Distributed Computing Project Report - Distributed Text Editor - Team 5

Faculty of Engineering - Ain Shams University - Computer Engineering and Software Systems

Subject: CSE354 (Distributed Computing)

18P2491 Mazen Mostafa Fawzy Moussa

18P7232 Mohamed Mokhtar Abdelrasoul

18P7713 Mohamed Yasser AbdelSamad

18P6713 Youssef Maher Nader Halim

Group: 2

Section: 2

Due Date: 26/6/2022

Table of Contents

List of Figures

1. Introduction

For the "Distributed Computing" (CSE354) course's project we were tasked with creating a distributed text editor in the style of "Google docs". To this end, this document will serve as documentation and a user guide to how our implementation was done and how to interact with the program.

2. Detailed Project Description

The distributed text editor needed to have multiple vital features these included, as the name implies, the editor being distributed over multiple clients and/or servers, with the clients being able to contend for shared resources (i.e., the documents) and perform real-time updates to a shared state (i.e., editing the documents). The system should be able to handle the crash of any of the participating nodes and should recover the state of a crashed node once it can resume operation.

These requirements were fulfilled on the server-end by utilising Amazon Web Services such as Lambda and DynamoDB. Lambda provided us with a world-class robust and distributed back-end that is fault-tolerant, whereas in a traditional system if a server were to crash the crash would disrupt the whole system, Lambda overcomes this problem using its serverless architecture and the user's experience should not be heavily impacted. Additionally, DynamoDB provides us with a consistent database that is resistant to data loss, as there is no single point of failure.

3. Beneficiaries of the Project

This project can be used by students, educators in academia, and professionals in the industry to collaborate with their colleagues in order to create professional-looking documents while not being restricted to working on their documents locally and allowing global collaboration.

4. Detailed Analysis

Structure and Logic

To construct this system, we utilised some AWS services namely, DynamoDB and Lambda. The front-end is a website that the user can access normally through any web browser and through a simple graphical user interface (further elaborated on in the user guide section) the user can perform all the functionalities of the system. These functionalities include creating a document and editing the document in real-time with other agents.

The exact nature of how the front-end and back-end communicate is further explained in the System architecture and design portion of this report, in this section however we will focus on the justification of why we used the technologies we used. For instance, we utilised Lambda to have a serverless back-end that allows for world-class availability and abstracts away the handling of multiple servers to ensure that no single point of failure exists in the system. This removes the limitation of a server going down while processing requests and thus needing to communicate to another server with the same previously sent requests, which may lead to many errors. Additionally, we use DynamoDB to create a consistent database that can be accessed by the back-end without fear of loss of data due to the crash of the database server, as once again DynamoDB abstracts the access to the shared resources (i.e., the deltas of the different documents found on the system).

We are using the MVC architecture where the model is DynamoDB, the view is the website accessible by all users, and finally, the control is Lambda functions. The website communicates with Lambda over a web socket.

We use pseudo-operational transformation, whereby the document is transformed into a set of operations that can be replicated to reconstruct the document. The "pseudo" in pseudo-operational transformation refers to the fact that the operations are done on the client end and not on the back-end. This is done to distribute the computational load between the client and the back-end.

Project Scope and Constraints

The system we created is capable of creating and maintaining multiple documents simultaneously. The number of documents and deltas in each is theoretically unlimited (It is technically limited by how much storage Amazon has, and by the amount of money that can be paid to AWS to maintain the database i.e., limited by Amazon's on-demand model and funding). However, when the number of deltas increases substantially this will slow down the joining of documents, as currently the deltas of each document are obtained in batches of 100 deltas or less per call so users may need to wait for multiple round-trip-times to load their documents successfully.

Topics Covered

- Creating documents
- Renaming a document
- Duplicating a document
- Deleting an existing document
- Multiple users joining the document
- Multiple users simultaneously editing the document

- Multiple users viewing the edits made by other users in real-time
- Handling when a user joins a document, and another user writes while they are joining
- Handling when multiple users create changes at the same time
- Handling the server crashing (Implicitly handled by Lambda's serverless architecture)
- Handling the client crashing by getting all the changes made by other users once the client is back online
- Handling the client crashing by sending the changes they made while offline once they reconnect
- Seeing the cursor of users when then connect and where they are editing
- The ability to revert the document to a certain version (this has a certain limitation that will be demonstrated later)
- Creating a different version of the document (i.e., creating a duplicate document from a certain version of the document) (this also has the previously mentioned limitation)

Constraints

- When a document is made up of a substantial number of deltas this slows down the joining of a document (this is due to the document's deltas being obtained in batches of 100 or fewer deltas i.e., the user may wait for several round-trip times)
- When reverting the document to a certain version or creating a different version of the document, if the document is made up of more than 5 million characters (this roughly translates to 35,000 pages of text which is relatively rare), the revert will not be a perfect replica of the original version.
- Since we use a random colour for the cursors of users it is technically a possibility that two users will have the same cursor colour

5. Task Breakdown

A system similar to agile was implemented where the entire team was continuously iterating over the highest priority tasks. At the beginning of the development effort, the highest priority task was learning how to utilise the AWS services such as Lambda and DynamoDB. The highest priority then transitioned into building a back-end system using Lambda and DynamoDB. We then moved on to developing the front-end of the system that was capable of supporting multiple users simultaneously.

However, during the development of the front-end new requirements for the back-end were made clear and as such, we needed to return to the back-end to develop the features needed.

The Milestones

1. Learning how the Amazon web services that we used work.
2. Implementing a basic back-end of Lambda communicating with DynamoDB.
3. Enabling the client to communicate with the back-end and the back-end being able to respond.
4. Enabling a user to create a document.
5. Enabling the front-end to display the available documents to the user.
6. Enabling a user to view edits made in real-time to the document and make edits themselves.
7. Enabling the server to resolve "race conditions" (i.e., when two different clients send their changes at the same time).
8. Handling when the client's socket closes or on socket error.
9. Sorting list of documents in the front-end by date and name.
10. Automatically list the documents and UI upgrade for listing documents.
11. Handling when deltas are received out of order.
12. Displays the cursors of other users in a document (Each user has a cursor highlighted).
13. Enabling a user to revert and version control a document.
14. UI overhaul.
15. Enabling a user to search for documents by name.
16. Enabling a user to duplicate and delete a document.
17. Enabling a client to recover from a crashed state (i.e., connection lost).
18. Removing a user's cursor when they disconnect.
19. Enabling a client to create an account, change password, and sign in.
20. Linking documents with accounts and setting document permission on account access.
21. Displaying Username on the cursors.
22. Enabling a client to share a document with a person by giving their username.

The Project can be broken down into two major tasks that in turn are broken down into smaller tasks these two major tasks are building the back-end and building the front-end. These two tasks are dependent on each other, therefore, editing the back-end will need an edit in the front-end to make the system recognize the functionality, and as we work on the front-end some functionalities require a change in the back-end.

Building the Back-End

This includes writing code to receive the messages being sent from the front-end, augmenting the database as required and replying to the messages accordingly. This task is further elaborated upon in the "System Architecture and Design" section.

Building the Front-End

Building the front-end can be broken down into constructing the UI that the user will interact with and writing code that calls the Lambda functions from the back-end in order to create new documents, show the available documents, allow a user to join a document, allow a user to edit the document, and show the user what other users are editing in real-time. This task is further elaborated upon in the "System Architecture and Design" section

6. Time Plan

One week was spent on developing the initial backend design. Followed by two weeks of front-end development and discovering new features that needed to be added to the backend and integrated with the front end.

Roles:

- **Mazen Mostafa:** Team Leader, Worked on Connecting Front-End to the Back-End
- **Mohamed Mokhtar:** Worked on Backend regarding Lambda and DynamoDB
- **Youssef Maher:** Worked on Front-End and worked on Connecting Front-End to the Back-End
- **Mohamed Yasser:** Worked on Lambda and Connecting Front-End to Back-End

Task Distribution:

- **Mazen Mostafa:** Worked on quill cursors, browser UI, and backend integration
- **Mohamed Mokhtar:** Worked on DynamoDB integration, backend main loop and backend pseudo-operational-transformation
- **Youssef Maher:** Worked on frontend main loop, out-of-order pseudo-operational transformation and reconstructing the document on opening the document using pseudo-operational transformation
- **Mohamed Yasser:** Added getDeltas and Broadcast functionality to Back-End.

7. System Architecture and Design

Building the Back-End

For this project, we concluded that the best architecture to use would be a serverless architecture. Due to the real-time nature of the system, we are trying to create, using a serverless architecture would provide us with the most scalable, efficient and secure backend system. There are a few cloud service providers that we could have gone for. Namely:

- AWS (AWS Lambda)
- Azure (Azure Functions)
- GCP Firebase (Cloud Functions)

AWS was chosen because it provides the most scalable and resilient architecture of all. The architecture we designed utilises several components from the AWS service list. The most prominent of those components are:

- WebSocket API Gateway
- AWS Lambda
- DynamoDB

The typical cycle is as follows:

1. The client connects to the WebSocket API gateway
2. The client and the AWS Lambda function can send to each other freely
3. The Lambda function reads/writes important data to/from the DynamoDB database

Since the user only has access to the WebSocket API gateway, the Lambda function can enforce any kind of authentication/security policies needed. This prevents the user from deleting/modifying previous document versions, which are immutable as long as the document exists.
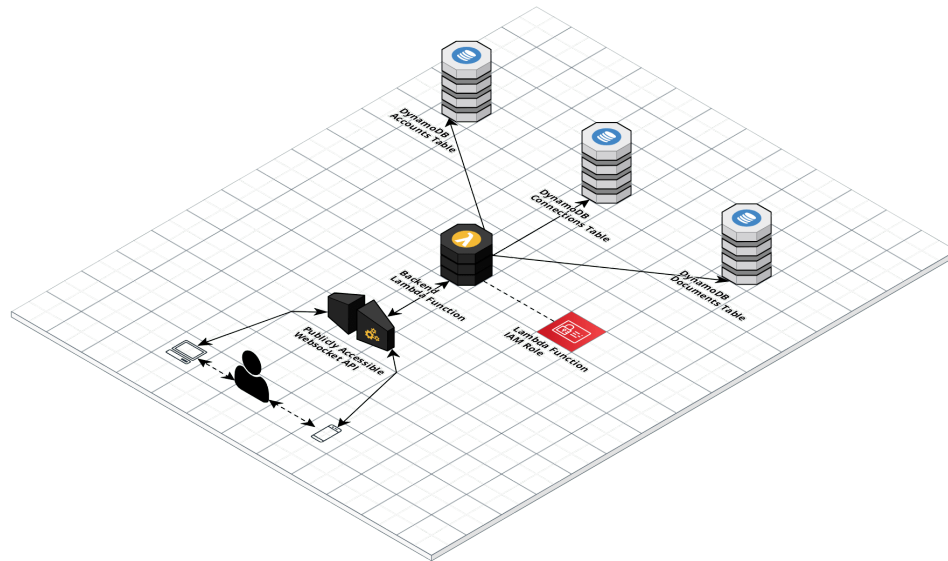
Figure 1 shows the back-end serverless architecture

The Lambda function accepts multiple types of requests. Namely:

- newDocument
    - Description: Create a new document
    - Parameters:
        - documentName: The name of the new document to create
        - userName: The name of the user that is creating the document
    - Constraints: None
- listDocuments
    - Description: Lists the documents accessible by that user
    - Parameters:
        - userName: The name of the user that is listing the documents
    - Constraints: None
- joinDocument
    - Description: Attempts to join an existing document that is accessible by the user
    - Parameters:
        - documentName: The name of the new document to join
        - userName: The name of the user that is joining the document
    - Constraints: The user must have access to that document
- addDelta

- ○ Description: Attempts to add a change to a document. Any changes added are sent to all clients in the form of a "newDelta" message.
- ○ Parameters:
  - ■ documentVersion: The version of the document that this delta is intended for
  - ■ delta: A string containing the delta's information
- ○ Constraints: The user have called joinDocument or newDocument successfully before
- getDeltas
  - ○ Description: Gets a list of deltas that already exist in a document which turn a document from oldVersion to newVersion
  - ○ Parameters:
    - ■ oldVersion: The version before the deltas
    - ■ newVersion: The version after the deltas
  - ○ Constraints: The user have called joinDocument or newDocument successfully before and newVersion - oldVersion <= 100
- sendBroadcast
  - ○ Description: Sends a broadcast message to other clients in the document
  - ○ Parameters:
    - ■ message: The message to send to the other clients
  - ○ Constraints: The user has called joinDocument or newDocument successfully before.
- deleteDocument
  - ○ Description: Deletes a document that the user has access to.
  - ○ Parameters:
    - ■ documentName: The name of the document to delete
    - ■ userName: The user attempting to delete the document
  - ○ Constraints: The user has access to the document
- renameDocument
  - ○ Description: Renames a document that the user has access to
  - ○ Parameters:
    - ■ documentOldName: The old name of the document
    - ■ documentNewName: The new name of the document
    - ■ userName: The user attempting to rename the document
  - ○ Constraints: The user has access to the document

- duplicateDocument
  - Description: Duplicates a document that the user has access to
  - Parameters:
    - documentOldName: The name of the document
    - documentNewName: The name of the duplicated document
    - userName: The user attempting to rename the document
  - Constraints: The user has access to the document
- shareDocument
  - Description: Shares access to the document
  - Parameters:
    - documentName: The name of the document
    - userName: The name of the user that has access
    - newUserName: The name of the user that the document is to be shared with
  - Constraints: The user has access to the document
- createAccount
  - Description: Creates a new user account
  - Parameters:
    - userName: The name of the user to be created
    - userAccountName: The name of the user account to be created
    - userEmail: The email of the account to be created
    - userPassword: The password of the account to be created
  - Constraints: No account already exists with that username
- loginAccount
  - Description: Logs in as a given user
  - Parameters:
    - userName: The name of the user to be logged in as
    - userPassword: The password of the account to be logged in as
  - Constraints: The username and password match an existing account
- changeAccountPassword
  - Description: Changes the password of an existing account
  - Parameters:
    - userName: The name of the account to change the password of
    - userOldPassword: The password of the account to change the password of
    - userNewPassword: The new password of the account
  - Constraints: The username and password match an existing account

Building the Front-End

Constructing the UI

SignIn.html

Figure 2 shows Sign in page for user already registered on the system

On opening the website the sign in page would appear in case if the user wasn't logged in the system because all our pages do direct the user to sign in page if user wasn't logged in where the user could sign in using their account or go to the sign up page which can be used to create an account on our system, in case the user was already logged in our system they would be redirected to the index.html page which can be considered the home page for our Online Docs System, same scenario would apply to documents search bar at the top of the page.

Signup.html

Figure 3 shows Signup page for a new user on the system

Our sign-up page is pretty basic and simple for the user where only a handful of information is needed from them, and the system would check if the information is correct specially the mail should be valid and the passwords should be matching otherwise the system will not sign the user up.

Email Validation

Figure 4 shows email validation error if email was incorrect

Index.html

Figure 5 shows Documents homepage for users

Upon signing in this would be the home page for the user to see the user can join the documents he was added to, create a new document, search for a certain document by name using the search bar, change password or sign out.

changePassword.html
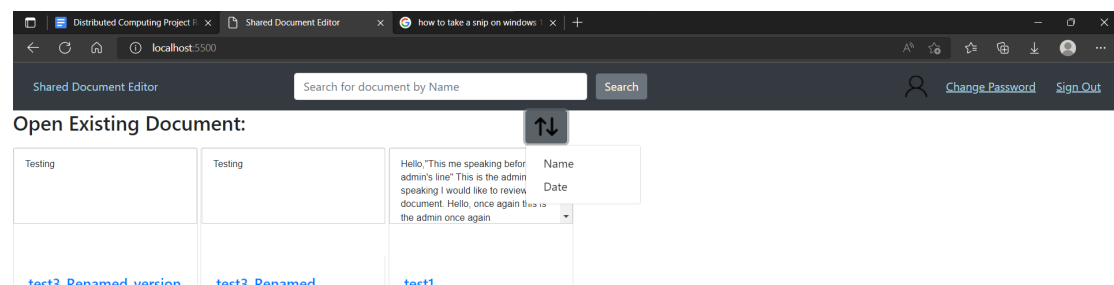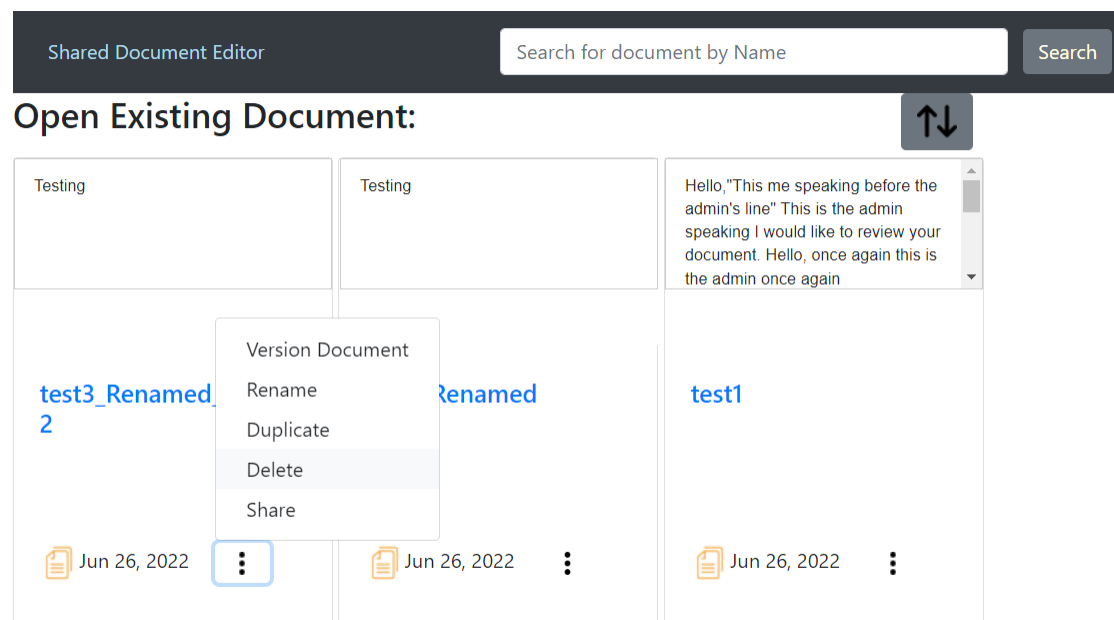

Figure 6 shows change password for user window

Changing password for the user is a very simple procedure with a simple interface too where the old password is required from the user then the user makes his new password and repeats it and if the two passwords are not the same the system would show an error to the user that passwords do not match.

Options on homepage

Figure 7 shows document sorting options (Date, Name)

Figure 9 shows the search results when searching for test3

Users can sort his documents using time or name based on their reference , the document itself does have plenty of options either to rename the document, duplicate it, delete, or share the document with other users on the system using their username and the most important feature would be version control of the document.
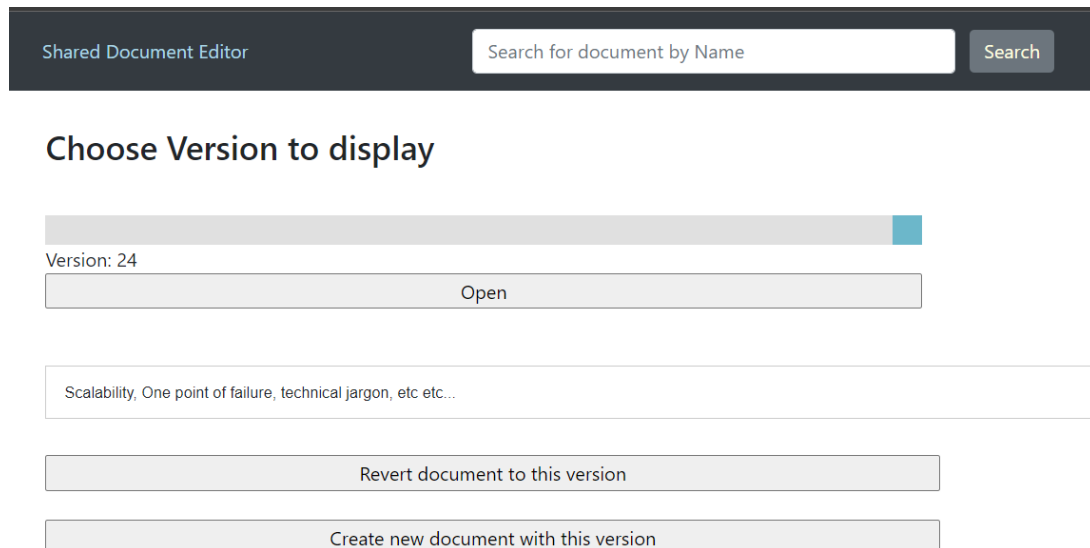
Versioning.html

Figure 10 shows the latest version of the document

13
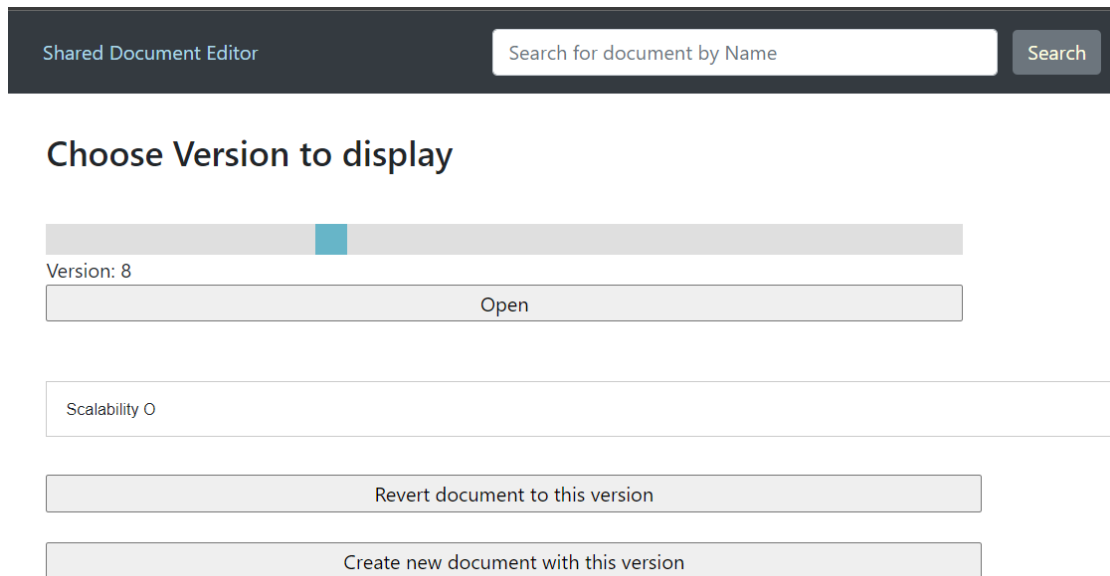
Figure 11 shows version 8 of the document (earlier old version)
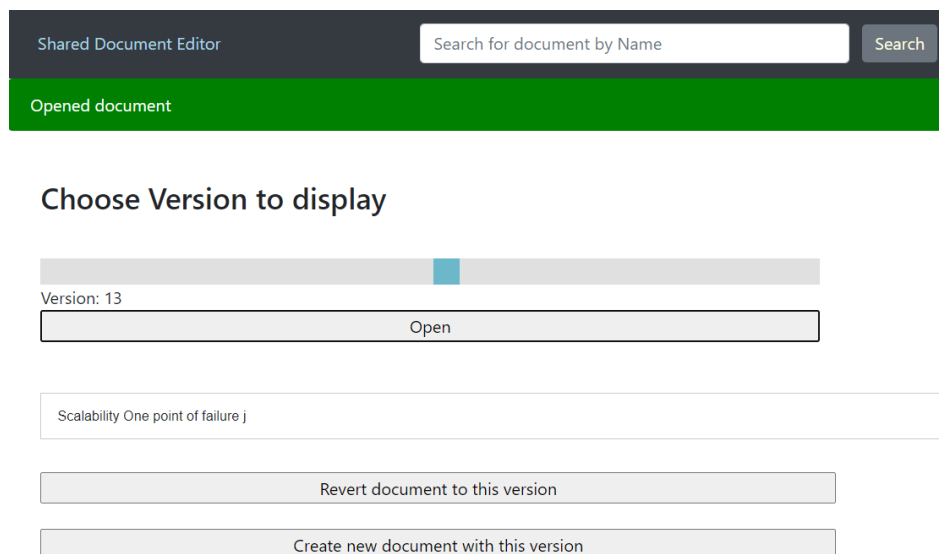
Figure 12 shows version 13 of the document shows the changes between version 8 (old) and version 24(latest)

Version control view would keep track of all changes that happened on a document since it was created, the user would choose the version he wants from the document then clicks open and they will see the content in that document during that time.

Users can revert changes in the current document using revert document to this version button or they can create a new document from the version chosen like branching in GitHub.

Communicating with the Back-End

14

As previously mentioned in a typical use case the client will need to communicate with the back-end to perform the functionality of the system. The communication is done by creating a web socket to the gateway URL of our Lambda back-end. We can then send messages from the opened socket to the Lambda back-end. The messages' formats are tailor-made for this application to fulfil the needed requirements. Below is a demonstration of how the front-end interacts with the back-end.

Listing documents: When a user first visits the webpage a list of all available documents is made available to them this is done by sending the "listDocuments" message to the back-end. This message sends the text "listDocuments" to the back-end. The back-end then replies with a "listDocuments" message with its body including a list of documents available in the database. Upon receiving the reply, a function named "listDocumentsHandler" is called, which extracts the documents from the list of documents sent by the back-end and generates UI elements to show them to the user.

Joining a document: When a user wishes to open a document the front-end sends the "joinDocument" message to the back-end this message also includes the name of the document that the user wishes to join. The back-end then replies using a reply "joinDocument" message which includes the number of the latest version of the document, upon receiving this reply the function "joinDocument" is called which builds the document using the "Deltas" of the document stored in the database. "Deltas" are an object found in the "QuillJS" library, which represents the differences between two versions of a document. The front-end then sends a "getDeltas" message to the back-end which includes the current version the user has, and the version they wish to reach, i.e. the latest version received from the "joinDocument" reply (getDeltas has a limitation where only 100 deltas can be retrieved at once so if more than 100 deltas are needed we handle this case in the function that is called when the "getDeltas" reply is received). The back-end then replies with a "getDeltas" reply message that includes the deltas requested. This reply triggers a function called "composeDocumentOnJoin", which joins the deltas with each other and displays the document once done. (Additionally, the case where more than 100 deltas are needed is handled within this function by calling "getDeltas" once again as many times as needed to get all the required deltas).

Adding changes to the document: the previously mentioned QuillJS library provides us with a traditional text editor, additionally, it provides us with the ability to add a handler to the event that the text within the editor has changed. We utilise this handler (named "textChangeHandler" to communicate to the server any changes that the user has made. Firstly this function checks if the change made was done by the user or by an API call (i.e. in the case another user added changes and the system added it to the first user's editor, in this case, the change is ignored as it was already

handled by the system). If the change was made by the user then we create a delta object of the changes the user made and send the "addDelta" message to the back-end which contains the delta object as well as the version number the front-end wishes to sync this delta to. If the user has made many changes before the server responds to them that their changes have been accepted then we store all the changes made after the first delta in a different delta object and add all the following changes to this delta object.

Showing changes in the document: When the server receives an "addDelta" message it replies with a "newDelta" reply message which includes the delta sent by the user who sent the "addDelta" message, whether the delta came from this client or another, the version number (it also includes details about the sending user's cursor this will be explained in further detail later on). When the "newDelta" reply is received the client calls the newDeltaHandler function which stores the delta in the array "allDeltas" according to the version number.

If the delta's version number is the same version number that the user is synced to we additionally call the function "inOrderDeltaHandler" which simply places the new delta into the document if the client did not send any deltas recently. Alternatively, if the user did send a delta recently we check if the delta originated from the current user or another by checking the "isOwn" value from the reply, if the delta belongs to the current user we send a new "addDelta" message with the user's "blockedDelta" (i.e. any further changes made by the user after the delta currently being processed). If "isOwn" is false then the user has added a delta recently however another user also added a delta and theirs was accepted first, in this case, we need to perform some transformation to show the correct order of the document. First, we see if a "blockedDelta" exists if there is one we add its change to the delta sent to the server, we then revert all changes made by this new combined delta we then add the changes made by the delta accepted by the server then re-add the combined delta containing the current user's changes. We then send another "addDelta" message to send the combined delta to the server with the new version number.

Cursor Details: If the client opens a document (not in versioning nor display of homepage), the client sends his cursor as a Cursor object, which contains his cursor position, the length of the selection, the type of Cursor and the username on the cursor. The client uses "getMyCursor" method to retrieve the cursor object and sends it to the back-end using "sendCursorChanges" method, which is called every 500ms or on a new delta to update the cursor to be displayed frequently approaching real-time. Upon the client receiving other cursors from users through the back-end via "newBroadcastHandler", the client checks if the cursor is from the same document version and not old cursor and if so then the client creates a new cursor for the user (if the user doesn't have cursor) with a random color and displaying the username. If

the position is out of document, the cursor will be toggled off and when the client closes the document, his/her cursor is removed from all other clients.

Versioning: When the client wants to revert a document. The client has to have permission to access that document with his/her account. Then, the client will be able to revert the document either by entering the revert page relative to the document or accessing the relative page through the homepage from the edit menu on the document. The client will be then redirected to the revert page, where initially it will open the latest version by using "openDocumentHandler" and show any upcoming changes live with the cursors and names of who are responsible for the changes. The client can then proceed to open a specific version, which will show the document of that version and in addition, it will show cursor placement of users who are currently active in the document. The client can open by pressing the Open button, which will call "openDocumentHandler" while manipulating to join the deltas till the slide version value.

Any new change will result in new versions that will increase the slider max value but won't change the display nor change the slider of the selected version. The client can then choose from two options, which are to revert the current document by making a new version that has only the display content (can access previous versions later) or the second option, which is to make a brand-new document that has that version content with the first version of it being the version displayed. The client is provided by confirmation on pressing and allowed to enter the document name.

8. Testing Scenarios and Results

In this section, we will focus on the testing of edge cases that may be found during the execution of the system. For the testing of normal cases, this can be seen in the user guide section.

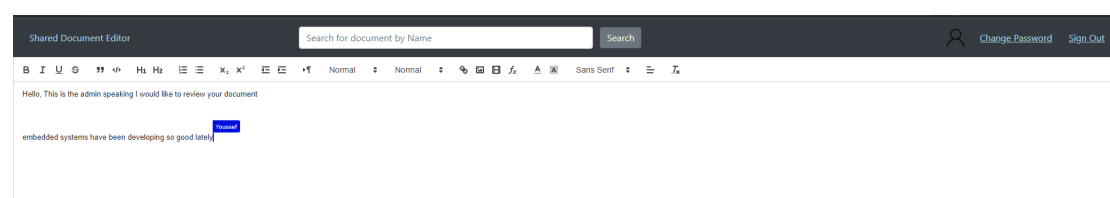Testing if multiple users can type simultaneously in the document

Figure 13 shows multiple users typing together successfully

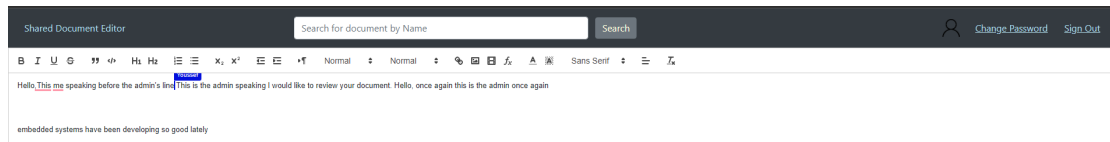## Testing if multiple users can type on the same line

Figure 14 shows multiple users typing successfully on the same line

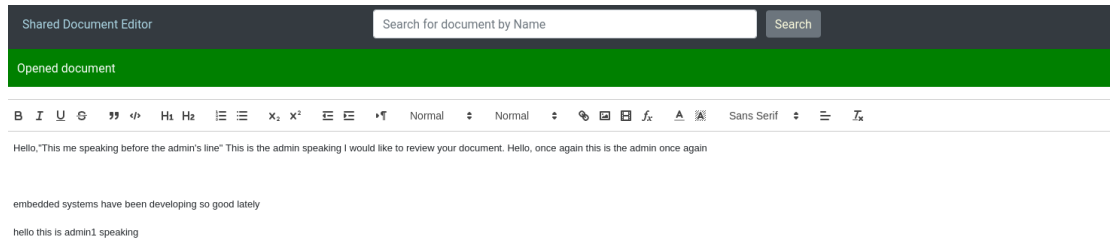## Testing if a client disconnects if the system will be able to recover their state

Figure 15 shows joining a document normally
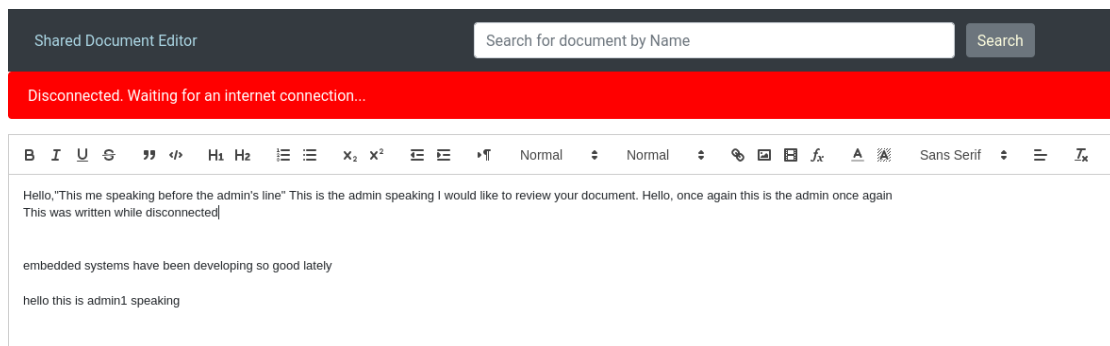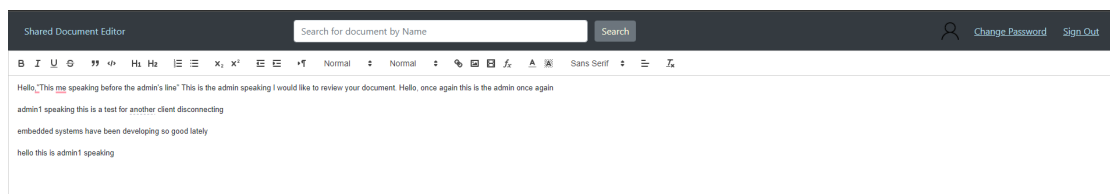

Figure 16 shows a user disconnecting

Figure 17 shows another client typing normally

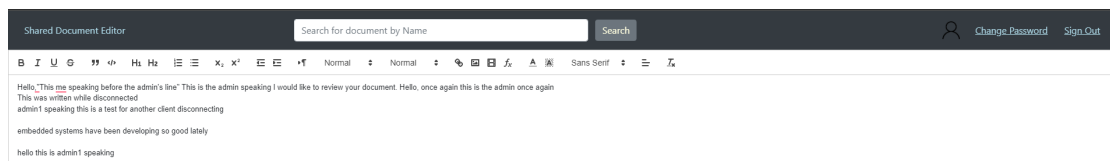*Figure 18 shows that the disconnected client's edits are then sent to the second client upon reconnecting*

9. End-user Guide

The website can be accessed using the following URL "https://mazen-ghaleb.github.io/Distributed-Text-Editor/".

Upon joining the website, the user will be redirected to the sign-in page if they are not signed in.
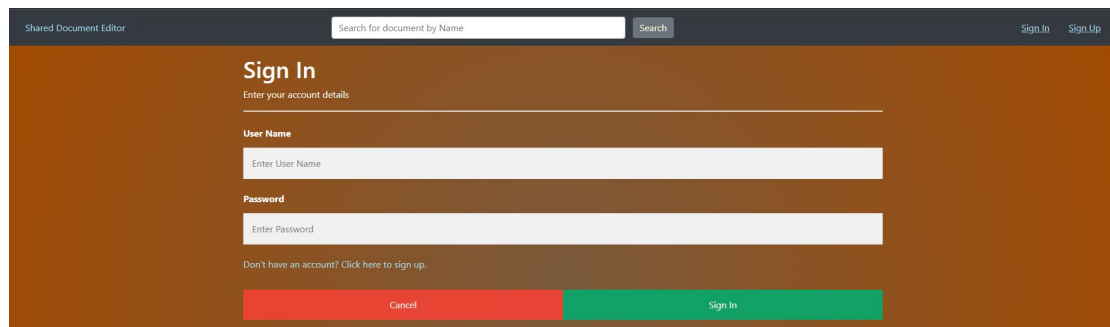
Figure 19 shows the sign in page

On this page, the user can sign in or can alternatively press the sign-up link to create an account. (For the intent of this user guide the admin account will be used). Once the user has signed into their account they are redirected to the main page.
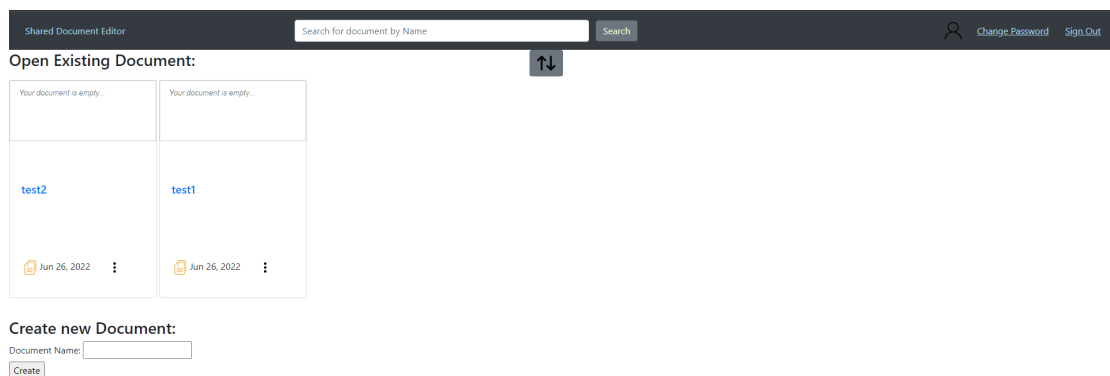
Figure 20 shows the main page

The main page includes the existing documents the user has access to and enables the user to search for a document by name at the top and to create a new document. Additionally, through the main page, a user is able to see if any users are currently editing their accessible documents by seeing their current cursor positions as well as seeing the text of the document. For now, we will create a new document by entering the name in the "Document Name" textbox and pressing the create button.
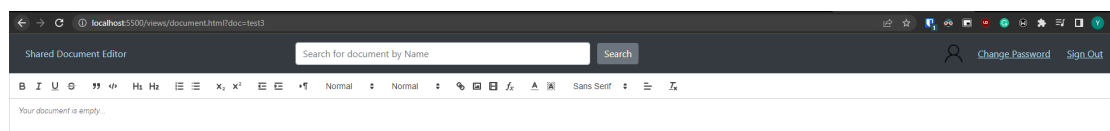
Figure 21 shows when the document page is opened

The user can now edit their document as they see fit.
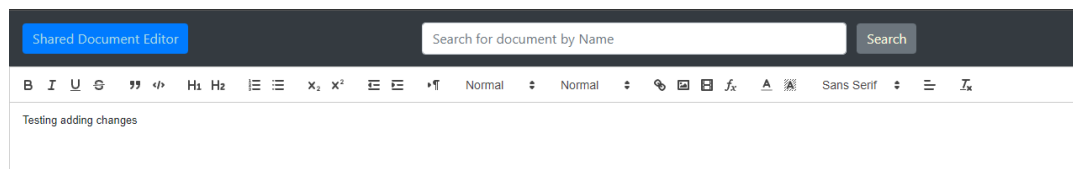
Figure 22 shows the edits made by the user

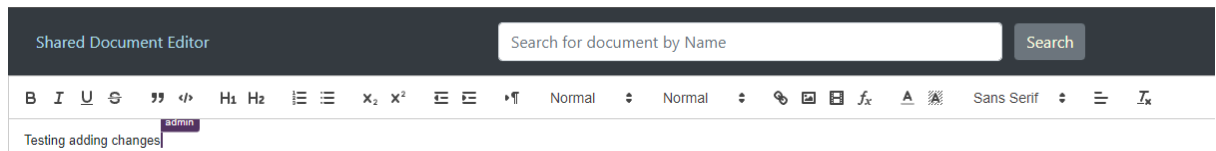The edits can then be seen by other users as follows

Figure 23 shows the edits made by the user as viewed by another user

The system allows for multiple agents to edit the document in real-time and handles displaying the changes in the correct order.

The user is capable of editing the name of the document this is done by returning to the main page and selecting the options list of the wanted document and selecting the "Rename" option
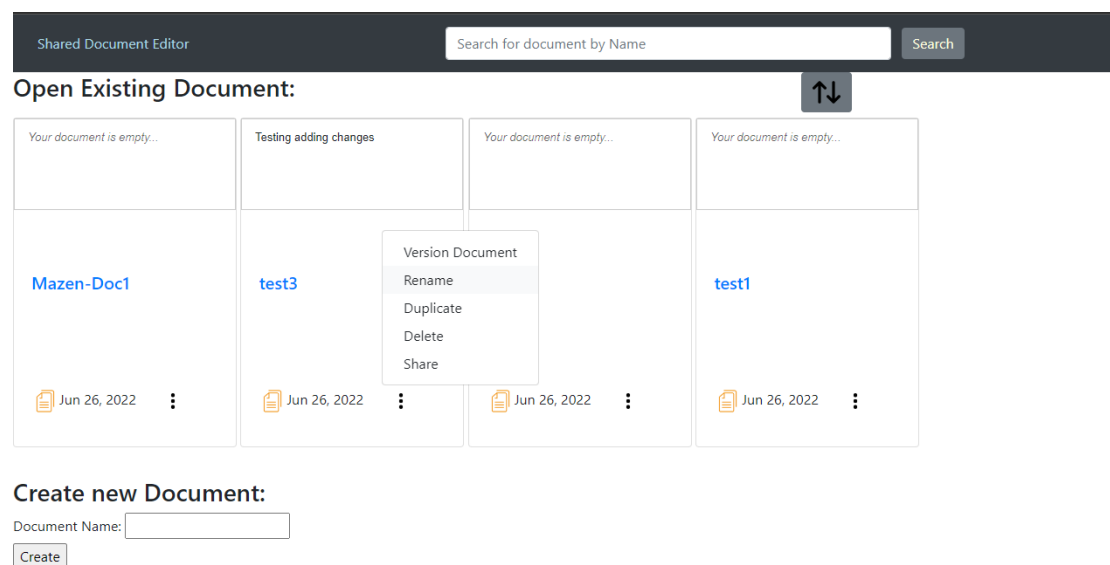
Figure 24 shows the "Rename" option

After entering the new name
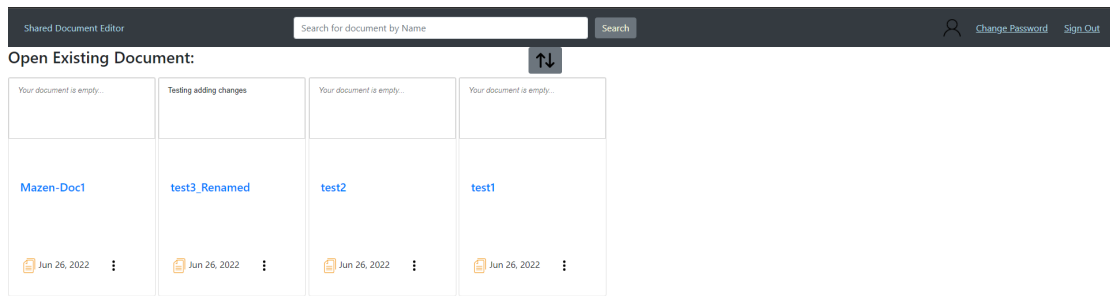
Figure 25 shows the result of renaming the document "test3" to "test3_Renamed"

The user is capable of creating a duplicate version of a document by selecting the "Duplicate" option in the options list.
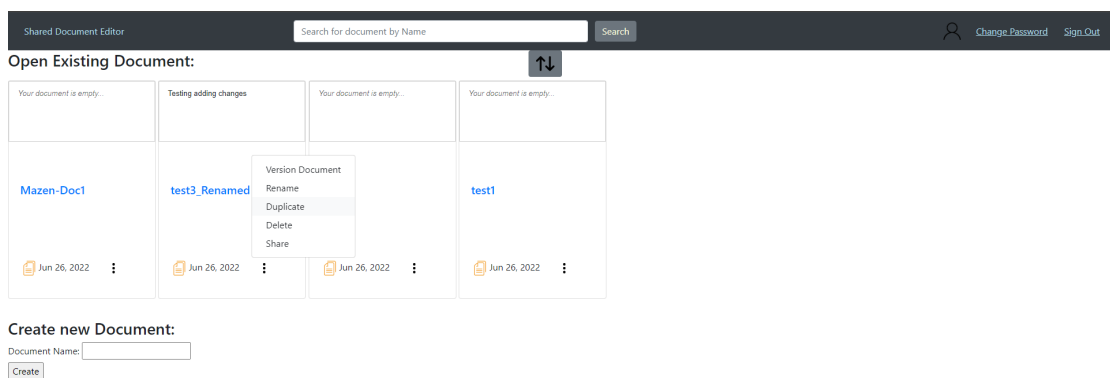
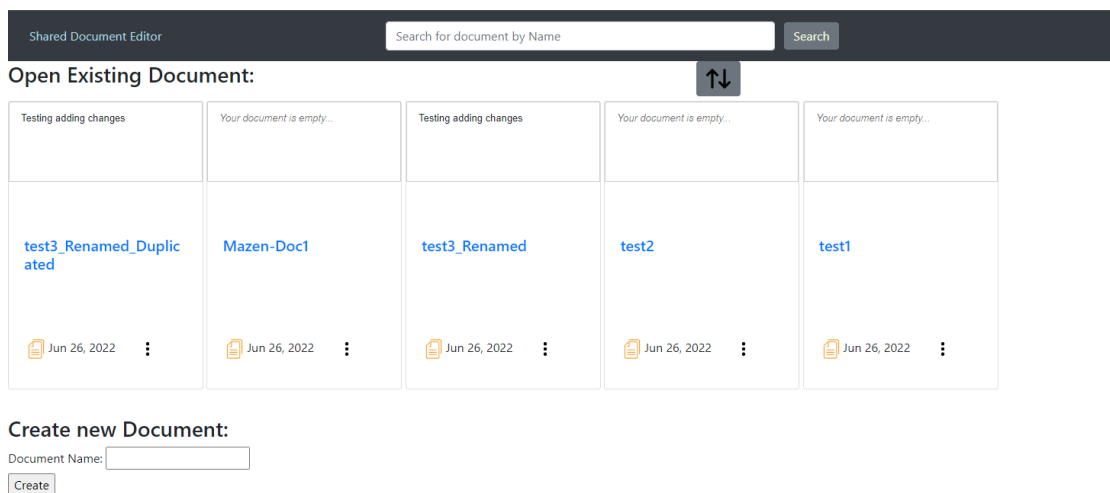*Figure 26 shows the "Duplicate" option*

Figure 27 shows the result of duplicating "test3_Renamed"

The user is capable of deleting a document by pressing the "delete" option in the options menu.
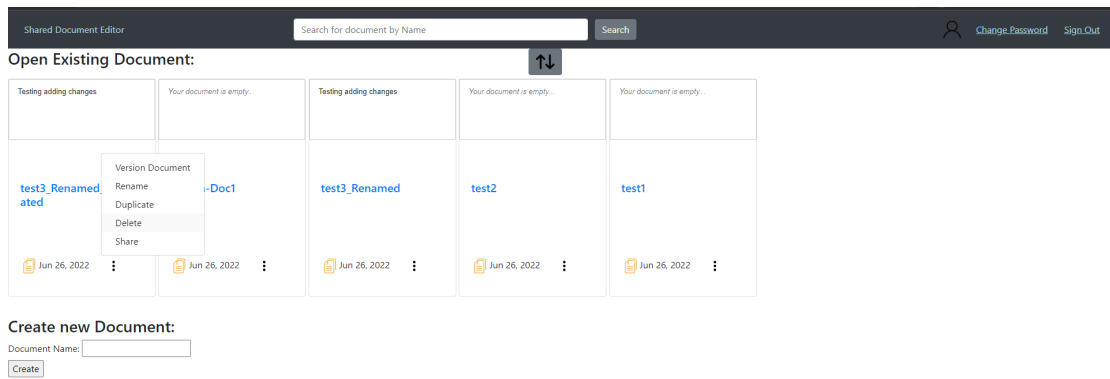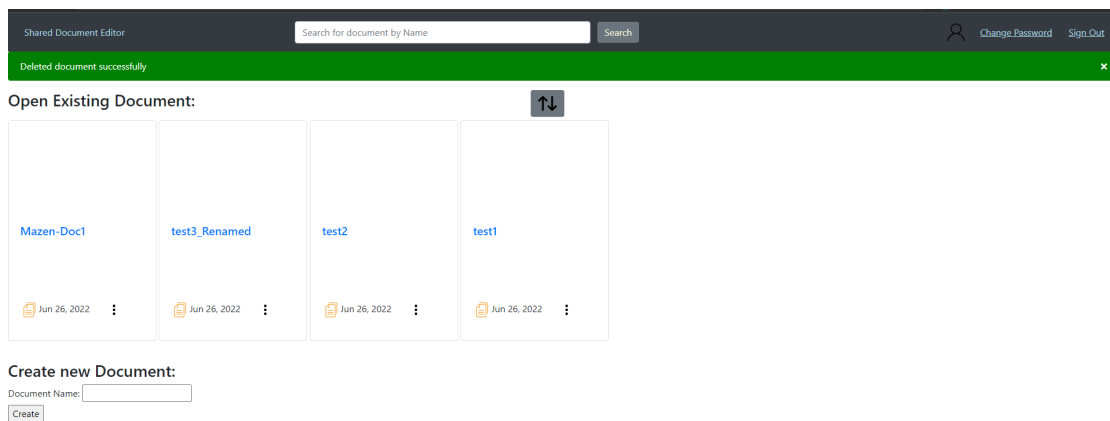
21

Figure 28 shows the "Delete" option

Figure 29 shows the results of deleting "test3_Renamed_Duplicated"

The user can share their document with another user through their username by selecting the "share" option in the options list.
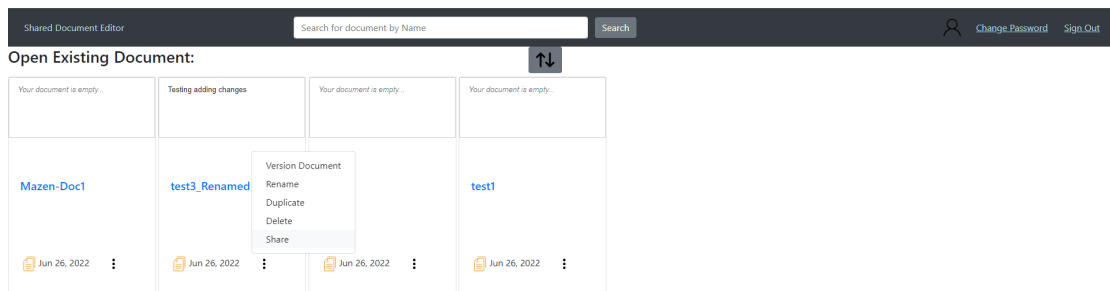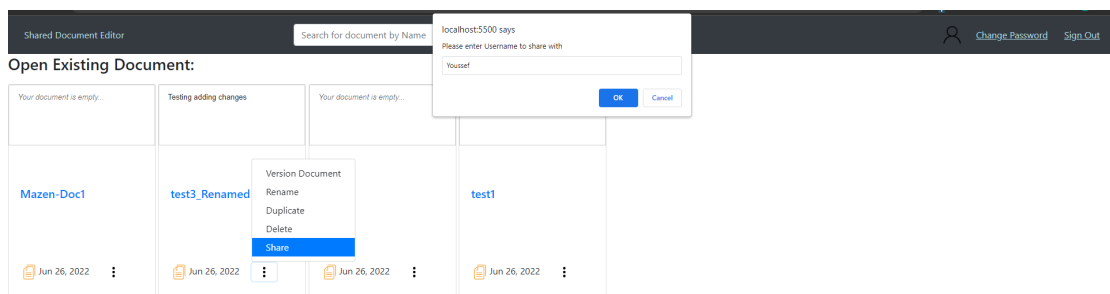

Figure 30 shows the "Share" option

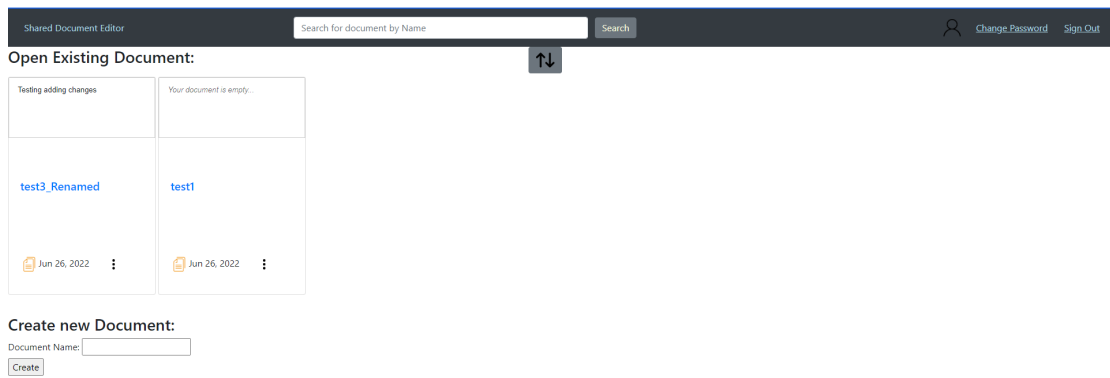*Figure 31 shows choosing a username to share with*

Figure 32 shows the document "test3_Renamed" being shared with another user

The user is capable of reverting the document to a certain version or creating a duplicate document at a certain version of the document by pressing the "Version Document" option in the options list.
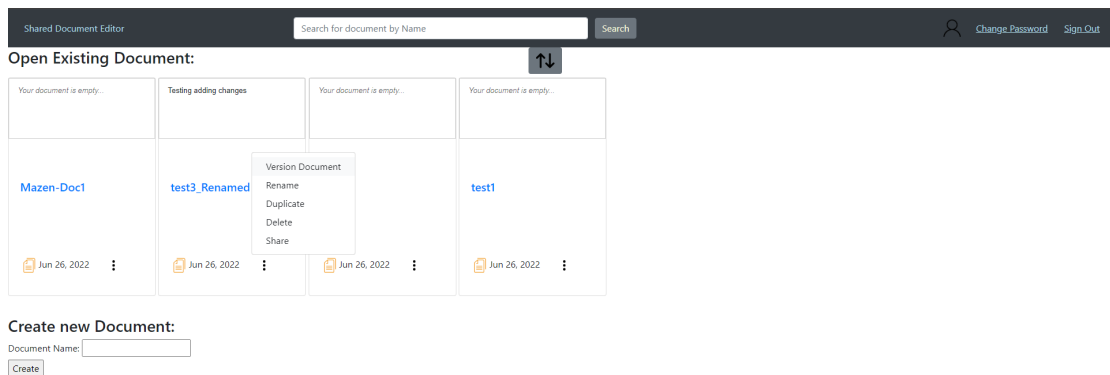

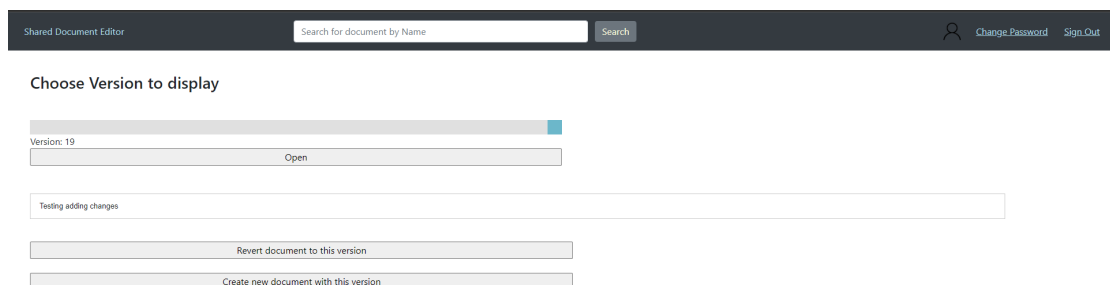*Figure 33 shows the "Version Document" option*
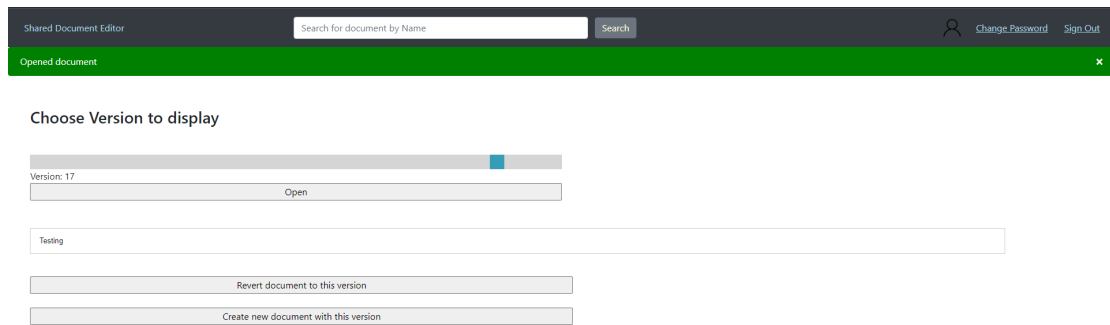

Figure 34 shows the versioning page

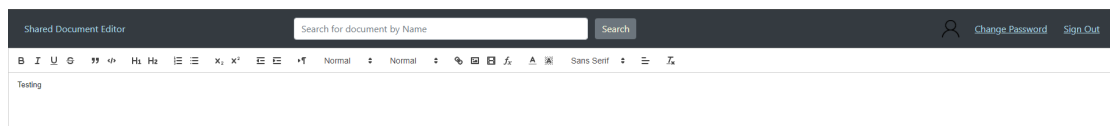*Figure 35 shows choosing a different version to revert to*
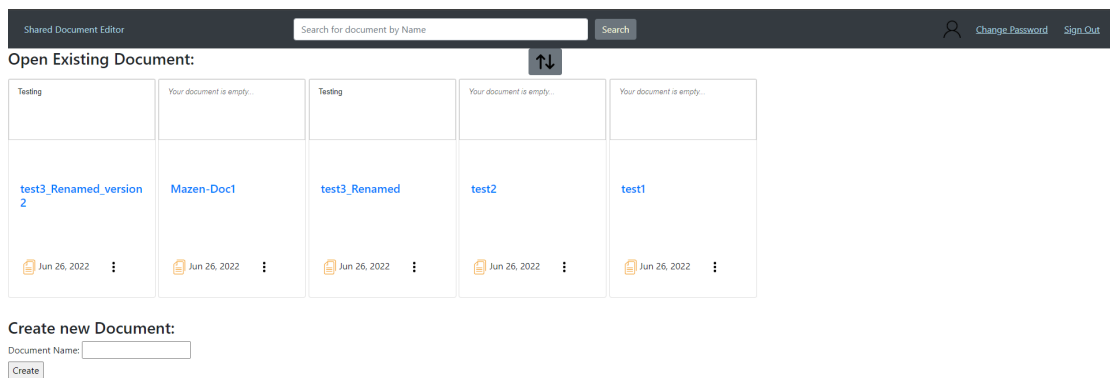

Figure 36 shows the reverted document

Figure 37 shows the document created from versioning

The user is capable of search for a certain document by name by utilising the search bar found at the top.
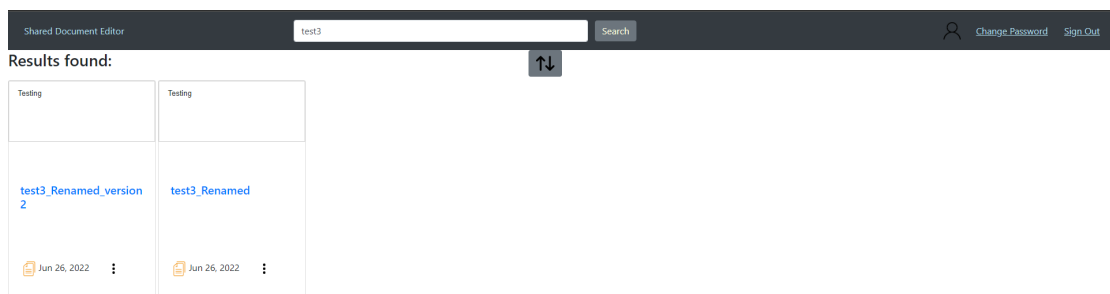

Figure 38 shows the search results for the term "test3"

10.Video

The video can be found at the following https://youtu.be/2E2165A7NYM

11. Code

The code can be found at the following Github Repo ("https://github.com/Mazen-Ghaleb/Distributed-Text-Editor") or the following drive link ("https://drive.google.com/file/d/1kzScCMI0Uq27JxuBf10rRuZMXDGOleJP/view?usp=sharing")

12. Conclusion

Implementing this project enabled us to learn further about the fundamentals of distributed computing, as well as the tools used in the industry to implement distributed systems. This allowed us to create a distributed text editor with a very high level of robustness and reliability.

13. References

[1] Quill JS. (n.d.). Version (2.0.0-dev.4). Your powerful rich text editor. Retrieved June 2022, from https://quilljs.com/

[2] Quill JS Cursors. (n.d.). Version (3.1.2). Retrieved June 2022, https://github.com/reedsy/quill-cursors

14. Appendices

[Appendix 1] In this document we refer to "delta" many times. This is an object found in the Quill JS library that is a representation of the differences made in a document between two versions of a document.