Real-time Text Collaboration Tool

June 5, 2025

Natasja Vitoft **cph-nn194@cphbusiness.dk**

Lasse Baggesgård Hansen **cph-lh479@cphbusiness.dk**

1 Introduction

This project is created to be a proof-of-concept, web-based, real-time text collaboration tool, utilizing the fast query times of Redis, the text agnostic, horizontally scalable persistence of MongoDB and strong relation model of PostgreSQL. It is inspired by tools such as overleaf.com and docs.google.com.

The requirements of the system was for the user to:

- Create, edit and persist files
- Share files with other users with either editor or read-only rights
- Access and edit files in real-time with shared users on localhost
- Create groups of users for bulk-sharing

As of now, the user can create text-based files and edit them with real time persistence, sharing editor or read-only rights with other users individually, or by creating user collaboration groups. When multiple users are editing the same document, they will all receive updates of the document made by other users, as well as their own edits in real-time, without having to explicitly persist edits.

When a user requests access to a file, the content is loaded into a Redis store as key-value pairs for quick updates, and periodically persisted to MongoDB for long term storage while users are editing, making sure that data is not lost on abrupt disconnects, or server faults.

As this is a proof-of-concept we simulate different servers of the system architecture with docker containers running on localhost. Authentication security is no more than a placeholder. User passwords are stored in plaintext and user id's are used in place of authentication tokens.

2 Architecture

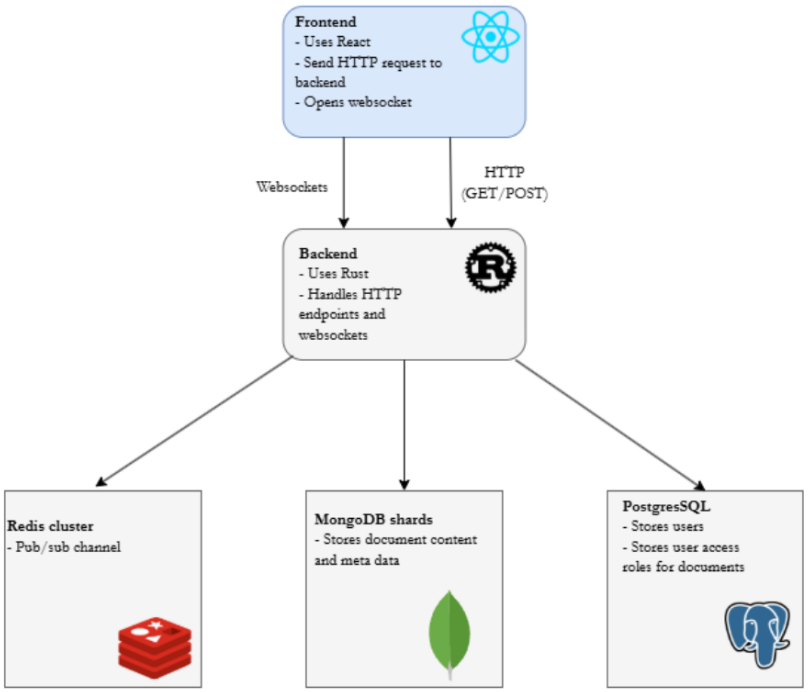

Figure 1: System Architecture

The architecture consists of a browser-based user interface, which connects to an backend API, which is the logical agent interacting with database systems.

- A PostgreSQL RDBMS instance is used for persisting sensitive information like user data for authorization and relations between users, groups and files. This allows us to keep a high degree of consistency in the relations between these, by using constraints to prevent illegal states, like user and relation uniqueness, together with preventing file relations created with non-existent users and groups. The structure consists of five tables. Two of them being entity tables representing users and groups with relevant attributes, and 3 being the entity tables' relations with files and each other. The tables have been designed to fulfill 3NF, for creating as little redundant data as possible.

  Figure 3 shows the database structure

- A MongoDB instance consisting of two database shards handles the actual content and metadata persistence of the files created by users. An entry has 4 fields:

  – id: Auto generated identifier which is related to in PostgreSQL

  – title: Title of the file

  – content: file content

  – format: the specified file format

  This is to ensure scalability of the system by storing the bulk of file content in a system that supports horizontal scaling by sharding, offloading the data bulk of the PostgreSQL instance, which is not tolerant to partitioning in the same way. MongoDB also handles different file formats well by encoding data into BSON, making it resilient to different character encodings. It also decouples created files from anything but their own content and metadata, making the file data more flexible in the architecture. The non-restrictive nature of MongoDB also means we can dynamically expand different kind of metadata on files, making the system resilient for future file type expansions. The API connects to a router, which the distributes queries and inserts within the correct database shards.

- A Redis instance acting as "shared memory" between users. When a user accesses a file, a websocket connection is established between API and client, file content is fetched from long-term persistence (MongoDB), and inserted into a Redis store. A Redis Pub/sub channel is created for the file, and the client is subscribed to the channel. This allows the client to write to the stored content, and receive content updates immediately on edits. The instance is replicated, so as to minimize data loss by server outage. If the primary Redis database goes down, the replica is ready to take over as primary, preventing data loss between persistence flushes.

  Program flow is illustrated in Figure 2

This architecture helps to provide tolerance to partitioning by allowing us to horizontally scale the bulk of content with MoingoDB, while keeping the singly partitioned RDBMS on a low data load. This means we can utilize its strong relational and consistency capabilities, while providing high availability for file content with a temporary Redis store receiving and pushing updates to clients by a websocket connection.

3 Reflection

In this development application we demonstrate the use of of sharding in MongoDB and replicating in Redis, but the setup we use is meant for demonstration of the architecture and not production, as it uses minimal hardware resources, instead of following best practices for setting up clusters and replicas.

As described in the architecture section, we implemented a simple MongoDB setup where we utilized horizontal scaling through the use of shards. However, there are a few considerations we have to take into account with our straightforward setup.

Because we only configured a single-node configurations replica set for each shard, we introduced some potential errors. Each replica set contains only one member, even though it's recommended to have at least three members for production. We chose this minimal setup to simply demonstrate the architecture for development purposes. If it were a production environment, it would be a good idea to introduce at least three members per replica set to ensure fault tolerance. It should always be an odd number to avoid split-brain situations, where the nodes are not synchronized correctly if a node outage happens and need to consolidate again after reconnecting.

For our solution with Redis, we chose a simple master-slave setup, as mentioned earlier. This is also a fairly simple setup where we only have one slave. This is also for demonstration purpose that we do this, but in a production environment, we might want to scale a bit we would have a cluster of at least three primary nodes each with a replica, to ensure horizontal scalability. At the moment we only have one slave, so if that one goes down, then we don't have others to take over. Also, we haven't really taken into account how we will handle a scenario where the Redis master might go down. What we could do here, and what we should do when it gets to production, is ensuring that if the Redis master goes down, one of the Redis slaves should take over. We however chose a simple setup for our solution as we are still in development phase and we can scale eventually.

We tried to keep application logic as much as possible out of the PostgreSQL database, because we believe it is easier to maintain an architecture where business logic is kept in the application, than in for example a procedure or function within the database, which can be hard to test, keep track of in the codebase, and to reproduce if for example reinstallation of the database is necessary. However, we still wanted to utilize the restrictive power of PostgreSQL, by adding constraints to tables that can be run initially from a setup script for recreation if necessary. For example we created a "role" domain for restricting the possible strings inserted for a user role. This makes sure an error will be thrown from the database if an inserted user role is not either an "editor", "reader" or "owner". This will make the RDBMS directly refuse to reach an illegal state, instead of relying only on bug-free application logic.

We use table constraints for enforcing uniqueness of group names, since they are tied directly to a user as owner, as opposed to files which we only store a users relation to in the RDBMS, and instead storing actual file data in MongoDB. This allows users to create files with duplicate names, but also gives flexibility of decoupling files from users, potentially making it possible to redistribute ownership of files to other users, or letting them exist even if the user who owned them initially doesn't exist anymore.

Since we wanted to utilize the speed of Redis, we needed to connect the user as directly as possible without compromising security. Since Redis is not a secure service, we instead managed access by authorizing the user on data from PostgreSQL before setting up a connection to Redis. One of the

biggest challenges of this project, was to efficiently both send and receive updates in Redis. We opted for a solution where we always treat content updates as a capture of the whole content. This is fine for initial development because it is simpler and let's us create testable scenarios, but is not ideal for files with a lot of content. Ideally we would treat updates as positional inserts and deletes, and update database content accordingly, but it almost exclusively required application logic, and we deemed it unnecessary for this project scope.

The file format metadata stored is meant as a possibility to expand the system to supports multiple file formats. As we are storing content in binary encoding, the only thing we need to expand the system to handle a specific file format, is to implement rendering and generation of content in the client application.

4 Conclusion/Summary

All in all, we believe we have achieved our system requirements for the project scope with promising results by the tools implemented.

We believe that the system is tolerant to illegal states and relations by using an RDBMS to manage relationships and constraints between data, and still be tolerant to scaling data loads, by offloading actual bulk data from the RDBMS.

We believe the concept of using Redis as a form of shared memory between users, and keeping the system scalable by using MongoDB as a partition tolerant bulk data store, is a robust way of keeping the data availability high, and still keep the persisted data consistent across shards.

This is a prototype however, so for production use considerations still have to be made:

For safeguarding against data loss, both the database clusters should be setup according to best practices, which is:

- Dividing nodes onto different network connected hardware
- Having an odd number of nodes in a cluster, and at least three
- Having each node consist of a replica set of one primary and two secondary members

These practices will make sure that data is preserved in a server or network fault event, and that reconnection of servers after an outage will not result in inconsistent states or split-brain situations.

5 Appendix

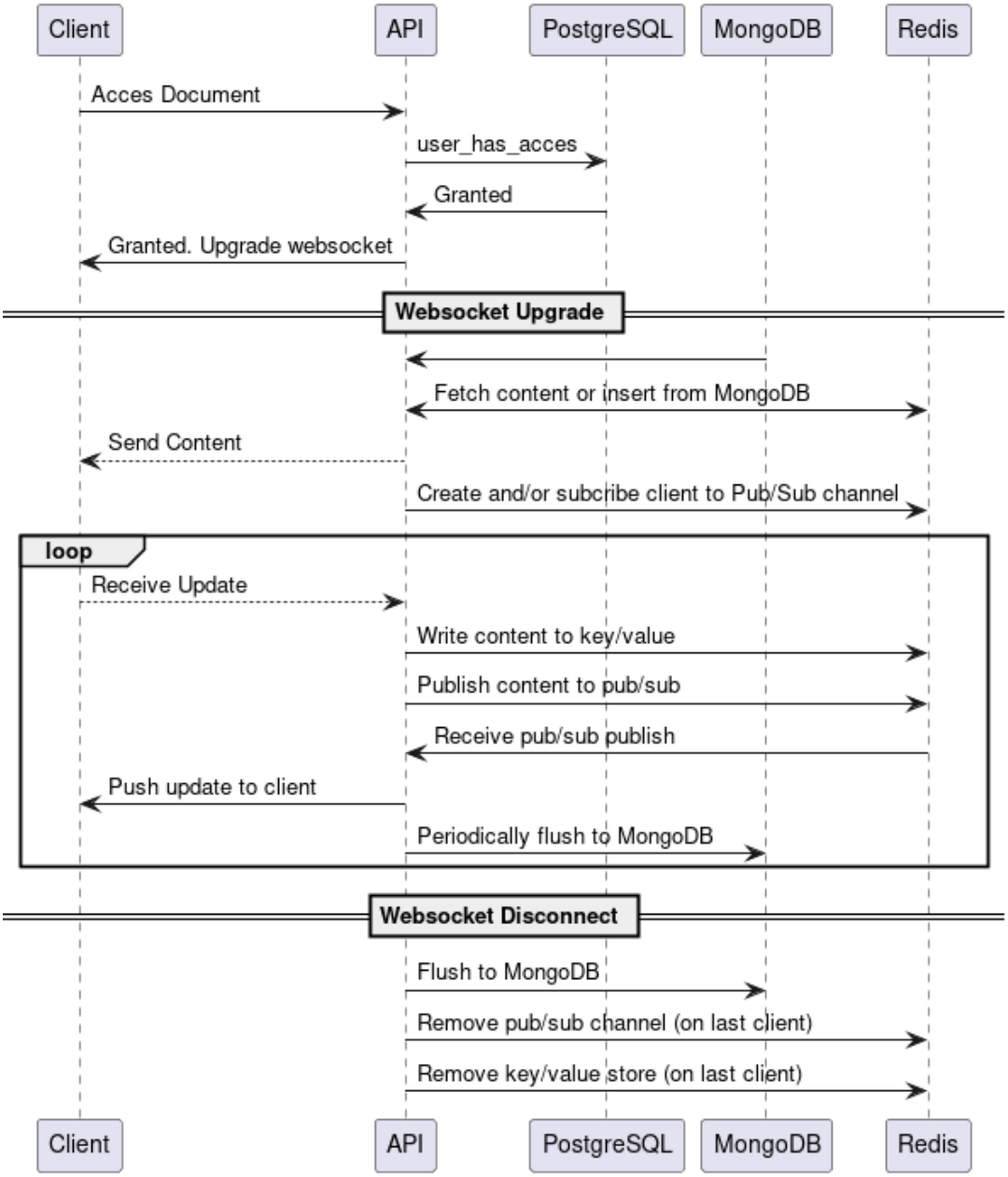

Figure 2: Client accessing and editing file

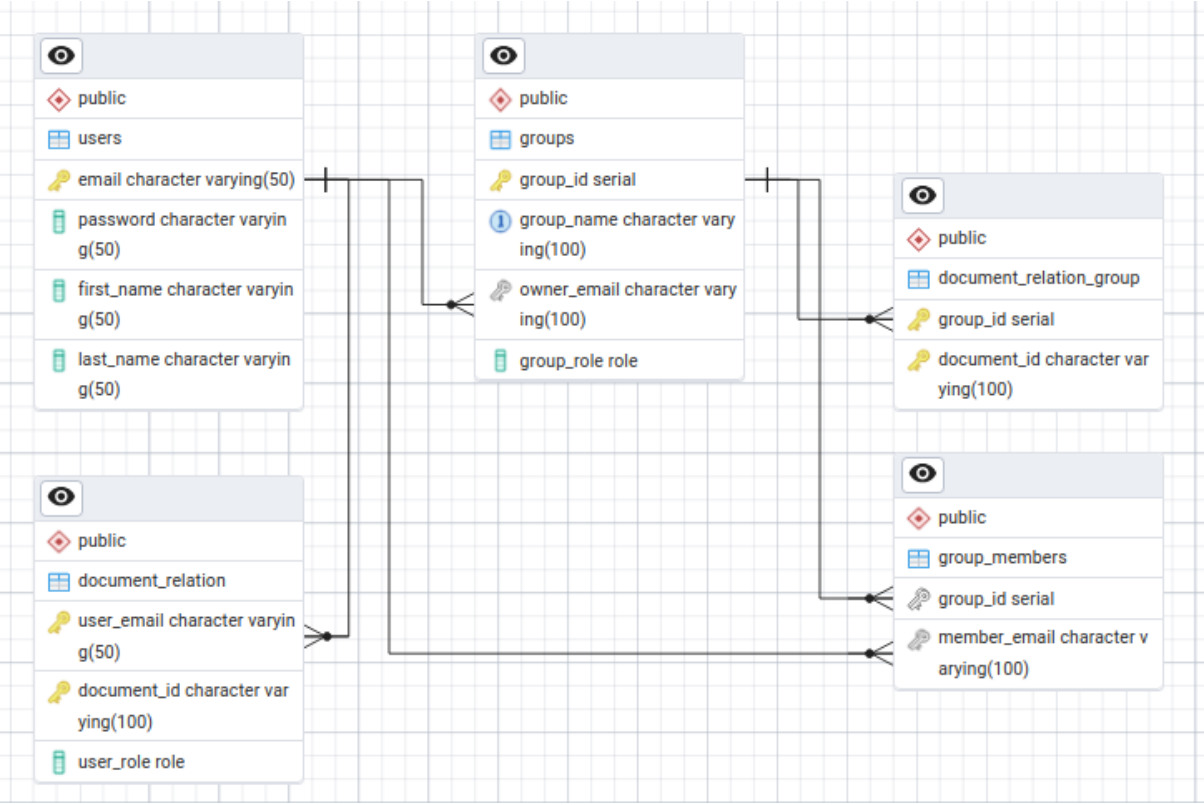

Figure 3: ER diagram