Distributed System Project final Report: Collaborative Text Editor

Matteo Iorio
matteo.iorio2@studio.unibo.it

Stefano Furi
stefano.furi@studio.unibo.it

Fabio Vincenzi
fabio.vincenzi2@studio.unibo.it

April 18, 2024

1 Introduction

1.1 Goals

The project's objective is to create a **real-time collaborative text editor**. This web application will allow a user to *create* a text document and then *share* it with any number of other users by granting them either **read-only** or **write permissions**. Users with write permissions can collaboratively *contribute* to the document. Their typing will be visible in real-time to others viewing the document. Users with write permissions can also initiate document saves, which will be executed accordingly.

The file owner has the flexibility to **modify a user's permissions** at any given time. If a user's permission kind changes they will *receive immediate notification*. If permissions are revoked, the document will *automatically close* for the affected user, and they will lose their permission capabilities. Conversely, an upgrade to write permissions will make the document editable for the user. If a user gets a read-only permission he will not be able to edit the editor, but he will only see the other users write on that.

Furthermore, the text editor will **feature auto-save functionality**, automatically saving the document at predefined intervals to ensure work is not lost.

1.2 Q&A

Question: What is the primary aim of the project described in the text?

Answer: The primary aim of the project is to develop a real-time collaborative text editor. This web application allows users to create and share text documents with others.

Question: Is It possibile to change permission while the other users are using the editor?

Answer: Yes, It is possible to change at run time the permission of every user. The effected user will be noticed about his new permission and if his permission is revoked the document will be closed otherwise if He has a read-only permission He will not be able to edit the file otherwise He will have the ability to edit the document.

Question: How can I use this editor?

Answer: If you want to use our collaborative text editor, as first thing you have to signup into the system by creating an account. Each account is characterized by an *username* and a *password*. After that the user can *log in* into the system and use the editor by creating document or join the document shared with him.

Question: How can I see the documents shared with me?

Answer: The first thing that a user must do is log in into the system. After that, the user will be able to see in a section all his documents and in another section all the documents that are shared with him.

1.3 Requirements

This section outlines the specific requirements necessary for the successful completion of the project and to ensure clarity and comprehensive coverage of the project's needs.

Functional Requirements

- *Architectural Requirements*: the system will operate on a Client-Server architecture which should provide a highly available system, capable of handling a considerable amount of user's request, and in the same time keeping every document in a consistent state, so that every client sees the same version of the document they are working on. All the architectural requirements must not interfere with user's interaction with the system (i.e. interactions with the system must be flawless as if it is a simple single client-server architecture).
- *Database*: The database must be able to handle a reasonable amount of read and writes operation, while keeping a consistent view of the underlying data. Most of the reads and writes operations will occur inside those tables who are responsible for document's contents, while other table will be subject of a much less stress.

The whole database architecture must operate in order to address the single point of failure problem, providing an overall *highly available* service.

- *Web Server*: The web server should be able to handle all users' operations successfully through the exposure of a web page in which the application runs. The web server should be *highly available*, meaning it should support a reasonable amount of users using its service, and in the same time implementing mechanisms of *fault tolerance*.

- *User Requirements*:
  - *User Data Interactions*: the user uses REST APIs for the data access. The APIs return the data only if the user is currently logged inside the system;
  - *User Operations*: a user must be able to:
    * Create an account;
    * Log in (if the account exists);
    * Create a new document (if logged);
    * Edit his own documents (if logged);
    * Manage the permissions on his own documents (if logged);
    * Edit documents where He has write permission (if logged);
    * Read documents where He has read-only permission (if logged);
    * Sign out (if logged);
  - *User Navigation*: the user must be able to navigate into the website only if he is logged in;

- *Document Interaction*: all the users must be able to collaborate on the same shared document all together at the same time;

Non-Functional Requirements Non functional requirements includes:

- *High performances*: while the system must be able to keep up to a reasonable amount of users, it should also implement real-time collaboration functionalities with the highest performance as possible.

- *Limit system resources usage*: considering the previous requirement, the system should be quite lightweight in terms of system resources, both in server side computation and network transmissions/communications.

- *Security*: having among the user's requirements operations like log in and registration, user's related informations should be kept encrypted inside the system, and properly secure procedures should be taken in place in order to establish secure connections from server to client.

Now that all the requirements are explained, in Figure 1 is shown the *UML usecase diagram* of user's operations.

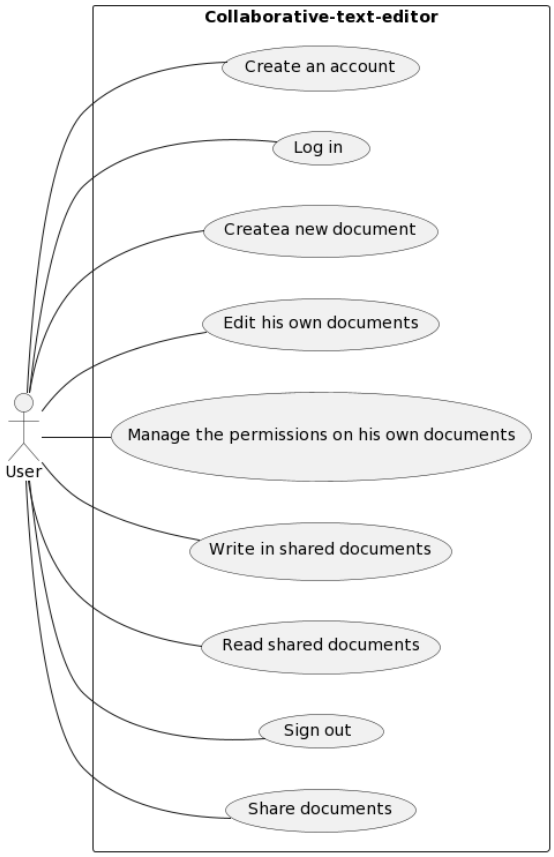

Figure 1: User's use case diagram

1.4 Scenarios

In the development of any software, especially one designed for collaborative work, understanding the context within which it will be used and the users who will interact with it is paramount. This section of our report delves into the **scenarios** and **user profiles** for our project: a collaborative text editor. The purpose of exploring these aspects is twofold. Firstly, scenarios provide us with vivid, concrete examples of how our text editor can facilitate collaboration among users in various contexts. These scenarios help in illustrating the practical applications of our editor's features, shedding light on its *potential* to enhance productivity and streamline the *collaborative writing* process. Secondly, by identifying and describing the user profiles, we gain insights into the needs, goals, and challenges of our potential users. This understanding enables us to tailor the design and functionality of our collaborative text editor to better meet these needs, ensuring a more intuitive and effective user experience.

With the ever-evolving landscape of collaborative projects, our text editor is designed to address a wide array of *use cases*, fostering teamwork and creativity among its users. This section outlines several *key scenarios* that illustrate the versatility and utility of our collaborative text editor. Each scenario is crafted to showcase how different user groups can harness the editor's features to collaborate efficiently, overcome challenges, and achieve their goals. By exploring these scenarios, we aim to demonstrate the practical applications of our text editor in real-world contexts, highlighting its potential to transform collaborative writing experiences.

1.5 Detailed Scenarios

- **Scenario**: Collaborating on University Notes
  Description: A group of university students uses the collaborative text editor to work on their university notes. They leverage the real time editing and commenting features to brainstorm, draft, and revise their work collectively.
  Actors: University students;
  Steps: One of the student has to create the document. After the document creations, the owner has to share It with all the other students interested in this document, the owner can decide for each user's permission.

- **Scenario**: Organizing a Conference
  Description: A team of organizers collaborates on planning a professional conference. They use the collaborative text editor to draft the agenda, coordinate keynote speeches, and compile biographies of speakers. The real-time collaboration feature allows them to work together simultaneously from different locations.
  Actors: Conference Organizers, Keynote Speakers
  Steps: Organizers create a shared document for the agenda and speaker bios. They assign sections to each organizer to input information. As speakers confirm, their details are updated in real-time. The final agenda is then shared with speak-

ers and attendees using a read-only permission.

- **Scenario**: Planning Job Schedule
  Description: This scenario can take place in a generic company. We assume for example that is a city restaurant. In this scenario, there is only one actor that is allowed to write in the document, the job scheduler. This person is in charge to write for each worker their job schedule. In the meantime all the other workers can only see the job schedule. By using the real-time collaborative text editor allows the job scheduler to share the job schedule to all the other workers;
  Actors: Job Scheduler, Company Workers
  Steps: The Job Scheduler creates a shared document for each week. The Job Scheduler share this document to all the workers inside the company by using a read-only permission. This allows the job scheduler to write in real-time to all the users in the company without wasting time in creating a single file for each worker and share It to him.

1.6 Self-assessment policy

Assessing the quality of a collaborative text editor, like the one developed in our project, requires a multifaceted approach that considers the unique challenges and goals of such a platform. Quality assessment should focus on several key areas:

1. **Functionality and Feature Completeness**:
   - **Correctness**: ensure that all the features works as intended, including text-editing , real-time collaboration and document sharing;
   - **Feature Coverage:** evaluate whether the software includes all the essential features identified during the requirements gathering phase, catering to the needs of various user scenarios outlined.
2. **Usability and User Experience (UX)**
   - **Ease of use:** the interface should be intuitive for new users and facilitating easy navigation.
3. **Performance and Scalability:**
   - **Responsiveness:** measure how quickly the software responds to user inputs, especially in collaborative scenarios with multiple users editing simultaneously;
   - **Scalability:** evaluate the system's ability to handle a growing amount of work or its potential to be enlarged to accommodate that growth;
4. **Collaboration Efficiency:**

- **Real-time Performance:** assess the latency in reflecting changes made by one user in the view of another user, aiming for near-instant updates;
- **Conflict Resolution:** evaluate the software's ability to handle conflicts when simultaneous edits occur, ensuring that data integrity is maintained;

The quality of our *collaborative text editor* software has been thoroughly assessed by meticulously evaluating all course arguments, ensuring comprehensive coverage of key functional and non-functional requirements. Additionally, this evaluation process has been rigorously conducted within the strict time availability allocated for the project, demonstrating our commitment to delivering a completed software under tight deadlines.

2 Requirements Analysis

The requirements exposed in the previous chapter illustrate broadly which characteristics our system must provide. Firstly, there is the explicit constraint of a web-based application, where there are no constraints about development platforms (e.g. the Java Virtual Machine (JVM)). Moreover, the system must be thought and designed keeping in mind three main concept: real-time consistency among users, high availability of the system and consequently its tolerance to faults. The first is explained later in Section 2.2, while high availability will be covered along in this chapter introducing the overall system architecture, and in Section 3.

2.1 System Architecture

Analyzing system requirements, the first thing to think about is the system architecture. While Client Server architecture are by far the most common design implementation, a collaborative editing scenario can benefits from other kinds of system architectures. In the following sections we dive into analyzing two different kind of system architecture, highlighting pros and cons of both, leading the way to further choices taken in the design phase.

2.1.1 Client-Server Architecture

Client-Server architectures are the most common system architecture when it comes to designing and implementing web services. In a CS architecture a server acts as the central hub processing requests and managing data state. This allows for seamless horizontal scaling, mitigating single points of failure and improving overall system performances. The whole logic of the application and its state is encapsulated inside this component, which has among its responsibilities tasks like consistency handling and security and access control policies. In such scenario, clients acts as consumers of the service exposed by the server. Their capabilities are limited to set of features implemented by the server and every operation made on top of application resources must be regulated and checked on top of the server.

This architecture separates the user interface from the application logic and data storage, offering several benefits: firstly, client side development is greatly simplified, making them lightweight and focusing on rendering and communicating with server. Also, such separation allows for flexibility in choosing different client technologies (e.g. web or desktop apps) as long as they can communicate with the server. Lastly, client and server uncoupling make client and server updates independent one to another, improving deployment and maintenance efficiency.

However, making use of a CS architecture also comes with some challenges, especially in the use case of this project. First of all, designing such distributed server will require strong coordination techniques, along with the use of consensus algorithms (e.g. RAFT) for keeping server replicas in a consistent state. Another problem can be related to network latency in a real-time environment which can be hard to manage, especially with strong requirements of responsiveness.

2.1.2 Peer To Peer (P2P)

P2P architectures remove the central server, fostering a more democratic and potentially scalable environment. Each user's device acts as both a client and a server, directly communicating with other collaborators. Theoretically, a P2P architecture can scale effortlessly with increasing users, as the processing load is distributed amongst all participants. While this can be true in certain scenarios, in the context of a collaborative editing environment the higher the number of users is, the slower the processes of propagating edits is. Another problem that can arise from the use of a P2P architecture is the lack of a single source of truth, especially considering edits that can be in conflict with each other. While not having a centralized server improves service capabilities (e.g. can scale up without complex management of resources), a challenge lies in user discovery and connection: without a central server to coordinate connections and the establishment of communication channels, a set of procedures must be accounted in order to provide this functionalities among all peers autonomously.

2.1.3 Hybrid Architecture

A mix of a CS and P2P architecture can be used in order to mitigate problems that arises when using one or the other. The main problems with P2P architecture is the lack of a source of truth for the documents, especially in case all users log out from the P2P session, and the complexity in handling user churns and the establishment of an initial communication channel: these two critical tasks can be delegated to a server. The server now becomes the source of truth for shared data, keeping them safely inside a database. In the same time, clients handle data modification and communications, keeping the server free from handling conflicts and propagating changes amongst all users. In this scenario functionalities like explicit saving of the document or autosaving, will trigger a write operation to the database executed by the server.

Thanks to this hybrid approach, the server will not be in charge of complex tasks like conflicts resolutions in edits made by users, but its task will include:

- *Security*: the server can implement security measures to keep unauthorized users out of the connection between other peers;
- *Permissions*: server is now responsible of ensuring each user will join peers network with the right permission assigned to them;
- *Data Consistency*: the server is the source of truth for shared data. Each peer that queries the server, will get the same, up-to-date, version.

In this scenario, the complex task of handling edits must be implemented inside clients, which then must propagate computed edits to all other peers, which may eventually converge to the same shared data version.

2.2 Granting document consistency

One of the core problem of our project is to understand how to grant the **document consistency** for each shared document. It is crucial to highlight that maintaining consistency across collaborative editing sessions is a fundamental challenge of our project. This challenge stems from the necessity to ensure that all users see the same version of a document, even when changes are made simultaneously by multiple participants. In the following sections, we will illustrate how real-time synchronization and consistency can be archived in order to address our core problem.

2.2.1 Operational Transformation (OT)

OT is a set of mechanisms and algorithms for supporting a range of collaboration functionalities in advanced collaborative software systems. It is designed to ensure that all participants of a collaborative editing application have a synchronized view of the shared data, despite potentially conflicting edits.

In a collaborative editing scenario using OT, each user maintains a local copy of the shared data (e.g. the document). This will allow responsive editing even in high-latency environments. Every edit performed by a user is represented as an operation (e.g. insert, delete, retain), that is a transformation applied to the local data state. When concurrent edits target the same portion of the data, conflicts arise. In these scenarios OT algorithms detect these conflicts and transform the operations to ensure they can be applied sequentially without altering the intended outcome.

This is done thanks to the *operational representation* of each edit made by users, where every edit is represented as a function that takes the current data state and produces the modified state as output. When it comes to transforming the state and resolving conflicts, OT mainly uses two kinds of *transformation functions*:

1. *Inclusion Transformation* (forward) where an operation `a` is modified in order to account the effects of a subsequent operation `b`, ensuring `b` can be applied after `a` without altering the outcome (e.g. two insertion at different positions);

2. *Reordering Transformation* where operations are rearranged to ensure consistent order of execution, even if they were received in a different order (e.g. concurrent insertions at the same position).

Once conflicts are identified and resolved using transformation functions, the modified operations are distributed and applied to all user's local data copies, maintaining consistency across all replicas.

Conclusions While OT enables smooth and responsive editing for multiple users without requiring locks or centralized control, the design and implementation of such technology can be challenging due to the need to handle various conflict scenarios. Other challenges are represented by the **Causality preservation**, i.e. ensuring the order of operations reflects the intended user actions, and the strong dependency on **operational commutativity**, which might not hold for all circumstances.

2.2.2 Conflict-free Replicated Data Types (CRDT)

CRDT are a class of data structures specifically designed for distributed systems where data is replicated across multiple devices or servers, with the following features:

1. Each replica can be updated independently, concurrently and without coordination with other replicas;

2. An algorithm (itself part of the data type) automatically resolves any inconsistencies that might occur;

3. Replicas are guaranteed to eventually converge to a consistent, shared state.

Eventual consistency is obtained leveraging specific *properties of the operations*. This removes the complexity of handling real-time conflict detection and resolution, simplifying data management in distributed environments. More specifically, in a collaborative editing environment, a *Sequence* CRDT can be suitable to the needs of such system, due to the fact that updates are Operation based (rather than State based) meaning that the bandwidth usage is drastically reduced, and the representation of the data and operations upon them is very simple: the text is represented as an array of characters, and operations are like OT's ones (insert/delete/retain at position X within the sequence). The key advantage of sequence CRDT lies in their straightforward merge logic, where incoming changes are simply appended to the local sequence regardless of their origin or position, and in case of conflicts, they are inherently resolved during insertion. Concurrent insertions at the same position are handled by appending them in the order received. This might lead to a temporary inconsistency, but eventually all clients will converge to the same state by applying all operations in the received order.

Conclusions The use of CRDT eliminates the need for complex coordination protocols among replicas in a distributed system, providing a way to handle concurrent updates efficiently and facilitating the overall scalability of the system. All of this comes to the

cost of eventual consistency (which might not be very suitable for a collaborative editing environment) and the challenges that arises when designing and implementing complex merge logics to ensure the desired behavior.

3 Design

The system design can be split into three major components:

- Server design;
- Client design;
- Distributed service architecture design.

The latter is preemptive for the development of the server component, which design is led to a distributed-driven design.

In the following sections we will dive into the design choices that has been taken, in light of the analysis done in previous chapters. We will start discussing how the system must be distributed in order to keep up to reasonable amount of users requesting the service. Having cleared distribution aspects, we can proceed to define the structure and the behavior of server and clients, and finally how they interact with each other.

3.1 High-Level Design overview

In figure Figure 2 is shown a simple high level overview of the system architecture.

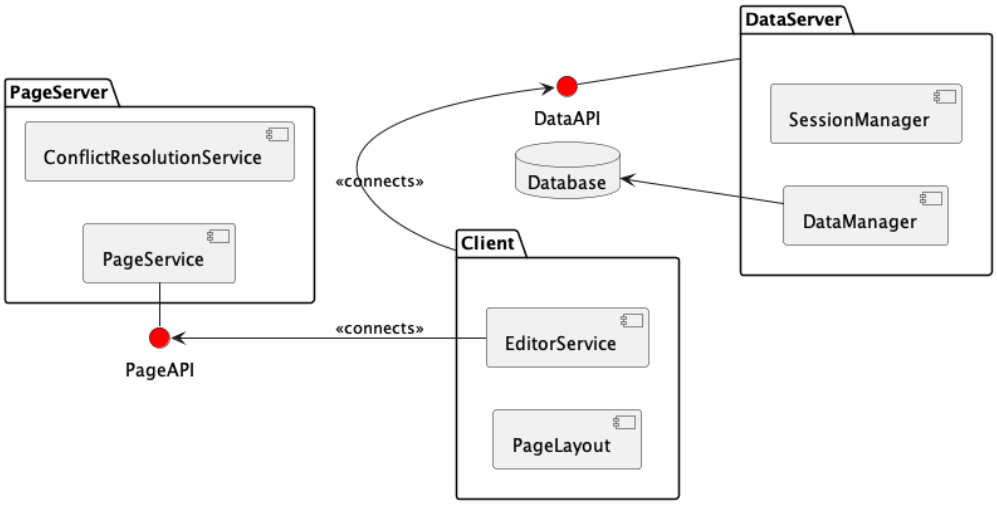

Figure 2: High Level UML component diagram of the system architecture, illustrating how server and client are composed and interact.

Following is listed the structure and the behavior of each component:

- `DataServer`: this package contains the components responsible for **data consistency** and the data service exposure. More in particular, the `DataManager` component handles all interactions with the database and exposes CRUD operations through a set of REST APIs. User sessions and permission are then delegated to the `SessionManager` component.

- `PageServer`: this package exposes two main services: the first will serve application pages to clients, while the second, `ConflictResolutionService` will be responsible for automatically resolves conflicts that may arise while editing the shared document concurrently.
- `Client`: this package contains the `PageLayout` component, which structure is deeply described in Section 3.5, and the `EditorService` which responsibilities include communicating with the APIs exposed by the page server.

Having defined the overall structure of the wanted design, in the next sections we will go more in detail explaining components' responsibilities and interactions with each other, but first, let us discuss about how the system must be designed keeping in mind the requirements of high availability and distribution.

3.2 Distributed System Architecture

As explained in system requirements, the system must implement a set of mechanisms in order to provide high availability and data consistency. In the following sections a high level design overview is provided for both of those aspects.

3.2.1 Data Consistency and Availability

To provide data consistency, both the database and the data server must be designed keeping in mind that operations must occur in a concurrent fashion. A *relational database*, thanks to its **ACID** properties, can grant that concurrent transactions can occur sequentially without worrying about problems like dirty reads, lost updates or phantom rows problems. The biggest problem that arises with relational database, in compliance with the CAP theorem, is that when availability and consistency are granted, distribution of the database can be challenging if not impossible. Considering this aspect, the adopted databse design is illustrated in Figure 3: data servers query the database cluster as expected (i.e. without knowing its a distributed database), while depending on the operations, requests are redirected to specific replicas; more in particular, writes operations are handled by the `MasterDB`, who is also responsible for notifying all replicas that some data changed. All the other replicas are read-only databases, acting as slaves of the primary database. While this configuration is more suitable for scenarios where read operations are much more frequent than write operations, another important role is played by slave replicas: when the primary database fails, the system will be aware of the failure, and an election is done with all remaining replicas for choosing a new primary database. After the end of this procedure, a slave is promoted as primary replica, and the system can continue to operate as expected.

Designing the database in this way, not only improves drastically read performances, but also provide an overall high availability of the data system. Finally, making use of a relation database can ensure a robust level of data consistency.

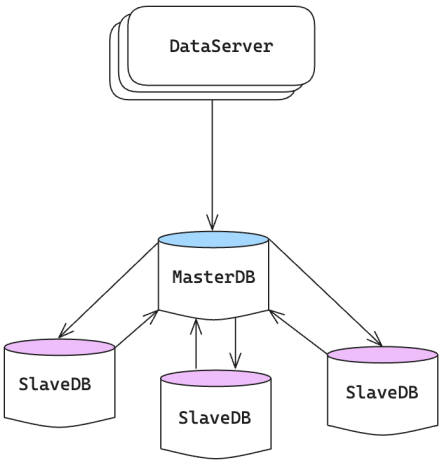

Figure 3: High level design of the distributed database: the `MasterDB` is responsible for writes and propagation to slave of updated data, while `SlaveDB`s are in charge of read operations.

3.2.2 Data server availability

The data server, critically responsible for both communicationm with the database and handling user requests (CRUD operations) in a timely and efficient manner, requires high availability to minimize any potential downtime or keep up with a reasonable amount of users requesting its service. Simply replicating the data service itself, however, proves to be an insufficient solution, due to the fact that among the data server responsibilities, is included user sessions and permission management. A more suitable approach is to implement a shared ledger for session information. This allows all data server replicas to base their session decisions on the same, central source. Nevertheless, it becomes equally crucial to ensure that this session management ledger itself maintains a state of high availability to guarantee the continued proper functioning of the data servers: here a distributed in-memory store can be employed, ensuring that its functioning is regulated by a robust consensus algorithm (e.g. **RAFT**) in order to archive session data consistency among all store replicas.

With a data server built in this way, we can obtain an overall high availability, with some interesting properties on fault tolerance: `DataServer`s can fail and later be restarted without causing significant problems on the service (excluding Byzantine Fault problems), while the session store fault tolerance is strictly dependant on its consensus algorithm properties. With the design described in this section, we can provide a highly available data and session management service, which can help also improve the overall system capabilities in terms of requests served.

3.3 Data Server

The design of the Data Server resulted in a clear distinction between two fundamental concepts within our system: **documents** and **users**. This approach allowed us to achieve greater separation and, concurrently, greater extendibility of the project. As depicted in the figure below (figure fig. 4), we outlined a set of classes to manage data-user interactions based on the requests made. Whenever a user submits a request, it is done through one of the *REST APIs* provided by the Ktor server. We have different classes that helps us to handle the user's requests:

- **API Routing**: as mentioned earlier, we decided to adopt a clear separation between the concepts of document and user. In the classes listed below, various exposed APIs have been defined and implemented.
  1. `DocumentRouting`: class that implements all *APIs* related to documents;
  2. `UserRouting`: class that implements all *APIs* related to users;
- **Data Resolver**: the second category of classes we need is those that *interface* with the database. These classes allow us to execute the queries requested by the client, collect the data, and return them in response to the requests. To manage this process, we decided to entrust this task to specific classes, while ensuring the respect for the concepts of document and user:
  1. `DocumentResolver`: class that does all the queries related to the *Documents*;
  2. `UserResolver`: class that does all the queries related to the *Users*.

Every object returned to the user will be converted into **JSON** format, thereby simplifying the exchange of information between the server and the user.

3.4 Page Server

The page server has two primary roles: the first is to provide application pages to users requesting the application service, and the second is handling users changes to documents while providing automatic algorithms for possible conflicts.

3.4.1 Document Handling

The `PageServer` major role deals with collecting and applying users changes to the shared document. The approach that could be suitable for our requirements is the implementation and application of OT algorithms for collaborative editing scenarios.

While this approach can be very challenging due to the set of problems that arise when dealing with automatic conflict resolution algorithms, in the end the system will have a set of robust and reusable procedures capable of handling a reasonable amount of edits sent by users. In figure Figure 5 is shown the overall behavior of the `PageServer`: this component is responsible for accepting connections from clients and providing them last up-to-date version of the document, while in the same time listens for users changes, applies the edits to document (resolving possible conflicts that can arise) and finally

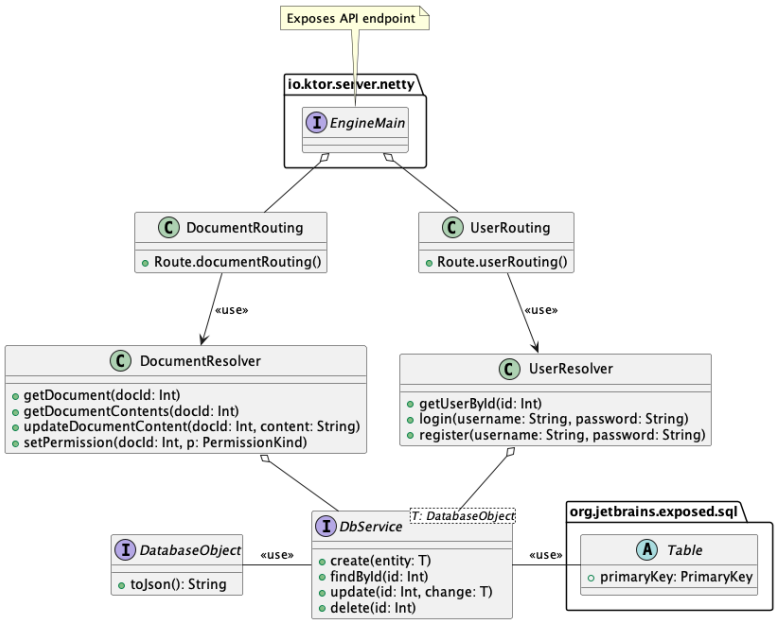

Figure 4: UML class diagram of the system data architecture, showing the process for the data retrieving.

propagate the updated version to all clients. Later in Section 4.2.1 will be in depth explained how the conflict resolution mechanism can be archived, and how client and `PageServer` interacts for obtaining a real-time collaborative editing environment.

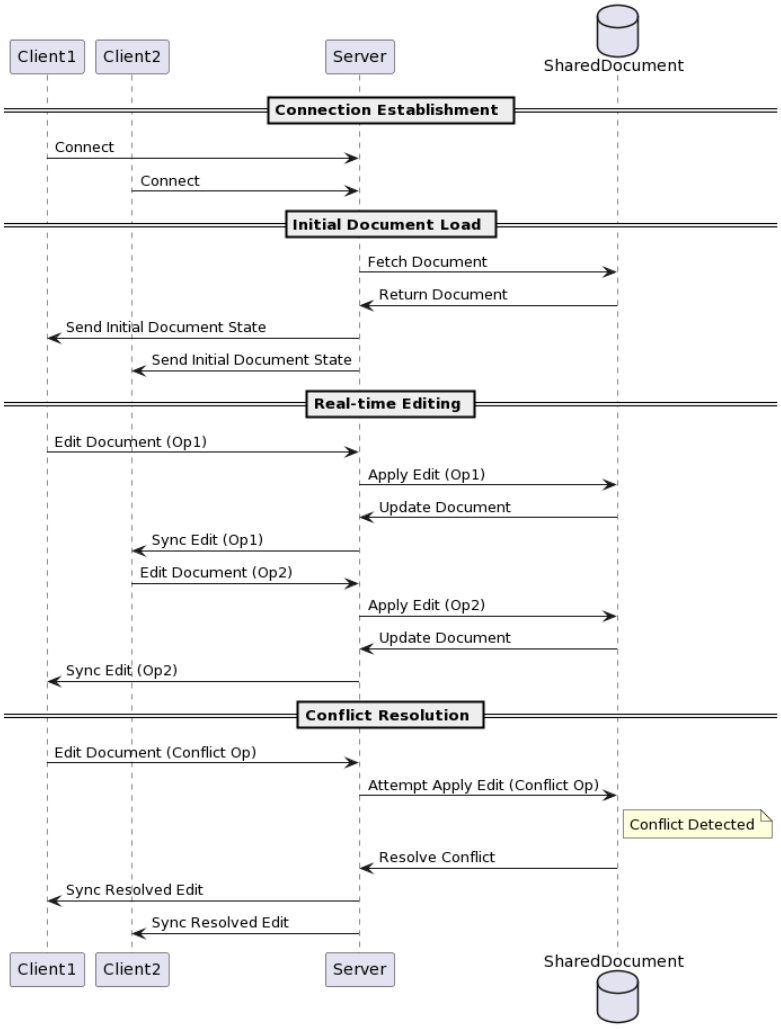

Figure 5: UML sequence diagram showing the document handling

3.5 Client

3.5.1 Login Page

The login page serves as the entry point for users into the editor. It presents an interface where users can login by providing their credentials, composed by a username and a password. Upon successful authentication, users gain access to their account and the main features of the application.

3.5.2 Signup Page

The signup page allows new users to create an account within the system. It offers a straightforward registration process where users provide the necessary information: username and password. After completing the registration, if no duplicated usernames are found, users can immediately proceed using the editor.

3.5.3 Main Page

Upon logging in, users are directed to the main page, which serves as the central hub of the application. Here, there is a navbar, providing easy access to different sections of the application. The main page displays a dashboard or summary of the user's documents, including those owned by the user and documents shared with them. Users can perform various actions from this page, such as creating new documents specifying the name and managing existing documents.

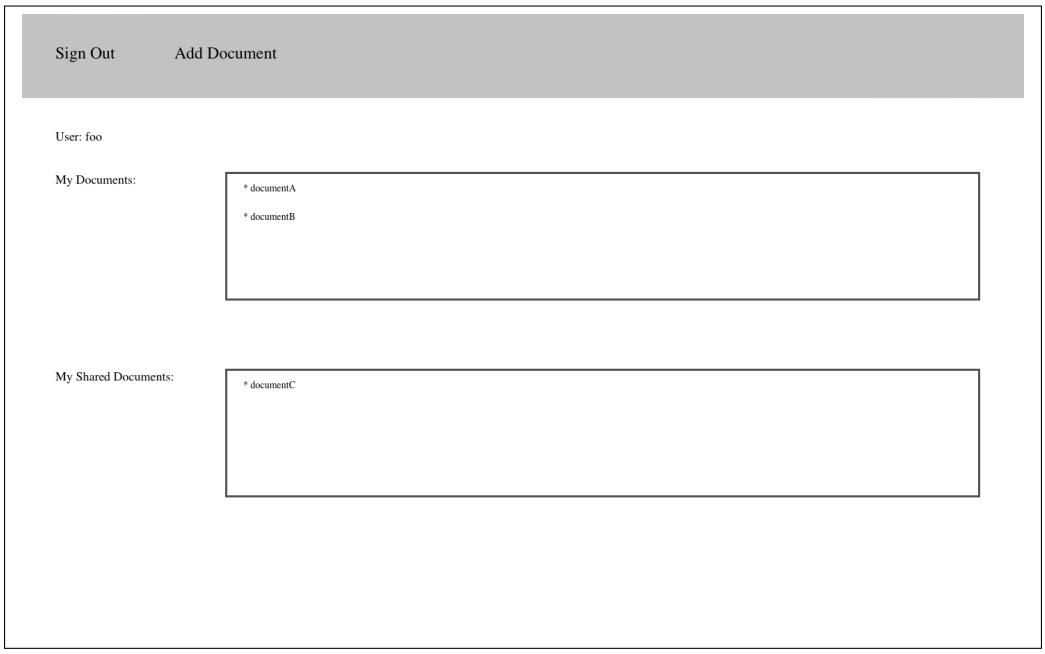

Figure 6: Home page mockup design

3.5.4 Editor Page

When a user selects a document from the main page, they are directed to the editor page, where they can view and edit the selected document. The editor allows users to make modifications to the document in real-time. Users can save their changes and collaborate with others by sharing the document and assigning specific permissions, such as read-only or editing rights.

3.5.5 Page Handling

In the development of our real-time collaborative document editor, we opted for Node.js as the primary technology for several reasons. Node.js facilitates efficient management of multiple connections, a crucial aspect of real-time collaborative applications.

Our server architecture is structured around Express.js, a minimalist web application framework for NodeJs. `ExpressJs` provides a robust set of features for building web applications and APIs, making it an ideal choice for handling **HTTP** requests and routing within our distributed system.

On the client side, we employ a combination of `HTML`, `JavaScript`, and `CSS` to create a seamless user experience. JavaScript plays a central role in enabling real-time collaboration features, such as simultaneous editing.

4 Implementation Details

4.1 Server Tecnologies

4.1.1 Exposing Server's API

One of the primary functions of our Ktor server is to *provide* clients with the appropriate data in response to their requests. To achieve this goal, it's crucial to offer users various ways to *interact with the server*. In doing so, the server, upon receiving a request, is able to process it and then return the requested data accurately. This process requires our Ktor server to be equipped with a **REST API** service implementation. This implementation entails that the server makes various routes externally available, allowing users to submit requests via HTTPS methods. Regarding the development of this system, we decided to divide the REST API implementation into two separate packages: one dedicated to **documents** and the other to **users**. The realization of this service has thus been organized into two sections:

1. **Routes Creation**: In classes `UsersRoutes` and `DocumentRoutes`, each route is represented by a constant variable, whose content is a simple string representing the relevant route being exposed. This approach has allowed us to clearly distinguish between the concept of the path to be reached and its actual implementation. Thanks to this separation, not only have we made the distinction between the two concepts clearer, but we have also simplified the future modification of the paths;

2. **Routes Implementation**: For the actual implementation of the exposed routes, we opted to create two distinct classes: one dedicated to users `UserRouting` and the other to documents `DocumentRouting`. Within these classes, we implemented the respective routes. Each route was developed by defining the corresponding *HTTPS method* and the path to use, thus allowing the client to interact with the route using the method specified by the server.

4.1.2 Data Server

While Node.js is a powerful and popular choice for many web applications, particularly due to its asynchronous, non-blocking I/O model, there are specific considerations that led us to prefer Kotlin over Node.js for our Data Server:

- **Type Safety Concerns:** Node.js, with JavaScript at its core, is dynamically typed. While this can offer flexibility, it also introduces a **higher risk of runtime type errors**, which might not be acceptable in the critical role our Data Server plays in managing database queries and information exchange;
- **Scalability and Maintainability:** As our project scales, the static typing in *Kotlin offers clearer contracts* between different parts of our server code, *enhancing maintainability*. Large, complex applications can become harder to manage with dynamic typing due to the potential for subtle bugs and the increased effort required for understanding and ensuring type correctness throughout the codebase;
- **Ecosystem and Tooling:** While Node.js has a vast ecosystem, the choice of Kotlin and Ktor allows us to leverage the mature ecosystem of the JVM for performance optimization, security, and reliability, aspects critical for the backend of our collaborative text editor.

The decision to use a JVM-based technology like `Ktor` reflects a strategic continuation of our learning journey in **Java**. It represents our intent to apply the course's teachings in a practical project while embracing newer technologies that enhance our application's design and functionality. By doing so, we aim to ensure that our collaborative text editor stands at the forefront of innovation, offering a robust and efficient platform for real-time collaboration.

4.1.3 Managing Sessions and Verifying User Access

One of the key steps in the development of our system has been *managing client sessions across various distributed servers*. Given that the Ktor server implementation is distributed, it became necessary to develop a strategy that allowed for the creation and retrieval of sessions for each user. Due to the distribution of the servers, it is neither efficient nor wise to manage sessions by storing them locally on each server. A much more intelligent solution to this problem, which also represents our chosen implementation approach, involves the use of a distributed database in shared memory. This approach allows all instances of the Ktor server to access the database in an extremely efficient

manner. In order to achieve this result, we created the class `EtcdConnector`, which exposes two methods

- `putSession`: using this method, it's possible to add a new entry to the database containing the session of the user authenticated through login.
- `getSessionForUser`: using this method, the calling Ktor server will be able to verify the existence of a specific user's session.

Through this session management, it was possible to distribute the Ktor servers, ensuring that session management was delegated to another technology, separating responsibilities and providing a more agile approach to this problem.

4.2 Client Technologies

4.2.1 ShareDB

In our project, **ShareDB**, a real-time database backend based on *Node.js* plays a pivotal role in facilitating collaborative editing features. **ShareDB** is sedigned to handle concurrent document editing and synchronization, making it an invaluable technology for projects that require real-time collaboration capabilities. At Its core, ShareDB allows *multiple users* to connect to the same document and make changes *simultaneously*, with all modifications being seamlessly merged and synchronized across all user's views. one of ShareDB's standout features is its *Operational Transformation* (OT) capabilities, which enable It to *efficiently* manage conflicts that arise when users edit documents *concurrently*. This library is used inside our *Page Server*. By doing this we were able to introduce the real-time collaboration between multiple users.

4.2.2 How does ShareDB work

ShareDB utilizes *WebSockets* primarily because of WebSockets'ability to facilitate **full-duplex** communication over a single, long-lived connection. This communication model is essential for *real-time* applications, where the *immediate exchange of data* between client and server is crucial for synchronizing document changes as they occur.

`WebSockets` provide a persistent connection between the client and server, allowing for instant data transfer without the need for repeated *HTTP* request-response cycles. This means updates made by one user can be quickly pushed to all other users working on the same document ensuring that everyone sees the most current version of the text.

Here's a close look at how document creation and usage work within ShareDB (Figure 5).

1. **Initial Setup**: To start using ShareDB, a server instance is set up with ShareDB connected to a database. This server will manage all document operations and client connections;

2. **Client Connection**: Clients, such as web applications, connect to the ShareDB server;

3. **Creating a Document**: When a user wants to create a new document in ShareDB, the client sends a request to the server. The server then creates a new document in Its database with an unique identifier;

4. **Share the Document**: Once the document is created, its ID can be shared with other users. Subscribing allows them to receive real-time updates to the document.

4.2.3 Real time web pages

In the process of developing the project, our goal was to integrate **real-time data** capabilities into the client's web pages. This feature aims to eliminate the need for unnecessary page reloads or manual data updates by the user. To achieve this, we created an automatic data retrieval system applicable to all pages, except the text editor that involves other mechanisms. We implemented a polling system within each page to facilitate this. Although we recognize that polling may not be the most efficient method, given our time constraints, it was the chosen approach. This system ensures that pages are always *displaying the most current information*. For instance, if user-A shares a document with user-B, and user-B is currently on their documents page, the shared document will immediately appear in the "shared documents" section without any action required from user-B.

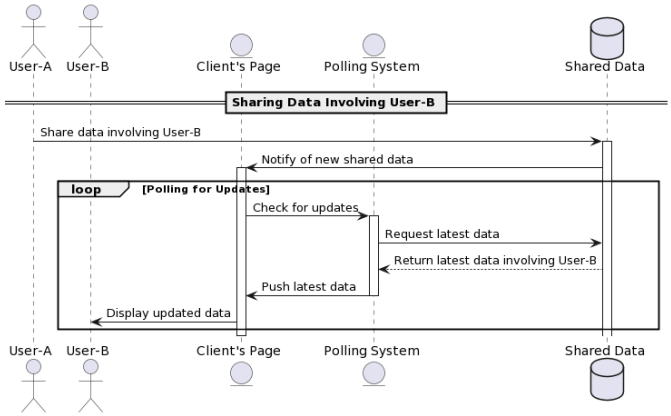

Figure 7: UML sequence diagram of the client page system, showing how the real-time is achived by using the polling.

4.2.4 Real time permissions

During the development of this project, we have committed ourselves to providing the capability to modify user permissions in real-time on a specific document. In detail, the document owner can, through a dedicated page (fig. 12), grant or revoke permissions to each user registered in the system. These permission changes occur immediately. If a user who has access to the shared document is working on it, they might experience one of three possible events:

1. **Complete permission revocation**: should the document owner choose to cancel access for a specific user, they can do so by setting the permission to `NONE`. Simultaneously, the user affected by this change will receive a notification through an alert message, informing him of their access loss to the document, which will automatically close;

2. **Read-only permissions**: if it is desired to assign a user the read permission (`READ`), they will have the ability to view the document's content without making any changes. If the user previously had `WRITE` permission, he will receive a notification through an alert message, which will inform them that from now on, he can only read the document without altering its content;

3. **Editing Permissions**: if a user is granted the write permission (`WRITE`), they will have the ability to make changes to the document without any restrictions, and will also be able to interact with other users who are working on the same file simultaneously. If the user previously had a `READ` permission, he will be notified of their updated status, which will now allow him to edit the document.

5 Self-assessment / Validation

5.1 Git Hook

During the development of our project, we implemented an automatic mechanism that initiates a full code check with every push made, allowing the push only if the compilation of the entire code successfully completes. For this purpose, we utilized **Gradle**, specifically using the `gradle build` command, which enabled us to perform this particular operation. If the code builds and all the tests are successfully executed, the commit will be pushed into the repository.

5.2 Database Tests

We have implemented a series of *unit tests* that run against the database to verify the correctness of **CRUD** operations provided by the system using `Junit`.

6 Deployment Instructions

To initiate the utilization of our software, the preliminary step involves cloning the project repository to your local environment. This action can be achieved through the execution of the following command:

```
git clone https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/ds-project-furi-iorio-vincenzi-ay2324.git
```

Listing 1: Clone the Gitlab repository

Following the successful cloning of the repository, it becomes imperative to navigate into the corresponding directory. Subsequently, one must execute the commands listed below. It's crucial to note that prior to executing the first command, the Docker daemon must be active, and the installed Docker version supports compose and docker swarms:

```
./service.sh up
```

Listing 2: Run the script that creates and setups the backend services

Upon the completion of the aforementioned script, the database and the `DataServer` will be up and running. The next step involves initiating the client through the execution of the script provided here (NodeJS is required for this step):

```
sudo ./start_node_server.sh
```

Listing 3: Start the NodeJS server

Now, the application will be available visiting `http://localhost:8080`.

In the event that there is a necessity to dismantle the Docker images that have been created, the execution of the following command is required:

```
./service.sh down
```

Listing 4: Tear down of backend services

7 Usage Examples

7.1 Login and Account Creation

7.1.1 Accessing the System

When accessing the system, users are prompted to log in with their credentials, including username and password. If they don't have the credentials, they can easily create a new account by clicking on the "Sign Up" button, where they can provide a username and a password. Existing User Login: For returning users, the login process is straightforward. They simply input their username and password, which, upon successful authentication, grants them access to the application's main interface.

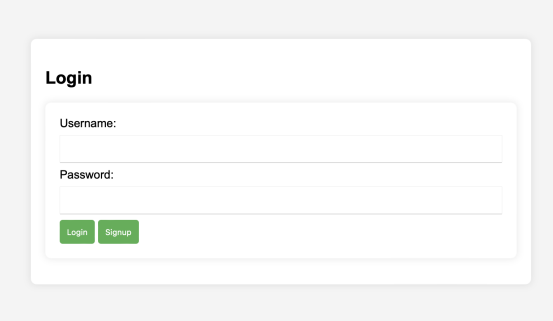

Figure 8: Login page.

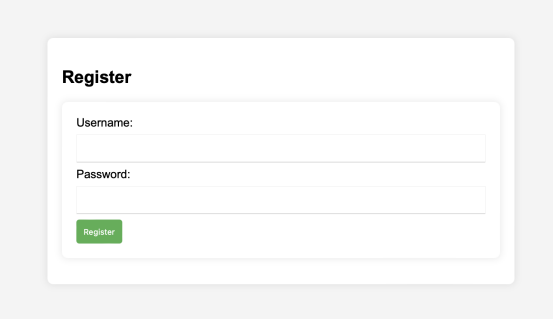

Figure 9: Sign In page.

Upon logging in, users access the application's dashboard the central hub for managing documents and collaboration. At the top of the dashboard, there is a navigation bar that allows different options such as "Signout" and "Add a new document" for initiating new collaborative projects, which takes the user to a new page where he can create a new document specifying the name. Below the navbar, the dashboard is divided into two primary sections: "My Documents" showcasing files authored by the current user, and "Shared Documents" displaying files contributed by others yet accessible for collaboration.

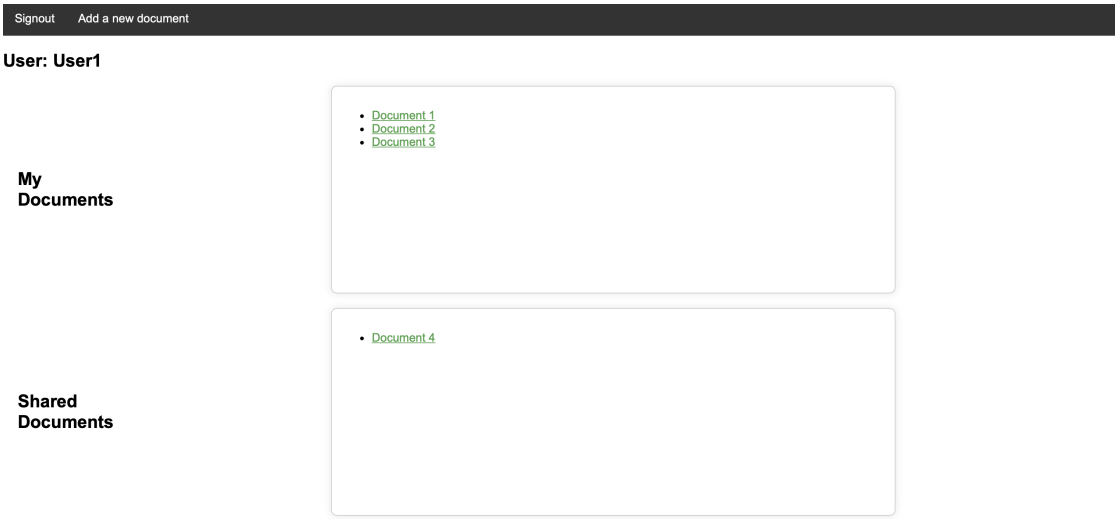

Figure 10: Dashboard page of users.

Navigating through the list of documents, users can open any file with a simple click on the link. Each document opens into the collaborative text editor.

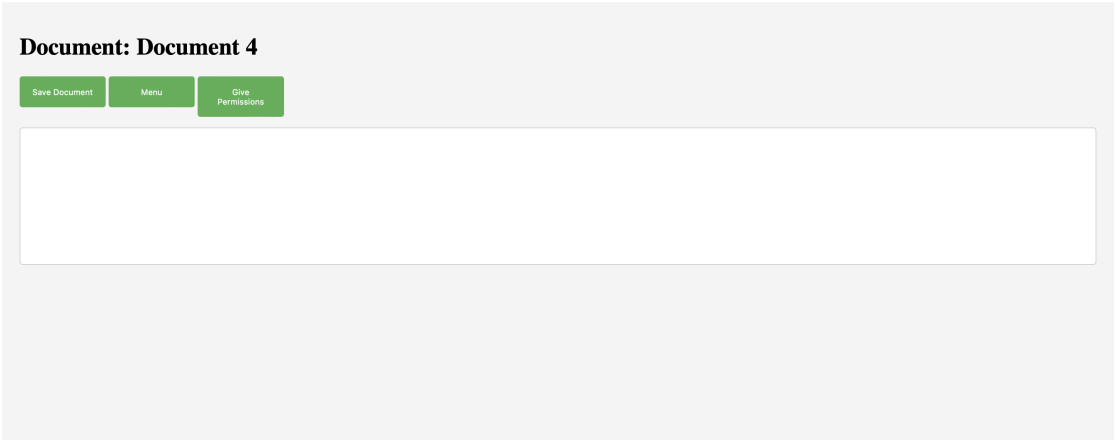

Figure 11: Editor page.

By clicking on "Give Permissions" users can extend collaboration roles by sharing their documents with other users. Specifying the username and the roles He wants to grant to him, such as "Read" or "Write," ensuring controlled access to the document. When documents are shared with a user, they appear in the "Shared Documents" section. Depending on the assigned role, users can either edit collaboratively or simply see the content of a document shared with them.

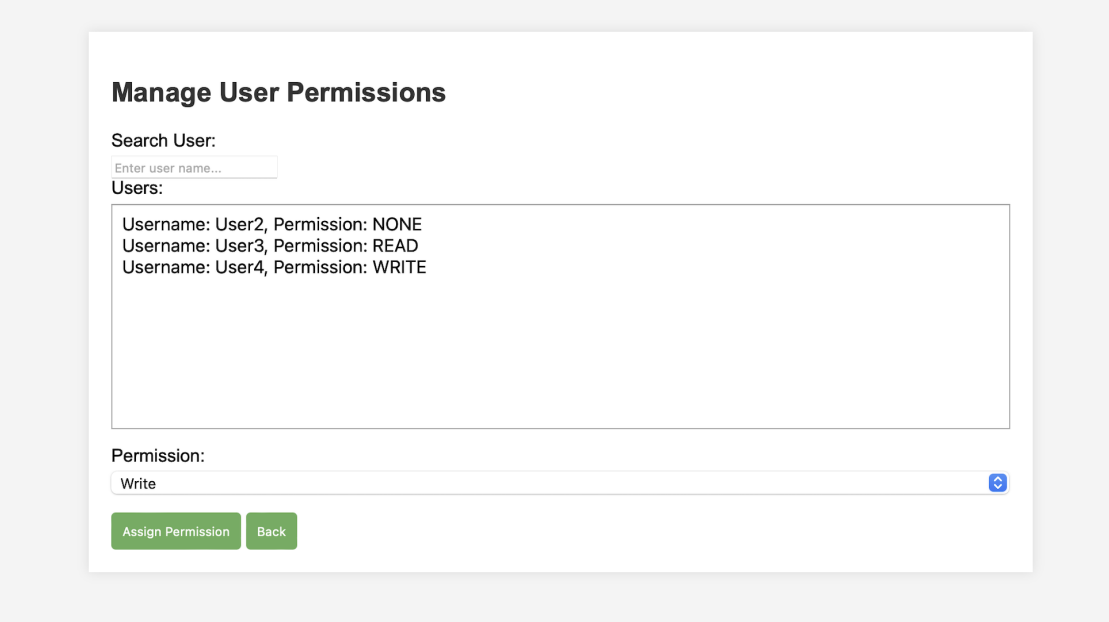

Figure 12: User permissions management page.

8 Conclusions

Our project to develop a collaborative real-time text editor was divided into several stages, summarized below. Throughout the project, we alternated study sessions on the various potential technologies to be used with the implementation of the structures necessary to complete our work.

The project's initiation proved complex, mainly due to the introduction of a mechanism that allowed for real-time document editing. After spending weeks studying various libraries, we achieved exceedingly positive results. Having overcome the challenge of real-time collaboration, we proceeded to develop APIs on the Ktor server and web pages, complemented by JavaScript scripts for the client, to monitor and refine the developed APIs for any errors.

Subsequently, we focused on creating sessions for the client, exploring various approaches to manage this aspect in a distributed context, as one of our primary goals was to create an architecture with distributed Ktor servers. Regarding the distributed storage of sessions, we evaluated several options, ultimately deciding to discard the idea of an inter-server messaging system due to potential overload and unnecessary traffic, opting instead for the use of a distributed database in memory, a solution we found thrilling and particularly effective.

For distribution, we employed Docker Swarm, integrating a load balancer into our architecture to balance the load among the servers. Finally, we implemented a *pre-push hook* that allows accepting user pushes only if the tests are successfully passed. Despite being unable to integrate this code into a `CI` due to technical issues related to the use of Docker containers, we opted for a `CI` solution that executes the `gradle assemble` command, failing and returning an error in case of failure. At the conclusion of this project, we can express great satisfaction for having achieved a significant result in such a substantial endeavor. We succeeded in implementing a real-time text editor, distributing it, and effectively addressing the complex challenges that arose, even in the face of imposed time restrictions.

We are extremely pleased with our decision to personally undertake the implementation of a project that initially seemed to be highly complex—a perception that proved accurate over time. However, through collaboration, we successfully managed to complete this project.

8.1 Future Works

In future developments, it will be crucial to replace the *polling system* with an alternative, shifting towards an event-based architecture. This change will eliminate the need for continuous requests: the server will contact the client only when it is strictly necessary. Implementing this new behavior in JavaScript should not be overly complex, as JavaScript offers constructs to manage and implement this type of behavior. Equally important will be strengthening the security system for managing passwords and documents, adopting encrypted communications during the drafting of documents to ensure their content remains confidential. Furthermore, introducing a password hashing mech-

anism is essential to prevent their storage in a readable form. To address the issue of password salting, a hash function can be used to store them, keeping their digest as a value in the database. As for encrypted communication between clients, the `WebSocket` communication could be revised, attempting to create a mechanism through which the text is encrypted each time.

A significant challenge to address stems from the substantial dependency on NodeJs for the library managing real-time collaborations. The aim is to explore the development of a hybrid architecture, as described in chapter section 2.1.3. In this architecture, clients will play the primary role in exchanging messages and information related to the documents being worked on, thereby reducing the reliance on the NodeJs server and assigning it only the task of providing web pages.

In the future, we also plan to add advanced editor features, enabling the user to format text by choosing a font among multiple available ones, write words in different styles such as bold, italic, and underlined.

Another feature that we would like to introduce to the editor is the possibility of constantly seeing the cursor of all the users who are writing in the same shared document simultaneously.

As a final goal, we plan to adopt a different type of database for our project, due to the need for a system that is highly efficient and quick in *writing operations*. Given the extensive use of the project by users and the fact that most of their activities involve writing operations, transitioning to a *non-relational database* could provide significant benefits in terms of performance and speed, reducing the operational burden and facilitating the execution of core operations.

In essence, these are the main future tasks that need to be carried out on this project.

8.2 What did we learned

During the development of this project, we had the opportunity to delve into various topics, for what concern the *design* of the system and it's realization, we can summarize the main concepts in:

- **How to manage a Distributed System**;
- **How to design a Distributed Architecture**;
- **The technologies involved for the realization of a Distributed System**;
- **How to divide the project in multiple parts in order to separate the different problems**;
- **How to design the communication between the system's components**

Moving to all the technologies used in this project, more in particular:

- **Kotlin and it's libraries (e.g.** `kotlinx.coroutines`**,** `kotlinx.exposed`**)**;
- **ETCD**

- **PostgreSQL**
- **NodeJs and it's libraries**;
- **How to use GitLab for a project**;
- **Gradle**;
- **Docker**;
- **Docker Swarm**

In conclusion, this project has provided us with the opportunity to delve into the creation of a distributed system, with a specific focus on its design and implementation. We have gained the ability to tackle and solve issues related to server distribution and the implementation of mechanisms to facilitate collaboration among various data servers. Moreover, to accomplish this project, we had to learn from scratch the `Kotlin` language and its libraries, as well as the `NodeJs` language, which were essential to complete our work. During development, we also deepened our understanding of `Docker Swarm` software, in order to figure out how to effectively distribute our architecture.

These represent the main learnings obtained from the project, although some minor topics have not been mentioned, as they were considered of lesser importance.