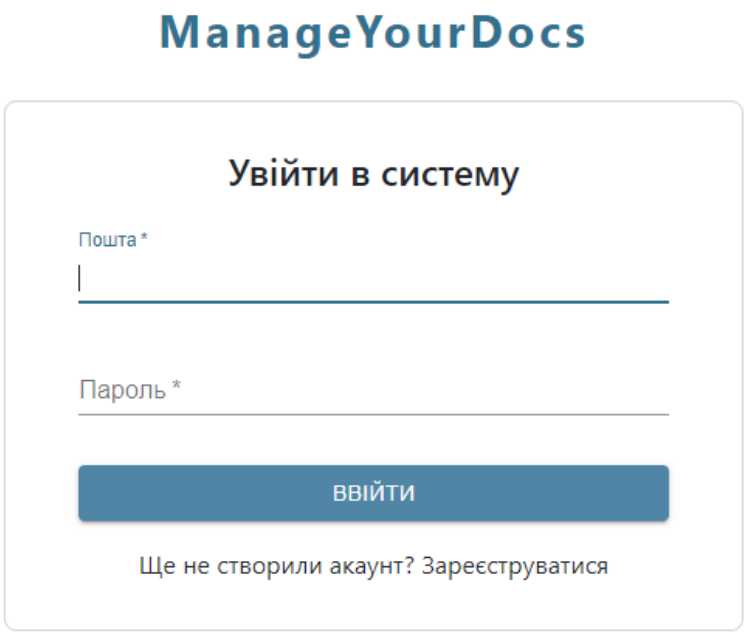
```
        public async Task ChangeText(OperationModel operation)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                var task = Clients.OthersInGroup(group).SendAsync("ReceiveOperation",
operation);

                var document = _documentService.GetDocument(Guid.Parse(group));
                lock (document)
                {
                    var newContent = OT.ApplyOperation(document, operation);
                    if (document.Content != newContent)
                    {
                        _documentService.ChangeText(group, newContent);
                    }
                }
                await task;
            }
        }
```

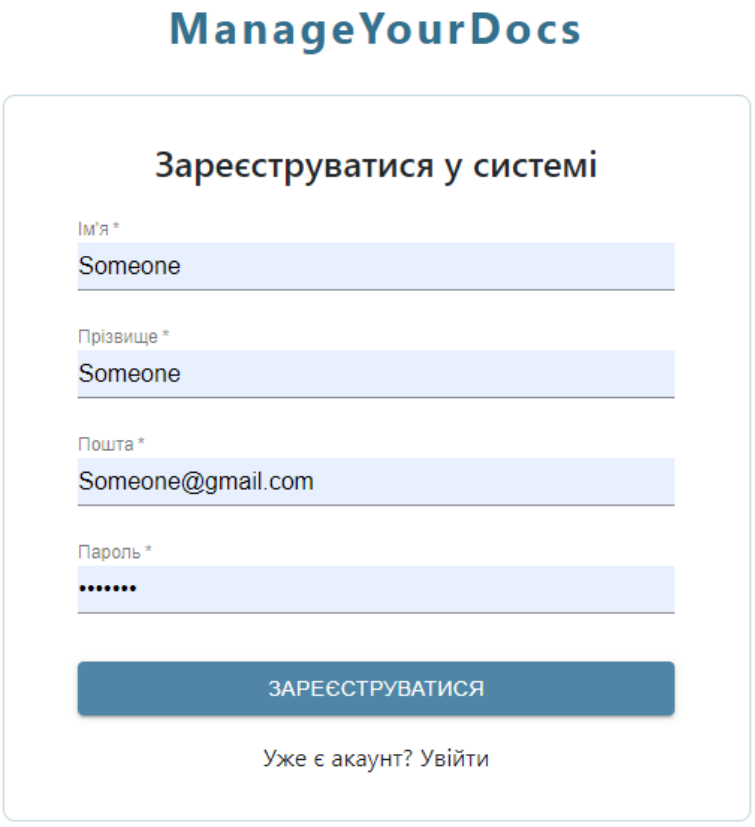
Всі інші клієнти відслідковують зміну, яка приходить із сервера, та відповідно оновлюють свої локальні версії. Важливим моментом є збереження позиції курсора, саме тому у кожного клієнта вона локально зберігається та після застосування змін відповідно зсовується. Для керування курсором встановлено вказівник на textarea та при зміні оновлюються його значення для *selectionStart* і *selectionEnd*.

```
    const changeCursorPosition = (idx) => {
        textareaRef.current.selectionStart = idx.start;
        textareaRef.current.selectionEnd = idx.end;
        textareaRef.current.focus();
    }
```

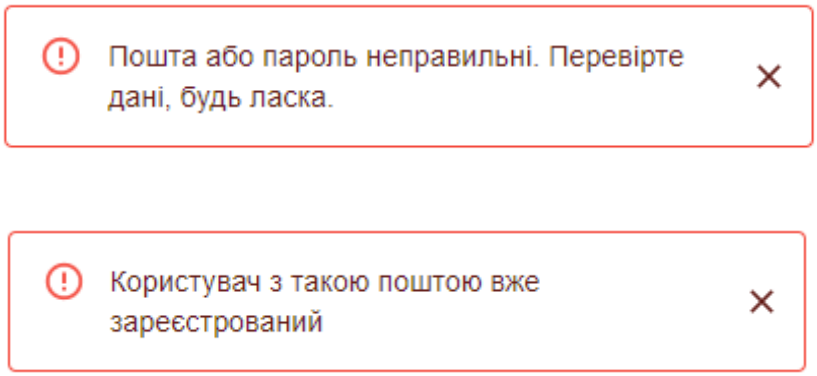
3.5. Тестування програми і результати її виконання

Щоб отримати доступ до основного функціоналу застосунку, користувачу спочатку треба ввійти у систему або зареєструватися. Так виглядає сторінка входу та реєстрації:

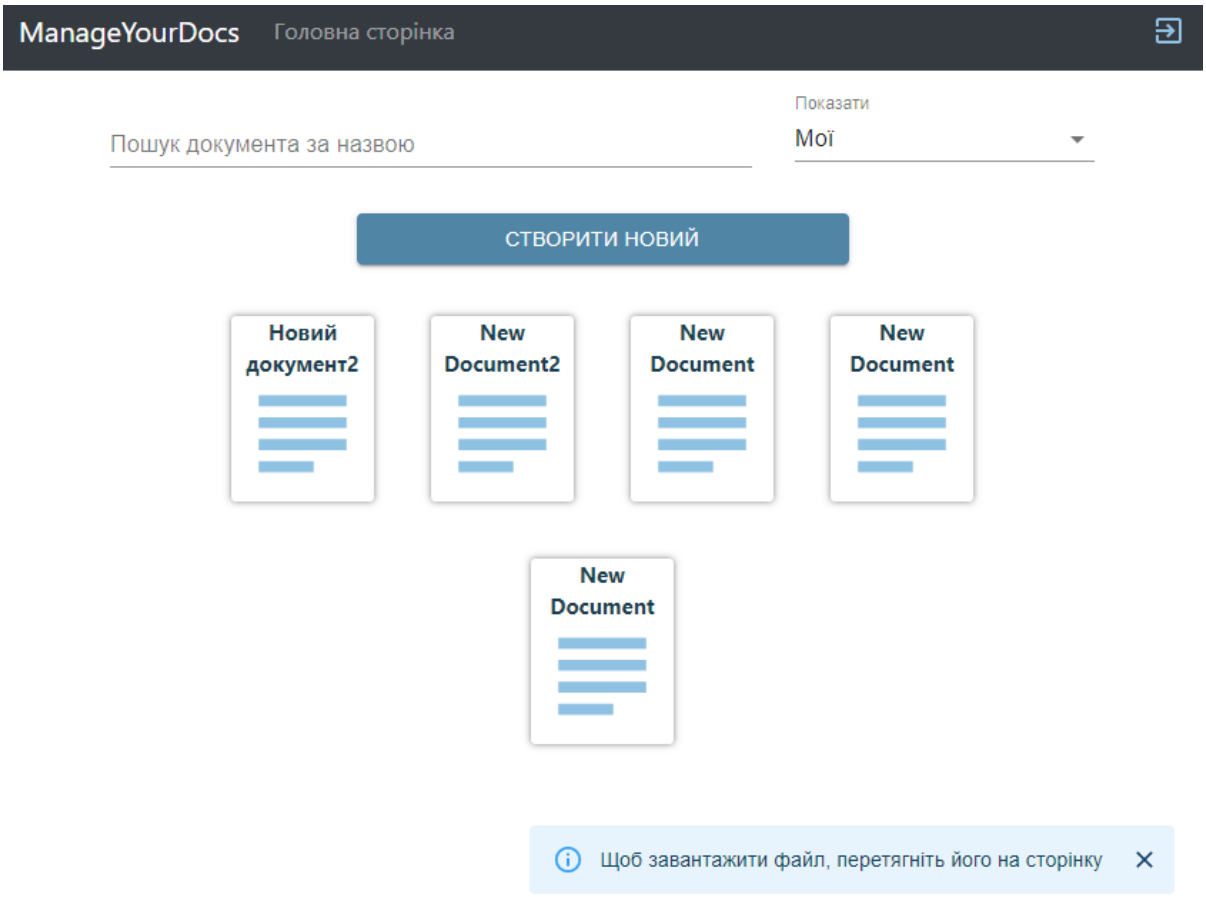




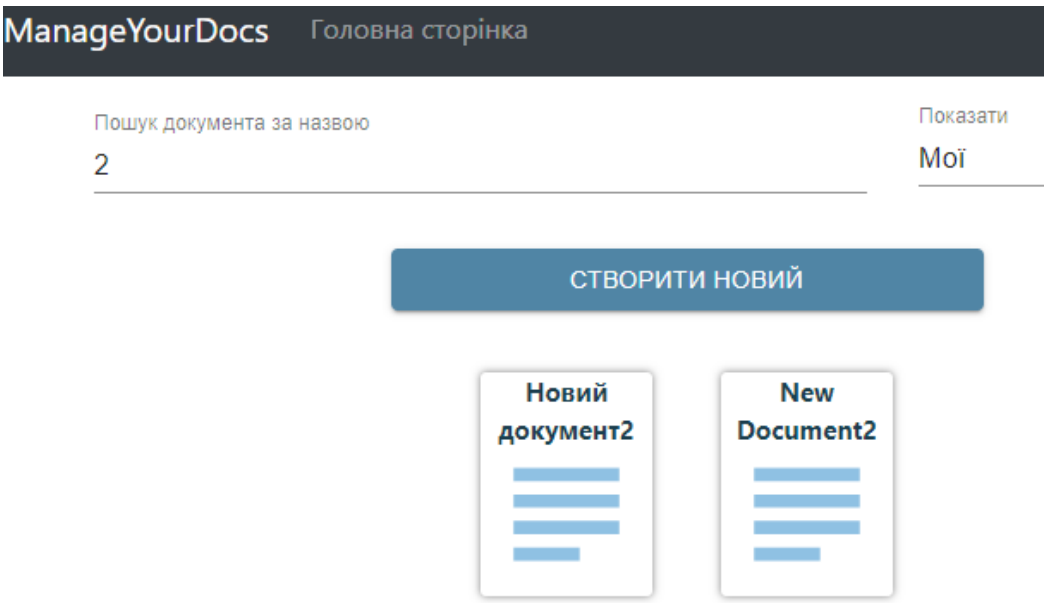
Якщо ж користувач введе неправильні дані, система його про це повідомить:



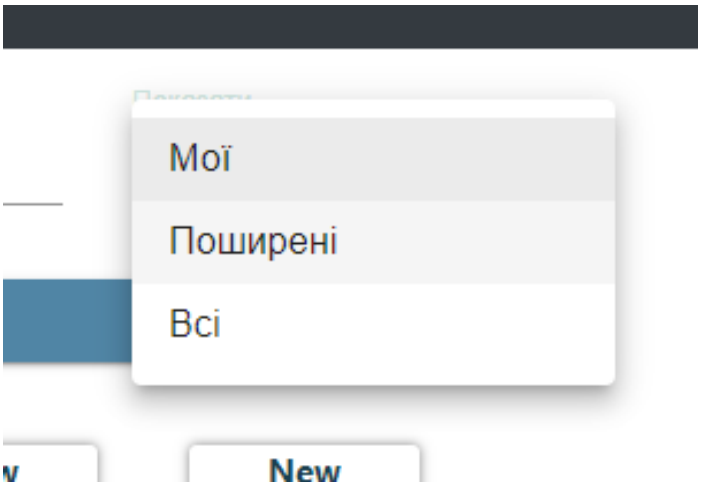
Після успішної аутентифікації, користувач буде перенаправлений на головну сторінку. Тут він може вибрати список документів, які хоче переглянути – тільки його, якими з ним поділилися та всі йому доступні. А також може знайти документ за його назвою. Окрім цього, користувачу висвічується підказка, як можна завантажити вже існуючий файл. Відкрити документ можна натиснувши на нього.



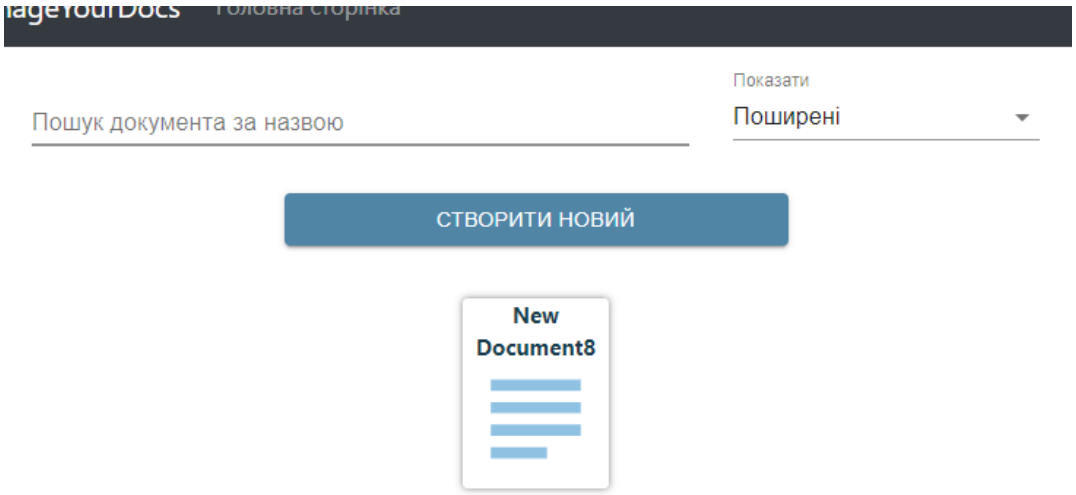
Для пошуку документа необхідно ввести частину його назви у відповідне поле:



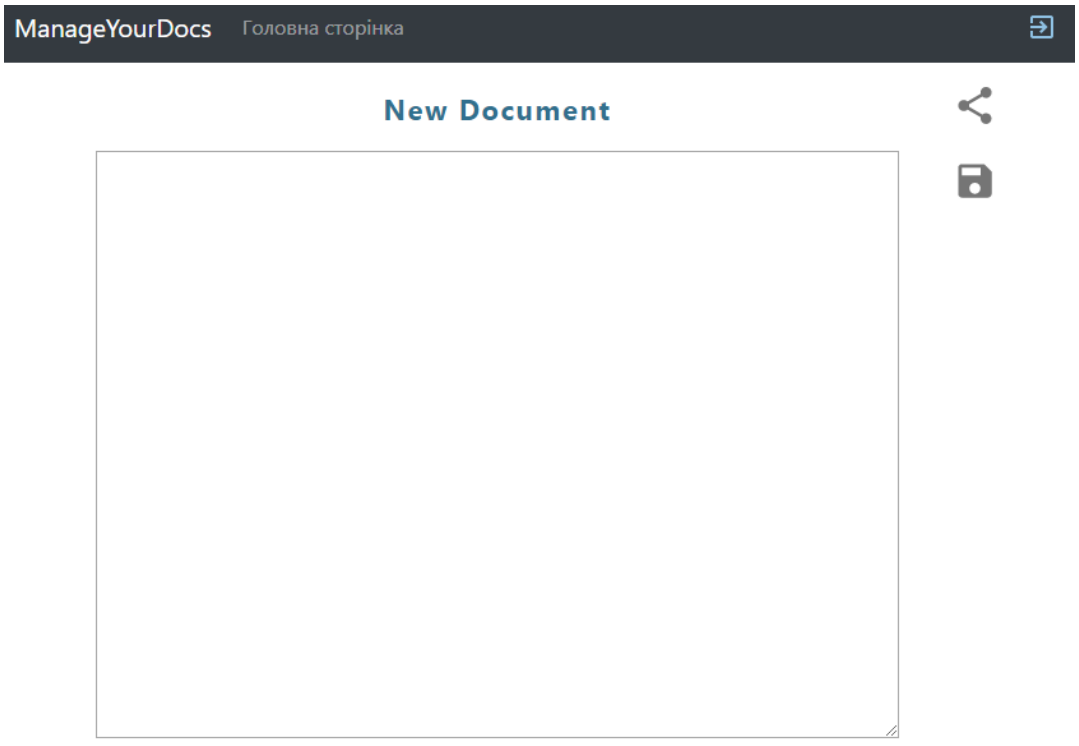
Якщо користувач хоче переглянути документи, якими з ним поділилися, необхідно вибрати відповідну опцію у даному списку:



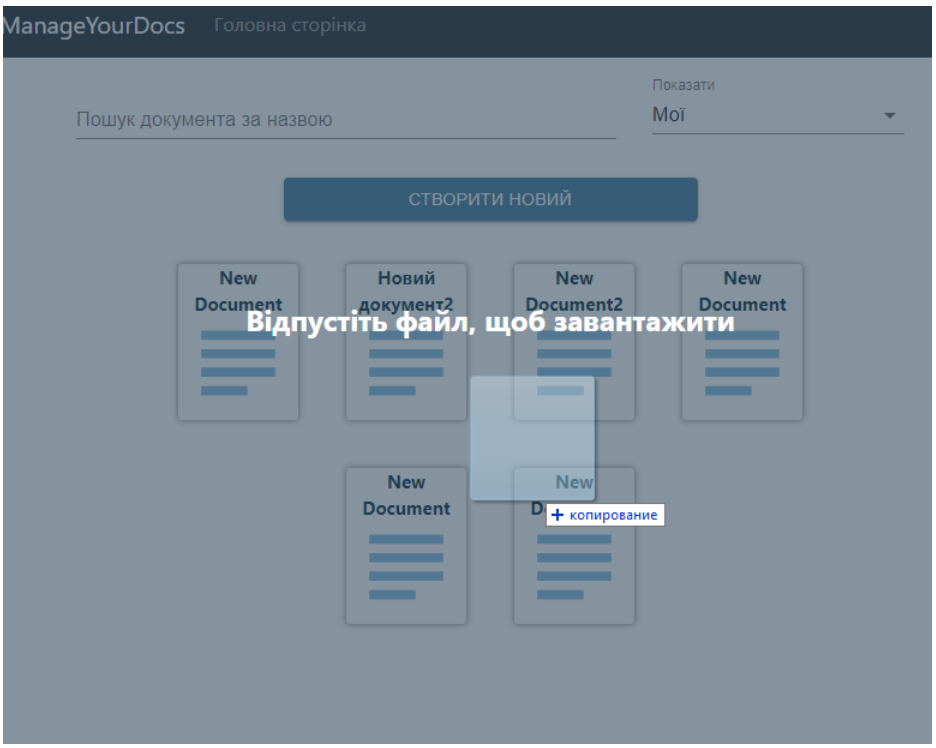
Після чого на сторінці будуть відображені тільки документи, якими з ним поділилися:



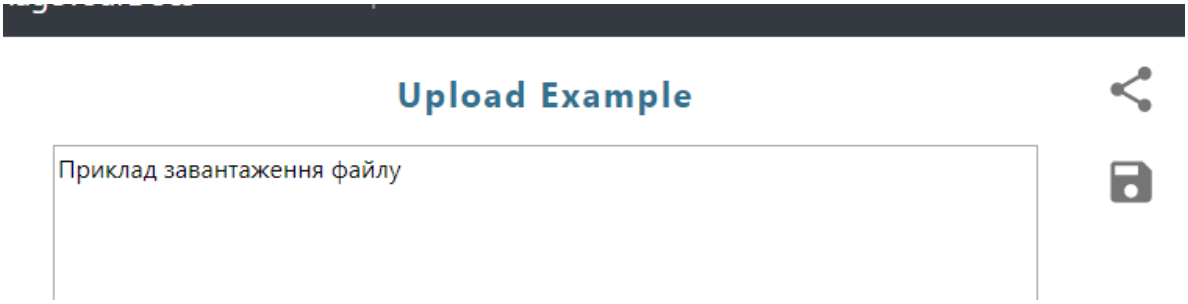
Для того, щоб створити новий пустий документ, потрібно натиснути на кнопку «Створити новий», після чого користувача буде перенаправлено на сторінку цього документа. Документ отримає назву “New Document” та не буде мати тексту:



Для того, щоб завантажити файл необхідно перетягнути його на головну сторінку та відпустити:



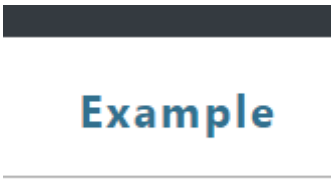
Після цього буде відкрито завантажений документ:



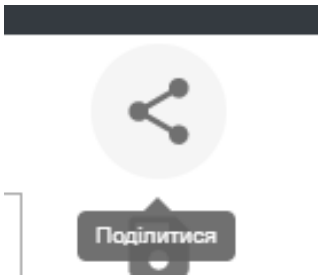
На сторінці документа користувач може відредагувати його текст, назву та зберегти відповідні зміни. При наведенні на назву документа з’являється наступна підказка:

Після її виконання можна буде змінити назву:

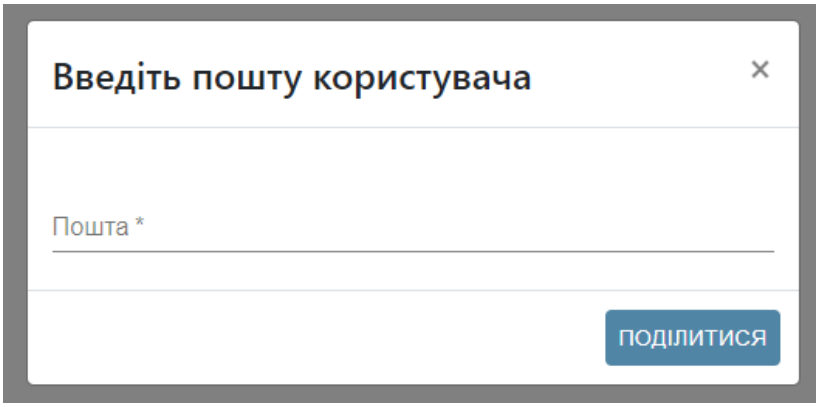
Для збереження змін потрібно натиснути за межами поля:



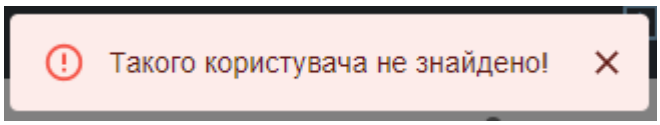
Для того, щоб декілька користувачів могли редагувати документ одночасно, його власнику треба ним поділитися. Для цього необхідно натиснути на наступну іконку:

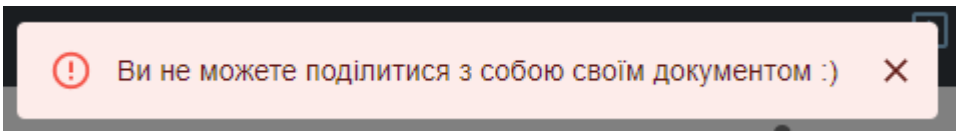


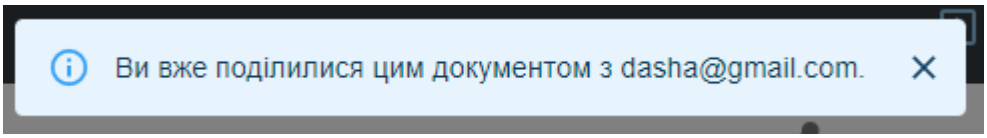
Після чого з'явиться модальне вікно, в якому потрібно ввести пошту користувача:

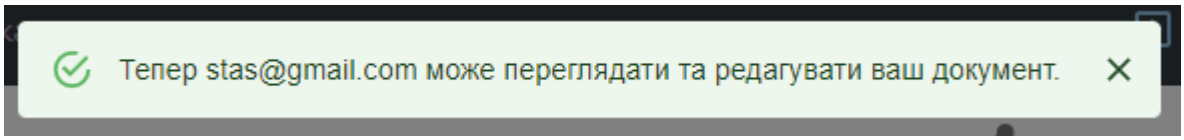


Після того, як користувач ввів пошту та натиснув «Поділитися», буде виведено повідомлення про результат дії:



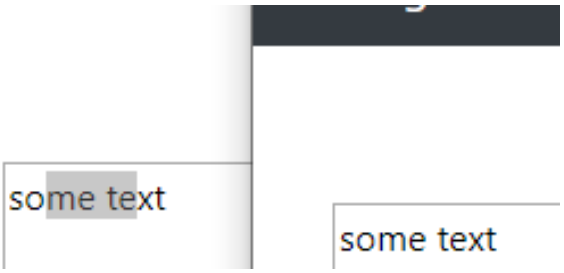




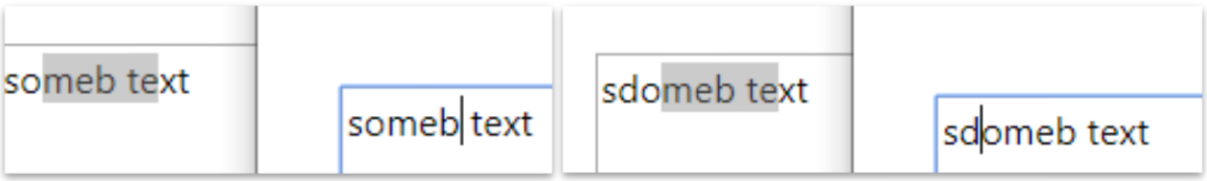


Якщо користувач отримав доступ до документу, він зможе переглядати та редагувати його разом із власником та іншими користувачами, у яких доступ вже є.

Далі показано збереження позиції курсора та відповідно і виділення при одночасному редагуванні. У двох користувачів відкритий однаковий документ із вхідним текстом «some text». Перший скріншот показує початкове виділення:



На наступних двох видно, що після вставки символу у межах виділення або друку перед ним, воно зберігається коректно:



Висновок

Отже, мета роботи була досягнута і розроблений проект відповідає вимогам, описаним у постановці завдання. Було детально розглянуто методи реалізації функціоналу в режимі реального часу та наведено недоліки та переваги кожного з них. Окрім цього, були описані основи алгоритму ОТ та проблеми, які він допомагає вирішити.

Наразі можливе редагування тільки звичайного тексту, але в подальшому функціональні можливості системи можна розширити за рахунок підтримки тексту з форматуванням. Незважаючи на це, даним сервісом можна користуватися як повноцінним веб-застосуванням.

Список використаної літератури

1. Головний сайт Collabedit [Електронний ресурс]: http://collabedit.com/
2. Головний сайт Google Docs [Електронний ресурс]: https://docs.google.com/
3. Головний сайт Confluence [Електронний ресурс]: https://www.atlassian.com/ru/software/confluence
4. Long Polling [Електронний ресурс]: https://javascript.info/long-polling
5. Using server-sent events [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
6. Server Sent Events [Електронний ресурс]: https://javascript.info/server-sent-events
7. Ian Fette & Alexey Melnikov, “The WebSocket Protocol”, RFC 6455, December 2011
8. Introduction to SignalR [Електронний ресурс]: https://docs.microsoft.com/en-gb/aspnet/signalr/overview/getting-started/introduction-to-signalr
9. Operational Transformation [Електронний ресурс]: http://operational-transformation.github.io/index.html
10. Understanding and Applying Operational Transformation [Електронний ресурс]: http://www.codecommit.com/blog/java/understanding-and-applying-operational-transformation
11. React [Електронний ресурс]: https://reactjs.org/
12. Fetch API [Електронний ресурс]: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/
13. What's new in ASP.NET Core 2.1 [Електронний ресурс]: https://docs.microsoft.com/en-gb/aspnet/core/release-notes/aspnetcore-2.1?view=aspnetcore-2.1
14. Visual Studio 2019 [Електронний ресурс]: https://visualstudio.microsoft.com/vs/
15. Thinking in React [Електронний ресурс]: https://reactjs.org/docs/thinking-in-react.html

16. Richards M., Software Architecture Patterns. O'Reilly Media, Inc., 2015. 54p.
17. Creating a Database with Code First in EF Core [Електронний ресурс]: https://docs.oracle.com/cd/E17952_01/connector-net-en/connector-net-entityframework-core-example.html
18. Lea Doug, "Concurrent Programming in Java", Second Edition, Addison-Wesley, 1999. 432 p.

ДОДАТОК А. Клієнтський код – компонент документа

```
import React, { useState, useEffect, useRef, useCallback } from 'react';
import { connect } from 'react-redux';

import * as signalR from "@microsoft/signalr";

import { Container, Row, Col } from 'react-bootstrap'
import Snackbar from '@material-ui/core/Snackbar';
import Alert from '@material-ui/lab/Alert';
import Layout from "../Layout.jsx";
import { useStyles } from "./documentStyles";
import { getDocument } from "../../store/actions/documentActions";
import { Operations, Insert, Delete } from '../../utils/OT';
import { DocumentMenu } from "./DocumentMenu.jsx";
import { DocumentTitle } from "./documentTitle.jsx";

const Document = props => {
    const { match: { params } } = props;

    const [doc, setDoc] = useState(null);
    const [title, setTitle] = useState("");
    const [hubConnection, setHubConnection] = useState(null);
    const [content, setContent] = useState("");
    const [message, setMessage] = useState(null);
    const [cursor, setCursor] = useState({start: 0, end: 0});

    const contentRef = useRef();
    const textareaRef = useRef();
    const cursorRef = useRef();

    useEffect(() => {
        contentRef.current = content;
        cursorRef.current = cursor;
    });

    useEffect(() => {
        props.getDocument(params.docId)
            .then(data => {
                setDoc(data);
                setTitle(data.title)
            });

        const connection = new signalR.HubConnectionBuilder()
            .withUrl("/hub")
            .build();

        setHubConnection(connection);
    }, []);

    const changeCursorPosition = (idx) => {
        textareaRef.current.selectionStart = idx.start;
        textareaRef.current.selectionEnd = idx.end;
        textareaRef.current.focus();
    }

    const receive = useCallback((res) => {
        if (res.operation == Operations.Insert) {
            var result = Insert(contentRef.current, res.startIndex, res.endIndex,
res.text);
            setContent(result);
            if (cursorRef.current.start >= res.startIndex) {
                var a = {
                    start: cursorRef.current.start + res.text.length,
```

```
                    end: cursorRef.current.end + res.text.length
                };
                setCursor(a);
                changeCursorPosition(a);
            } else if (cursorRef.current.start < res.startIndex && res.startIndex <
cursorRef.current.end) {
                var a = {
                    start: cursorRef.current.start,
                    end: cursorRef.current.end - (res.endIndex - res.startIndex) +
res.text.length
                };
                setCursor(a);
                changeCursorPosition(a);
            } else {
                changeCursorPosition(cursorRef.current);
            }
        } else if (res.operation == Operations.Delete) {
            var result = Delete(contentRef.current, res.startIndex, res.endIndex);
            setContent(result);
            if (cursorRef.current.start > res.startIndex) {
                var a = {
                    start: cursorRef.current.start - 1,
                    end: cursorRef.current.end - 1
                };
                setCursor(a);
                changeCursorPosition(a);
            } else if (cursorRef.current.start <= res.startIndex && res.startIndex <
cursorRef.current.end) {
                var a = {
                    start: cursorRef.current.start,
                    end: cursorRef.current.end - (res.endIndex - res.startIndex)
                };
                setCursor(a);
                changeCursorPosition(a);
            } else {
                changeCursorPosition(cursorRef.current);
            }
        }

    }, [contentRef, cursorRef]);

    const receiveTitle = useCallback((title) => {
        setTitle(title);
    }, [title]);

    useEffect(() => {
        if (hubConnection !== null && doc !== null) {
            setContent(doc.content);

            hubConnection
                .start()
                .then(() => hubConnection.invoke("JoinGroup", doc.guid)
                    .catch(err => console.error(err)))
                .catch(err => console.log('Error while establishing connection :('));

            hubConnection.on('ReceiveOperation', (operation) => receive(operation));
            hubConnection.on('ReceiveNewTitle', (title) => receiveTitle(title));
        }
    }, [hubConnection, doc]);

    const onSelect = (e) => {
        setCursor({ start: e.target.selectionStart, end: e.target.selectionEnd});
    }
```

```
    const onChange = (e) => {
        setContent(e.target.value);
        e.persist();
        console.log({ e });

        var inputType = e.nativeEvent.inputType;
        console.log(inputType);

        const cursor = cursorRef.current;

        if (inputType == "insertText") {
            const data = e.nativeEvent.data;
            hubConnection
                .invoke("ChangeText", {
                    operation: Operations.Insert,
                    startIndex: cursor.start,
                    endIndex: cursor.end,
                    text: data
                })
                .catch(err => console.error(err));
        } else if (inputType == "insertFromPaste") {
            return;
        } else if (inputType == "deleteContentBackward") {
            hubConnection
                .invoke("ChangeText", {
                    operation: Operations.Delete,
                    startIndex: cursor.start == cursor.end ? cursor.start - 1 :
cursor.start,
                    endIndex: cursor.end,
                    text: ""
                })
                .catch(err => console.error(err));
        } else if (inputType == "deleteContentForward") {
            hubConnection
                .invoke("ChangeText", {
                    operation: Operations.Delete,
                    startIndex: cursor.start,
                    endIndex: cursor.end + 1,
                    text: ""
                })
                .catch(err => console.error(err));
        }
    }

    const onPaste = (e) => {
        setContent(e.target.value);

        const data = (e.clipboardData || window.clipboardData).getData('text');
        hubConnection
            .invoke("ChangeText", {
                operation: Operations.Insert,
                startIndex: cursor.start,
                endIndex: cursor.end,
                text: data
            })
            .catch(err => console.error(err));
    }

    const changeTitle = () => {
        hubConnection
            .invoke("ChangeTitle", title)
            .catch(err => console.error(err));
    }
```

```
    return (
        <Layout>
            <Container>
                <Row>
                    <Col>
                        <DocumentTitle title={title} changeTitle={changeTitle}
                            setTitle={(title) => setTitle(title)} />

                        <textarea ref={textareaRef} className={useStyles().textarea}
                            value={content}
                            onSelect={(e) => onSelect(e)}
                            onChange={(e) => onChange(e)} onPaste={e =>
onPaste(e)}></textarea>
                    </Col>
                    <Col xs={12} md={1}>
                        <DocumentMenu setMessage={(data) => setMessage(data)} doc={doc}/>
                    </Col>
                </Row>
            </Container>

            {message != null &&
                <Snackbar
                    open={true}
                    onClose={() => setMessage(null)}
                    anchorOrigin={{ vertical: 'top', horizontal: 'right' }}
                    autoHideDuration={6000}>
                    <Alert onClose={() => setMessage(null)} severity={message.severity}>
                        {message.text}
                    </Alert>
                </Snackbar>
            }
        </Layout>
    )
}

const mapStateToProps = (state) => {
    return {
    };
};

const mapDispatchToProps = (dispatch) => {
    return {
        getDocument: (docId) => dispatch(getDocument(docId)),
    };
};

export default connect(mapStateToProps, mapDispatchToProps)(Document);
```

ДОДАТОК Б. Серверний код – концентратор ManageYourDocsHub

```
using ManageYourDocs.Data.Abstract;
using ManageYourDocs.Models;
using ManageYourDocs.Services.Abstract;
using ManageYourDocs.Tools;
using Microsoft.AspNetCore.SignalR;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace ManageYourDocs
{
    public class ManageYourDocsHub : Hub
    {
        IDocumentRepository _docRepository;
        IDocumentService _documentService;

        public ManageYourDocsHub(IDocumentService documentService, IDocumentRepository
docRepository)
        {
            _documentService = documentService;
            _docRepository = docRepository;
        }

        public override async Task OnConnectedAsync()
        {
            await base.OnConnectedAsync();
        }

        public override async Task OnDisconnectedAsync(Exception exception)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                await Groups.RemoveFromGroupAsync(Context.ConnectionId, group);
                _documentService.RemoveConnectionGroup(Context.ConnectionId);

                if (!_documentService.connectionGroups.Values.Any(v => v == group))
                {
                    var doc = _documentService.documents.FirstOrDefault(d => d.Guid ==
Guid.Parse(group));
                    _docRepository.Update(doc);
                    _docRepository.Commit();
                    _documentService.RemoveDocument(doc);
                }
            }

            await base.OnDisconnectedAsync(exception);
        }

        public async Task ChangeText(OperationModel operation)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                var task = Clients.OthersInGroup(group).SendAsync("ReceiveOperation",
operation);

                var document = _documentService.GetDocument(Guid.Parse(group));
                lock (document)
                {
                    var newContent = OT.ApplyOperation(document.Content, operation);
                    if (document.Content != newContent)
```

```
                    {
                        _documentService.ChangeText(group, newContent);
                    }
                }

                await task;
            }
        }

        public async Task ChangeTitle(string title)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                var task = Clients.OthersInGroup(group).SendAsync("ReceiveNewTitle",
title);
                var doc = _documentService.ChangeTitle(group, title);
                _docRepository.Update(doc);
                _docRepository.Commit();

                await task;
            }
        }

        public async Task JoinGroup(string groupName)
        {
            var group = _documentService.GetConnectionGroup(Context.ConnectionId);
            if (group != null)
            {
                await Groups.RemoveFromGroupAsync(Context.ConnectionId, group);
                _documentService.RemoveConnectionGroup(group);
            }
            _documentService.AddToGroup(Context.ConnectionId, groupName);
            await Groups.AddToGroupAsync(Context.ConnectionId, groupName);
        }
    }
}
```