ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

NoteSpace

Multi-platform Web Application for Real-Time Document Collaboration & Sharing

Guilherme Ferreira	49472
Ricardo Costa	49511

Supervisor: Paulo Pereira, ISEL

Final version of the report written for Project and Seminar BSc in Computer Science and Computer Engineering

July 2024

Instituto Superior de Engenharia de Lisboa

NoteSpace

Multi-platform Web Application
for Real-Time Document Collaboration & Sharing

49472 Guilherme Filipe Cardoso Ferreira

49511 Ricardo Manuel Sousa Costa

Supervisor: Paulo Pereira

Final version of the report written for Project and Seminar BSc in Computer Science and Computer Engineering

July 2024

Abstract

Both modern real-time collaboration apps like Google Docs and version control platforms like GitHub have become the standards when it comes to collaborative work. Nevertheless, both approaches lack major features the other addresses; while both of them are sufficient for either basic real-time collaboration or fully asynchronous collaboration, there is still a major gap between them. What if there was an hybrid approach? This kind of application would allow out of the box real-time collaboration, while still taking advantage of features from version control platforms. This project aims to address this question.

NoteSpace is a web application that provides markdown-supported document-based repositories and borrows features such as real-time collaboration from platforms like Google Docs, as well as version control and a sharing platform from GitHub. Through this hybrid solution, we implemented a bare-bones version to what can be seen as the 'in-between' approach to collaborative work.

Keywords: real-time collaboration, conflict resolution algorithms, version control, web application, markdown editor, sharing platform.

Resumo

Tanto aplicações modernas de colaboração em tempo real, como o Google Docs, quanto plataformas de controlo de versões, como o GitHub, tornaram-se os padrões no que diz respeito ao trabalho colaborativo. No entanto, ambas as abordagens têm funcionalidades em falta que a outra aborda; enquanto ambas são suficientes para colaboração em tempo real ou colaboração completamente assíncrona, ainda existe uma grande lacuna entre ambas. E se existisse uma abordagem híbrida? Este tipo de aplicação permitiria colaboração em tempo real, tirando partido de funcionalidades de plataformas de controlo de versões. É através deste projeto que tentamos responder a essa pergunta.

O NoteSpace é uma aplicação web que oferece repositórios de documentos com suporte markdown, combinando funcionalidades de colaboração em tempo real de plataformas como o Google Docs com controlo de versões e uma plataforma de partilha inspirada no GitHub. Essa solução híbrida permite criar uma versão base de uma abordagem ”intermediária” para trabalho colaborativo.

Palavras-chave: colaboração em tempo real, algoritmos de resolução de conflitos, controlo de versões, aplicação web, editor markdown, plataforma de partilha.

Contents

List of Figures

List of Acronyms

API Application Programming Interface

CI/CD Continuous Integration and Continuous Delivery/Deployment

CLI Command Line Interface

CRDT Conflict-Free Replicated Data Type

CRUD Create Read Update and Delete

CSRF Cross-Site Request Forgery

CVS Concurrent Versions System

DFS Depth First Search

DOM Document Object Model

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JSON JavaScript Object Notation

NPM Node Package Manager

OT Operational Transformation

PWA Progressive Web Application

REST REpresentational State Transfer

RBAC Role-Based Access Control

SCSS Sassy Cascading Style Sheets

SDK Software Development Kit

SPA Single Page Application

UI User Interface

VCS Version Control System

Chapter 1

Introduction

For a long time, project collaboration often relied on the exchange of local copies for each iteration. Soon after, tools like CVS Concurrent Versions System (CVS) emerged and allowed for features such as version-tracking, branching, and merging. One of CVS's major disadvantages, however, was the existence of pessimistic locking on resources, which meant that only a single user could edit a resource at a time. Then, with the emergence of Git, a distributed version control system, in 2005, a significant shift was made by enabling asynchronous collaborative work. With Git, collaborators could keep a local copy of the repository, a significant difference from previous centralized repositories. Now, changes could be made locally and pushed to a remote repository, allowing for greater flexibility.

In 2006, Google Docs emerged with its groundbreaking approach to collaboration, addressing the challenges posed by traditional project collaboration methods and even Git. These challenges were mainly focused with collaborators not being able to work on the same version at the same time. With its real-time collaboration features, multiple users could now edit the same document simultaneously from anywhere, across multiple devices. This meant no local copy was even needed and users could still enjoy the freedom and flexibility given by previous tools.

However, many problems still exist within both approaches: version control system (VCS) solutions like Git and GitHub lack the aforementioned real-time collaboration features, as users need to rely on other solutions to enable real-time collaboration. This requires that at least one user keeps a copy of the repository's content, which can be undesirable. Similarly, real-time collaboration applications like Google Docs are missing some major features such as an explicit VCS, as the granularity of the content of each saved version is ambiguous and dependent on the system's implementation, and a public platform for sharing documents and repositories, which is only done through link sharing.

Most tools already address most problems that emerge with collaboration, such as version merging, version control and real-time collaboration; what seems to be lacking is an approach that bridges both solutions, providing the same robust features as well as the freedom and control needed for efficient organization, sharing and collaboration on repositories.

To address this, we decided to develop NoteSpace. NoteSpace is a web application with markdown-document based workspaces which allow users to collaborate in real-time similarly to Google Docs, and also take advantage of workspace management features, a basic version control system and a public workspace sharing platforms in a manner analog to GitHub. The project serves as a solution to the previously described problem, by attempting to take advantage of features from both synchronous and asynchronous collaborative platforms.

1.1 Outline

This document is composed of seven chapters and aims to provide a comprehensive understanding of the project.

- Chapter 2 presents the underlying background of collaborative editing problems and solutions; it will also focus on explaining progressive web apps.
- Chapter 3 describes the problem and the solution's requirements.
- Chapter 4 presents the project's architecture, the codebase organization and technologies that comprise it.
- Chapter 5 delves into the implementation of the system, exploring how the different features were implemented.
- Chapter 6 explains how the project was tested and deployed.
- Chapter 7 wraps the report addressing what was achieved and what still could be done to improve the project, with a couple of final remarks about the development of the project.

Chapter 2

Background

2.1 Collaborative Editing

Collaborative editing enables multiple users to edit shared resources simultaneously. This process has gained a significant importance with the increase in remote work and teamwork. Collaborative editing systems can be divided into two main approaches: asynchronous and synchronous, each with its distinct methodologies and conflict resolution strategies.

2.1.1 Approaches

Asynchronous Collaborative Editing

Asynchronous collaborative editing allows multiple users to make changes to a local copy of a document, independently. This approach requires manual commits, where each user's changes need to be merged into the main version of the document at a later time. This technique is used in version control systems, such as Git, where changes are committed and then merged, potentially leading to conflicts that need to be later manually resolved [2].

Synchronous Collaborative Editing

Synchronous collaborative editing, on the other hand, involves automatic synchronization of changes in real-time, where users can see each other's changes instantaneously. This approach eliminates the need for manual commits but introduces the challenge to manage concurrent changes automatically.

A common method for handling concurrency in general is pessimistic locking. With this approach, the editing user would need to lock the document or a portion of it while making changes; this method is, however, not practical for synchronous collaborative editing since it prevents multiple users to edit the same document simultaneously. Hence, a strategy with an optimistic locking approach is required, which would need to allow multiple users to edit concurrently while still resolving potential conflicts.

2.1.2 Conflict Resolution Algorithms

The two main properties that should be guaranteed in a conflict resolution algorithm are commutativity and idempotency. Commutativity ensures that the order in which operations are received and applied do not matter. Idempotency guarantees that the number of times an operation is applied also does not affect the outcome.

In the example described in Figure 2.1, a user performs an insertion and the other user performs a deletion in the same position. After both users apply both operations, they reach an inconsistent state, with each one observing a different state of the document, meaning that the operations do not commute.

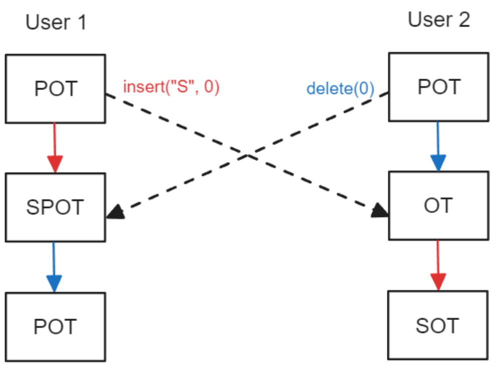

Figure 2.1: Lack of commutativity of operations, image based on the Conclave case study [1]

In the another example shown in Figure 2.2, both users attempt to delete the first character of the document simultaneously, each with the intention to delete the same character. In this case, both states converged, but the same operation was applied more than once, meaning that the operations are not idempotent.

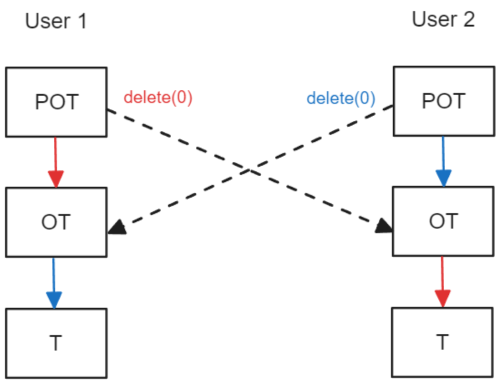

Figure 2.2: Lack of idempotency of operations, image based on the Conclave case study [1]

In order to ensure state consistency across all users editing a resource, we need an conflict resolution algorithm. This algorithm effectively guarantees that the same state should always be eventually reached across all clients, regardless of the type of operations performed by each. There are two main types of conflict resolution algorithms: Operational Transformation (OT) and Conflict-Free Replicated Data Types (CRDTs).

Operational Transformation (OT)

The core mechanism of the OT algorithm relies on a transformation function that modifies operations based on the context of the preceding operations before being applied. For example, if in a document, a user inserts a character and another user deletes a character at the same position, these operations might conflict. This function modifies the operations ensuring both users end up with the same desired document state, as seen in Figure 2.3.

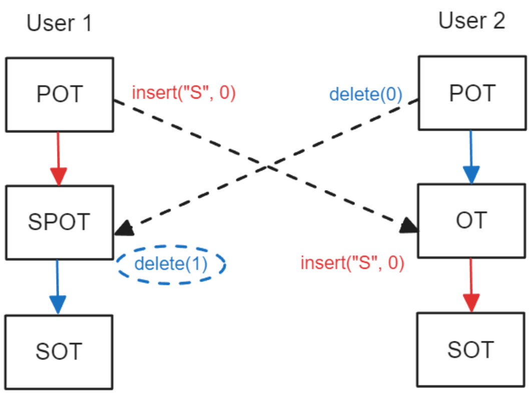

Figure 2.3: Example of the OT algorithm, image based on the Conclave case study [1]

All user-originated operations in a given time frame are then processed by the server and transformed independently per user, in order to reach a state consistency, even though the operations received by each client can diverge; however, this approach comes with some drawbacks, mainly regarding scalability.

However, in OT algorithms, all users editing the same document must communicate through the same server instance to ensure synchronization and consistency [3]. This stateful centralized server acts as an orchestrator by managing operation order, resolving conflicts, and maintaining a single source of truth. Consequently, the server needs to be aware of all previous operations. By limiting clients to communicating through the same server instance, a bottleneck can quickly arise as the number of users in the same document grows. Additionally, the OT algorithm is extremely complex to implement effectively [1]. For those reasons, we decided that OT was not the most viable approach for handling conflicts.

Conflict-Free Replicated Data Type (CRDT)

CRDTs provide an alternative approach to conflict resolution. They are designed such that their operations are always commutative and idempotent, as mentioned previously. Rather than implementing a complex algorithm to handle conflicts like OT, it uses a more complex data structure.

The CRDT algorithm works by assigning a globally unique identifier to each character in the document, which is a combination of a unique site identifier for each client session and a unique character identifier. Also, the CRDT algorithm requires a way to globally order characters, instead of absolute indices, which depends on the type of CRDT algorithm.

When a user inserts a character, the algorithm generates a new identifier between the neighboring characters. When a user deletes a character, the algorithm simply marks the character as deleted without actually removing it from the data structure, creating what it's called a *tombstone character.* This way, the algorithm can maintain the order of the characters correctly and handle concurrent operations without conflicts.

In the example represented in Figure 2.4, two users make simultaneous edits to a shared document. One user inserts a new character, while the other deletes another character. Both users then exchange these changes. As a result, despite making different edits concurrently, both users' documents synchronize to a consistent state, showing how CRDT handles conflicts and ensures uniformity.

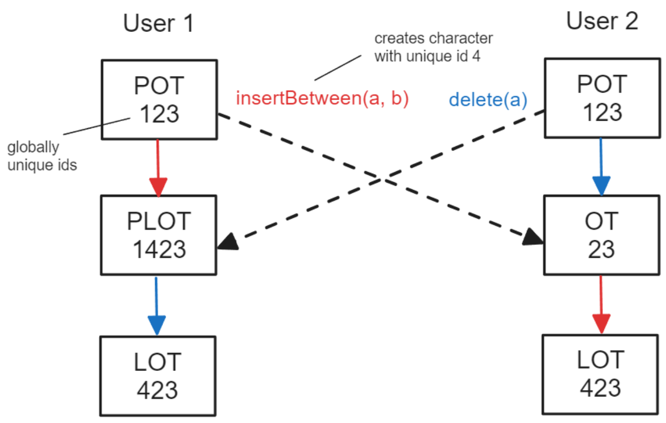

Figure 2.4: Example of the CRDT algorithm, image based on the Conclave case study [1]

This algorithm provides a variety of advantages over OT: it is easier to implement, it allows users to communicate through different stateless server instances even when in the same document and changes sent to the server by offline clients can be arbitrarily delayed. Additionally, this algorithm is even viable in a peer-to-peer environment, without the need for a centralized server. This makes CRDT a more viable and scalable approach, since it allows a decentralized and more efficient architecture.

Fugue CRDT

There are a lot of CRDT variants, but most of them share the same problem - an anomaly that occurs in both OT and most CRDT algorithms. It is called ”interleaving”, which can happen when concurrent changes made by different users overlap or are mixed together, with the merged outcome resulting in an unreadable or undesirable state. This situation can arise when two users type in the same position simultaneously or in an online/offline scenario. An example showcasing this issue is shown in Figure 2.5.

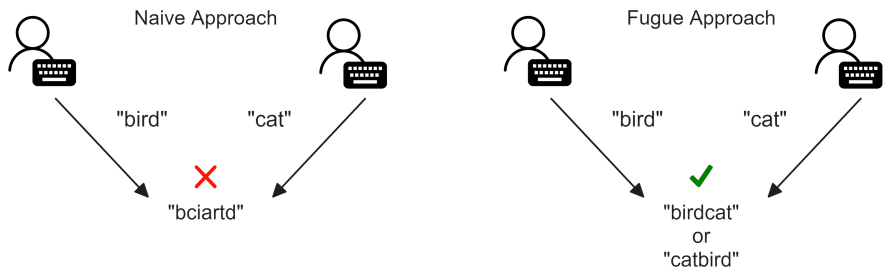

Figure 2.5: Example of the interleaving anomaly

To address this, we explored a recent CRDT algorithm called ”Fugue”, that introduces a unique mechanism to handle such anomalies, ensuring the correct merging of concurrent edits, thereby maintaining data integrity even in such cases. *”It is named after a form of classical music in which several melodic lines are interwoven in a pleasing way”* [4]. Fugue is particularly notable for its simplicity and effectiveness in maintaining a *”uniquely dense total order”* [5] among elements.

In the Fugue algorithm, each character is considered a node in a tree. Each node in the tree, except for the root, has a parent and a side (indicating if it's a left or right child). Multiple nodes can have the same parent and side, making it an n-ary tree. Each node is labelled by a causal dot (combination between the site ID and the character ID), and the order of characters is obtained through an in-order traversal: first traverse the node's left children, then visit the node itself and then traverse its right children. If there are ties between children on the same side, they are resolved by sorting alphabetically the causal dots. Put simply, rather than specifying a position to insert a character, it specifies inserting it to the right or left of another character, thus preventing the interleaving anomaly.

There are two ways of implementing this algorithm: a string-based approach or an actual tree structure approach, each with its upsides and downsides. The string implementation uses ”string positions” [6] as a proxy for a tree structure, that implicitly establishes the tree-like algorithm with the identifiers of characters. The traversal of the tree is done simply by alphabetically sorting the characters. In the tree implementation, the tree structure is explicitly maintained, with nodes representing the positions of elements in the list. Each node has a unique causal dot and references to its left and right children.

Comparison of Algorithms

The string-based implementation has several advantages:

- **Simplicity**: Positions can be compared using standard string comparison, making it easy to implement.
- **Efficiency**: Sorting and insertion operations can be performed efficiently by leveraging string comparison.

However, it also has some downsides:

- **Positional Growth**: As more insertions occur, position identifiers may grow in length very quickly, potentially impacting performance.
- **Imprecision**: The reliance on lexicographic ordering might lead to inefficiencies in densely populated segments of the list.

The tree-based implementation offers a more explicit and flexible structure:

- **Direct Manipulation**: Nodes can be directly manipulated, allowing for more fine-grained control over the structure.
- **Scalability**: The explicit tree structure can handle a high volume of operations without significant performance degradation.
- **Clear Semantics**: The tree structure provides a clear and intuitive representation of the order of elements.

The downsides include:

- **Complexity**: Managing an explicit tree structure requires more complex algorithms and data management.
- **Overhead**: The additional metadata and references required for each node can introduce overhead.
- **Synchronization**: Ensuring that all replicas maintain a consistent view of the tree can be challenging in a distributed environment.

In summary, the choice between string-based and tree-based implementations depends on the specific requirements and constraints of the application. The string-based approach offers simplicity and efficiency for general use cases, while the tree-based approach provides greater flexibility and scalability for more demanding scenarios. Initially, we adopted the string-based approach, and later transitioned to the tree-based approach. This decision is further explained in the chapter 5.

More information about the actual implementations of this algorithm can be found in Matthew Weidner's original web article on Fugue [7].

2.2 Progressive Web Apps (PWAs)

For a long time, when an app was said to be multi-platform, it usually meant that developers had to implement a different codebase for each platform. This approach was necessary because different platforms support different technologies. Additionally, for every update, the app would need to go through a validation process specific to each platform's app store before reaching the final user. Once approved, the update would need to be re-installed on the user's device. This process is both time-consuming and resource-intensive.

In contrast, the web paradigm offers a more seamless development and deployment process. Developers could create a website once, and it would run on different devices and browsers without the need for multiple codebases. Each update could be instantly deployed to users without any validation processes, providing immediate benefits. However, traditional web applications have some disadvantages compared to native apps, such as limited offline support and the inability to send push notifications on mobile devices.

The desire to combine the seamless development and deployment process of the web with the full functionality of native applications led to the creation of Progressive Web Apps (PWAs). PWAs are developed using the JavaScript/TypeScript ecosystem, similarly to traditional websites, but with key enhancements that allow them to mimic native apps. These enhancements include:

1. Existence of a web app manifest, a JSON file that provides metadata about the application. This allows the website to be installed as an app on the user's device, and provides information about details like the app logo, its theme and different icons for different screen sizes.

2. A service worker, a script that runs in the background separate from any currently displayed web page. It enables crucial features like offline support, background sync, and push notifications. The service worker can also intercept network requests and cache resources to ensure the app works offline or in low-connectivity environments.

PWAs must also be performant and responsive, and they can only be deployed from HTTPS sites to ensure security. When all these requirements are met, the user can install the app on its device. To note that in order to support the original JavaScript codebase, every PWA is run on a WebView, a browser engine without the usual user interface. This allows the use of the engine to render and handle the HTML, CSS and original JavaScript files, whilst providing a similar user experience to a native app.

Chapter 3

Problem Description

3.1 Problem

Google Docs excels in real-time collaboration, allowing multiple users to edit documents simultaneously with changes appearing instantly for all users. However, it lacks sophisticated explicit version control mechanisms, such as branching, merging, and detailed commit histories, which are essential for tracking changes over time and managing contributions from multiple team members. On the other hand, GitHub provides these advanced version control features, enabling teams to manage and review changes efficiently, handle conflicts through merges, maintain a clear history of modifications, among other useful features. However, GitHub has no native support for real-time collaborative editing, requiring users to manually synchronize their changes. By merging the strengths of these two systems, our solution aims to offer the best of both worlds: seamless real-time editing and workspace management, with an expanded set of tools such as a comprehensive version control system and a public platform for sharing and discover public workspaces.

3.2 Solution

We propose NoteSpace, a web application that facilitates real-time document collaboration and sharing. It supports many functionalities, mainly a **markdown-supported editor** that supports the creation and editing of documents, **workspaces** that help users organize multiple documents into a single workspace, allowing organization into folders as well; **real-time collaboration** which enables multiple users to work simultaneously on the same workspace and/or document, whilst still ensuring consistency; a **public sharing platform** that allows the discovery and sharing of public workspaces and content, and finally a basic **version control system** that allows users to commit, rollback, clone and review document versions. In addition to the main functionalities, we also aim to develop a system that is: performant, as real-time editing should be responsive and with acceptable latency; secure, as authentication and authorization systems must prevent unauthorized access to resources; reliable, as there should be no data loss on concurrent editing.

Chapter 4

Architecture

The system is comprised of two main modules: the backend application and the frontend application, as can be seen in Figure 4.1. The frontend is composed of a web application that provides simple interfaces and user interaction with the system, while the backend enforces the business logic and processes requests from the frontend; it's responsible for validating resource access attempts, resource modification and data storage, as well as communicating with other clients any events received through existing WebSocket channels, as to ensure consistency across all users. Communication between the two modules is achieved through the HTTP protocol [8] for request-response communication and the WebSockets protocol [9] for bidirectional streaming communication.

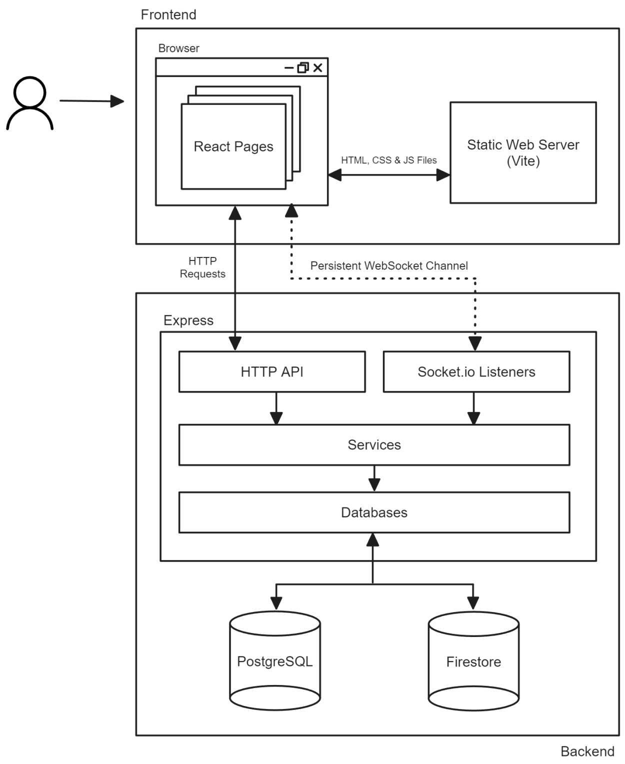

Figure 4.1: System Architecture

4.1 Frontend

The frontend application acts as the intermediary between the client and the system. It provides a simple interface and allows users to manage workspaces, folders, resources, and collaborate in real-time with other users. Its composed by a single page application (SPA) with multiple layouts. Each layout is comprised of a multitude of components and the application's service layer is responsible to communicate with the backend through either HTTP [8] or WebSockets [9]. The frontend application's codebase is organized as follows:

- **/assets**: Static files like images and fonts used throughout the application.
- **/contexts**: React context providers for managing global state and data sharing.
- **/domain**: Domain-specific logic and models, namely editor and workspace logic.
- **/pwa**: Configuration and files for the PWA functionality.
- **/services**: Modules for API calls and backend communication.
- **/ui**: Pages and reusable UI components.
- **/utils**: Utility functions and helpers for common tasks.

The frontend application supports deep linking, enabling users to directly access specific content through URLs. The non-static UI interface dynamically generates and updates components on the browser-side, ensuring a responsive user experience.

4.1.1 Navigation Graph

The navigation graph of our frontend application shown in Figure 4.2 visualizes the main interconnected paths that can be taken, serving as a structured representation of the user journey, depicting how different screens are linked together.

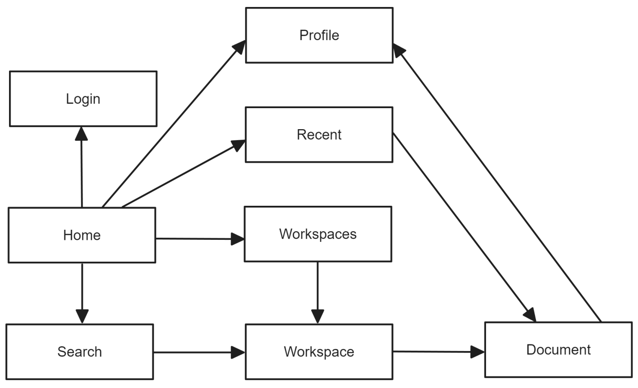

Figure 4.2: Navigation graph of the frontend application

4.2 Backend

The backend is responsible for handling all incoming requests from the clients, processing them to ensure request validity. It enforces business logic and ensures users can't access or modify resources without the proper permissions. It also handles data handling and storage as well as a synchronization agent to ensure consistency across all users on real-time operations. The backend application's codebase is organized as follows:

- **/controllers**: Contains the logic for handling incoming requests and sending responses.
  - **/http**: Contains the controllers for handling HTTP requests.
  - **/ws**: Contains the event handlers for WebSockets connections using Socket.io.
- **/services**: Contains the business logic and services used by controllers for validation and operations, interacting with the databases.
- **/databases**: Contains database access logic.
  - **/postgres**: Handles PostgreSQL database logic.
  - **/firestore**: Handles Firestore database logic.
  - **/memory**: Implements in-memory databases for testing or temporary data storage.
- **/utils**: Contains utility functions and helpers for common functionality.

4.2.1 Data Model

System data is divided into two categories: content data and metadata. Content data refers to large chunks of data that represent the content of a given resource[1]. On the other hand, metadata refers to the descriptive information about a resource, namely its identification and relation with other resources.

Data Splitting

As content data and metadata serve different roles, each type of data comes with diverging requirements. Content data handling requires fast read/write speeds, specially for highly recurring writes in smaller time-frames, as well as flexibility regarding the data structure for storage. In this regard, No-SQL databases are the ideal choice. In the end, we opted with Firestore [10], because of its simple API as well as the previous experience we had with it. Similarly, metadata requires strong data structuring, as well as more complex queries. For this reason, we opted for a SQL database, namely a PostgreSQL one [11].

[1]To note that content data is not a raw representation of the content of a resource, as it could hinder some processes, such as state consistency across users, which will be further explained in chapter 5

In addition to the previously mentioned requirements, we took advantage of internal tools such as triggers to automate database-related tasks, removing strain from the server and improving request-processing times. The entity relationship diagram for the PostgreSQL database is illustrated in Figure 4.3.

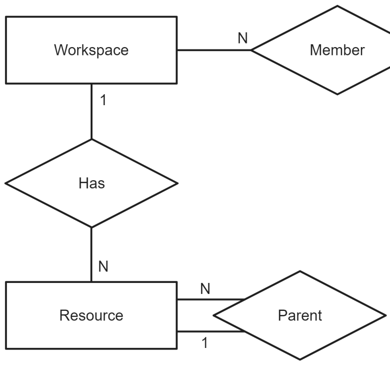

Figure 4.3: Entity relationship diagram of the PostgreSQL database schema

Metadata

As mentioned previously, metadata provides necessary data to identify and relate datasets, as well as provide any additional information; it's comprised of two main entities: Workspace and Resource. A workspace acts as an analog to a Git repository. Its purpose its to organize multiple related resources to improve ease-of-use and organization for the user. Each workspace is independent of each other, as each has their own resources and related data. Resources have a similarly function to workspaces, but in a smaller scale. They can act as means to organize multiple related resources, but are not independent from other resources, given that those share the same Workspace, as they keep a parent-child relation, as seen in Figure 4.4. In the frontend this is represented as documents and folders, which can organize multiple documents and/or other folders together.

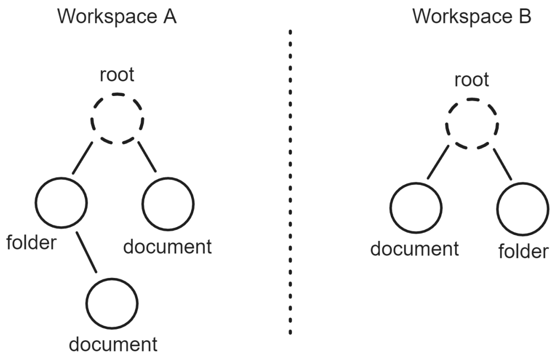

Figure 4.4: Relationship between workspaces and resources

Content Data

The main goal of content data is to store content-related data, which can be significantly larger than the metadata that represents it; a simple example of this can be seen with documents. To represent a document while still ensuring state consistency across users, each user-performed operation is stored in an array of operations, instead of the actual document content. This allows the server to be stateless since all it does is provide and store the operations. The document operations are then stored in Firestore, more specifically in a collection that represents the workspace and a firestore document that represents the workspace document.

This allows for fast read/write speeds, specially as a large amount of operations can be done in smaller time-frames. Note that folders to not have any data other than the metadata, so the only data stored in the document database are the document operations, and the different versions of each document, as explained in section 5.3.

Data Organization

In order to improve read/write speeds, all data is stored in its most basic form, keeping the minimum data needed to represent it and its relations with other datasets. The implementation of these relations is done only on the client side. This vastly improves performance as, for example, representing the tree structure of resources in a workspace directly in the database would slow all CRUD operations [12]; instead, the tree is built on the client side only when needed. More details will be further explored in chapter 5.

4.3 Technology Stack

In this section, we describe the technologies used throughout the development of this project and why we used them.

4.3.1 Core

The core technology stack of NoteSpace includes the common frameworks, libraries, and tools that were used in both the frontend and backend components of the application.

TypeScript

Chosen as the main language of development for both the frontend and backend applications, TypeScript [13] simplified the development process with its type safety and support. It also allowed for building common modules that could be used for both the client and the server, further improving simplicity and robustness while avoiding code repetition.

Node.js

Used as the runtime environment for both the backend and frontend applications, Node.js provides great tooling and public libraries through NPM [14], which maximized our productivity.

Socket.io

Running over the WebSocket protocol [9], Socket.io [15] was used to enable real-time bidirectional communication between the client and server applications. It provided the necessary tools to build this collaborative application with real-time events, rooms, and a variety of other useful functionalities, such as HTTP long-polling fallback, automatic reconnection, and message broadcasting.

4.3.2 Frontend

React

To create a single-page application (SPA) with a simple user interface, we chose React [16]. It allows us to create reusable components, making the development process more efficient and the codebase easier to maintain. By leveraging React's virtual DOM and state management, we quickly built a user interface that updates seamlessly with user interactions.

Vite

Vite [17] served as the bundler and development server, offering fast and optimized bundling of the application and fast iteration speed through its hot module replacement. Vite's modern architecture and use of native ES modules in development provided a significant improvement in build performance, reducing the time required for initial builds and incremental updates.

Slate.js

For the editor styles, both block (headings, lists, etc.) and inline styles (bold, italic, etc.), we needed a framework that could be flexible while still compatible with React. The most popular frameworks that would allow us to achieve this were Slate.js [18], Draft.js [19], and BlockNote [20]. Overall, we agreed that Slate.js was a better choice due to its flexibility and plugin system. Slate's rich API allowed us to implement complex text editing features and customize the editor to meet our specific needs.

Material UI

Occasionally, we used Material UI [21], which provides a library of pre-made React-compatible reusable components for a better and easier way to implement user interfaces.

SCSS

Sassy Cascading Style Sheets (SCSS) [22] was used for styling our application. As a CSS preprocessor[2], it provides a more powerful and flexible syntax than traditional CSS, with features like variables, nested rules, mixins for style-templating, and functions for dynamic styles. This improved the readability and maintainability of the style-sheets.

Vitest

Vitest [23] was chosen as the testing framework for the client application, as it seamlessly integrates with Vite and provides a fast and efficient testing environment.

4.3.3 Backend

Express

Express [24] was used as our framework to build the backend API, providing a robust middleware system, routing, authentication, and error handling, which allowed for faster development of the RESTful API.

Firestore

For storing the document-related content, we decided to use Firestore [10], a No-SQL database that allows for efficient storage and fetching of smaller-sized documents. It was chosen over other No-SQL databases due to its ease of use and simplicity. A No-SQL database was also chosen over a relational database due to its efficiency with larger documents compared to SQL databases.

PostgreSQL

For storing all metadata, such as user, workspace, and document-related information, we used a PostgreSQL database management system [25], which provides a powerful, reliable, and structured database solution that handles structured data and complex queries efficiently. It also supports a robust set of tools like triggers that allow for the automation of tasks on the database layer, removing responsibility from the server.

Docker

Docker [26] was used to deploy our PostgreSQL database, ensuring a better production environment, better portability, scalability, and simplifying deployment and maintenance processes.

[2]In the context of computer science, preprocessors are used to process data input into some other valid format to be compiled later. SCSS preprocesses files into CSS, so that they can be used in browsers.

Jest

For testing our backend code, we used Jest [27], a powerful testing framework with features such as test runners, assertion libraries, and mocking capabilities.

4.3.4 DevOps

Git/GitHub

Git [2] was used for the version control of our project, enabling collaborative development and efficient management of our codebase. GitHub [28] provided a platform for hosting our repositories, organizing our tasks in issues, and managing our project on a project board.

GitHub Actions

GitHub Actions provided us with continuous integration and continuous deployment (CI/CD) by automating our tests on each commit pushed to the main branch, reducing the risk of errors and improving our overall development efficiency.

ESLint/Prettier

ESLint [29] and Prettier [30] were used to enforce quality and consistency across our codebase. ESLint provided static analysis to identify and fix potential issues in our code, while Prettier ensured a consistent style by automatically formatting the codebase.

Chapter 5

Implementation

5.1 Real-Time Editor

The primary goal of this project was to enable real-time editing, allowing multiple users to collaboratively edit the same document simultaneously, ensuring that all users eventually reach the same consistent state. This section details the main implementations and challenges encountered in achieving seamless synchronization and conflict resolution.

5.1.1 Conflict Resolution

In order to maintain all documents states' consistent across all users, we employed the previously mentioned conflict resolution algorithm, called Fugue, by Weidner and Kleppman [4]. Our first version was based on their "string implementation", which initially seemed promising, since the order of the characters was established lexically, removing the need of explicitly implementing a tree data structure. However, we soon realized that it had many drawbacks, mostly regarding the complexity of the algorithm for creating the string positions, code maintainability and future compatibility with other features, such as supporting markdown styles. Moreover, scalability was also a problem, since we also observed that the size of string positions would get exponentially larger over time.

For these reasons, we transitioned to the tree-based Weidner and Kleppmann's approach [31], aligning more closely with their original proposal [4]. In this implementation, a class called *FugueTree* was implemented to represent the tree itself. This class consists of two main fields: root and nodes; the root represents the node from which every other node is a child, and acts as the starting point for tree traversal. It also stores other details which will be further explained. The nodes field is a map that maps the site ID (randomly generated ID for each client session) to all nodes inserted by them. To traverse the tree, we opted for an iterative in-order traversal algorithm instead of a recursive one to prevent stack overflow issues, especially with larger trees. This wrapper also handles cursor-node translation by traversing the tree until the implicit cursor reaches the target. This can be extended to two cursors (a selection) for operations like deletions that affect multiple nodes.

Besides wrapping the tree and adding functionalities, the wrapper provides additional information, such as the site ID. This ID distinguishes different clients sessions, ensuring differentiation across multiple clients.

Operation Flow

Users are able to perform text operations by interacting with the text editor, but in order to integrate both the interface and the conflict resolution algorithm, we needed a wrapper that acts as an intermediary between the tree and the editor. This intermediary wraps around the tree in order to support text-based operations, expanding upon the tree's general structure. We called this wrapper *Fugue* and the flow between the two modules is as follows:

1. The user types in the editor, triggering a DOM input event.
2. An operation in the editor is triggered, calling the event handlers in the Editor component.
3. Through intermediary connectors, a fugue method is called which modifies the tree in order to represent the editor's current state.
4. The connector calls the service layer, in order to send the operation to the server, which is then stored and broadcasted to the other clients in the document.

The architecture and operational flow of the editor, showcasing the interactions in the frontend application, can be seen in Figure 5.1.

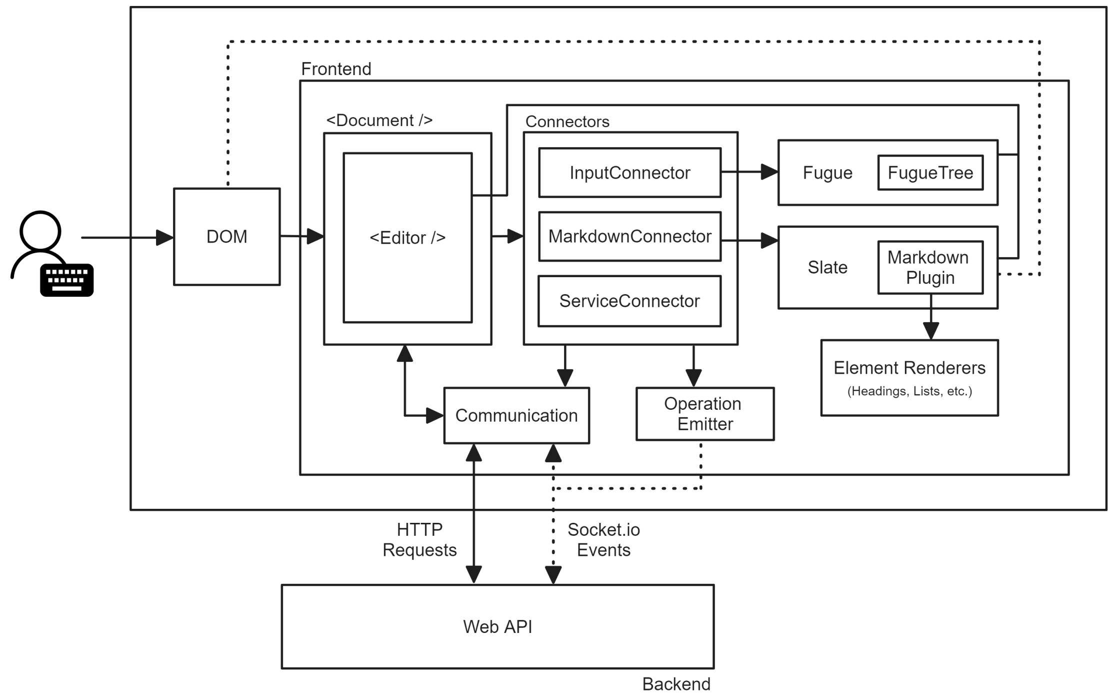

Figure 5.1: Editor operation flow in the client application

Operation Transmission & Storage

The use of intermediary connectors enables a two-way decoupling: one between the editor and the *Fugue* wrapper, and another between the wrapper and the service layer. This improves testability and component isolation.

Using the Communication context, which is used for both HTTP requests and WebSocket events, the service layer sends the events to the server. However, when the event is an operation event, it is not immediately sent.

The transmission of operations is managed through a class called *OperationEmitter*, which transmits operations to the server using an operation buffer to manage network resources more efficiently. Operations are sent through the web-socket stream under two conditions:

1. If the buffer reaches its max chunk size, all operations are sent in a single stream to optimize network usage.
2. If the specified time passes (100ms) and no other operations are added to the buffer, all the operations in the buffer are also sent, even if the max chunk size is not reached.

When the server receives said operations, it firstly broadcasts them. With this optimistic approach, the server does not validate any operations, but only who sends it, which is later explained in the chapter 5.4.2. Moreover, this means that every client, when receiving new remote operations, must apply them locally, which is also done through the *Fugue* wrapper, meaning that the processing needed to convert them into the equivalent tree is deferred to the client. Finally, the backend stores the operations in the Firestore database, in the corresponding document, instead of storing a copy of the Fugue tree, as this extra step would require that the server kept an internal copy of the tree, modify it on every operation, and finally store it in the data layer. This would quickly cause a huge bottleneck in the server if multiple operations are sent at the same time. A diagram demonstrating the document operation handling is shown in Figure 5.2.

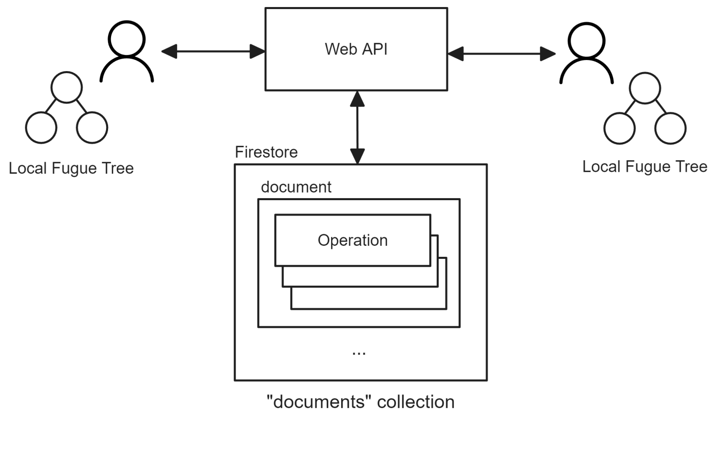

Figure 5.2: Diagram of client trees and operation storage

Optimizations

In order to improve the tree traversal, two main optimizations were added to the basic structure. The first one was adding the 'depth' property on the basic node structure. This allows to early checking if traversal can be done on a node's child, instead of only doing it during the loop's iteration. The second optimization was made by indexing lines' root nodes. As previously mentioned, operations are cursor-based; this means that, in order to translate a cursor into a valid node, we need to traverse the tree. This can cause a bottleneck if a full tree traversal is needed for every single operation so, by indexing nodes which represent line breaks (nodes with character value of '`\n`', which we call line root nodes), the traversal cost for an operation will significantly decrease. Let's suppose the tree is comprised of 20 lines, each with 30 characters. Now let's suppose we want to insert a new character in line 20 at index 10, resulting in the following cursor: (20,10). In the old version, we would need to traverse the whole tree until we reached that position.

This would mean the following:

```
19 lines * 30 characters + 20th line * 10 characters = 580 iterations
```

Now, by indexing lines' root nodes, the cost would be the following:

```
20th line * 10 characters = 10 iterations
```

As we can see simply by indexing lines' root nodes, we get a better performing traversal algorithm. All this allowed for an algorithm that ensured not only state consistencies, faster processing speeds both in the client and server-sides, as well as capabilities for future features, such as offline support, which through the use of Weidner and Kleppman's algorithm [4], requires minimal modification due to its resistance against interleaving.

Additional Features

For a better collaborative experience, there were implemented additional collaborative features. These include real-time cursors, which display the exact cursor or text selection of each user within the document. Users can also see who else is currently in the document and can navigate to any user's cursor position or profile.

5.1.2 Markdown Support

To support markdown styles, such as headings, lists, bold and italic, we used the Slate.js library, as it was one of the few React-compatible rich text editor libraries that also allowed for further expansion, which was vital as we needed a way to integrate user operations on the editor with our conflict resolution algorithm, which will be explained in the next section.

Slate's internal structure is fairly straightforward, as it organizes nodes into *Descendants*, which are higher-order nodes that each represent a line in a document and information about said line, such as a its style and children, which are sub-parts of a *Descendant* and could represent different parts of a line, such as texts with different inline styles, like bold, italic or underline. For every operation, Slate provides data that we then convert into our domain cursors. A cursor is made of a *line* and a *column* field; when an operation gets executed - Slate returns a *path* and *offset* properties; the path is an array made of 1 or more entries, being the first one the index of the *Descendant*, and the second the correspondent children; the offset is the internal offset relative to the children indicated by the path array.

Furthermore, we implemented the markdown support as a plugin, with many supported styles. Our implemented styles can be categorized based on how they apply to the text. We have block styles, which are styles that are only applied to a line, such as headings and lists and inline styles, which can be applied to any character of the text, such as bold and italic.

5.2 Workspace Management

Similarly to other platforms' approaches, resources in workspaces tend to have a hierarchical relationship, as users can nest documents and folders inside one another for better organization. This means we must not only keep information about the relations between different resources, but also provide a way to visually show them to the user.

5.2.1 Resource Metadata Management

All data regarding resources are stored as metadata in the PostgreSQL database, except for data concerning document content, which is stored in Firestore, as explained in section 5.1.1.

Firstly, we represent the workspace itself as an entity with two main attributes: ID and name, which are used to identify it. Next, we must represent resources in a way that allows us to associate them with a workspace and represent relations between different resources, as previously mentioned.

For this reason, resources are represented by a unique ID, a workspace id, a name, a type (document or folder), a parent, and an array of children IDs. Through the 'parent' and 'children' fields, we can relate different resources to represent hierarchy.

Finally, we had to implement some rules regarding resource creation, editing and deletion, as they are crucial for maintaining data consistency. The rules are as follows:

1. When a parent resource is deleted, all children resources are also deleted.
2. When a resource is added as a child of another resource or moved to another resource, both the new and the old parent's children fields must be updated with the ID of the new resource.
3. When a workspace is deleted, all resources within that workspace must also be deleted.

We can assure points 1 and 3 through the use of foreign keys, but not point 2. We consider resource insertion in the context of two different states of the parent. In one state, the parent is null, indicating that the resource is either the root node[1]of a workspace or a child of the root node. In the other state, the parent is non-null, signifying that it represents a valid resource.

For both situations, after the resource insertion and parent's ID validation, we update the corresponding parent by appending the node's id using SQL triggers. In case the node already existed but its parent field was modified, such as the result of moving a resource to another parent, we also remove the node's ID from the old parent's children field.

5.2.2 Workspace State & Tree Management

The state management of workspaces is handled by the workspace context. This context provides a single source of data accessible by any component within the workspace provider. Consequently, when a real-time update occurs, all components using this data are updated, ensuring consistency.

The management of the workspace tree is done via the *useWorkspaceTree* hook, which uses the *useState* hook to store resource nodes. This ensures that updates are reflected in the UI and provides the necessary operations to add, remove, update, and move nodes in the tree.

The nodes in this tree are then used in the Sidebar component to organize documents and folders in a file system-like structure. To convert these nodes into a tree object, we used a Depth First Search (DFS) approach to build the tree object by traversing the children of each node.

The rendering of resources in the tree is then done recursively using the *TreeResourceView* component, which represents a single resource (either a document or a folder) and its children, which are also rendered as *TreeResourceView* components recursively. This algorithm is shown with pseudo-code in Algorithm 1.

[1]The root resource node is created automatically when a workspace is created with an SQL trigger; its ID matches the workspace ID for fast querying in other operations. This serves as a way to be able to represent root-level resources on the hierarchy tree.

Algorithm 1 traverseWorkspaceTree Algorithm

```
Input: $id : string, nodes : Map < string, Resource >$
  $node \leftarrow nodes[id]$
  $children \leftarrow []$
  for $childId \in node.children$ do
      $children \leftarrow children \cup \text{traverseWorkspaceTree}(childId, nodes)$
  end for
Output: tree = {node, children}
```

Note that not only folders can have children, but also other documents. This design choice allows users greater flexibility in organizing their documents.

To move a resource, the user can simply drag it onto another resource, thereby making it its parent. This was achieved by making the resources in the tree draggable and using the drag-and-drop events of the DOM. To get the resource ID of the dragged element, the DOM ID had to match the resource ID, making this operation possible.

To create a new resource in the tree, the user clicks the plus button at the desired location, triggering a popup where they can choose to create either a document or a folder. They can then rename the document, and if it's a document, click it to navigate to the document page.

Both the Workspaces and the Workspace components use a common component, called DataTable. This reusable component displays content in a table format with built-in functionalities, such as sorting elements by columns (ascending or descending), selecting or deselecting rows, and creating or deleting resources, with the handlers passed as arguments defining these behaviors.

5.2.3 Member Management

Effective management of workspace members is essential to maintain proper access control and collaboration. Members are added and removed from a workspace using their email addresses, which are subsequently converted into user IDs. These user IDs are then combined with the workspace ID in a relational table that manages the many-to-many associations between workspaces and users.

5.2.4 Access Control

Through these relationships, it is possible to determine which user is a member of which workspace, enabling the possibility to allow or deny read or write operations for users, ensuring appropriate workspace access control. Workspace permissions are managed through two different middlewares, depending on the type of operation: one for performing read operations, which requires that either the workspace is public or the user is a member of that workspace, and another for write operations, which ultimately requires that the user is a member of that workspace, even if the workspace is public.

All of this also applies when performing an operation on a resource (document or folder), since that resource is also part of that workspace.

It is important to note that all members in a workspace have the same privileges, without any roles. This is an aspect that could be improved in the future, as mentioned in chapter 7.

5.2.5 Isolation of Real-time Updates

In section 5.1.1, we discussed how the service layer sends all user-performed document editing operations to the backend, which in turn broadcasts the operations to all clients in the same document. Documents and folders in a workspace can be modified at any moment, also requiring real-time updates. However, a problem quickly arises with more than one workspace or document. For example, when a user performs an operation in a document, there needs to be a way to only send that operation to the users in that document, and not users in another document.

To achieve this, we took advantage of Socket.io's rooms [32], which can be imagined as, for example, chat groups on messaging apps—users can only receive messages in rooms they are part of; similarly, users can only receive events occurring in rooms they are part of.

We implemented two types of rooms: workspace and document rooms. When a user navigates to the workspace page, they join the respective workspace room. Similarly, upon entering a document, they join the corresponding document room. When the user leaves these pages, the client exits the respective rooms. This way, when a user in a workspace creates a new document, it appears to the other users in the same workspace in real-time, just as when a user inserts a character in a document the other users are in.

5.3 Version Control System

The implemented version control system (VCS) was inspired by GitHub, though with a much simpler approach and functionalities for the specific needs of our application.

5.3.1 Data Model & Storage

The commits are stored in the Firebase database within a dedicated collection named "commits". Each commit stores the currently applied operations on the document, encoded in Base64, along with its metadata, namely a unique id, the commit author, and timestamp.

The diagram shown in Figure 5.3 provides a simple overview of the commit operation flow. It demonstrates the process of creating a new commit in the Firestore collection based on the current version of the document, detailing how the data is stored and managed.

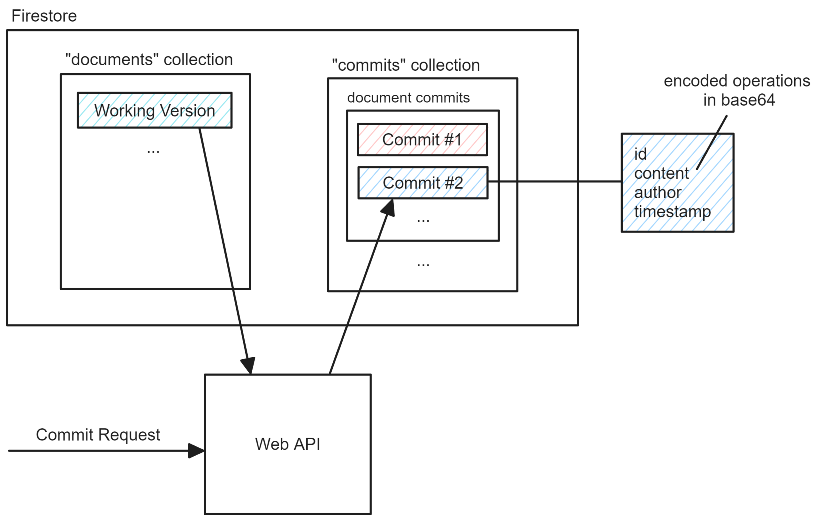

Figure 5.3: Diagram demonstrating the commit operation flow

The rollback and clone operations follow a similar flow to the commit operation, with distinct differences in their specific actions. The rollback operation retrieves the saved commit and updates the working version with its content. In contrast, the clone operation creates a new document using that content.

5.3.2 Functionalities

The implemented version control system supports the following operations:

- **Commit**: Allows users to save a snapshot of the current version of the document, storing it in the database for future reference.
- **Rollback**: Enables users to revert a document to a previous state by rolling back to a specific commit, restoring the document to the state it was in at that point in time.
- **Clone**: Allows users to create a new document based on a specific commit from an existing document within the same workspace.
- **Version History**: Provides users with the ability to retrieve the list of commits associated with a document, allowing them to track changes over time.
- **Detailed Commit Information**: Offers detailed information about a specific commit, including the operations performed, which allows users to view the snapshot of the document from that commit.

While the current version control system provides essential functionalities, there are several enhancements that could be made to bring it closer to the capabilities of platforms like GitHub. These improvements are mentioned in more detail in chapter 7.

5.4 Authentication & Authorization

The user authentication is achieved through the Firebase Authentication SDK, namely using OAuth 2.0, with Google and GitHub as authentication providers. This way, the authentication was easily implemented, with quick integration, support for multiple providers, secure session management and reliability.

5.4.1 Authentication Flow

The authentication flow process, illustrated in Figure 5.4, involves a combination of client-sided user authentication with server-sided user authorization:

1. **Provider Choice**: The user selects their preferred authentication provider (Google or GitHub) in the frontend application.
2. **Firebase Authentication**: Firebase initiates the OAuth 2.0 flow with the chosen provider, handling the authorization process.
3. **ID Token**: Upon successful authentication, Firebase retrieves an ID token used to authenticate the user.
4. **API Request**: The frontend application sends a POST request to the web API with the received ID token.
5. **Session Cookie**: The web API creates an HTTP-only session cookie, verifying the ID token in the process. This cookie is then included in subsequent requests to the web API.
6. **User Registration**: If the user is not already in the system, the user data is fetched using the ID token and is stored in the PostgreSQL database.

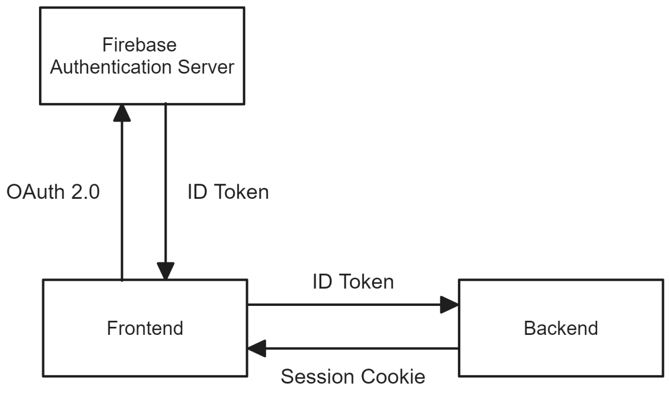

Figure 5.4: User authentication process

5.4.2 User Authorization

User authorization is managed through a combination of middlewares in the web API. These middlewares ensure that only authenticated and authorized users can access protected routes and resources.

- **Session Cookie Middleware**: This middleware retrieves the session cookie from the request. If the cookie is present, it validates the session and extracts the user information, injecting it into the request object.
- **Authentication Enforcement Middleware**: This middleware ensures that protected routes are accessed only by authenticated users. If the user information is not present in the request object, indicating that the user is not authenticated, the middleware responds with a unauthorized status code.
- **Workspace Access Middlewares**: These middlewares check if the authenticated user has the necessary permissions to read or write a specific resource, more specifically, it verifies if the user has read or write permission to a particular workspace. If the user lacks the required permissions, the middleware responds with a forbidden status code.

5.4.3 User Session Management

To ensure secure and efficient session management, session cookies were used to maintain user authentication states. These cookies were configured with a limited lifespan, accessible only through HTTP(S) requests, transmitted exclusively over HTTPS and protected against Cross-Site Request Forgery (CSRF) attacks, using the *maxAge*, *httpOnly*, *secure*, and *sameSite* fields, respectively. These settings ensure robustness against common web vulnerabilities, maintaining the integrity and security of user sessions.

5.4.4 User State Management

On the client side, user authentication state and operations are managed by the *AuthContext*. This context provides the necessary functionalities for user authentication and the current authenticated user information for other components in the application. It ensures that authentication status and user information are consistently available across all components, providing a single source of truth for authentication data.

5.5 Sharing Platform

Beyond the previously mentioned functionalities, additional features were implemented to enhance the platform's usability. These include:

- **User profiles**, which allow users to explore detailed information about others on the platform.
- **Workspace searching** with pagination for easier workspace discovery.
- **Sorting capabilities** that enable users to organize workspaces and documents by various columns.
- **Recently edited documents page** that provides quick access to users' most recent work.
- **Workspace feed** that showcases public workspaces on the homepage, encouraging engagement and collaboration among users.

These enhancements collectively aim to improve user experience and facilitate seamless interaction within the platform.

5.6 PWA Support

To make the application installable on devices, we implemented it as a Progressive Web Application (PWA). Instead of manually creating the service worker and the Web App Manifest JSON file like general PWAs, we took advantage of Vite's PWA support. This is done by a plugin called 'Vite PWA' which allows for fast setup of a PWA with a basic and simple auto generated service worker and manifest files, as well as fully support for granular control of both for more complex behaviours. This allows us to both configure the manifest file programmatically and without the need to manually create the JSON file, as well as setup custom template features, like resource caching and request intercepting for custom responses, which will then result in a generated service worker with the desired behaviours. Through simple typescript configuration files, we can now implement our app as a simple PWA and in the future implement features like offline support and background sync.

Chapter 6

Testing & Deployment

This chapter outlines the testing process undertaken to ensure the reliability and functionality of the project and the deployment of the system.

6.1 Testing

6.1.1 Automated Testing

Automated testing is essential in any software development project, as it enables developers to quickly identify and resolve issues in their code. In our project, automatic tests were executed either through the CLI before pushing updates or by the CI/CD pipeline to automatically test parts of the codebase. We primarily used unit tests, supplemented by some integration tests. Additionally, we employed mocking and other techniques, such as abstracting the interface by which two components communicate. An example of this can be seen on the operation handlers described in section 5.1.1. Another example is the usage of an abstract Communication class, which abstracts the interface used to communicate with the service layer, for both HTTP and WebSocket communication, which in turn communicates with the backend. This allows for mocking the service layer in components that use it, and test them without worrying about side effects from the service layer. This allows for faster testing cycles as well as the isolation of modules.

In the backend, Jest was chosen as our testing framework, used for both unit and integration tests, while in the frontend, we used Vitest for unit testing, namely for testing the domain logic.

6.1.2 Memory Modules

The memory modules are specialized components designed solely for testing purposes. They simulates the behavior of the production databases, allowing efficient testing of the rest of the system without unnecessary overhead. This also allows us to test and validate the project without the database setup, which is useful in the CI/CD environments, like GitHub actions. These modules can be used by running the server in dev mode.

6.2 Deployment

The backend and the frontend applications were deployed in Render [33]. The backend was deployed as a web service while the frontend was deployed as a static site. Additionally, we also deployed the PostgreSQL database in a web service using docker compose, for containerized database management. This approach provides a robust, scalable and secure environment for running the applications. Render's automation and monitoring features, combined with Docker Compose's flexibility in managing containerized applications, ensure that the deployment process is efficient and reliable.

Chapter 7

Conclusions

This chapter provides an overview of the completed work and outlines future plans to expand the project's functionality, with some final remarks at the end.

7.1 Completed Work

Over the five months of development, the project has evolved significantly since its initial proposal. The most significant milestones completed were:

- **Conflict Resolution**: We were able to implement a functional conflict resolution algorithm that allows for idempotent operations that maintain data consistency across users, while still keeping the backend server stateless.
- **Text & Style Operations**: We were able to implement a markdown-supported editor that supports most markdown styles, that integrates with the fugue algorithm to allow for real-time collaboration on documents. Moreover we also implemented some useful collaborative features, such as live cursors and selections, that allows one person to see where each other user is on the document, as well as what they are selecting.
- **Workspace Management**: We also implemented basic workspace management through CRUD operations of documents and folders, as well as displaying the workspace hierarchy to users through the workspace tree.
- **User Authentication & Authorization**: We also implemented user profiles as well as user authentication and authorization through the Firebase Authentication SDK, with Google and GitHub as OAuth 2.0 authentication providers.
- **Version Control**: We implemented a simple version control system, where users can commit, rollback, clone and review versions of each document;
- **Sharing Platform**: Finally, we implemented a basic sharing platform, where users can discover, search and share public workspaces.

7.2 Future Improvements

Even though we achieved a lot throughout these months, there is a lot of room for improvement, namely regarding:

- **Offline Mode**: This feature would allow users to edit workspaces and documents offline, as well as background sync upon re-connection.
- **Workspace Member Roles**: Through Role-Based Access Control (RBAC), we could allow for a more granular control of each user's permissions to the workspace.
- **Version Control**: The system would have to be expanded to not only support document-scoped versions but workspace-scoped as well. We would also like to implement:
  - **Branching**: Implement the ability to create and manage branches to facilitate parallel work.
  - **Forking**: Allow users to create forks of documents to specific workspaces, providing a better way to share and contribute to the platform.
  - **Change Reviews**: Introduce mechanisms for reviewing changes before they are committed and for viewing differences between commits.
  - **Pull Requests**: Enable users to propose changes and discuss them through pull requests, enhancing collaboration and contributions from users outside the workspace.
- **History**: Currently undo and redo operations are not supported as they were very costly in terms of development.
- **Final Touches**: Implement responsive layouts for mobile UI/UX, additional optimizations and potential bug fixes.

7.3 Final Remarks

This project was a significant milestone for the team in our software development journeys, as it allowed us to experience the development of a project of our authorship, instead of relying solely on guidelines. This came with its set of lessons as we learned how to realistically plan features as well as discuss ideas for efficient implementations. Even though most of the original planned features were not reached by the end, we take the experience of developing such a large-scoped project as irreplaceable.

Bibliography

[1] Sun-Li Beatteay Nitin Savant, Elise Olivares. Conclave - a private and secure real-time collaborative text editor. https://conclave-team.github.io/conclave-site, 2023.

[2] Phoenix NAP. How does git work? https://phoenixnap.com/kb/how-git-works, 2021.

[3] Mehul Ghala. The enigma of collaborative editing, 2019.

[4] Matthew Weidner and Martin Kleppmann. The art of the fugue: Minimizing interleaving in collaborative text editing. https://arxiv.org/pdf/2305.00583, 2023.

[5] Matthew Weidner. Fugue: A basic list crdt. https://mattweidner.com/2022/10/21/basic-list-crdt.html, 2022.

[6] Matthew Weidner. "position strings" for collaborative lists and text. https://mattweidner.com/2023/04/13/position-strings.html, 2023.

[7] Matthew Weidner and Martin Kleppman. Fugue: A basic list crdt. https://mattweidner.com/2022/10/21/basic-list-crdt.html, 2022.

[8] mdn web docs. An overview of http. https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Overview.

[9] mdn web docs. Web sockets. https://developer.mozilla.org/pt-BR/docs/Web/API/WebSockets_API.

[10] Google. Firestore: Nosql document database. https://cloud.google.com/firestore?hl=en.

[11] The PostgreSQL Global Development Group. Postgresql: The world's most advanced open source relational database. https://www.postgresql.org.

[12] Codecademy Team. What is crud? urlhttps://www.codecademy.com/article/what-is-crud.

[13] Microsoft. https://www.typescriptlang.org.

[14] NPM Inc. https://www.npmjs.com.

[15] Socket.io. 1urlhttps://socket.io.

[16] Meta Open Source. https://react.dev.

[17] Vite—next generation frontend tooling. https://react.dev.

[18] Ian Storm Taylor. https://github.com/ianstormtaylor/slate.

[19] Draft.js. https://draftjs.org.

[20] Blocknote. https://blocknote.net.

[21] Material UI SAS. https://mui.com.

[22] Sass. https://sass-lang.com.

[23] Matías Capeletto Anthony Fu. https://vitest.dev.

[24] Open JS Foundation. https://expressjs.com.

[25] Postgresql. https://www.postgresql.org.

[26] Docker. https://www.docker.com.

[27] Jest. https://jestjs.io.

[28] Github. https://github.com.

[29] Eslint. https://eslint.org.

[30] Prettier. https://prettier.io.

[31] Matthew Weidner. Tree fugue. https://github.com/mweidner037/uniquely-dense-total-order/blob/master/src/implementations/tree_fugue.ts, 2023.

[32] Socket.io. Rooms — socket.io. https://socket.io/docs/v4/rooms/.

[33] Render. https://render.com/.

[34] Sarah Laoyan. What is agile methodology? (a beginner's guide). https://asana.com/pt/resources/agile-methodology, 2024.

[35] Whimsical. https://whimsical.com/.

[36] Excalidraw. https://excalidraw.com.

[37] Luke Vu. Pwa vs spa: Alike but different. https://www.simicart.com/blog/pwa-vs-spa/, 2019.