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

Розробка веб-застосування для спільного доступу до документів

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 121

Керівник курсової роботи
к.ф.-м.н., ст.викл. _____ *Гречко А.В.*
(прізвище та ініціали)

_____
(підпис)
“____” _____ 2021 р.

Виконав студент *ФІ-4*
Коношенко С.А.
(прізвище та ініціали)
“____” _____ 2021 р.

Київ 2021

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)

"____" _____ 2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

Студента Коношенка Сергія Артуровича факультету інформатики 3 курсу ТЕМА: Розробка веб-застосування для спільного доступу до документів.

Вихідні дані:
Зміст ТЧ до курсової роботи:
Календарний план
Перелік умовних позначень
Вступ
Частина 1: Аналіз предметної області. Постановка завдання курсової роботи
Частина 2: Теоретичні відомості
Частина 3: Опис реалізації програмного продукту
Висновки
Список використаної літератури
Додатки

Дата видачі "____" _____ 2020 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання курсової роботи

Тема: Розробка веб-застосування для спільного доступу до документів

Календарний план виконання роботи:

№ п/п	Назва етапу курсового проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	05.10.2020	
2.	Ознайомлення з предметною областю	09.11.2020	
3.	Огляд і аналіз аналогів	23.11.2020	
4.	Огляд літератури за темою роботи	21.12.2020	
5.	Вибір і аналіз технологій для розробки	25.01.2021	
7.	Розробка застосунку і аналіз результатів	18.03.2021	
8.	Оформлення пояснювальної записки	28.03.2021	
9.	Корегування роботи	06.04.2021	
10.	Створення презентації та написання доповіді	09.04.2021	
11.	Остаточне оформлення пояснювальної роботи та презентації	11.04.2021	
12.	Здача курсової роботи на перевірку	12.04.2021	

Студент Коношенко С. А.

Керівник Гречко А. В.

“_____”_____

Зміст

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ 5
Анотація 6
Вступ 7
РОЗДІЛ 1 9
1.1. Аналіз сучасного стану питання та обґрунтування теми 9
1.2. Огляд існуючих аналогів розробки 9
1.3. Постановка завдання 10
РОЗДІЛ 2 12
2.1. Архітектура системи операціональних трансформацій 12
2.2. Архітектура системи на основі безконфліктних репліційованих типів даних 19
РОЗДІЛ 3 23
3.1. Аналіз технічного завдання 23
3.2. Обґрунтування алгоритму й структури програми 23
3.3. Обґрунтування вибору засобів розробки 29
3.4. Опис розробки програми 34
3.5. Створення об’єктів і розробка головної програми 35
3.6. Опис файлів даних та інтерфейсу програми 38
3.6.1. Опис файлів даних 38
3.6.2. Опис інтерфейсу програми 40
3.7. Тестування програми і результати її виконання 42
ВИСНОВКИ 51
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 52
Додаток А. Код серверної частини 54
Додаток Б. Код клієнтської частини 62

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

БД – бази даних.

ПЗ – програмне забезпечення.

СКБД (Система керування базами даних) – набір взаємопов’язаних даних і програм для доступу до цих даних.

AJAX (Asynchronous JavaScript and XML) – підхід для побудови користувацьких інтерфейсів веб-застосунків, який заснований на “фоновому” обміні даними браузера і сервером.

API – application programming interface

CRDT (Conflict-Free Replicated Data Type) – типи даних, які можна скопіювати на велику кількість мережевих вузлів і оновлювати паралельно без координації.

IDE – Integrated Development Environment

JSON (JavaScript Object Notation) – текстовий формат обміну даних між системами, який зручний для читання людиною.

ORM (Object-Relational Mapping) – технологія програмування, яка пов’язує бази даних з концепціями об’єктно-орієнтованого програмування.

OT (Operational Transformation) / ОП (Операціональні перетворення, операціональні трансформації) – технологія, що використовується для автоматичного вирішення конфліктів в складних системах за допомогою перетворення операцій над даними.

Анотація

Робота присвячена розробці веб-застосування для спільного доступу та редагування документів.

В даній роботі розглянуто основні способи автоматичного вирішення конфліктів під час спільної роботи, а саме алгоритми OT та CRDT.

Було розглянуто основі способи взаємодії сервера та клієнта в застосунках у реальному часі, а саме long polling, SSE та WebSockets, їхні переваги та недоліки.

Клієнтська частина реалізована на бібліотеці для JavaScript – React. Серверна частина реалізована на програмні платформі Node.js, з використанням модулів Express, ShareDb та нереляційної бази даних MongoDb.

Вступ

Ми пройшли різноманітні етапи розвитку інформаційних технологій – від централізованих обчислень на мейнфреймах до настільних систем і розподілених обчислень. Через поширення Інтернету та поширеного використання комп’ютерів нам довелося перейти від персональних до спільних систем. Оскільки в наш час люди виконують багато завдань, працюючи разом і оскільки комп’ютер використовується в великій частині їх робіт, важливо дослідити способи використання комп’ютера для забезпечення такої співпраці. По мірі того як організації становляться глобальними, а вартість пересування стають вищими, дослідження спільної роботи з використанням комп’ютера стають все більш значущими. Також нинішня ситуація з епідемією сприяє тому, що синхронні системи для спільної роботи, такі як спільні дошки, редактори, використовують все більше і більше людей, тому такі системи є дуже актуальними в наш час.

За мету курсової роботи було поставлено дослідження і реалізація системи для спільного редагування документів.

Завданням курсової є створення веб-застосунку для паралельного редагування текстових документів

Об’єктом дослідження є способи організації синхронної взаємодії користувачів через мережу Інтернет.

Предметом дослідження є різні архітектури і проблеми проектування, які можуть виникнути про розробці систем для спільної роботи.

Основною мовою програмування для розробки було обрано JavaScript. Серверна частина реалізована на програмній платформі Node.js, а саме на веб-фреймворку Express. Для зберігання та використання даних використано нереляційну базу даних MongoDB.

Для відображення сторінок було використано JavaScript бібліотеку для розробки користувацьких інтерфейсів React.

Текстова частина роботи має вступ, 3 розділи, висновок, список використаних джерел та додатки.

В першому розділі буде описано стан розвитку обраної галузі. Буде проаналізовано існуючі аналоги, охарактеризовано основні переваги та недоліки обраних аналогів.

В другому розділі буде описано основні теоретичні відомості побудови редакторів з паралельним доступом, способи взаємодії клієнтів і серверу, синхронізації змін, які вносять користувачі.

В третьому розділі буде присвячено опису реалізації програмного застосунку, файлів даних, інтерфейсу користувача, конкретизації постановки задачі, пояснення структури програми, тестування.

У висновку буде зазначено підсумки роботи, результати виконання кожного з поставлених завдань і роботи в загальному.

Після висновку буде надано список використаних джерел та додаток з вихідним кодом.

РОЗДІЛ 1

1.1. Аналіз сучасного стану питання та обґрунтування теми

На сьогоднішній день завдяки поширенню мережі Інтернет широкого застосування набираються різноманітні системи з можливістю спільного редагування контенту, серед яких є застосунки для управління різних процесів управління проектами, документами, графіками. Такі застосунки дають можливість швидко вносити зміни та відстежувати зміни іншими користувачами в реальному часі.

Більшість сучасних застосунків для спільної роботи працюють за схемою розподіленої системи – набору комп’ютерів, які є незалежними один від одного та представляються як одна ціла система. Для підтримки такої системи є два основних варіанти їх взаємодії: інформація передається та зберігається лише між учасниками без втручання посередників та система з наявним посередником, який делегує інформацією між учасниками системи. Перший випадок має значний недолік – вся інформація зберігається в кожного з учасників. В другому випадку система виходить простіше оскільки існує єдине джерело істини.

1.2. Огляд існуючих аналогів розробки

Для визначення вимог до веб додатку, що є метою розробки курсової роботи варто проаналізувати аналогічні застосунки, що вже існують в мережі. Комплексний аналіз дозволить сформувати чіткий перелік необхідного функціоналу, виявити переваги та недоліки аналогічних застосунків.

1.2.1. Google Docs [1]– це веб-орієнтований онлайн сервіс, який включає в себе текстовий і табличний процесор, інструмент для створення презентацій і також хмарне сховище. Система підтримує багатокористувацьке спільне редагування.

Серед недоліків можна виділити обмежений функціонал редактора та прив’язку до аккаунтів Google.

1.2.2. Firepad [2] – спільний текстовий редактор з відкритим вихідним кодом. Зазвичай він використовується для вбудови в інші проекти. Firepad має такі основні можливості:

- Інтелектуальне злиття та вирішення конфліктів на основі OT.
- Підтримка форматування тексту і редагування коду.
- Синхронізація положення курсору.
- Відміна і повтор дій.
- Перевірка версій.
- Зображення.

Основні переваги:

- Швидкість дуже висока
- Розташування елементів редагування зручне
- Просто додати користувачів до редагування

Серед недоліків можна виділити:

- Має невелику кількість функцій, для їх розширення необхідно власноруч дороблювати застосунок.
- Обмежена стилізація тексту.

1.3. Постановка завдання

Основною задачею є аналіз існуючих програмних продуктів, виділення їх переваг та недоліків, визначення алгоритму синхронізації і створення веб-застосунку для паралельного редагування тексту. Застосунок має містити такі можливості:

- Реєстрація та авторизація
- Перегляд інформації про користувача
- Перегляд доступних до редагування документів
- Створення та видалення документів
- Додавання інших користувачів до редагування документів

- Основні можливості текстового редактору

Створена система має містити такі особливості:

1. Розширюваність – можливість додавання функціоналу системи без значної переробки всієї системи.
2. Кросплатформеність – можливість створення клієнтського застосунку під будь-яку платформу.
3. Стійкість клієнтів до розширення функціоналу системи – додавання або розширення функціоналу не має впливати на користувачів системи.
4. Синхронізація відкритих текстових документів в реальному часі.

РОЗДІЛ 2

2.1. Архітектура системи операціональних трансформацій

Сумісне редагування документів складний процес, оскільки декілька користувачів можуть змінювати вміст документу в один і той же час. Оскільки швидкість передачі даних через Інтернет має обмеження[3] і унеможливлює миттєво оновлювати стан документу, то для імітації миттєвої передачі даних кожен клієнт працює зі своєю версією локальною версією документу. Інакше користувачам потрібно було б чекати поки відбудеться синхронізація їх дій, а це зайняло б помітну кількість часу. Отже основна проблема одночасного редагування це забезпечення консистентність (послідовність) між локальними версіями документів, тобто всі локальні версії документу врешті решт мають сходитись в одну версію документу.

Перші версії застосунків для сумісного редагування використовували порівняння версій документу. Працювало це за такою схемою: два користувача – Користувач А та Користувач Б – мали синхронізовані версії документу. Коли сервер отримував зміни від Користувача А він визначав різницю між своєю версією документу і версією Користувача А, намагався поєднати їх і відправляв об’єднану версію до Користувача Б. Якщо Користувач Б мав свої несинхронізовані зміни, то він намагався об’єднати свою версію і версію, отриману від серверу. І так це процес повторювався.

На перший погляд такий підхід забезпечував синхронізацію версій документу, але це не завжди працювало. Уявімо таку ситуацію: Користувач А, Користувач Б і сервер мають однакову версію документу з текстом “Великий чорний кіт”.

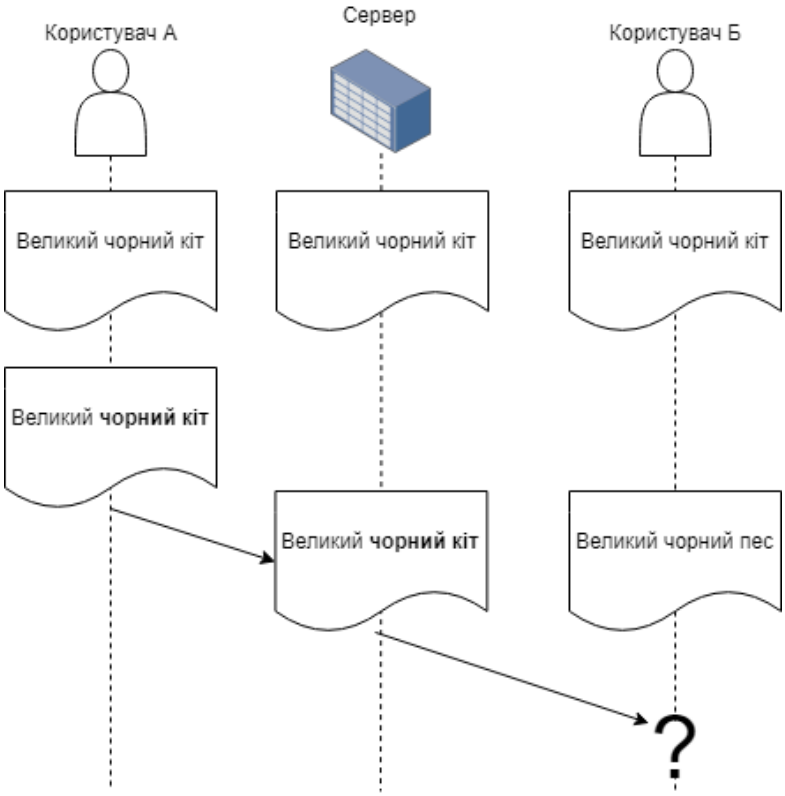


Рисунок 2.1 – Невизначений результат об’єднання

Користувач А виділяє жирним слова “чорний кіт” і в цей самий час Користувач Б змінює “кіт” на “пес”. Нехай зміни Користувача А прибудуть на сервер першими, вони збережуться і зміни відправляться Користувачу Б. Правильним результатом об’єднання є “Великий **чорний пес**”, проте алгоритм об'єднання не має достатньо інформації для правильної обробки інформації. З перспективи алгоритму “Великий **чорний кіт** пес”, “Великий **чорний** пес”, “Великий **чорний** пес **кіт**” є правильними результатами об’єднання. Це основна проблема такого підходу – ми не можемо бути впевненими в результаті об’єднання.

В нових текстових редакторах з паралельним редагуванням використовується інших підхід – замість порівняння версій документ зберігається як послідовність лінійно впорядкованих операцій.

Нам потрібно зрозуміти коли треба коли приймати зміни – для цього потрібен протокол взаємодії. В класичному варіанті всі операції представляються 3 типами:

- Вставка тексту

- Видалення тексту
- Використання стилів на текст

Коли користувач редагує текст, то всі зміни зберігаються як набір з 3 операцій і потім коли користувач відкриває документ то ці зміни застосовуються один за одним.

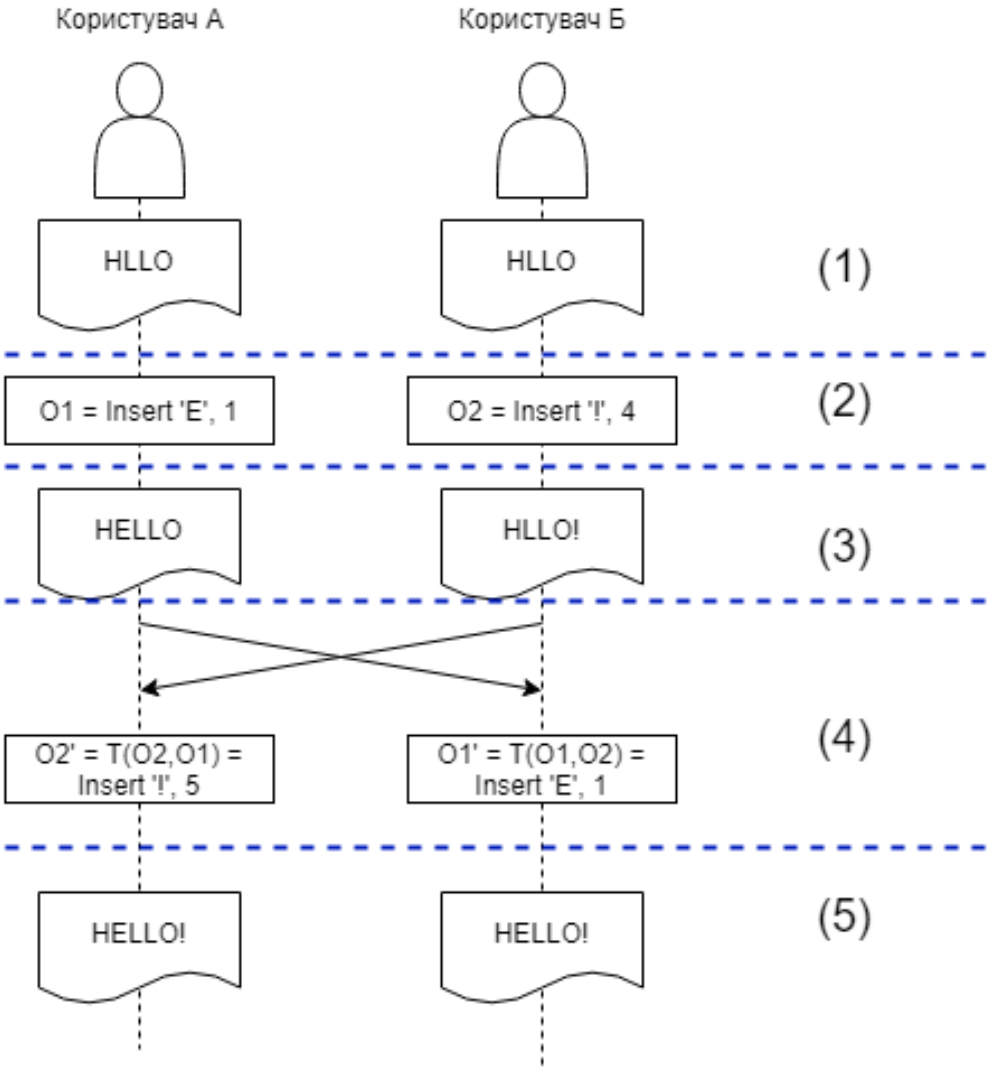


Рисунок 2.2 – Схема операційних трансформацій документу

Нехай X = (1), останній синхронізований стан.

Користувач А додає літеру ‘E’ на позицію 1, а Користувач Б додає символ ‘!’ на позицію 4. Нехай операції Користувача А та Б є O1 та O2 відповідно (2).

Після цих операцій в кожного з двох користувачів свій локальний стан (3).

Локальні зміни користувача передаються іншому користувачу як певні операції. Інший користувач приймає відправлені операції і виконує функцію T(Прийняті операції, Локальні операції), яка синхронізує певним чином операції. Таким

чином за допомогою функції T, яка враховуючи локальні операції перетворює прийняту операцію O2 в O2’ та прийняту операцію O1 в O1’ (4).

Після цього кожний користувач має однаковий вміст документа, тобто O1’(O2(X)) = O2’(O1(X)) (5).

Описаний вище алгоритм має назву Operational Transformation[5].

Для забезпечення узгодженості було створено декілька моделей узгодженості. Розглянемо одну з них – модель CCI.

Модель CCI містить наступні властивості:

1. Збіжність (convergence): всі репліки одного документа мають бути ідентичними після виконання всіх операції

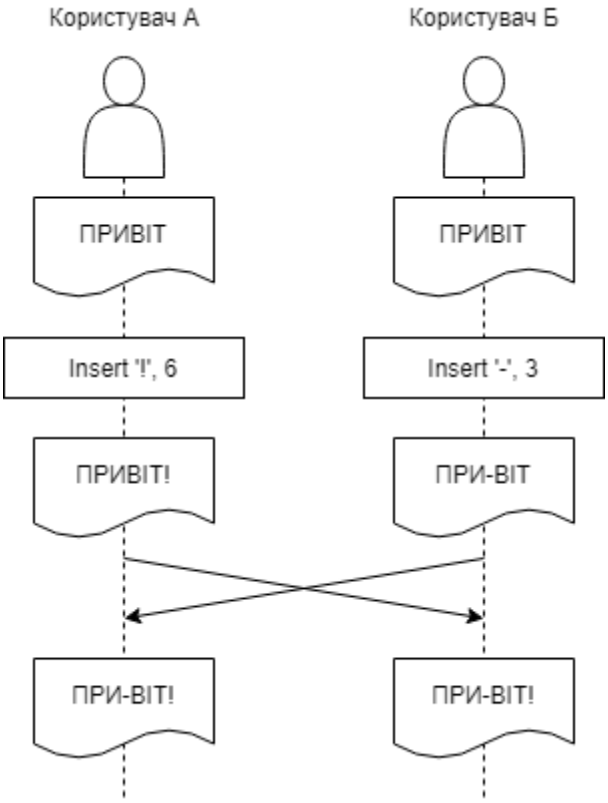


Рисунок 2.3 – Властивість збіжності

На рисунку 2.3) Користувач А та Користувач Б почали роботу з однакових версій документу. Потім вони роблять локальні зміни і репліки документів розходяться. Властивість збіжності вимагає, щоб Користувач А та Користувач Б мали однаковий зміст документу після того як вони прийняли зміни один одного.

2. Збереження намірів (intention preservation): гарантує, що ефект від виконання операції на будь-якому стані документу буде таким же як і намір операції. Намір операції (intention of operation) визначається як ефект її виконання на створеній нею репліки.

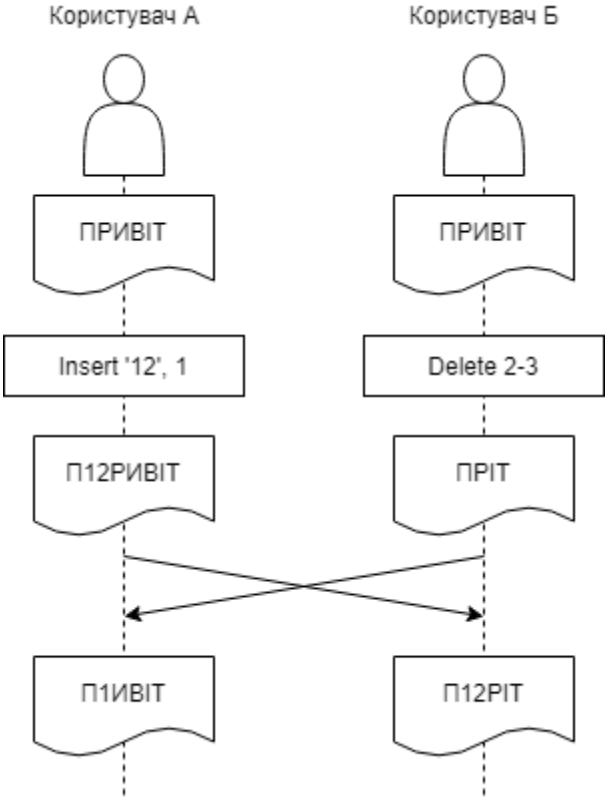


Рисунок 2.4 – Властивість збереження намірів

На рисунку 2.4) Користувач А та Користувач Б почали з однакового стану документа, потім зробили свої локальні зміни. Намір операції Користувача А це додати ‘12’ на позицію 1, а намір операції Користувача Б це видалити ‘ИВ’. Після синхронізації намір операції яку виконав користувач Б порушився. Також ми можемо спостерігати, що після операцій стан реплік у користувачів різна, проте це не вимога властивості збереження намірів.

3. Збереження причинності (causality preservation): в процесі співпраці операції мають виконуватися відповідно до їхнього природнього причинно наслідкового порядку (відповідно до відношення “Happened-before”).

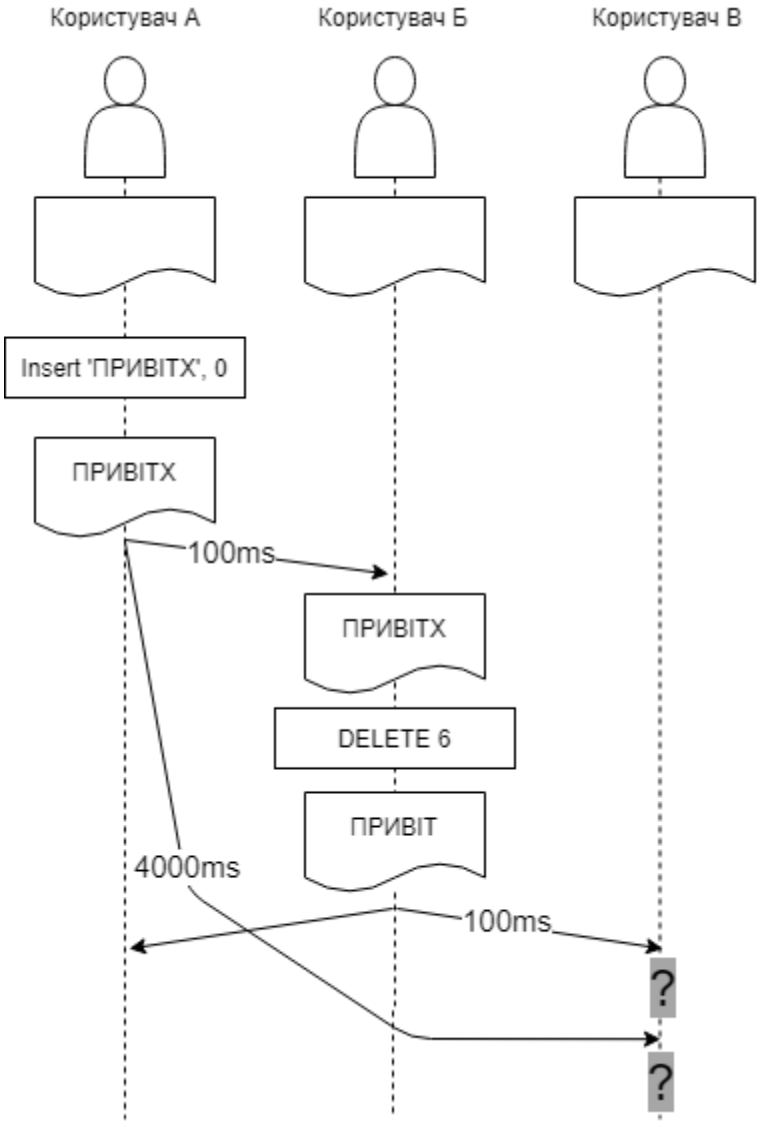


Рисунок 2.5 – Властивість збереження причинності

На рисунку 2.5) Користувач А, Користувач Б та Користувач В почали з однакового стану документа. Користувач А зробив зміни і відправив їх до інших користувачів. Користувач Б отримав зміни першим, виправив опечатку і відправив зміни іншим користувачам. Через повільну мережу Користувач В спочатку отримав операцію від Користувача Б, яка намагається видалити символ, який ще не існує в цій репліці. Операціональна трансформація не може забезпечити виконання властивості збереження причинності. Для виконання властивості збереження причинності потрібно використовувати інший алгоритм, наприклад векторний годинник.[4]

Основними елементами системи операціональних трансформацій є:

1. Алгоритми контролю трансформацій відповідальні за те, щоб визначити коли операція готова для трансформації, відносно яких операцій виконувати трансформації і в якому порядку здійснювати перетворення.
2. Функції трансформації відповідальні за виконання перетворення актуальних трансформацій на цільові операції враховуючи пов'язані операції. Ці функції можна розділити на два типи:
   - Трансформація включення/пряма трансформація: перетворити операцію O_a перед операцією O_b так щоб включався ефект від O_b (наприклад, дві вставки).
   - Трансформація виключення/ обернена трансформація: перетворити операцію O_a перед O_b так щоб виключався ефект від O_b (наприклад, вставка після видалення).
3. Властивості і умови визначають взаємозв'язки між алгоритмами контролю трансформацій та функціями та слугують вимогою точності алгоритму OT.

Операціональні трансформації (Operational Transformation, OT) представляють собою технологію, що використовується в системах управління даними і являє собою комплекс функціональних можливостей для роботи з ними. З самого початку дана технологія була розроблена для підтримки сумісного редагування текстових документів. Можливості сумісного паралельного редагування зросли з появою в операціональних трансформацій здатності вирішувати користувацькі конфлікти, блокування операцій при роботі з даними, редагування XML (eXtensible Markup Language) та HTML (HyperText Markup Language) документів, а також різних документів деревовидної структури[5].

2.2. Архітектура системи на основі безконфліктних репліційованих типів даних

CRDT (Conflict-free Replicated Data Type) – технологія на основі типів даних, які можуть працювати на багатьох вузлах і паралельно оновлюватись без координації. CRDT виник пізніше, ніж OT і пропонував все те, що міг і OT. Ця технологія подібна до технології OT, але на відміну від них не зберігає попередню історію операцій. Також при передачі даних у вигляді коміту, система CRDT оперує не переліком операцій, а поточним станом даних.

Основними моделями реплікації в розподіленій системі є підхід на основі стану і підхід на основі операцій. На основі моделі реплікації вони визначають два типи CRDT: CvRDT (convergent replicated data type) та CmRDT (commutative replicated data type), при чому ці два типи CRDT є еквівалентними.

2.2.1. Реплікація на основі стану

Коли репліка отримує оновлення від клієнту, спочатку вона оновлює локальний стан і через деякий час відправляє весь поточний стан на іншу репліку. Час від часу кожна репліка відправляє повний свій стан іншій репліці в системі. І той, хто отримав повний стан іншої репліки, виконує операцію злиття для об’єднання свого локального стану зі стоном, що був отриманий. Так само ця репліка час від часу відправляє свій повний стан іншому користувачеві, тому кожне оновлення досягне всіх користувачів в системі.

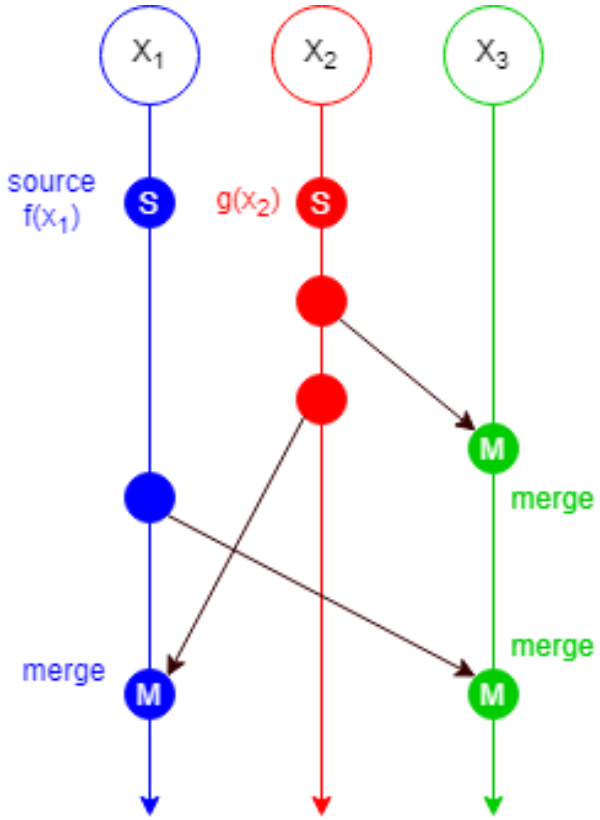


Рисунок 2.6 – Реплікація на основі стану

Якщо набір значень, який може прийняти стан, утворює напівгратку [6] (частково впорядкований набір з операціями join та least upper bound) і оновлення це операція збільшення (наприклад, стан – це ціле число, а оновлення – це збільшення цього числа) і якщо функція злиття обраховує найменшу верхню границю (least upper bound), то репліки гарантовано сходяться до одного і того ж значення (яке є найменшою верхньою границею останніх оновлень). Для того щоб множина всіх можливих станів системи була напівграткою, операція злиття має бути ідемпотентною, асоціативною і комутативною. Об’єкт репліки, який задовольняє цим властивостям є одним з типів CRDT – CvRDT (convergent replicated data type, конвергентний реплікаційний тип даних).

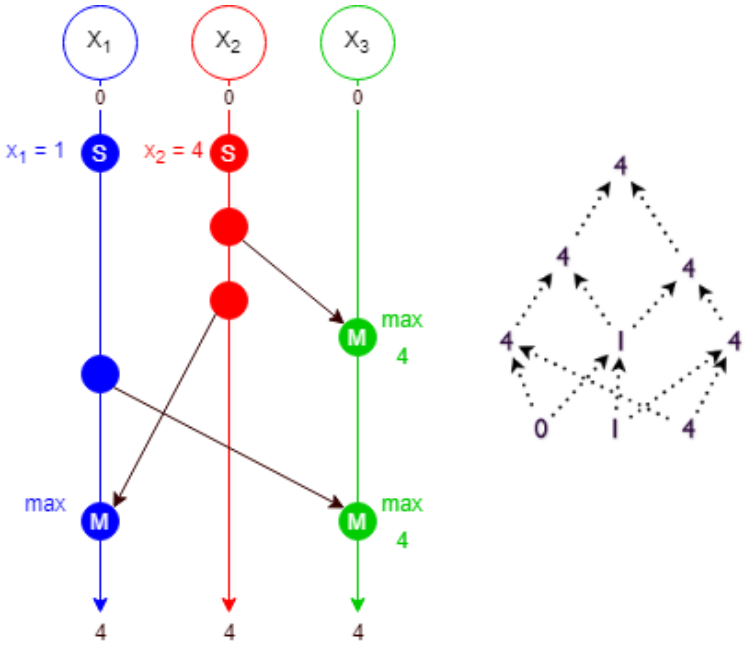


Рисунок 2.7 – Приклад реплікації на основі стану

2.2.2. Реплікація на основі операцій

В цьому підході репліка не відправляє свій повний стан (який може бути великим за розміром) до іншої репліки. Замість цього вона відправляє бродкастом операцію оновлення всім іншим реплікам в системі і очікує, що інші репліки приймуть оновлення. Оскільки це бродкастна операція, то якщо є два оновлення *up_1* та *up_2* виконані на репліці *repl_0* і якщо *repl_0* відправляє ці оновлення до інших реплік *repl_1* та *repl_2*, то можлива ситуація, що оновлення прийдуть в різному порядку. Тобто *repl_1* може отримати спочатку *up_1*, а потім *up_2*, *repl_2* може отримати спочатку *up_2*, а потім *up_1*. Ці репліки можуть збігатися якщо оновлення будуть являтися комутативними – незалежно від порядку прийняття оновлень, результуючий стан буде однаковим.

В цій моделі, де оновлення відправляються всім учасникам, об’єкт, для якого всі паралельні оновлення є комутативними називається CmRDT

(commutative replicated data type, комутативний реплікаційний тип даних)

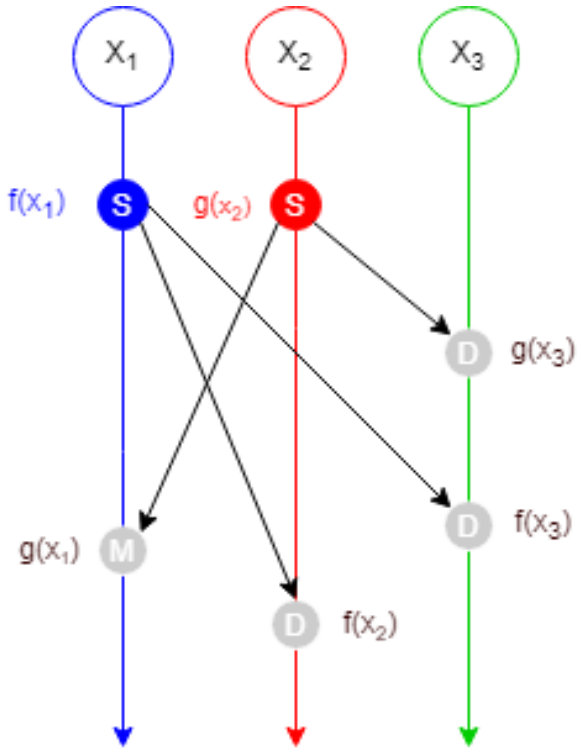


Рисунок 2.8 –Реплікація на основі операцій

РОЗДІЛ 3

3.1. Аналіз технічного завдання

Основне призначення створюваного веб-застосунку – можливість одночасного редагування документу декількома користувачами. Для роботи з документами користувач має бути авторизованим.

Алгоритм роботи застосунку:

1) Користувач створює новий документ, який зберігається в списку створених цим користувачем документів.
2) Користувач може переглядати список створених ним документів і список документів, до яких він має доступ.
3) Після переходу на сторінку певного документу, всі дані, що були раніше додані в документ завантажуються з серверу.
4) За бажанням користувач може додати іншого користувача до редагування документу за поштою.
5) Будь-які зміни, які провів один користувач з’являються в усіх інших користувачів.

3.2. Обґрунтування алгоритму й структури програми

3.2.1. Обґрунтування вибору способу взаємодії клієнта і сервера

Одним з основних способів передачі інформації між сервером і клієнтом є HTTP (Hypertext Transfer Protocol). Хоча цей протокол і є зручним, проте він має ряд недоліків, які заважають його використовувати в веб-застосунках з великою кількістю переданої інформації, а саме:

- Запит завжди ініціюється клієнтом, тобто для отримання нової інформації клієнт має сам звернутися до серверу
- На один запит від клієнта завжди повертається лише одна відповідь від серверу. Після цього зв’язок між клієнтом і сервером закривається. Тобто якщо користувач хоче відправити 50 повідомлень в чаті, використовуючи HTTP, йому потрібно було б виконати 50 окремих

циклів запит-відповідь. Крім того, отримувач повідомлень мав би оновлювати сторінку, щоб отримати повідомлення.

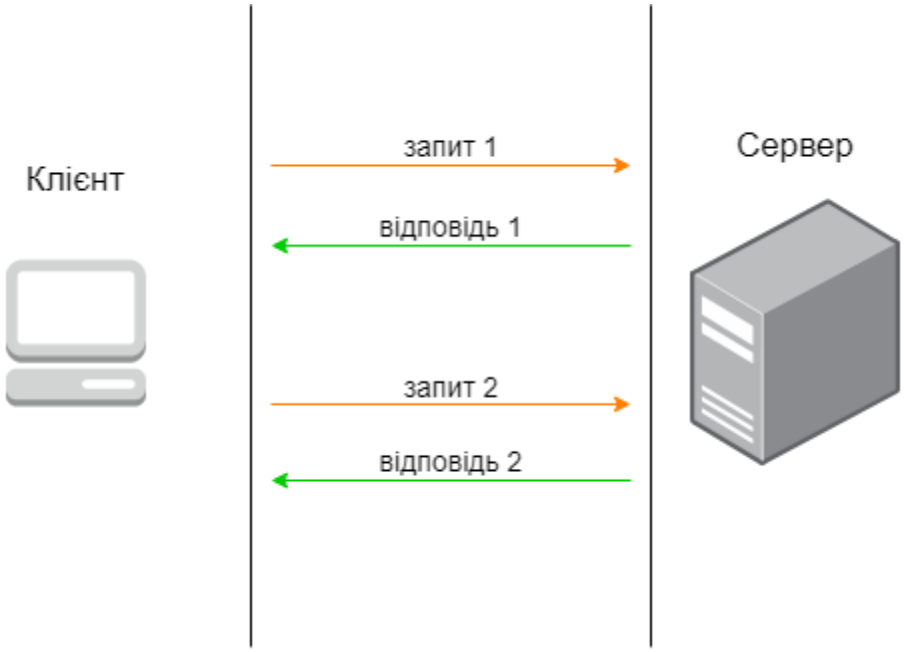


Рисунок 3.1 – Клієнт-серверна взаємодія за допомогою HTTP

HTTP – це протокол, який не зберігає стан [7]. Тому для застосунку, в якому постійно оновлюється велика кількість інформації, потрібно щось інше.

Існує декілька технологій, які допомагають у вирішенні цієї проблеми. Ці технології діляться на два підходи: client pull – клієнт запитує сервер про наявність оновлень з певним інтервалом та server push – сервер власноруч відправляє оновлення клієнтам.

Розглянемо три способи вирішення завдання: довге опитування (long polling) - client pull, server-sent events - server push, web sockets - server push.

3.2.2. Long polling

Довге опитування (long polling) – більш ефективна форма базового методу опитування [8]. Виконання запитів до серверу, які повторюються багато разів, витрачають велику кількість ресурсів для встановлення нового з’єднання, аналізу заголовків запиту, формування відповіді (яка часто не має нових даних) і доставки відповіді. Після цього потрібно ще зачинити з’єднання і очистити ресурси. Замість того, щоб повторювати цей процес багато разів поки не з’являться нові дані, long polling пропонує спосіб, при якому сервер тримає

з’єднання відкритим до тих пір, поки не будуть доступні дані або закінчиться час очікування.

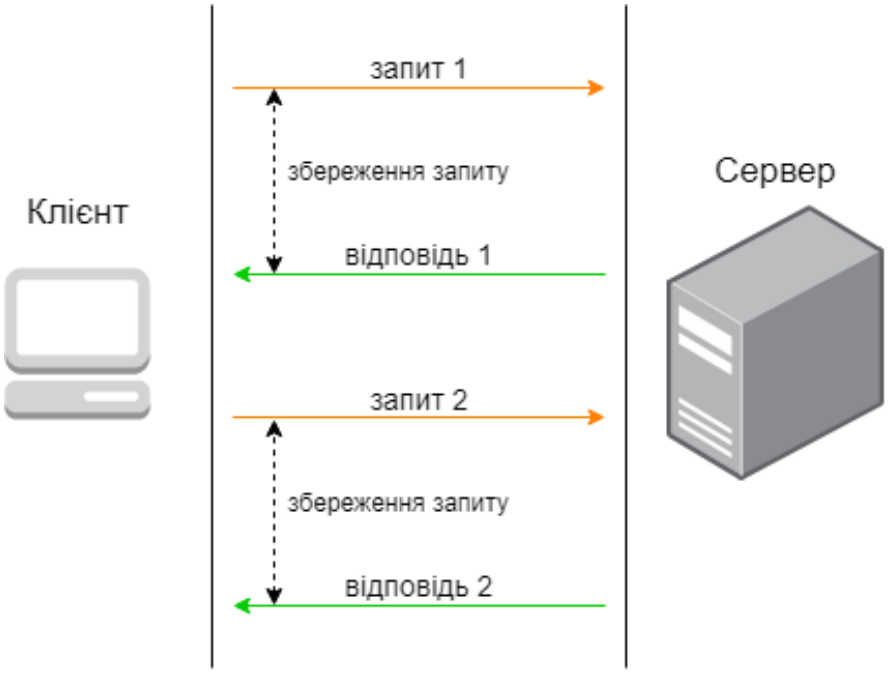


Рисунок 3.2 – Клієнт-серверна взаємодія за допомогою Long polling

Серед переваг такого підходу можна виділити зменшення кількості запитів до серверу. А серед мінусів варто виділи те, що залишається велике навантаження на сервер, оскільки йому необхідно зберігати відкритий стан запитів. Також такий підхід не підходить для редактору тексту, оскільки для кожного введеного символу буде створюватись окремий запит. Тобто такий підхід не змінює кількість запитів під час редагування.

3.2.3. Server-Sent Events

Server-sent event (SSE) – це стандарт, який описує як саме сервер може почати передачу даних клієнту після встановлення початкового клієнтського з’єднання. Він забезпечує реалізацію потокової передачі XMLHttpRequest з ефективним використанням пам’яті.

SSE описує вбудований клас EventSource, який дозволяє підтримувати з’єднання та отримувати події від серверу [9].

З’єднання через SSE зазвичай починається з клієнта, який ініціалізує взаємодію між ним і сервером. Клієнт створює новий об’єкт EventSource і

передає адресу серверу через HTTP запит. Сервер повідомляє клієнта про успішне з’єднання і залишає його відкритим до тих пір, поки не залишиться подій для відправки, або сервер вирішить, що з’єднання настільки довге, що його можна вважати застарілим, або поки клієнт сам не закінчить початкове з’єднання.

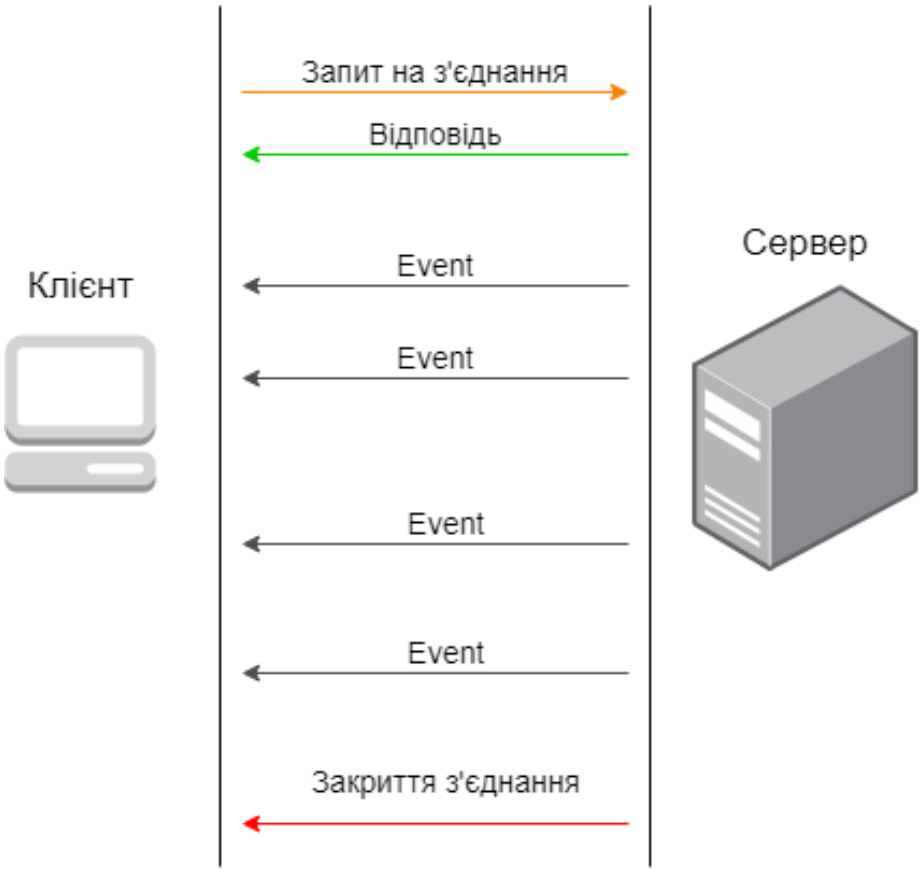


Рисунок 3.3 – Клієнт-серверна взаємодія за допомогою Server-Sent Events

SSE міг би підійти для розробки текстового редактора, оскільки всі події передаються в рамках однієї сесії, але через те, що дані можна відправляти з лише з боку сервера, то цей спосіб не підходить.

3.2.4. WebSocket

WebSocket визначає API, який встановлює з’єднання сокета між клієнтом і сервером. Це дозволяє веб браузерам відправляти дані в будь-якому напрямку. Крім цього, він також має декілька покращень в порівнянні з HTTP, що робить його кращим для зв’язку в реальному часі. WebSocket має такі ключові особливості:

1) Зв’язок в реальному часі.

Під час кожного HTTP запиту браузер відправляє файли cookie і інші заголовки, які збільшують витрати під час зв’язку в реальному часі. Проте WebSocket використовують менше даних, кожне повідомлення містить лише 6 байтів службової інформації (2 байти для заголовку та 4 для маски).

Таким чином WebSocket краще підходить для передачі інформації в реальному часі і особливо для застосунків, де важлива невелика затримка між отриманням актуальної інформації.

2) З’єднання WebSocket.

Відкриття WebSocket з’єднання дуже просте. Також можна відрити з’єднання з додатковими протоколами.

```
const socket = new WebSocket(“ws://yoursite.com”);
// or
const socket = new WebSocket(“ws://mysite.com”, [“soap”];
```

Після створення з’єднання до нього можна приєднати обробників подій, що дозволить дізнатися коли з’єднання відкрито (подія onopen виникає коли завершене рукостискання), коли воно отримує повідомлення (подія onmessage виникає коли клієнт отримує дані) та коли виникла помилка (onerror).

3) Шаблони передачі даних

При використанні WebSocket існує декілька варіантів передачі даних.

Перший варіант це відправка повідомлень про наявність нових даних використовуючи WebSocket. Тобто сокет лише повідомляє про оновлення, а нові дані передаються іншим способом, наприклад через звичайний HTTP запит.

Другий варіант це безпосередня передача даних через WebSocket з’єднання. Такий сценарій важливий для забезпечення мінімальної затримки

оновлень даних, наприклад в онлайн іграх, чатах, або онлайн редакторах тексту.

4) Повторне підключення для забезпечення відмовостійкості.

Важливою особливістю при роботі з WebSocket є правильний алгоритм дій при розриві з'єднання. Це відбувається коли клієнт або сервер не відповідають. Для запобігання такої проблеми необхідно реалізовувати механізм коректного закриття з'єднання. Особливо якщо строк служби з'єднання великий то для забезпечення відмовостійкої системи необхідно реалізовувати метод оновлення з'єднань.

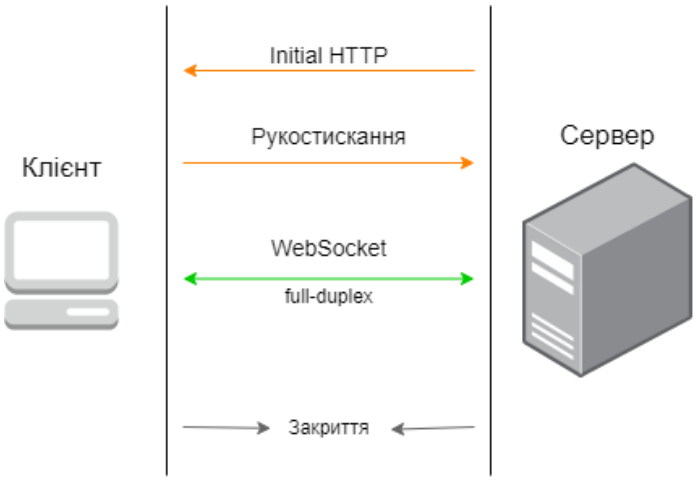


Рисунок 3.4 – Клієнт-серверна взаємодія за допомогою WebSocket

Отже, проаналізувавши різні способи з'єднання клієнту і серверу, оцінивши переваги і недоліки кожного із способів можна зробити такі висновки:

- Polling/long polling використовує HTTP(S) і має низьку швидкість роботи, оскільки потребує нового з'єднання для кожного оновлення даних.
- Server-sent event забезпечує постійне з'єднання між клієнтом і сервером, але лише в один бік, лише зі сторони серверу. Для текстового редактору необхідно, щоб клієнт часто відправляв нові дані, тому такий підхід також не підходить. Серед плюсів можна виділити те, що SSE забезпечує автоматичне перепідключення при втраті з'єднання.

- WebSocket забезпечує швидку взаємодію в сторону і серверу і клієнта, тому такий підхід задовольняє вимогам текстового редактору з паралельним редагуванням. Серед мінусів треба виділити те, що необхідно власноруч контролювати стан з’єднання і за потребою виконувати перепідключення.

3.3. Обґрунтування вибору засобів розробки

Для розробки серверної частини було обрано програмну платформу Node.js [10]. Ця платформа працює на рушію Google Chrome – V8, який може компілювати код JavaScript в машинний код. Node.js використовує event-driven модель і неблокуючий ввід/вивід.

3.3.1. Неблокуючий ввід/вивід

Node.js використовує неблокуючі операції вводу/виводу, а це означає що:

- Основний потік не буде блокуватися операціями вводу і виводу.
- Сервер продовжить опрацьовувати запити.
- Велика частина коду буде асинхронна.

Для того, щоб зрозуміти переваги неблокуючих операцій розглянемо приклад.

```
console.log("Getting Data1");
var data1 = getData('123');
console.log("Data is:", data1);

console.log("Getting Data2");
var data2 = getData('456');
console.log("Data is:", data2);

var sum = 1 + 2;
console.log("sum is:", sum);
```

Виконання вище написаного коду буде таке:

1) Виведеться напис “Getting Data1”.
2) Потік буде заблокований в очікуванні результату з виклику API і після цього продовжить вивід.
3) Потік буде заблокований в очікуванні результату з виклику API і після цього продовжить вивід.

4) Лише після повного виконання двох API викликів виконається обрахування суми.

Звичайно, цього можна уникнути при використанні багатьох потоків, але JavaScript пішов іншим шляхом – використання функцій-callback.

```
console.log("Getting Data1");
getData('123', function(data1) {
  console.log("Data is:", data1);
});

console.log("Getting Data2");
getData('456', function(data1) {
  console.log("Data is:", data1);
});

var sum = 1 + 2;
console.log("sum is:", sum);
```

Виконання вище написаного коду буде таке:

1) Ініціюється перший виклик API, проте console.log знаходиться в callback функції, тому потік не буде очікувати повного завершення виклику і виконання до другого API виклику.
2) Другий API виклик виконується за схожою процедурою, тому виконання переходить до обрахування суми.
3) Сума виводиться і як тільки ми отримаєм результати від викликів API, то виконується вивід в консоль.

Отже, в багатьох випадках неблокуючі операції виконуються швидше за блокуючі, тому це є важливою перевагою Node.js.

3.3.2. Цикл подій

Цикл подій (event loop) – це головна особливість JavaScript та Node.js. Це буквально безкінечний цикл, якій відбувається всередині рушія [11].
Цикл подій має 6 фаз (рисунок 3.5), кожна фаза називається tick.

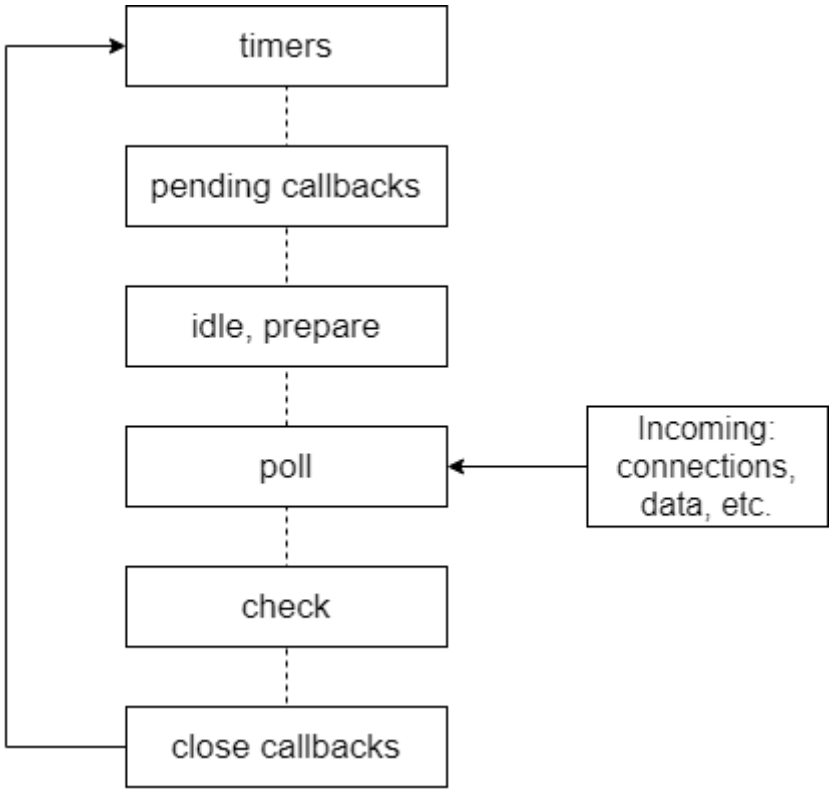


Рисунок 3.5 – Схема циклу подій

- timers – в цій фазі виконуються callbacks, які були створені методами setTimeout та setInterval
- pending callbacks – в цій фазі виконуються майже всі callbacks, окрім подій close, таймерів та створених методом setImmediate
- idle, prepare – використовується для внутрішніх цілей
- poll – в цій фазі отримуються нові події вводу та виводу
- check – в цій фазі виконуються callbacks, викликані методом setImmediate
- close callbacks – наприклад socket.on(‘close’, …)

З такою структурою може виникнути питання, коли та хто виконую операції вводу та виводу? Коли циклу подій треба виконати операцію вводу та виводу, він використовує потоки операційної системи з thread pool, а коли задача виконана, callback стає в чергу під час фази pending callbacks.

Node.js добре підходить для великої кількості застосунків, але помітним недоліком є задачі, які потребують великих затрат CPU. Коли головний потік зайнятий такою задачею, він стає недоступним для інших задач. Рішенням цієї

проблеми є Worker Threads, які дають можливість створювати потоки для паралельного обчислення ємких задач.

Для редагування коду було обрано редактор Visual Studio Code. Visual Studio Code – редактор коду для розробки застосунків різного типу. Даний редактор розроблений компанією Microsoft [12].

Visual Studio Code надає такі можливості:

- підтримка майже всіх сучасних мов програмування
- вбудовані інструменти інтеграції з GitHub, GIT
- підтримка TypeScript та JavaScript
- робота з Node.js за допомогою вбудованого debugger
- обширна бібліотека плагінів, яка дозволяє змінити Visual Studio Code так як бажає розробник.

Для розробки серверної частини було використано велику кількість фреймворків, серед яких є:

- Express це один з найпопулярніших веб фреймворків для Node.js, які надає ключові механізми для розробки веб-застосунку, такі як написання обробників HTTP запитів, інтеграція з різними рушіями для рендерінгу, додавання middleware в будь-якій точці процесу обробки запиту, встановлення загальних налаштувань веб-додатків, такі як порт для підключення, розташування шаблонів сторінок і інше.
- body-parser – пакет для перетворення тіла запиту з JSON в JavaScript об'єкт.
- jsonwebtoken – пакет для реалізації процесу авторизації з використанням JSON Web Token (JWT).
- ws – реалізація WebSocket комунікації між сервером та клієнтом
- sharedb – це база даних в реальному часі, в основі якої лежить операціональні трансформації документів в JSON форматі. Серед основних можливостей є синхронізація в реальному часі JSON

документів, паралельна багатокористувацька співпраця, API для синхронного редагування з асинхронною послідовністю, засоби для контролю доступу, синхронізація офлайн змін після відновлення з’єднання.

- sharedb-mongo – адаптер для інтеграції sharedb з нереляційною базою даних MongoDB.
- mongoose – інструмент для об’єктного моделювання MongoDB призначений для роботи в асинхронному середовищі.
- websocket-json-stream – це обгортка Note.js потоку з’єднання WebSocket.
- uuid – реалізація стандарту ідентифікації, який використовується при створенні програмного забезпечення.

Для збереження даних використовується нереляційна база даних MongoDB, а для зручної роботи з базою даних було використано пакет mongoose, яка побудована на технології об’єктно-реляційного зв’язку.

На стороні клієнту основною частиною була бібліотека JavaScript для побудови користувацьких інтерфейсів React [13]. З допомогою React можна створити веб-застосунки, які змінюють відображення без перезавантаження сторінки. Завдяки цьому застосунок швидко реагує на різноманітні дії користувача.

React має наступні особливості:

- Декларативність. З допомогою React ми описуємо як компоненти інтерфейсу виглядають в різноманітних станах.
- React використовує компоненти. Компонент – це частина користувацького інтерфейсу зі своїм станом. Цільний застосунок будується шляхом об’єднання багатьох окремих компонентів.
- Віртуальний DOM. Зміна класичного DOM є доволі затратною операцією, тому React використовує віртуальний DOM – об’єкт, в

якому зберігається інформація про стан інтерфейсу. При зміні стану (наприклад, після натиснення кнопки) React розраховує різницю між старим і новим станами. Після цього бібліотека рендерить новий стан. Завдяки такому підходу збільшується швидкість оновлення реального DOM.

- JSX. React використовує розширення синтаксису, яке називається JSX. Це розширення схоже на JavaScript, в якому можна писати і взаємодіяти з HTML елементами.

3.4. Опис розробки програми

Застосунок побудований з використанням клієнт-серверної архітектури. Клієнтська частина, яка складається з набору компонентів, виконує запити на сервер, отримує у відповідь певну інформацію і діє згідно отриманої інформації. Серверна частина приймає запити від клієнта, певним чином оброблює їх, за необхідністю звертається до бази даних і повертає певну інформацію.

Сервер розділений на такі пакети:

- config

  Містить конфігураційні дані, наприклад, адресу підключення до бази даних
- routes

  Містить код, який делегує які функції з пакету controllers мають викликатись для конкретних точок API
- middlewares

  Містить код, який виконується перед виконанням функцій з пакету controllers. Тут перевіряється чи вхідний запит є валідний, користувач, який виконує цей запит має валідний токен авторизації, чи користувач, який робить цей запит має необхідні права для цього
- controllers

Містить код, який дістає потрібну інформацію із запиту, за потребою звертається до бази даних, дістає необхідну інформацію і формує відповідь користувачу

- models

  Містить моделі документів, які зберігаються в базі даних

3.5. Створення об’єктів і розробка головної програми

При запуску створюється об’єкт Express, він передається функції створення серверу як слухач запитів.

```
const app = express();
const server = http.createServer(app);
```

Після цього створюється підключення до MongoDB бази даних і створюється shareDb сервер.

```
export const documentDb = shareDbMongo(`mongodb://${dbConfig.HOST}:${dbConfig.PORT}/${dbConfig.SHARE_DOCUMENTS_DB}`);
export const shareDBServer = new ShareDB({ db: documentDb });
```

Далі ми в shareDb реєструємо тип, який будемо використовувати при передачі даних від клієнта при паралельному редагуванні. За замовчанням sharedb використовує JSON0 OT тип. Після цього створюється WebSocket сервер, який буде знаходись в тому ж місці що й основний сервер. Також ми після створення WebSocket серверу створюємо обгортку WebSocketJsonStream, яка використовується для читання та запису JavaScript об’єктів.

```
ShareDB.types.register(richText.type);
const wss = new WebSocket.Server({ server: server });
wss.on('connection', (ws) => {
    const jsonStream = new WebSocketJSONStream(ws);
    shareDBServer.listen(jsonStream);
});
```

Далі створюється підключення до бази даних MongoDB. Дані для підключення беруться з конфігураційного файлу.

```
db.mongoose
    .connect(`mongodb://${dbConfig.HOST}:${dbConfig.PORT}/${dbConfig.DB}`, {
        useNewUrlParsed: true,
        useUnifiedTopology: true
    })
    .then(() => {
        console.log('Successfully connect to MongoDB.');
        initial();
    })
    .catch(err => {
        console.error('Connection error', err);
        process.exit();
    });
```

Після цього викликаються функції, які реєструють точки API, які опрацьовує сервер.

```
authRoutes(app);
userRoutes(app);
documentRoutes(app);
```

Кожна точка API має ряд параметрів:

- URL API, яку вона опрацьовує
- Список функцій-посередників, які виконуються перед подальшою обробкою запиту
- Функція, яка буде оброблювати цей запит

Нижче наведено приклад функції, яка реєструє точки API:

```
export default (app) => {
    app.use((req, res, next) => {
        res.header(
            'Access-Control-Allow-Headers',
            'x-access-token, Origin, Content-Type, Accept'
        );
        next();
    });

    app.post('/api/createDocument', [verifyToken], controller.createDocument);
    app.delete('/api/deleteDocument/:documentId', [verifyToken], controller.deleteDocument);
    app.post('/api/inviteUser', [verifyToken], controller.inviteUserToDocument);
    app.get('/api/documentInfo/:documentId', [verifyToken], controller.getDocumentInfo);
    app.post('/api/changeDocumentTitle', [verifyToken], controller.changeDocumentTitle)
}
```

Функції-посередники – це функції, які виконуються перед обробкою запиту функцією-контролером. Такі функції перевіряють, чи має даний користувач

доступ то тієї точки API, куди він доступається, або чи авторизований цей користувач. Нижче наведено приклад такої функції.

```
const verifyToken = (req, res, next) => {
    const token = req.headers['x-access-token'];

    if (!token) {
        return res.status(403).send({ message: 'No token provided' });
    }

    jwt.verify(token, config.secret, (err, decoded) => {
        if (err) {
            return res.status(401).send({ message: 'Unauthorized!' });
        }
        req.userId = decoded.id;
        next();
    });
}
```

Ця функція перевіряє чи авторизований користувач за допомогою JSON Web Token (JWT). JWT – це один із способів представлення даних для передачі між двома чи більше сторонами об’єкта JSON. JWT складається з трьох частин: заголовок (header, службова частина, в ній зберігається інформація про те, яким чином оброблювати отриманий токен), корисне навантаження (payload, зберігається інформація для ідентифікації користувача), підпис (signature, генерується він об’єднанням заголовку і корисного навантаження). Тобто

```
JWT = base64(header) + '.' + base64(payload) + '.' +
base64( HMASCHA256(base64(header) + '.' + base64(payload), SECRET_KEY)
```

Кожна точка API має функцію, яка оброблює запит. Ці функції отримують необхідну інформацію з запиту, за необхідності, звертаються до бази даних і формують відповідь відповідно до отриманої інформації. Нижче наведено приклад функції-контролеру, яка створює новий документ користувачу, зберігає його до бази даних і формує відповідь користувачу.

```
const createDocument = (req, res) => {
    const token = JSON.parse(
        Buffer.from(req.headers['x-access-token'].split('.')[1], 'base64').toString('ascii')
    );
    User.findOne({ username: token.username })
        .exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }
            if (!user) {
                return res.status(404).send({ message: 'User not found.' });
            }
            const document = new Document({
                documentId: uuidv4(),
                title: 'Title document',
                text: '',
                redactors: [user._id],
                owner: user._id
            });
            document.save(err => {
                if (err) {
                    res.status(500).send({ message: err });
                    return;
                }
                createShareDbDocument('' + document.documentId);
                res.send({ message: 'Document was created successfully!' });
            })
        });
}
```

В цій функції окрім додавання нового документу в базу даних викликається метод createShareDbDocument, який підключається до ShareDb і створює там новий екземпляр документа, тобто після цього клієнти зможуть підключатися до редагування цього документа.

```
export const createShareDbDocument = (documentId) => {
    const connection = shareDBServer.connect();
    const doc = connection.get('documents', documentId);
    doc.fetch((err) => {
        if (err) throw err;
        if (doc.type === null) {
            doc.create([{ insert: 'Hello World!' }], 'rich-text', () => {
                console.log('Created shareDB document ' + documentId);
            })
        }
    })
}
```

3.6. Опис файлів даних та інтерфейсу програми

3.6.1. Опис файлів даних

Для даного застосунку було використано нереляційну базу даних MongoDB та ORM Mongoose для Node.js.

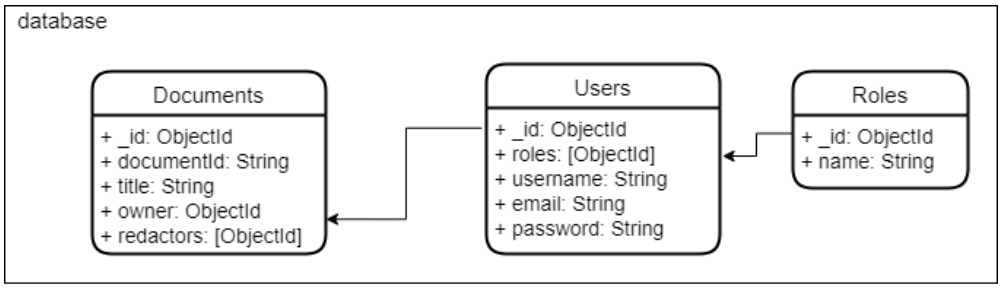
Нереляційні бази даних гарно підходять для даних, які погано структуруються у вигляді класичних таблиць. MongoDB є класичним представником нереляційних баз даний, а саме документно-орієнтованих. Документно-орієнтовані БД доволі новий вид баз даних, який розроблений для роботи з ієрархічними структурами даних (документами). Сховище даних має структуру дерева – кореневий вузол, від якого йдуть багато додаткових вузлів. Кожний вузол містить інформацію про доданий документ і формує індекси БД.

Документи можуть бути об’єднані в колекції. Колекції є певним аналогом таблиць з реляційних баз даних, але, на відміну від них, колекції можуть бути вкладені один в один для більш гнучкої індексації.

MongoDB має такі переваги:

- Легке масштабування
- Швидка робота при простих запитах (проте варто намагатися остерігатися запитів до різних видів документів).
- Збереження документів у вигляді JSON документів

На рисунку зображено схему колекцій бази даних MongoDB



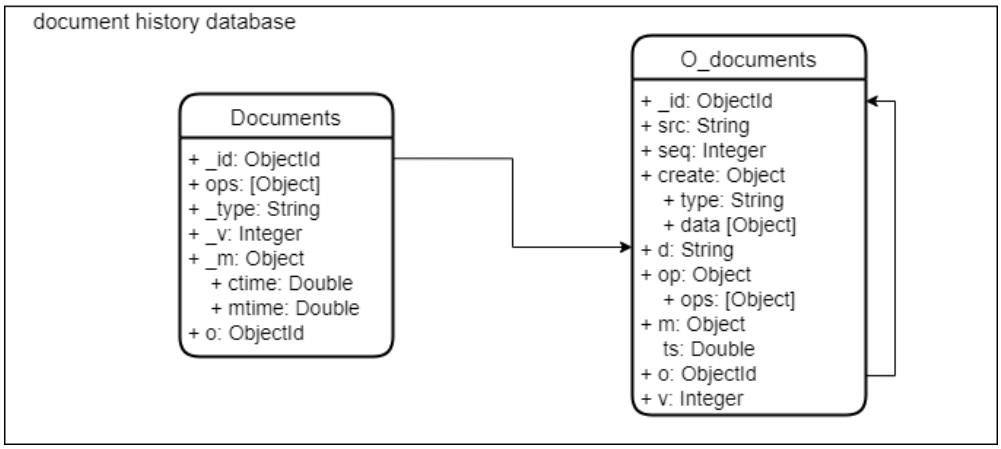


Рисунок 3.6 – Схема бази даних

3.6.2. Опис інтерфейсу програми

Коли неавторизований користувач заходить на сайт, він потрапляє на стартову сторінку, на якій може переглянути основну інформацію про застосунок.

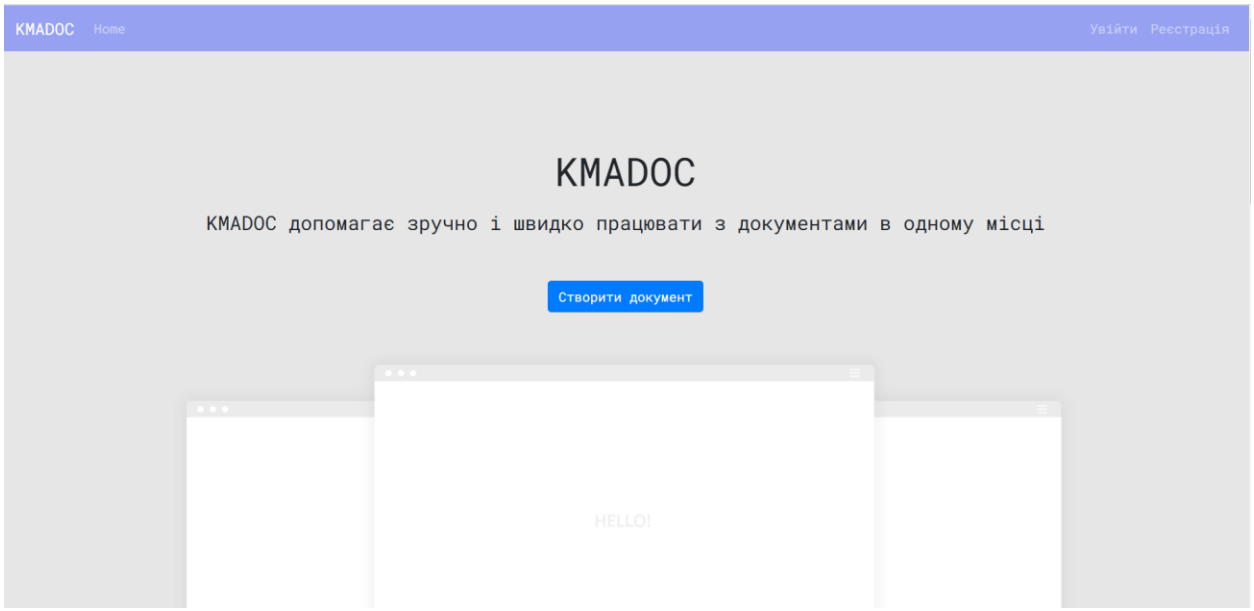


Рисунок 3.7 – Скріншот стартової сторінки

Рисунок 3.8 – Скріншот стартової сторінки (продовження)

Після авторизації користувач може створити новий документ. Після створення користувач може перейти до його редагування, змінити назву

документу, переглянути список тих, хто може редагувати текст і додати нового користувача до редагування тексту.

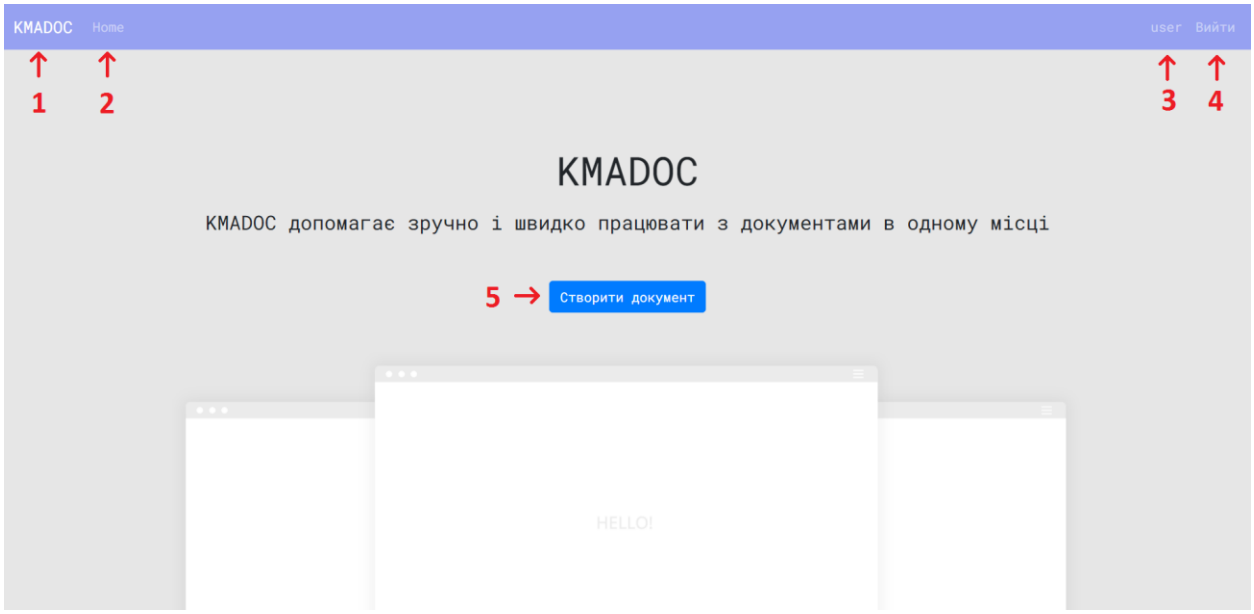


Рисунок 3.8 – Основні елементи на стартовій сторінці

1- Перехід на головну сторінку.
2- Повернення на головну сторінку .
3- Перехід на сторінку користувача, де знаходить інформація про користувача, список створених і список доступних для редагування документів.
4- Вийти з теперішнього аккаунту і перейти на сторінку авторизації.
5- Створити новий документ.

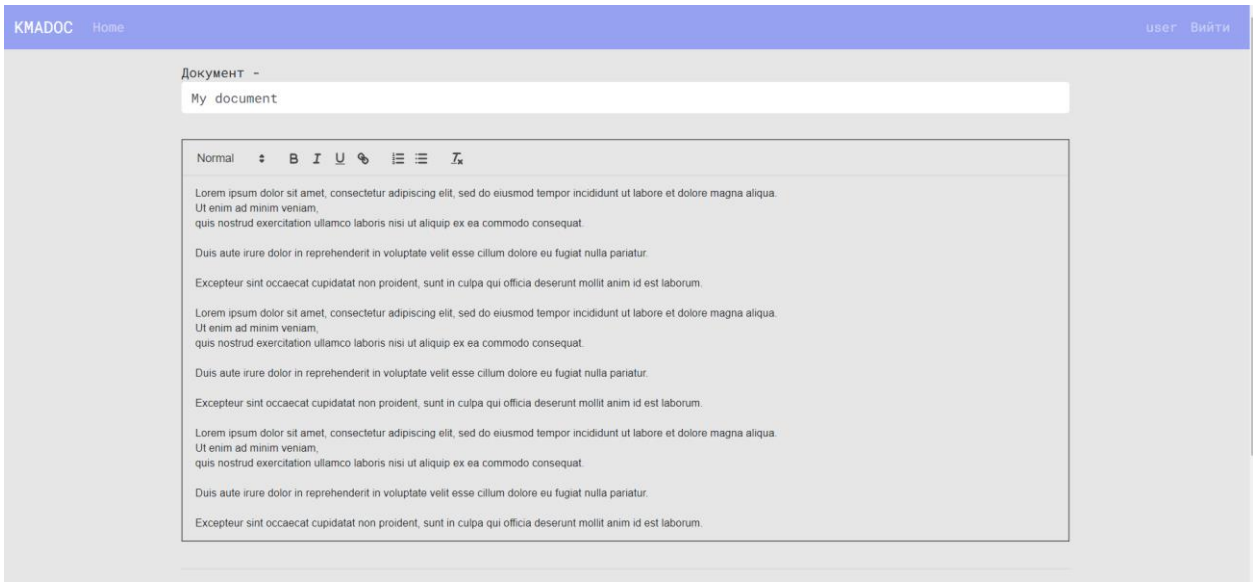


Рисунок 3.9 – Скріншот сторінки редагування документу

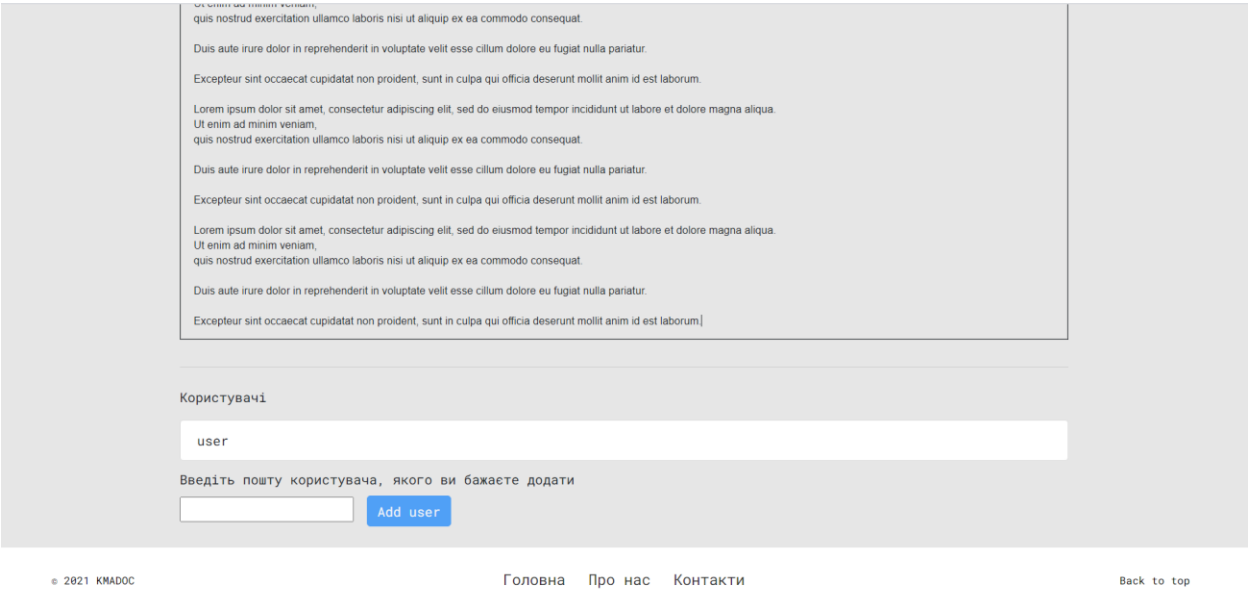


Рисунок 3.10 – Скріншот сторінки редагування документу (продовження)

3.7. Тестування програми і результати її виконання

При спробі створити новий документ користувач опиниться на сторінці авторизації, де йому буде запропоновано ввести свої данні або перейти на сторінку реєстрації.

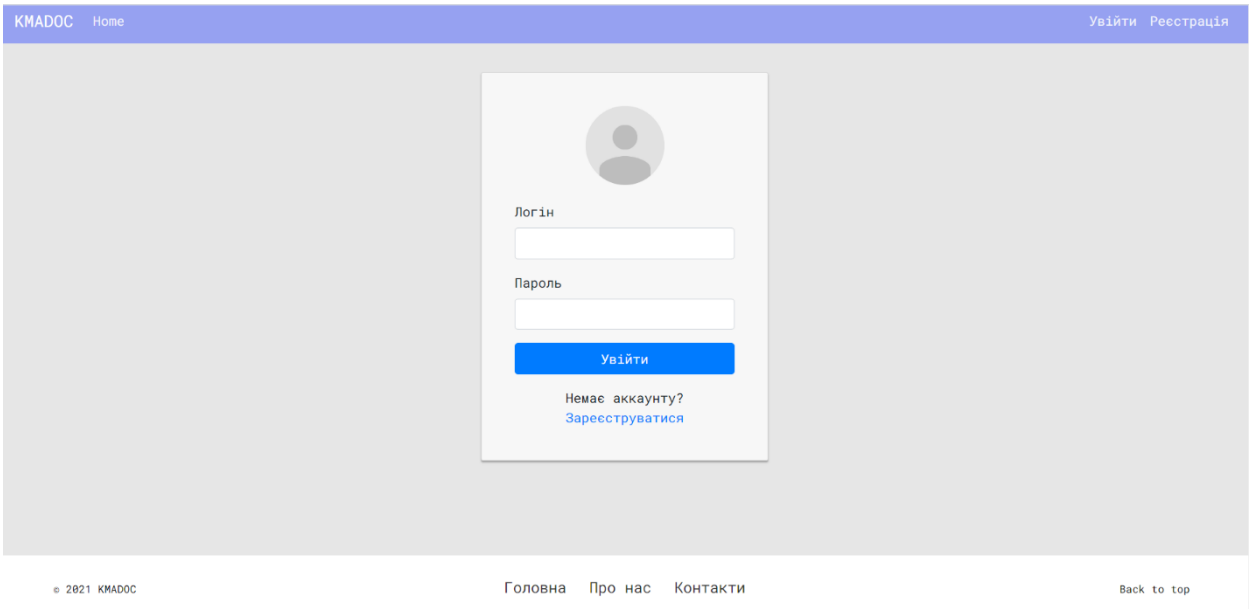


Рисунок 3.11 – Скріншот сторінки авторизації

При введені невірних даних буде виведено відповідно повідомлення.

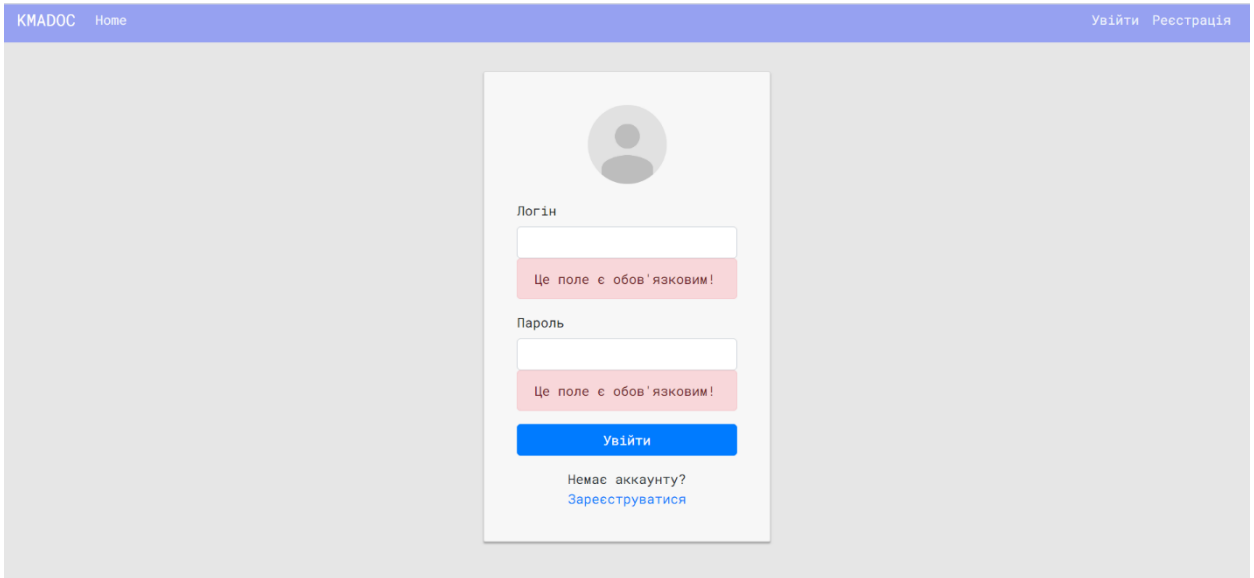


Рисунок 3.12 – Реакція на введення некоректних даних під час авторизації

Якщо користувач раніше не був зареєстрований, він може перейти на сторінку реєстрації та ввести необхідні дані.

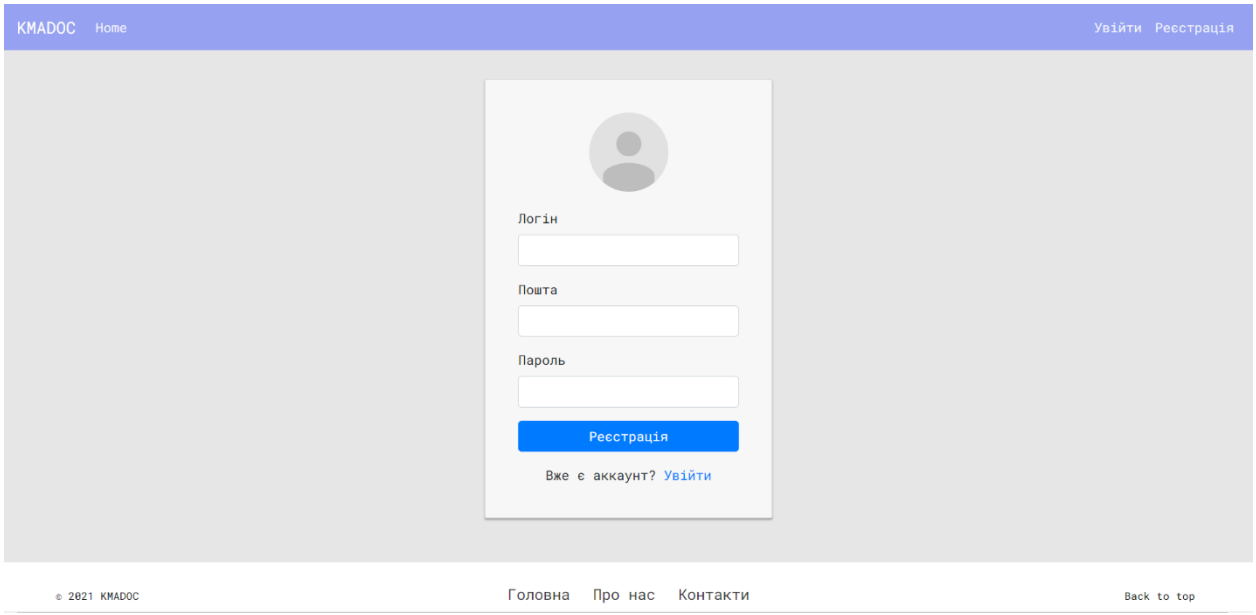


Рисунок 3.13 – Скріншот сторінки реєстрації

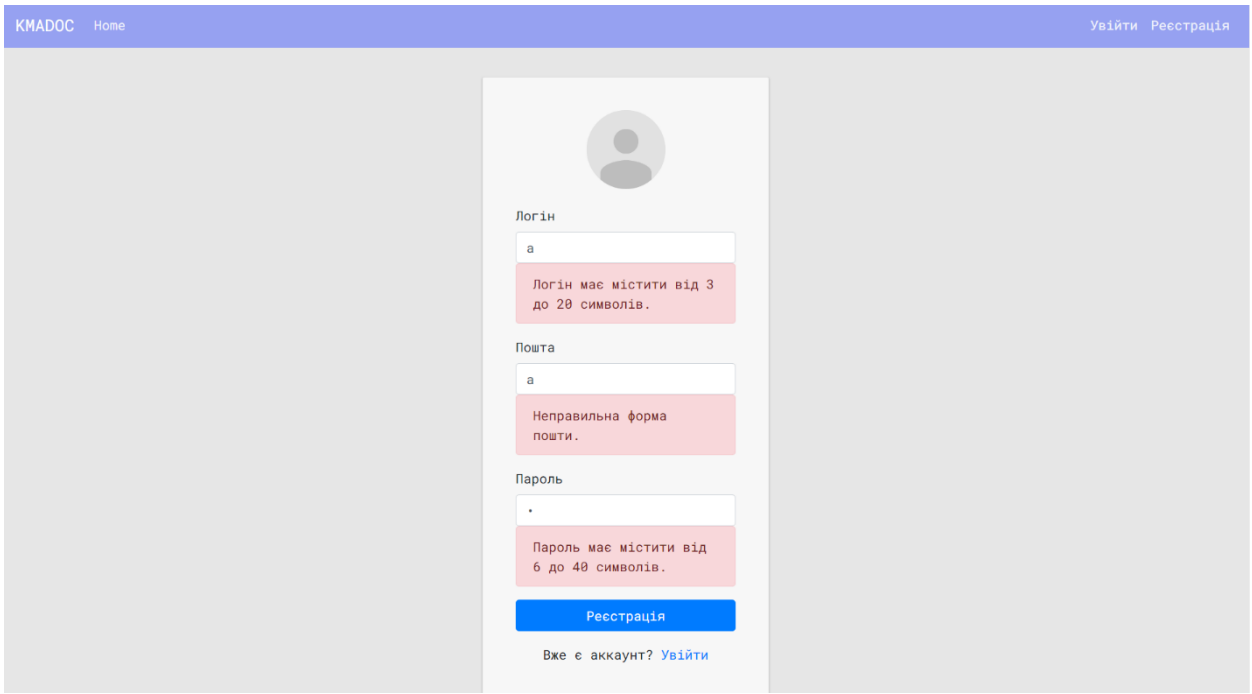


Рисунок 3.14 – Реакція на введення некоректних даних під час реєстрації

Після реєстрації та авторизації користувач потрапляє на головну сторінку, де є основна інформація про застосунок і користувач може створити новий документ.

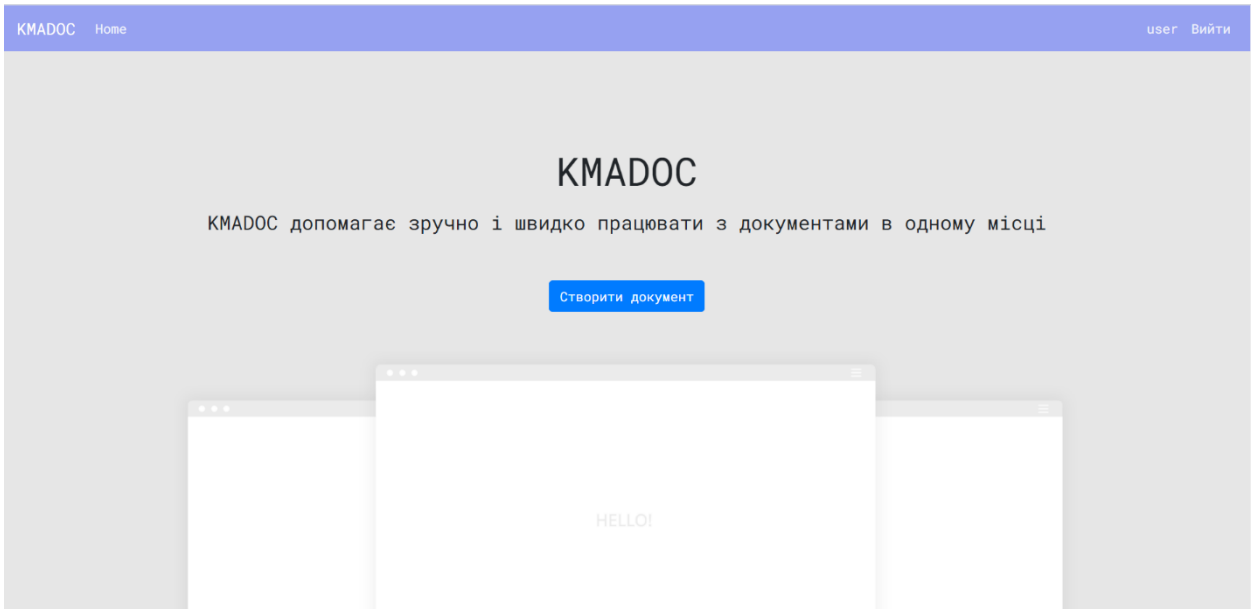


Рисунок 3.15 – Скріншот базової сторінки

Рисунок 3.16 –Скріншот базової сторінки (продовження)

Після створення нового документу користувач потрапляє на сторінку профілю. Тут знаходиться основна інформація про користувача, а також список створених та доступних до редагування документів. Власник документу може видалити цей документ.

Рисунок 3.17 – Скріншот сторінки користувача з його даними

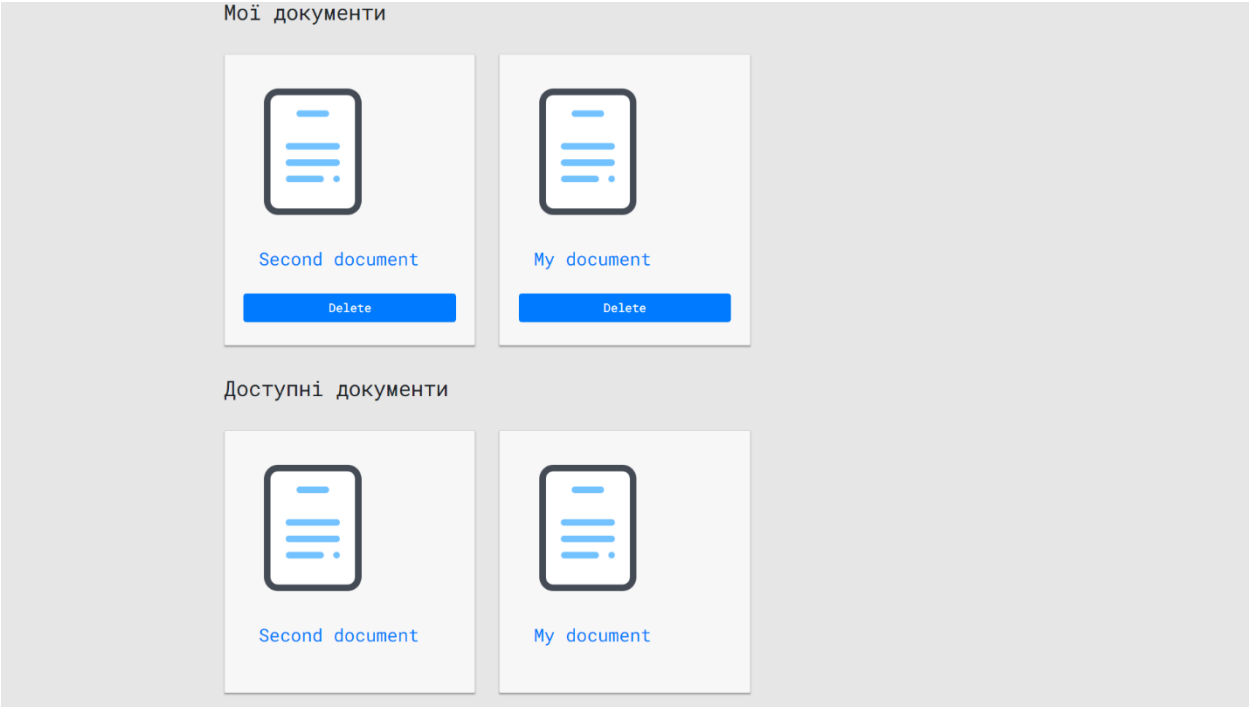


Рисунок 3.18 – Скріншот сторінки користувача з списком створених та доступних документів

Після натиснення на картку документу користувач потрапляє на сторінку редагування документу. Тут він має можливість змінити назву документу і редагувати сам документ. Для зміни назви документу достатньо ввести нову назву документу і вона автоматично збережеться в базі даних.

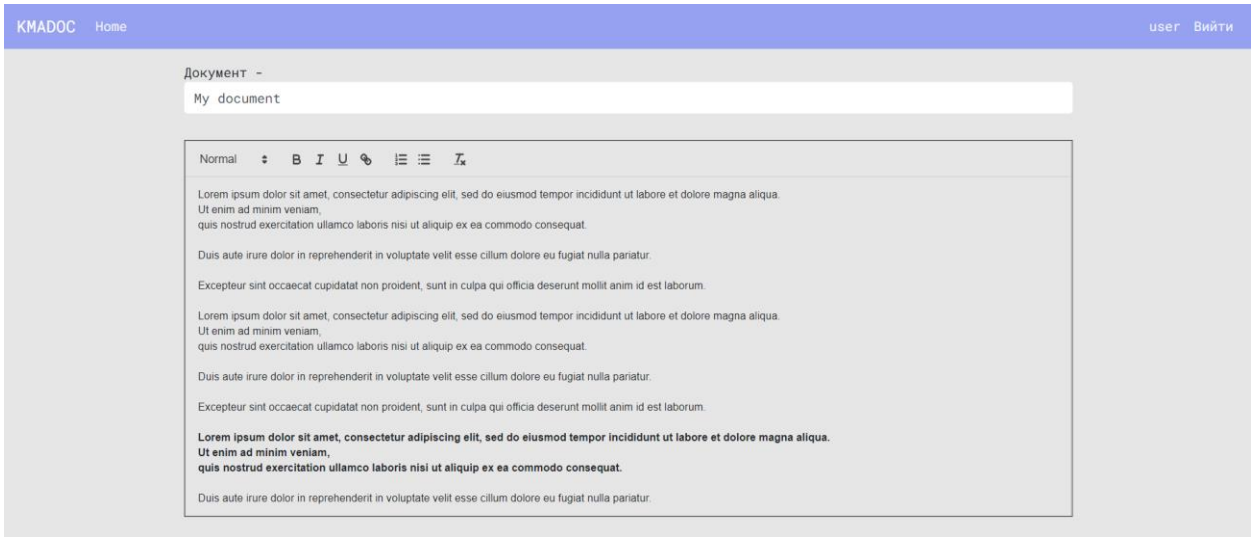


Рисунок 3.19 – Скріншот сторінки документу

Також користувач має можливість додати іншого користувача до редагування документу. Для цього треба ввести пошту користувача, якого треба додати до редагування.

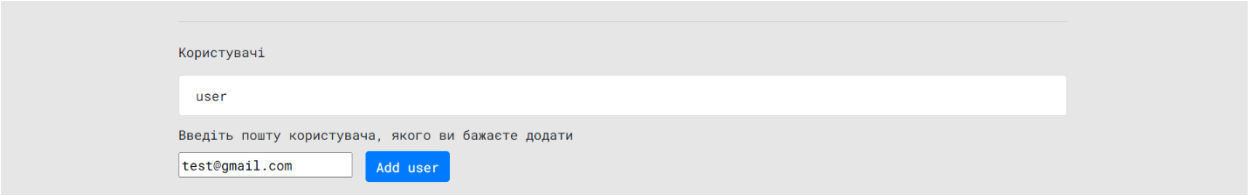


Рисунок 3.19 – Скріншот додавання користувача до редагування документу

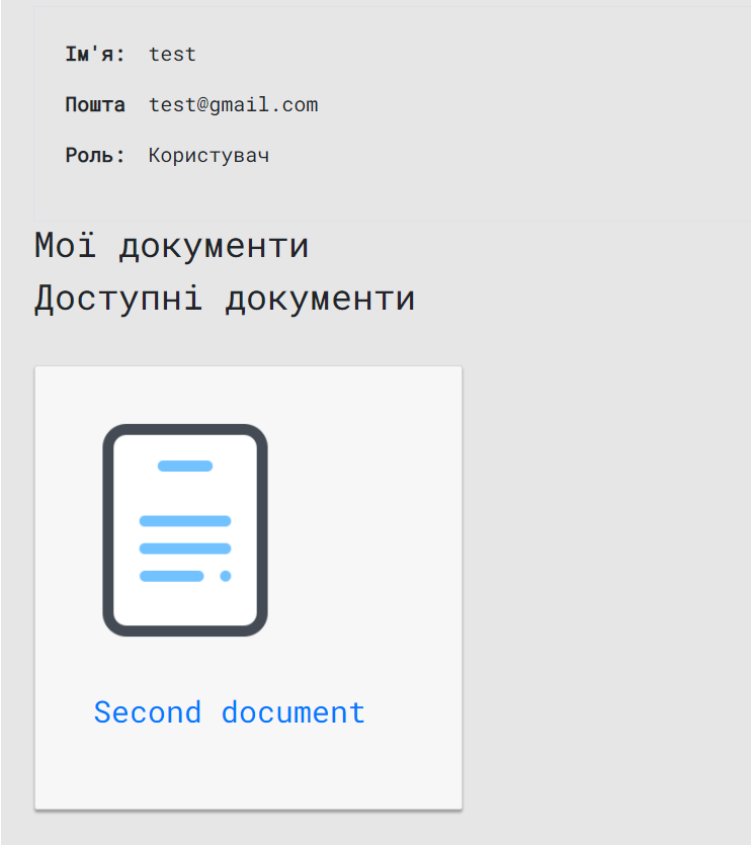


Рисунок 3.20 – Скріншот списку доступних документів користувача test

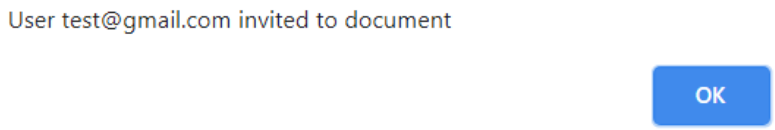


Рисунок 3.21 – Скріншот повідомлення про успішне додавання користувача до редагування документу

Після додавання користувач з поштою test@gmail.com отримав доступ до редагування документу.

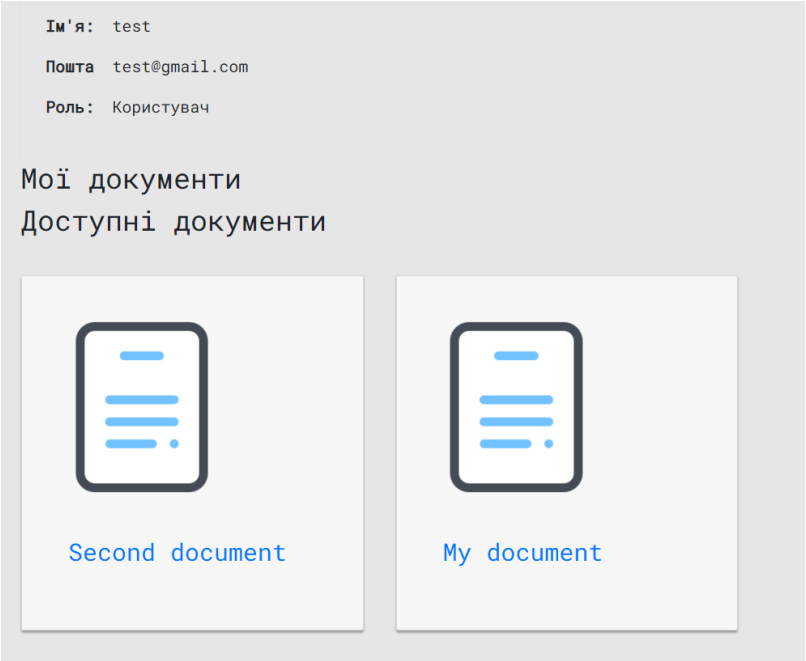


Рисунок 3.22 – Скріншот списку доступних документів користувача test (після додавання)

Під час редагування кожна зміна одного користувача відразу з’являється в іншого.

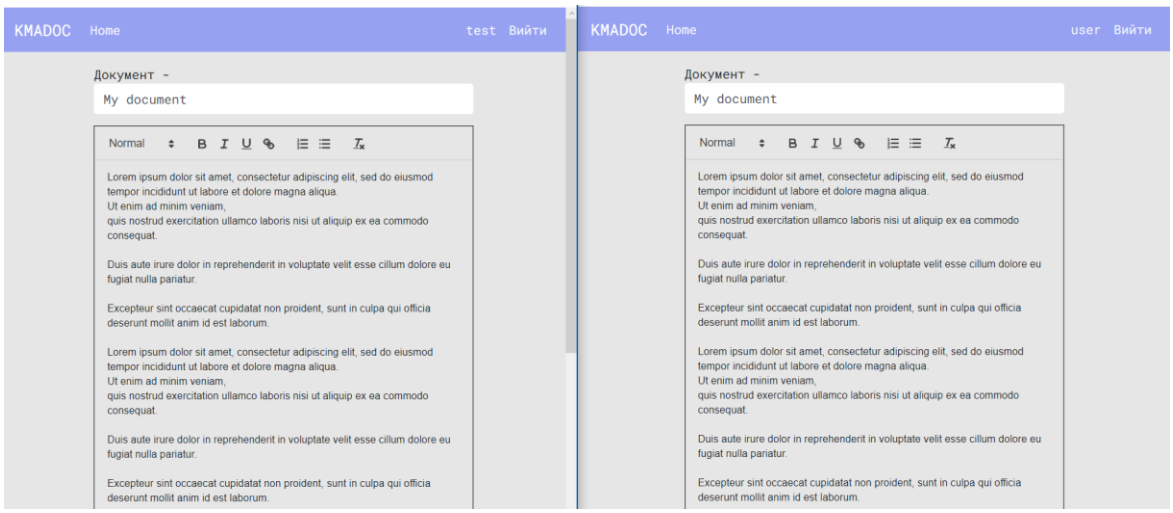


Рисунок 3.23 – Скріншот процесу паралельного редагування тексту

Серед доступних операцій є базові стилі текстового редактору, вставка та видалення тексту. Всі користувачі, які мають доступ до документу мають можливість працювати із його вмістом без затримок і очікування операцій інших користувачів.

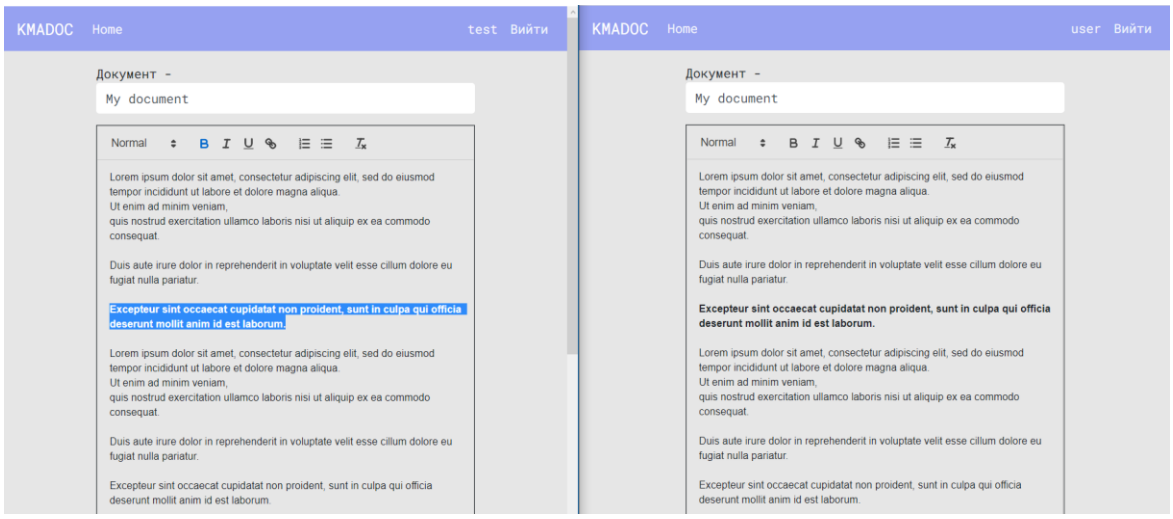


Рисунок 3.24 – Скріншот процесу паралельного редагування тексту (зміна стилю)

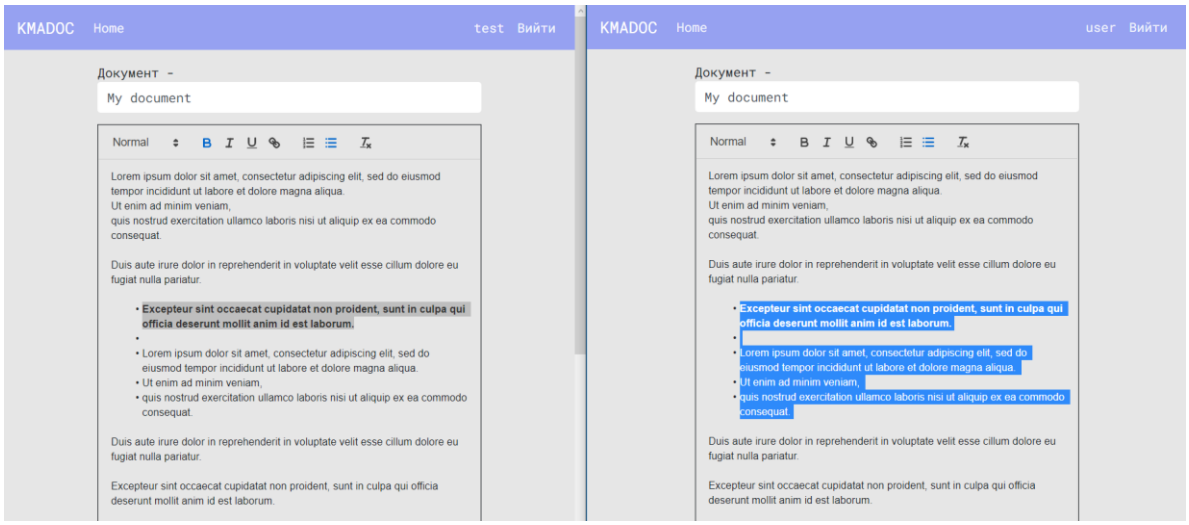


Рисунок 3.24 – Скріншот процесу паралельного редагування тексту (зміна стилю)

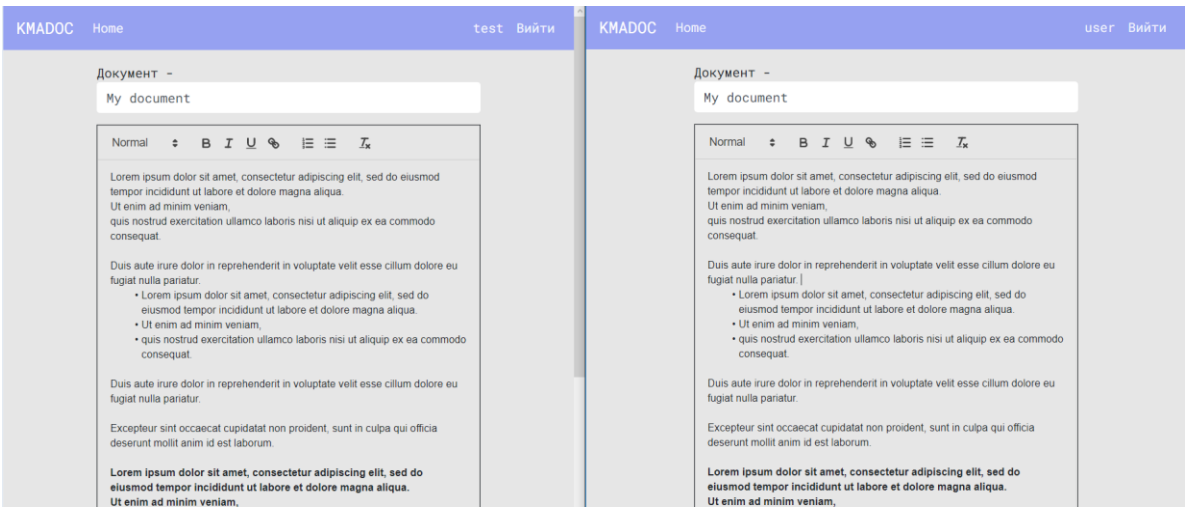


Рисунок 3.25 – Скріншот процесу паралельного редагування тексту (видалення тексту)

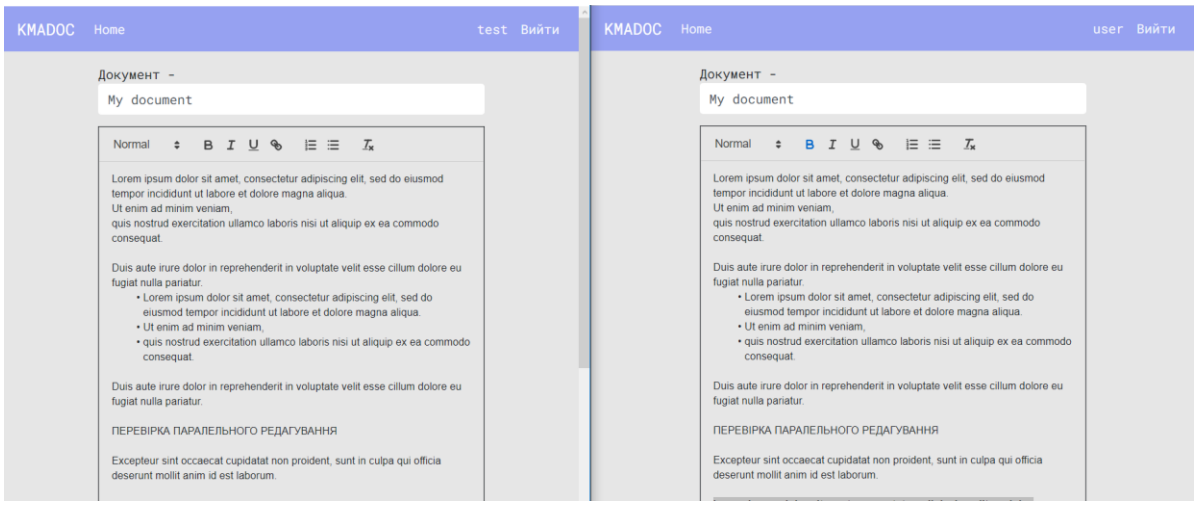


Рисунок 3.26 – Скріншот процесу паралельного редагування тексту (вставка тексту)

ВИСНОВКИ

Отже, мета роботи була досягнута і розроблений проект відповідає вимогам, описаним у постановці завдання. В ході виконання курсової роботи було оглянуто існуючі аналоги застосунків, виділено їх переваги; розглянуто основні методи синхронізації спільної роботи між користувачами, їх алгоритми та виділено переваги і недоліки кожного з методів. Окрім цього було розглянуто та проаналізовано основні способи передачі інформації між клієнтами та сервером і визначено який з розглянутих механізмів комунікації краще підходить для застосунку з спільним редагуванням документів.

В теперішньому варіанті застосунку можливе редагування звичайного тексту, але в подальшому розширенні функціоналу застосунку можна додати роботу з відео та фото, роботу з примітивними геометричними фігурами, розширення можливостей форматування тексту.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Google Docs [Електронний ресурс]. Режим доступу: https://docs.google.com/
2. Firepad [Електронний ресурс]. Режим доступу: https://firepad.io/
3. The Shannon-Hartley Theorem [Електронний ресурс]: https://www.ingenu.com/2016/07/back-to-basics-the-shannon-hartley-theorem/
4. Vector Clock [Електронний ресурс]. Режим доступу: https://riak.com/why-vector-clocks-are-easy/
5. OTFAQ: Operational Transformation Question [Електронний ресурс]. Режим доступу: https://www3.ntu.edu.sg/scse/staff/czsun/projects/otfaq/
6. Semi-lattice – Encyclopedia of Mathematics [Електронний ресурс]. Режим доступу: https://encyclopediaofmath.org/wiki/Semi-lattice
7. HTTP [Електронний ресурс]. Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP
8. Long polling [Електронний ресурс]. Режим доступу: https://javascript.info/long-polling
9. Server-Sent Events [Електронний ресурс]. Режим доступу: https://javascript.info/server-sent-events
10. Документація Node.js [Електронний ресурс]. Режим доступу: https://nodejs.org/en/docs/
11. The Node.js Event Loop [Електронний ресурс]. Режим доступу: https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/
12. Visual Studio Code [Електронний ресурс]. Режим доступу: https://code.visualstudio.com/
13. React [Електронний ресурс]. Режим доступу: https://reactjs.org/
14. База даних MongoDB [Електронний ресурс]. Режим доступу: https://www.mongodb.com/
15. Mongoose ODM [Електронний ресурс]. Режим доступу: https://www.mongodb.com/
16. Express – Node.js web application framework [Електронний ресурс]. Режим доступу: http://expressjs.com/

17. Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. [Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA., 2011. 50p.
18. Sharedb – real time database backend based on Operational Transformation (OT) of JSON documents [Електронний ресурс]. Режим доступу: https://github.com/share/sharedb
19. Cooperation, concurrency, conflict and convergence [Електронний ресурс]. Режим доступу: https://web.archive.org/web/20150928170646/http://opencoweb.org/ocwdocs/intro/openg.html
20. RFC 4122 - A Universally Unique IDentifier (UUID) URN Namespace [Електронний ресурс]. Режим доступу: https://tools.ietf.org/html/rfc4122
21. Key-CRDT Stores [Електронний ресурс]. Режим доступу: https://run.unl.pt/bitstream/10362/7802/1/Sousa_2012.pdf
22. Visualization of OT with a central server [Електронний ресурс]. Режим доступу: https://operational-transformation.github.io/
23. Operational Transformation, the real time collaborative editing algorithm [Електронний ресурс]. Режим доступу: https://hackernoon.com/operational-transformation-the-real-time-collaborative-editing-algorithm-bf8756683f66
24. WebSocket [Електронний ресурс]. Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket
25. Quill - Your powerful rich text editor [Електронний ресурс]. Режим доступу: https://quilljs.com/
26. Bootstrap [Електронний ресурс]. Режим доступу: https://getbootstrap.com/

Додаток А. Код серверної частини

server.js – файл запуску застосунку

```
import express from 'express';
import http from 'http';
import bodyParser from 'body-parser';
import cors from 'cors';
import db from './app/models/index.js';
import dbConfig from './app/config/db.config.js';
import authRoutes from './app/routes/auth.routes.js';
import userRoutes from './app/routes/user.routes.js';
import documentRoutes from './app/routes/document.routes.js';

import WebSocket from 'ws';
import WebSocketJSONStream from '@teamwork/websocket-json-stream';
import ShareDB from 'sharedb';
import richText from 'rich-text';

import { shareDBServer } from './serverShareDb.js'

const app = express();
const server = http.createServer(app);
ShareDB.types.register(richText.type);

const wss = new WebSocket.Server({ server: server });
wss.on('connection', (ws) => {
    const jsonStream = new WebSocketJSONStream(ws);
    shareDBServer.listen(jsonStream);
});

const corsOptions = {
    origin: 'http://localhost:3000'
};

app.use(cors(corsOptions));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: true }));

const Role = db.role;

db.mongoose
    .connect(`mongodb://${dbConfig.HOST}:${dbConfig.PORT}/${dbConfig.DB}`, {
        useNewUrlParsed: true,
        useUnifiedTopology: true
    })
    .then(() => {
        console.log('Successfully connect to MongoDB.');
        initial();
    })
    .catch(err => {
        console.error('Connection error', err);
        process.exit();
    });
authRoutes(app);
userRoutes(app);
documentRoutes(app);
const PORT = process.env.PORT || 8080;
server.listen(PORT);
console.log(`Server is running on port ${PORT}`);
```

shareShareDb.js – файл створення серверу shareDb та додавання до нього документів

```
import shareDbMongo from 'sharedb-mongo';
import dbConfig from './app/config/db.config.js';
import ShareDB from 'sharedb';

export const documentDb = shareDbMongo(`mongodb://${dbConfig.HOST}:${dbConfig.PORT}/${d
bConfig.SHARE_DOCUMENTS_DB}`);
export const shareDBServer = new ShareDB({ db: documentDb });

export const createShareDbDocument = (documentId) => {
    const connection = shareDBServer.connect();
    const doc = connection.get('documents', documentId);
    doc.fetch((err) => {
        if (err) throw err;
        if (doc.type === null) {
            doc.create([{ insert: 'Hello World!' }], 'rich-text', () => {
                console.log('Created shareDB document ' + documentId);
            })
        }
    })
}
```

document.controller.js – файл контролер роботи з документами

```
import db from '../models/index.js';
import { v4 as uuidv4 } from 'uuid'
import { createShareDbDocument } from '../../serverShareDb.js';

const Document = db.document;
const User = db.user;

const createDocument = (req, res) => {
    const token = JSON.parse(
        Buffer.from(req.headers['x-access-
token'].split('.')[1], 'base64').toString('ascii')
    );
    User.findOne({ username: token.username })
        .exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }

            if (!user) {
                return res.status(404).send({ message: 'User not found.' });
            }
            const document = new Document({
                documentId: uuidv4(),
                title: 'Title document',
                text: '',
                redactors: [user._id],
                owner: user._id
            });

            document.save(err => {
                if (err) {
```

```
                    res.status(500).send({ message: err });
                    return;
                }
                createShareDbDocument('' + document.documentId);
                res.send({ message: 'Document was created successfully!' });
            })
        });
}

const deleteDocument = (req, res) => {
    const token = JSON.parse(
        Buffer.from(req.headers['x-access-
token'].split('.')[1], 'base64').toString('ascii')
    );
    const documentId = req.params.documentId;
    User.findOne({ username: token.username })
        .exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }
            if (!user) {
                return res.status(404).send({ message: 'User not found.' });
            }

            Document.deleteOne({ documentId: documentId, owner: user._id })
                .exec((err, result) => {
                    if (err) {
                        res.status(500).send({ message: err });
                        return;
                    }

                    res.send({ message: 'Document was deleted successfully!' });
                });
        });
}

const changeDocumentTitle = (req, res) => {
    Document.findOne({ documentId: req.body.documentId })
        .exec((err, document) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }

            if (!document) {
                return res.status(404).send({ message: 'Document not found.' });
            }

            document.title = req.body.newDocumentTitle ? req.body.newDocumentTitle : 'D
 ocument title';
            document.save();

            res.send({ message: 'Document title was changed successfully!' });
        })
}

const getDocumentInfo = (req, res) => {
    const documentId = req.params.documentId;
    Document.findOne({ documentId: documentId })
        .exec((err, document) => {
```

```
            if (err) {
                res.status(500).send({ message: err });
                return;
            }

            if (!document) {
                res.status(404).send({ message: 'Document not found.' });
                return;
            }

            User.find({ _id: { $in: document.redactors } }, 'username', {},
                (err, users) => {
                    if (err) {
                        res.status(500).send({ message: err });
                        return;
                    }
                    res.send({ documentTitle: document.title, documentRedactors: users
});
                })
        })
}
const inviteUserToDocument = (req, res) => {
    const documentId = req.body.documentId;
    const userEmail = req.body.userEmail;
    const token = JSON.parse(
        Buffer.from(req.headers['x-access-
token'].split('.')[1], 'base64').toString('ascii')
    );

    User.findOne({ username: token.username })
        .exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }

            if (!user) {
                return res.status(404).send({ message: 'User not found.' });
            }

            User.findOne({ email: userEmail })
                .exec((err, inviteUser) => {
                    if (err) {
                        res.status(500).send({ message: err });
                        return;
                    }

                    if (!inviteUser) {
                        return res.status(404).send({ message: 'User with email ' + use
rEmail + ' not found.' });
                    }
                    Document.findOne({ documentId: documentId, owner: user._id })
                        .exec((err, docToModify) => {
                            if (err) {
                                res.status(500).send({ message: err });
                                return;
                            }

                            if (!docToModify) {
                               res.status(404).send({ message: 'Document not found' });
                                return;}
```

```
                            inviteUser.redactedDocuments.push(docToModify._id);
                            docToModify.redactors.push(inviteUser._id);

                            inviteUser.save();
                            docToModify.save();

                            res.send({ message: 'User invited to document successfully'
 });
                        });
                    });
        });
}

export { createDocument, deleteDocument, inviteUserToDocument, getDocumentInfo, changeD
ocumentTitle }
```

user.controller.js – файл контролер роботи з користувачем

```
import db from '../models/index.js';

const User = db.user;
const Document = db.document;

export const userDocuments = (req, res) => {
    const token = JSON.parse(
        Buffer.from(req.headers['x-access-
token'].split('.')[1], 'base64').toString('ascii')
    );
    User.findOne({ username: token.username })
        .exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }
            if (!user) {
                return res.status(404).send({ message: 'User not found.' });
            }
            Document.find({ owner: user._id }, 'documentId title', {},
                (err, documents) => {
                    if (err) {
                        res.status(500).send({ message: err });
                        return;
                    }
                    Document.find({ redactors: user._id }, 'documentId title', {},
                        (err, redactedDoc) => {
                            if (err) {
                                res.status(500).send({ message: err });
                                return;
                            }
                            res.send({ ownerDocuments: documents, redactorDocuments: re
dactedDoc });
                        });
                }
            );
        });
}
```

auth.Jwt.js – файл якій містить функції-посередники для перевірки валідності авторизованого користувача.

```
import jwt from 'jsonwebtoken';
import * as config from '../config/auth.config.js';

const verifyToken = (req, res, next) => {
    const token = req.headers['x-access-token'];

    if (!token) {
        return res.status(403).send({ mes: 'Такого токену не існує' });
    }

    jwt.verify(token, config.secret, (err, decoded) => {
        if (err) {
            return res.status(401).send({ mes: 'Такий користувач неавторизований' });
        }
        req.userId = decoded.id;
        next();
    });
}

export { verifyToken }
```

verifySignUp.js – файл якій містить функції-посередники для перевірки дублікатних даних під час реєстрації.

```
import db from '../models/index.js';
const ROLES = db.ROLES;
const User = db.user;

const checkDuplicateUsernameOrEmail = (req, res, next) => {
    User.findOne({
        username: req.body.username
    }).exec((err, user) => {
        if (err) {
            res.status(500).send({ message: err });
            return;
        }

        if (user) {
            res.status(400).send({ message: 'Такий користувач вже існує' });
            return;
        }
```

```
        User.findOne({
            email: req.body.email
        }).exec((err, user) => {
            if (err) {
                res.status(500).send({ message: err });
                return;
            }

            if (user) {
                res.status(400).send({ message: 'Користувач з такою поштою вже
існує' });
                return;
            }

            next();
        });
    });
};

export { checkDuplicateUsernameOrEmail }
```

document.routes.js – файл з реєстрацією в застосунку точок API для взаємодії з даними про документи.

```
import { verifyToken } from '../middlewares/index.js';
import * as controller from '../controllers/document.controller.js';

export default (app) => {
    app.use((req, res, next) => {
        res.header(
            'Access-Control-Allow-Headers',
            'x-access-token, Origin, Content-Type, Accept'
        );
        next();
    });

    app.post('/api/createDocument', [verifyToken], controller.createDocument);
    app.delete('/api/deleteDocument/:documentId', [verifyToken], controller.deleteDocum
ent);
    app.post('/api/inviteUser', [verifyToken], controller.inviteUserToDocument);
    app.get('/api/documentInfo/:documentId', [verifyToken], controller.getDocumentInfo)
;
    app.post('/api/changeDocumentTitle', [verifyToken], controller.changeDocumentTitle)
}
```

document.model.js – файл з моделлю документа.

```
import mongoose from 'mongoose';

const Document = mongoose.model('Document',
    new mongoose.Schema({
        documentId: String,
        title: String,
        owner: {
            type: mongoose.Schema.Types.ObjectId,
            ref: 'User'
        },
        redactors: [
            {
                type: mongoose.Schema.Types.ObjectId,
                ref: 'User'
            }
        ],
        text: String
    }));

export default Document
```

user.model.js – файл з моделлю користувача

```
import mongoose from 'mongoose';
const User = mongoose.model(
    'User',
    new mongoose.Schema({
        username: String,
        email: String,
        password: String,
        roles: [
            {
                type: mongoose.Schema.Types.ObjectId,
                ref: 'Role'
            }
        ],
        createdDocuments: [
            {
                type: mongoose.Schema.Types.ObjectId,
                ref: 'Document'
            }
        ],
        redactedDocuments: [
            {
                type: mongoose.Schema.Types.ObjectId,
                ref: 'Document'
            }
        ]
    })
);
export default User;
```

Додаток Б. Код клієнтської частини

App.js – основний файл застосунку

```
… import …
const App = () => {
    const { user: currentUser } = useSelector((state) => state.auth);
    const disp = useDispatch();

    useEffect(() => {
      hist.listen((location) => {
          disp(clearMessage());
      });
    }, [disp]);

    const logOut = () => {
        disp(logout());
    };

 return(
     <Router history={hist}>
         <div className="page-container">
             <div className="content-wrap">
             <nav className="navbar navbar-expand navbar-dark navbar-custom">
                 <a to={"/"} className="navbar-brand">KMADOC</a>
                 {currentUser ? (
                     <div className="navbar-nav ">
                     <li className="nav-item">
                         <a href="/profile" className="nav-link">
                         {currentUser.username}
                         </a>
                     </li>
                     <li className="nav-item">
                         <a href="/login" className="nav-link" onClick={logOut}>
                         Вийти
                         </a>
                     </li>
                     </div>
                 ) : (
                     <div className="navbar-nav ">
                     <li className="nav-item">
                         <a href="/login" className="nav-link">
                         Увійти
                         </a>
                     </li>

                       <li className="nav-item">
                           <a href="/register" className="nav-link ">
                           Реєстрація
                           </a>
                       </li>
                       </div>
                   )}
             </nav>

             <div className="container mt-3">
                 <Switch>
                     <Route path={["/", "/home"]} component={Home} />
                     <Route path="/login" component={LoginComponent} />
                     <Route path="/register" component={RegisterComponent} />
                     <Route path="/profile" component={ProfileComponent} />
                     <Route path="/document/:documentId" component={Document} />
```

```
                    </Switch>
                  </div>
                  </div>
                  <Footer/>
              </div>
          </Router>
      )


}

export default App;
```

document.jsx – сторінка редагування документу.

```
… import …
Sharedb.types.register(richText.type);
const WAIT_INTERVAL = 4000;
let timer = null;

const Document = (props) => {
    const [inviteUserEmail, setInviteUserEmail] = useState("");
    const [redactors, setRedactors] = useState();
    const [documentTitle, setDocumentTitle] = useState("");

    const handleClick = (e) => {
        documentService.inviteUserToDocument(props.match.params.documentId, inviteUserE
mail)
            .then((res) => {
                alert('User '+inviteUserEmail+' invited to document');
                setInviteUserEmail("");
            })
            .catch((err) => console.log('Error while inviting user to document ', err))
;
    }

    useEffect(() => {
        console.log('call user redactors effect');
        let mounted = true;
        documentService.getDocumentInfo(props.match.params.documentId)
        .then(response =>  mounted && (setRedactors([...response.data.documentRedactors
]) || setDocumentTitle(response.data.documentTitle)))
        .catch(error => console.log("error while get document redactors", error));
    }, [props.match.params.documentId]);


    useEffect(() => {
        // Connection to out socket server
        const socket = new ReconnectingWebSocket("ws://127.0.0.1:8080");
        const connection = new Sharedb.Connection(socket);
        const doc = connection.get("documents", '' + props.match.params.documentId);
        console.log('call subscribe effect');
        doc.subscribe((err) => {
            if (err) throw err;
            console.log('subscribed');
            let quill = new Quill("#editor", {
                theme: 'snow',
                modules: {
```

```
                toolbar: true,
                cursors: true,
                history: {
                     delay: 2000,
                     maxStack: 500,
                     userOnly: true
                 },
            },
        });
        quill.setContents(doc.data);

        quill.on("text-change", (d, oD, src) => {
            if (src !== "user") return;
            doc.submitOp(d, { src: quill });
        });

        doc.on("op", (op, src) => {
            if (src === quill) return;
            quill.updateContents(op);
        });
    });

    return () => {
        console.log('unsubscribe');
        connection.close();
        socket.close();
    };
}, [props.match.params.documentId]);

const handleChangeDocumentTitle = (newDocumentTitle) => {
    setDocumentTitle(newDocumentTitle);
    if(timer){
        clearTimeout(timer);
        timer = null;

    }
    timer = setTimeout(
        () => documentService.changeDocumentTitle(props.match.params.documentId, ne
wDocumentTitle ? newDocumentTitle : 'Document title')
            .then(res => console.log('change title'))
            .catch(err => console.log('error while change document title',err)),
            WAIT_INTERVAL);
}

const redactorsList = redactors && <ul className="list-group">
    { redactors.map(elem => {
    return <li key={elem._id} className="list-group-item">{elem.username}</li>
})
}
</ul>

return (
    <div className="container">
        <div>
            <span>Документ - </span>
            <input type="text" className="form-control document-title-
input" onChange={(e) => handleChangeDocumentTitle(e.target.value)} value={documentTitle
}/>
        </div>
        <div style={{ margin: "3% 0%", border: "1px solid" }}>
            <div id="editor"></div>
```

```
            </div>
            <hr className="my-4"></hr>
            <div>
                <div className="my-3">Користувачі</div>
                { redactorsList }
            </div>

            <div>
                <label>Введіть пошту користувача, якого ви бажаєте додати</label>
                <input type="email" name="addUserEmail" value={inviteUserEmail} onChang
e={(e) => setInviteUserEmail(e.target.value)} />
                <button type="button" disabled={!inviteUserEmail} className="btn btn-
primary ml-3" onClick={handleClick}>Додати користувача</button>
            </div>
        </div>
    );
};

export default Document;
```

footer.jsx – сторінка підвалу.

```
import React from 'react';
import './footer.css';
import { Link } from 'react-router-dom';


const Footer = () => {
    return (
        <div className="footer-basic footer mt-4 py-3">
            <footer>
                <div className="row">
                    <div className="col-2">
                        <p className="copyright">© 2021 KMADOC</p>
                    </div>
                    <div className="col-8">
                        <ul className="list-inline my-2">
                            <li className="list-inline-
item"><Link to={{pathname: "/"}}>Головна</Link></li>
                            <li className="list-inline-
item"><Link to={{pathname: "/about"}}>Про нас</Link></li>
                            <li className="list-inline-
item"><Link to={{pathname: "/contacts"}}>Контакти</Link></li>
                        </ul>
                    </div>
                    <div className="col-2">
                        <p className="copyright">Back to top</p>
                    </div>
                </div>

            </footer>
        </div>
    )
}

export default Footer;
```

home.jsx – сторінка з основною інформацією про застосунок.

```
… import …
const Home = (props) => {

    const handleClick = (e) => {
        documentService.createDocument()
            .then((response) => {
                props.history.push("/profile");
                window.location.reload();
            })
            .catch((err) => {
                if(err.message.includes('Request failed with status code 403')){
                    props.history.push('/login');
                    window.location.reload();
                }
                console.log('error', err);
            });
    }

    return (
        <div>
            <div className="home-head text-center">
                <h2 className="home-head-name">KMADOC</h2>
                <h4 className="home-head-
description">KMADOC допомагає зручно і швидко працювати з документами в одному місці</h
4>
                <button type="button" onClick={handleClick} className="btn btn-
primary btn block mt-5 create-document">Створити документ</button>

               <img className="mt-5" src={homePageImg} alt={'Home page'}/>
            </div>
            <div>
            <div className="container home-main-container">
                <h3 className="mb-5 text-center display-4">Чому KMADOC</h3>
                <div className="t-col t-prefix_2 t-col_8">
                    <div className="t-cell">
                        <img className="t-img" src={cloudImg} alt="Ніяких файлів"/>
                    </div>
                    <div className="t-cell pl-4">
                        <div className="t-name">Ніяких файлів</div>
                        <div class="t-
description">Більше ніяких електронних листів з вкладеними файлами. Всі версії документ
у знаходяться в одному місці.</div>
                    </div>
                </div>
                <div class="t-separator"></div>
                <div class="t-col t-prefix_2 t-col_8">
                    <div class="t-cell">
                        <img class="t-
img" src={textEditorImg} alt="Зручний текстовий редактор"/>
                    </div>
                    <div class="t-cell pl-4">
                        <div class="t-name">Зручний текстовий редактор</div>
                        <div class="t-
description">KMADOC - це редактор тексту. Ви можете писати на будь-
якій мові. Текст може бути відформатований жирним шрифтом, курсивом, підкресленням, мож
лива зміна розміру тексту, міжстрокового інтервалу і багато іншого.</div>
                    </div>
```

```
                </div>
                <div className="t-separator"></div>
                <div className="t-col t-prefix_2 t-col_8">
                    <div className="t-cell">
                        <img className="t-img" src={forkImg} alt="Просте поширення"/>
                    </div>
                    <div className="t-cell pl-4">
                        <div className="t-name">Просте поширення</div>
                        <div className="t-
description">Поділитися документом з іншою людиною можна відправивши URL адресу. Співав
тори можуть легко отримати доступ до редагування документу.</div>
                    </div>
                </div>
                <div className="t-separator"></div>
                <div className="t-col t-prefix_2 t-col_8">
                    <div className="t-cell">
                        <img className="t-img" src={collaborativeImg} alt="Співпраця"/>
                    </div>
                    <div className="t-cell pl-4">
                        <div className="t-name">Співпраця</div>
                        <div className="t-
description">Декілька користувачів можуть працювати одночасно. Зміни в документі бачать
всі автори та читачі в режимі реального часу.</div>
                    </div>
                </div>

               <div className="t-separator"></div>
                <div className="t-col t-prefix_2 t-col_8">
                    <div className="t-cell">
                        <img className="t-img" src={undoImg} alt="Збережена історія"/>
                    </div>
                    <div className="t-cell pl-4">
                        <div className="t-name">Збережена історія</div>
                        <div className="t-
description">Вся історія змін документу зберігається на сервері і вона доступна в будь-
який момент часу.</div>
                    </div>
                </div>
                <div className="t-separator"></div>
                <div className="t-col t-prefix_2 t-col_8">
                    <div className="t-cell">
                        <img className="t-img" src={secureImg} alt="Безпека"/>
                    </div>
                    <div className="t-cell pl-4">
                        <div className="t-name">Безпека</div>
                        <div className="t-
description">Доступ до ваших документів мають лише ті, кому ви надасте доступ.</div>
                    </div>
                </div>
                <div className="t-separator"></div>
            </div>
            </div>
        </div>
    )
}

export default Home;
```

profile.jsx – сторінка користувача.

```
… import …
const Profile = () => {
    const { user: currentUser } = useSelector((state) => state.auth);
    const [ownerDocuments, setOwnerDocuments] = useState();
    const [redactorDocuments, setRedactorDocuments] = useState();

    useEffect(() => {
        let mounted = true;
        userService.getUserDocuments()
        .then(response => mounted && (setOwnerDocuments([...response.data.ownerDocument
s]) || setRedactorDocuments([...response.data.redactorDocuments])))
        .catch(error => console.log("error while get user documents", error));
    }, []);

    if (!currentUser) {
        return <Redirect to="/login" />;
    }

    const handleDeleteDocument = (documentId) => {
        documentService.deleteUserDocument(documentId)
            .then(response => window.location.reload())
            .catch(err => console.log('error while delete ', err));
    }

    const ownerDocumentList = ownerDocuments && ownerDocuments.map(elem => {
        return <div key={elem._id} className="col-xl-4 col-lg-4 col-md-6 col-sm-12  p-
2 p-md-3 document-card">

                <div className="card">
                    <a href={{pathname :`/document/${elem.documentId}`}}>
                        <img className="card-img-
top" src={documentImg} alt="Document img"/>
                        <div>
                            <h4>{elem.title}</h4>
                        </div>
                    </a>
                    <button value={elem.documentId} className="btn btn-
primary" onClick={(e) => handleDeleteDocument(e.target.value)}>Delete</button>
                </div>


            </div>
    });

    const redactorDocumentList = redactorDocuments && redactorDocuments.map(elem => {
        return <div key={elem._id} className="col-xl-4 col-lg-4 col-md-6 col-sm-12 p-
2 p-md-3 document-card">

                <div className="card">
                    <Link to={{pathname :`/document/${elem.documentId}`}}>
                        <img className="card-img-
top" src={documentImg} alt="Document img"/>
                        <div className="card-body">
                            <h4 className="card-title">{elem.title}</h4>
                        </div>
                    </Link>
                </div>
```

```
            </div>
    });

    return (
        <div className="container">
            <div className="text-center">
                <svg
                    width="200"
                    height="200"
                    viewBox="0 0 200 200"                 >
                    <circle cx="100" cy="100" r="99.5" stroke="black" />
                    <path d="M95.84 57V141H102.44V57H95.84Z" fill="black" />
                </svg>
            </div>
            <div className="text-center display-4 my-2">
                <span>{currentUser.username}</span>
            </div>

            <div className="border">
                <div className="py-4 px-4">
                    <div className="form-group">
                        <span className="mr-2 font-weight-bold">
                            Ім'я:
                        </span>
                        <span> {currentUser.username}</span>
                    </div>
                    <div className="form-group">
                        <span className="mr-2 font-weight-bold">
                            Пошта
                        </span>
                        <span> {currentUser.email}</span>
                    </div>

                   <div className="form-group">
                        <span className="mr-2 font-weight-bold">
                            Роль:
                        </span>
                        <span> {currentUser.roles.map((role, index) => (
                            <span className="mr-
1">{role === 'ROLE_USER'? 'Користувач': ''}</span>
                        ))}</span>
                    </div>
                </div>
            </div>
            <h3>Мої документи</h3>
            <div className="row d-flex justify-content-start">{ownerDocumentList}</div>
            <h3>Доступні документи</h3>
            <div className="row d-flex justify-content-
start">{redactorDocumentList}</div>
        </div>
    );
};

export default Profile;
```

auth.service.js – файл з функціями відправки запитів на сервер (авторизація)

```
import axios from 'axios';
import { API_URL_SIGNIN, API_URL_SIGNUP } from '../config.js'

const register = (usrn, mail, pass) => {
    return axios.post(API_URL_SIGNUP, { usrn, mail, pass });
}

const login = (usrn, paswrd) => {
    return axios
        .post(API_URL_SIGNIN, { usrn, paswrd })
        .then((response) => {
            if (response.data.accessToken) {
                localStorage.setItem('user', JSON.stringify(response.data));
            }

            return response.data;
        });
}

const logout = () => {
    localStorage.removeItem('user');
}
```

document.service.js – файл з функціями запитів на сервер (робота з документами).

```
import axios from 'axios';
import { API_URL_ADD_USER_TO_DOCUMENT, API_URL_CHANGE_DOCUMENT_TITLE, API_URL_CREATE_DO
CUMENT, API_URL_DELETE_DOCUMENT, API_URL_GET_DOCUMENT_INFO } from '../config.js';
import authHeader from './auth-header.js';

const createDocument = () => {
    return axios.post(API_URL_CREATE_DOCUMENT, {}, { headers: authHeader() } );
}

const deleteUserDocument = (documentId) => {
    return axios.delete(API_URL_DELETE_DOCUMENT+'/'+documentId, { headers: authHeader()
 });
}

const inviteUserToDocument = (documentId, userEmail) => {
    return axios.post(API_URL_ADD_USER_TO_DOCUMENT, { documentId: documentId, userEmail
: userEmail }, { headers: authHeader() });
}

const getDocumentInfo = (documentId) => {
    return axios.get(API_URL_GET_DOCUMENT_INFO + '/'+documentId, { headers: authHeader(
) });
}

const changeDocumentTitle = (documentId, newDocumentTitle) => {
    return axios.post(API_URL_CHANGE_DOCUMENT_TITLE, { documentId: documentId, newDocum
entTitle: newDocumentTitle }, { headers: authHeader() });
}

export default { createDocument, deleteUserDocument, inviteUserToDocument, getDocumentI
nfo, changeDocumentTitle }
```