University of Cyprus

FRONT-END INTERFACE FOR A DISTRIBUTED STORAGE SYSTEM USING LARAVEL

Andreas Neofytou

A Thesis

Submitted in Partial Fulfillment of the Requirements for the Master Degree of

Computer Science

at the University of Cyprus

Recommended for Acceptance

by the Department of Computer Science June, 2023

ABSTRACT

This Thesis is about a distributed system application which is able to manage large shared data objects in distributed storage systems (DSS) while increasing the number of concurrent accesses, maintaining high levels of consistency assurance, and assuring smooth operation. So, the backend of the program manages the files using the COBFS framework where the files are fragmented to blocks. The fragments belong to the same object, each file is a linked list of objects that can be covered by blocks. The framework supports typical file operations, such as create, read, write/update, delete or change permission access. The users can concurrently access the same files at the same time by modifying or reading different blocks of the object. The front-end part which is a user interface is what this thesis focuses on. This user interface was created with Laravel, a popular PHP web application framework. Laravel is one of the best options for web developers looking to construct high-quality web applications since it offers reliable and effective tools for creating contemporary, dynamic user interfaces. The application starts with a login page where different users have different accesses (for example administrative access where the user has access to all files). When the user logs in, there are actions like rename, read and download a file while he/she can "drag and drop" or create a file in the directory of the application to have access on it. Also, the users can change password and username. Another, part of the interface is the Database. The database consists of three tables, the files table which has information about files, the users table which has information about each user and the permissions that has all the accesses of users on files. The database is running on MySQL. The interface is designed to be efficient and simple so that it helps users to understand how it works and operates. Lastly, the front-end application is connected to the servers that implement the COBFS framework. The files are displayed to the user through a dashboard of the interface. Overall, it is an application that aims to simplicity, efficiency, operability and concurrency.

APPROVAL PAGE

Master of Science in Computer Science Thesis

FRONT END INTERFACE FOR A DISTRIBUTED STORAGE SYSTEM USING LARAVEL

Presented by

Andreas Neofytou

Research Supervisor _____

Prof. Chryssis Georgiou

Committee Member _____

Prof. Anna Philippou

Committee Member _____

Associate Prof. George Pallis

University of Cyprus June, 2023

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor Prof. Chryssis Georgiou for his continuous support and guidance that he offered me throughout my thesis. Further, I would like to thank Dr. Nicolas Nicolaou and Ms. Andria Trigeorgi that were by my side throughout of the project. I would also like to thank my family and my friends for all for their support.

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

We live in a world where most of the data are digitalized and big data are a very important part in our life. The problem is that we need to manage these data, so *distributed storage systems* [1] and concurrent accesses are needed. Large volumes of data can be stored and managed across several network nodes using distributed storage systems. A distributed storage system's main objective is to ensure data consistency across all network nodes and to offer high availability, fault tolerance, and scalability. A distributed system is not centralized avoiding a single point of failure. Therefore, to let users use this distributed system a user interface is needed. A good user interface should have the following characteristics: it is easy-to-use, user-friendly and predictable for the user, meaning it will do the functionalities as expected from the user. It also, has the main functions the distributed system needs in order to operate correctly. This project focuses on the user interface that is built on the *Laravel framework* [6].

Data consistency [1,2] is one of the main obstacles to designing a distributed storage system. Data consistency is the quality of having the same view of the data at all times across all nodes in the network. Due to network delay, bandwidth restrictions and node failures, it is difficult to provide good consistency in a distributed storage system. As a result, weak consistency models rather than strong consistency are utilized by many distributed storage systems.

Weak consistency allows for eventual consistency, which means that while there may be a slight delay before all nodes have the same view of the data, updates to the data will eventually spread to all nodes in the network. Data inconsistencies may emerge from this delay, although it is frequently a fair trade-off given the scalability and fault tolerance advantages of a distributed storage system.

To this respect, the COBFS framework [1,2] was introduced towards leading to do a DSS with strong consistency and high concurrency guarantees. So, this study is about an interface for a system not to just store data, but also to enable users to easily maintain, update, delete and download files (data), avoiding problems like race conditions or crashes when multiple accesses happen to a file. The solution to these problems is a distributed storage system using COBFS, which manages huge items by using a block fragmentation method. The main focus of this study is the development of front-end prototype for providing an easy-to-use user interface for COBFS. The main focus was on creating a user interface that is effective and user friendly and lets the users work fast and learn quickly the capabilities of the system. Figure 1.1 shows the dashboard of the system which is one of the main pages of the user interface.

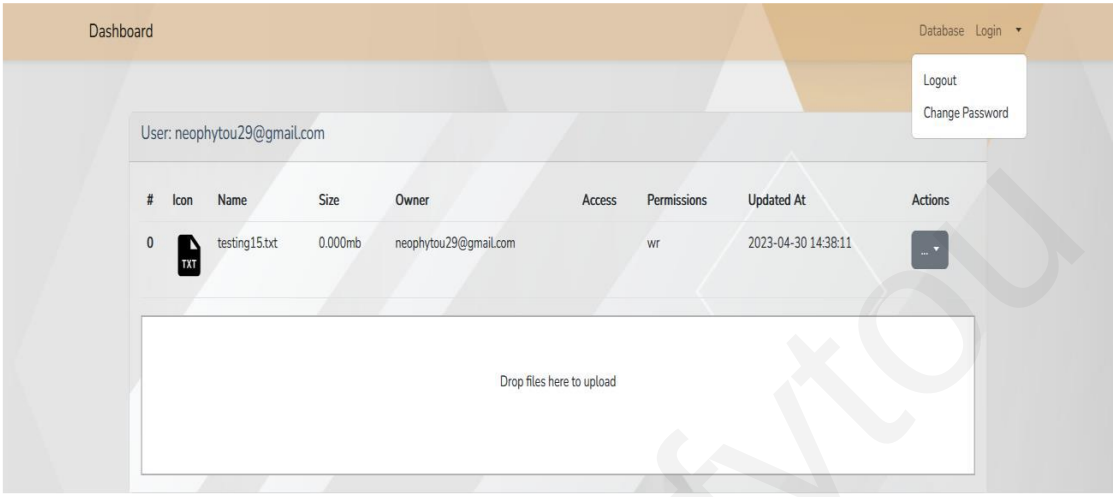

Figure 1.1 Dashboard of User Interface

1.2 Development Methodology

The study was initiated by becoming aware of the concept of the Distributed Storage System and of the COBFS framework. We were then given the assignment of building an intuitive, simple-to-use user interface using the Laravel framework.

Subsequently, we started to read and investigate the research of COBFS and the capabilities of distributed storage systems and before starting the project. Then, we studied research papers that we were given [1,2]. This made it easier to understand the capabilities and features that are needed to be built into the user interface.

We started working on the user interface once we had a better knowledge of the system. Then, the next step was to understand what is Laravel. So, we started learning how Laravel works and what exactly are the capabilities of this framework. Laravel is an open-source PHP web framework that is intended for the development of web applications. In other words, it is a framework that helps developers build websites for systems. The goal is to provide an administrator control panel and dashboard for all users that would make it simple for users to manage their files and other operations that are needed from the distributed storage system. The user interface was created to be simple to use and straightforward, including capabilities for routine tasks like changing passwords and read/write operations on files as well as obvious navigation.

To make sure the user interface satisfied COBFS, we routinely communicated with the research team throughout the project. To enhance the user experience, suggestions were included by the university researcher into the design. The user interface was also routinely tested to make sure it worked properly without getting any errors and crashes.

After implementing the dashboard, we started to work on the API integration as the project advanced. The Distributed Storage System utilizing COBFS was built to operate with APIs created using Laravel. These APIs include some functionalities for example, getting files which returns all the files information of a particular user. To make sure they worked properly and offered the user-required functionalities, these APIs underwent testing. The research team proved after testing that APIs would

enable the interface to be integrated to the COBFS system. We kept working on the user interface and APIs as the project came to a close, improving their functionality and design to make sure it complied with all requirements.

The outcome in the end was a user interface and API system that allowed COBFS to manage files. The APIs offered the essential capability for managing files and carrying out typical activities like renaming files and requesting user tokens, while the user interface was created to be simple to use and traverse.

1.3 Related work

As more data is created and has to be kept securely and effectively, distributed storage systems (DSS) are becoming more common. However, maintaining data in these systems can be difficult, especially for non-technical users. To solve this issue, user interfaces (UI) are required to let people to interact with the DSS. A study by M. I. Ali and M. D. Assaf [13], examines previous work on UI for DSS, concentrating on two major areas: UI design principles and UI features. These principles and features of the related work helped us to understand how our study should start and have a better view for what we should expect from our UI that is implemented in this study.

The article begins by going through the design guidelines that should be followed while creating a UI for DSS. Usability, learnability, adaptability, consistency, and beauty are examples of Distributed Storage System UI. Usability relates to how easily users can complete tasks using the UI, whereas learnability refers to how easily users can learn to use the UI. The capacity of the UI to adapt to varied user demands is referred to as flexibility, whereas consistency guarantees that the UI is predictable and recognizable to users. Finally, aesthetics relates to the UI's visual attractiveness, which might influence users' impressions of its usability and efficacy. All of these are very important for this kind of UIs and we were influenced by these guidelines as we tried to build the UI on Laravel to maintain all these design guidelines.

The study finishes by outlining future research areas in the field of UI for DSS. These include incorporating natural language processing and machine learning into UI design, as well as creating new UI elements that can assist users in better understanding and managing their data in DSS. Furthermore, the article emphasizes the importance of additional research on the usability and efficacy of UI for DSS, as well as the establishment of defined criteria for measuring UI performance.

Overall, this article presents a thorough analysis of the relevant work on UI for DSS, stressing the main design concepts and aspects to consider when creating an effective UI for these systems. The report offers significant insights for DSS academics and practitioners, as well as consumers who must engage with these systems on a daily basis. [13] This article was a great choice that helped us understand the ideology for our study. This article was very helpful and enabled us to understand all of the ideology that our UI needed to be implemented. All the examples of UIs that this related work includes, helped us to build our system (usability, learnability, flexibility, consistency and attractiveness)

1.4 Document Organization

In Chapter 2 we overview of what is a Distributed System and explain what is a Distributed Storage System in more detail. Also, emphasis is given on atomicity and why it is important for DSS.

Chapter 3 overviews the COBFS framework. It analyses the framework and explains basic information that help the reader to understand the framework better.

Chapter 4 gives all the interfaces' specifications in detail. Also, it explains all of the components of Laravel that were used to build successfully the User Interface of this study.

In Chapter 5, all of the pages of the User interface are analyzed in detail. This lets the reader understand all of the main UI functionalities and gives a good starting point for anyone that reads this study to be able to use in an effective way the User Interface.

In the last chapter, Chapter 6, some conclusions are given for this study, as identified during the implementation of the User Interface as well as some difficulties that we faced and how we managed to overcome them. Lastly, there is a section of future work with suggestions on how this work can be continued and expanded in the future.

Chapter 2

An Overview of Distributed Storage Systems

2.1 Distributed Systems

In a *distributed system* [1,2], different components are dispersed across a number of computers (or other computing devices) connected to a network. A system is decentralized if its components are found in different areas, with no or limited coordination. In other words, a distributed system is a collection of independed components that work together and appear to users as one coherent system. These devices divide up the labor and coordinated their efforts to do the task more quickly than if only one device had been in charge of it. Also, this kind of systems avoid the single point of failure where if a device fails the system will keep working avoiding the problem where the system will stop. This provides high availability or reliability to any system that is distributed.

There are a lot of objectives and principles of distributed systems, the most important are Transparency, Heterogeneity, Openness, Scalability, Performance, Dependability. Transparency is when hiding the distribution from users and making the system look like a single system. The different types of Transparency are:

- Access: gain unified access to resources.
- Location: conceal the location of a resource.
- Migration: is the movement of resources without affecting access.
- Relocation: is the movement of resources when they are used.
- Replication: Hide the fact that the resource has several copies. Copies of a resource must have the same name.
- Concurrency: concurrent access to a resource by competing users.
- Failure: Hiding a resource's failure and recovery.

Heterogeneity is when connecting users and resources like remote resource access and controlled resource sharing. Other components of Heterogeneity are variety and diversity which include communication network, material, operating systems, programming languages and databases and Implementations from various software development actors.

Openness is the ability that enables collaboration across implementations from various vendors operating on various platforms (the distributed systems have to be scalable). Scalability is the principle that allows the distributed system to be scalable if, following a significant growth in the systems' resources and users, the system remains efficient and effective.

Performance is the characteristic of the distributed system that aims for the system to be fast on processing process.

Dependability has three main objectives fault-tolerance, availability and security. All the systems have faults, but a distributed system should have the ability to troubleshoot and fix faults or hide them in order for the system to work properly.

Availability is the probability that the system will function successfully whenever it is accessible to service its users and security refers to the securities of information and data.

2.2 Distributed Storage System

A *distributed storage system* [4] is a distributed system that allows data to be dispersed over numerous physical servers or devices, and typically across different data centers. Distributed storage system is a software-defined storage solution that allows access to data when, where, and with whom someone chooses. Distributed storage system is a logical volume management system that is designed to handle scalability and data access in a HA (High Availability) environment while also detecting and responding to faults and cyber threats.

Massively scalable cloud storage systems like Amazon S3 and Microsoft Azure Blob Storage, as well as on-premise distributed storage systems like Cloudian Hyperstore, are built on distributed storage [4].

Several types of data can be stored in distributed storage systems:

- **Files:** a distributed storage system enables devices to mount a virtual drive, with the real files spread over several workstations.
- **Data:** is stored in volumes known as blocks in a block storage system.
- **Objects:** a distributed object storage system encapsulates data in objects, each of which is recognized by a unique ID or hash.

There are various advantages of using distributed storage systems (some of them have already been explained in Section 2.1):

- **Scalability:** the fundamental reason for spreading storage is to grow horizontally, providing additional storage capacity to the cluster by adding more storage nodes.
- **Redundancy:** for high availability, backup, and disaster recovery, distributed storage systems can store several copies of the same data.

- **Cost:** distributed storage allows for the use of less expensive, commodity hardware to store huge amounts of data at a low cost.
- **Performance**: In some cases, distributed storage can outperform a single server because it can store data closer to its customers or enable massively parallel access to enormous files.

- **Fault tolerance:** is the ability to keep data available even when one or more nodes in a distributed storage cluster fail.

Some of the main challenges of distributed system are Consistency, Availability, and Partition Tolerance. Partition Tolerance is the capacity to recover from a partition failure comprising a portion of the data. Consistency is when every node or user has the same view of the data at any given time, regardless of which client has modified the data [2]. Availability is the probability that the system is working properly at any given time and it is available to serve its users. As mentioned, there are many examples of distributed systems. In this study COBFS is considered, which is presented in the next chapter.

2.3 Atomicity

An important consistency property in distributed storage systems is *atomicity or linearizability* [1,2]. Atomicity is when every operation in an object seems to happen sometime between its invocation and its return. Atomicity is a solution where all operations are executed in a way as if executed on a single machine, strong consistency. *Coverability* [1] is a consistency condition that extends linearizability and concerns versioned objects. A coverable (versioned) object implementation is an object which the read operation returns both the version and the value of the object. Writes, on the other hand, try to write a "versioned" value to the object. If the reported version is older than the most recent, the write is ignored and transformed into a read operation, preventing the object from being overwritten by a newer version.

Even in the face of complex processes and erratic network conditions, a distributed storage system may offer strong assurances of consistency and dependability by assuring atomicity. This is crucial for applications that depend on the availability and integrity of data, such as those that deal with financial transactions, medical information, or data used in scientific research.

Atomicity access conditions:

- A function reading p of an object x returns either the value of x's most recent complete writing or the value of a writing that is concurrent with p (but does not have to be finished).
- If a read function p1 reads the value written by a write function g1 and a read function p2 reads the value written by a write function g2 and $p1 \neq p2$, then $g2 \rightarrow g1$.

Examples of atomicity:

Figure 2.1 shows two examples based on the atomicity conditions. The first example, shows a read operation that is concurrent with a write operation. The write operation is trying to write the value ‘8’. Based on the conditions the read can either return the number ‘8’ or ‘0’. It correctly returns ‘0’ which is the most recent complete writing. The next read operation returns the value ‘8’ which is the latest value of a completed write operation.

The second example in Figure 2.1, shows a write operation that is trying to write the value ‘8’ and two read operations, the one happening after the other, that are concurrent with the write operation. Both of the read operations, return the value ‘8’, satisfying the condition that a read operation can return the value of a writing operation which is concurrent with a read.

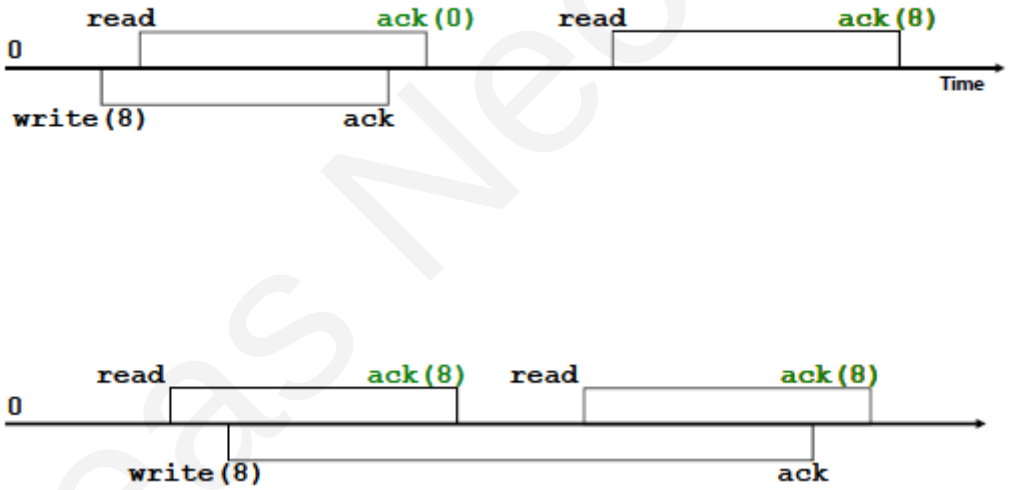

Figure 2.1 Atomicity Read/Write

2.4 Challenges of Distributed Memory

By using distributed systems some challenges have to be solved. One important is the memory management. With the correct memory management, it is ensured that the system will give correct data and work correctly to satisfy the requirements of the user. But, in order to have memory consistency some extra information are needed when the data are uploaded. These data specify when the data was updated, by who and what operation happened (write or delete).

Chapter 3

The COBFS Framework

3.1 Fragmented Objects

A concurrent object made out of a finite list of blocks is referred to as a *fragmented object* [1] since it can be accessed concurrently by many processes. Two clients can have access to different blocks at the same time of the fragment object, promoting concurrency. In the next pages we overview algorithms that implement fragmented objects (e.g. files) leading to robust DSS.

3.2 The COBFS Algorithm

COBFS (Consistent Object-Based File System) was introduced in [2] as a framework to manage large-scale, high-performance, and fault-tolerant storage in a distributed environment. COBFS basically relies on a block fragmentation technique to handle large fragmented objects. The object-based system fragmented architecture, on which the method is based, enables the system to manage files as fragmented objects and to store and retrieve them independently of one another.

The COBFS method employs a distributed metadata management strategy, with a piece of the metadata being stored on each system node. This makes it possible to update and access metadata quickly, and it also lessens the effect that metadata operations have on system performance as a whole.

The replication technique used by the COBFS algorithm ensures consistency and fault tolerance. Updates are distributed to all replicas to preserve consistency, and each file object is replicated over several system nodes. The system can continue to function and fulfil file requests even if a node fails or becomes unavailable by utilising the replicas that are still available.

When editing files, it is assumed that a value change would add to the object's current value. In such circumstances, a writer should be informed of the object's most recent value (i.e., via reading the object) before altering it. To keep this attribute coverable linearizable blocks have to be used. Coverability improves linearizability by ensuring that object writes succeed when linking the written value with the object's "current" version. In a different instance, a write action becomes a read operation and returns the object's newest version and associated value [2].

The file system provides fractured coverability as a consistency guarantee by leveraging coverable blocks. In the implementation, the underlying theoretical formulation enables for this implementation to be extended to handle many types of huge object [2].

The basic architecture appears in Figure 3.1 which shows the fundamental architecture of COBFS. COBFS is made up of two primary modules: Fragmentation Module (FM) and Distributed Shared Memory Module (DSMM). The FM implements the fragmented object, whereas the DSMM defines an interface to a shared memory service that enables read/write operations on individual block objects. Following this design, clients may access the file system via the FM, while servers keep the blocks of each file via the DSMM. The FM writes and reads blocks to shared memory using the DSMM as an external service. COBFS is adaptable enough to use any underlying distributed shared object technique in this regard.

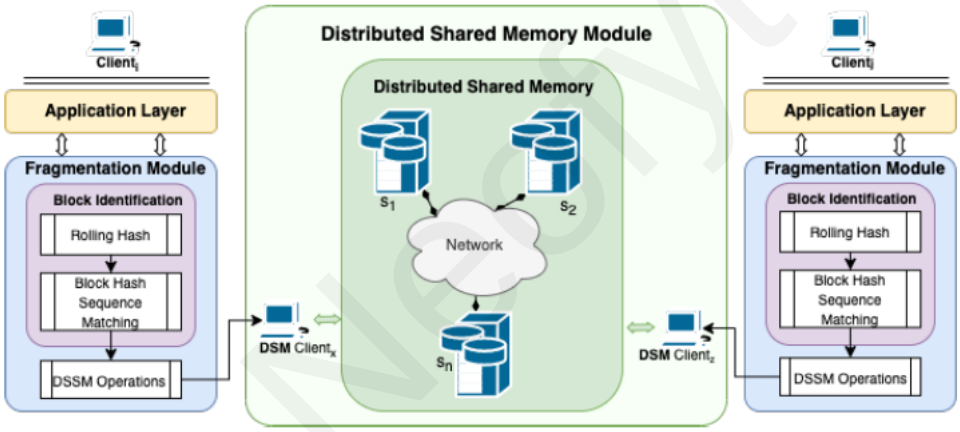

Figure 3.1 Basic architecture of COBFS [2].

In general, the COBFS algorithm offers distributed systems a highly scalable, fault-tolerant, and high-performance storage solution. It is an effective distributed storage system because it makes use of object-based file storage, distributed metadata management, replication, caching and prefetching.

3.3 COARESF Algorithm

In a research paper [3], COBFS is enhanced to provide COARESF algorithm, which is a dynamic consistent storage system ideal for large items. The COBFS framework was expanded to support *reconfiguration* [1].

Configuration [1]: is a set of servers maintaining the storage.

Reconfiguration: is the process of changing the set of servers that hold the object replicas. In other words, it is the transition of one configuration to another.

In order to expand COBFS the developers had to interact COBFS with ARES. When a reconfiguration operation is invoked in ARES, a client requests that the configuration of the servers hosting the single R/W object be changed. When dealing with a file (fragmented object) f that is made up of numerous blocks, the fragmentation manager tries to introduce the new configuration for each block in f. COARES, performs a dsmm-reconfiguration operation which is for each block in f. Concurrent write operations may result in the addition of additional blocks to the same file.

Chapter 4

Program Specifications

4.1 Why a Good Interface is Needed for COBFS

For COBFS, an interface is essential for various reasons [6]:

Usability: Users can interact with the system more easily when the interface is good. The intuitive user interface makes it simple for users to obtain the capabilities they require, navigate the system, and carry out their duties quickly. This enhances usability and increases the system's usability for a broader range of users.

Efficiency: COBFS can be made substantially more efficient with a decent interface. Users may search the files they need, upload and download files, and carry out other actions with the help of an easy and user-friendly interface. By reducing errors and saving time, this can enhance production.

Precision: The user interface can help to guarantee that users interact with the system appropriately and that data is entered precisely. The user interface that can walk users through the required procedures and verifies their input can lower the chance of errors and guarantee correct and consistent data.

Scalability: The interface can also increase the scalability of COBFS. The system can handle more users and data with an interface that is built to handle high traffic and big volumes of data without sluggishness or instability.

For COBFS, an easy-to-use interface is crucial ensuring usability, effectiveness, correctness, and scalability. The user interface can increase productivity, decrease errors, and allow the system to accommodate larger amounts of data and users by making it simpler for users to engage with the system. A straight forward user interface is vital because it helps users to navigate and use the system or program with little misunderstanding or irritation. An intuitive layout and simple navigation should be provided in a user-friendly interface in order to maximize user happiness and productivity. A simple user interface also makes it simpler for new users to grasp how to use the system and access its capabilities, which lowers the learning curve for them. Furthermore, a clear and concise interface can help to lower the possibility of user errors and mistakes, ensuring that users can complete their tasks with the least amount of effort and the highest level of accuracy.

4.2 User Interface Specifications for COBFS

4.2.1 General Specifications of the System

The User interface that is created has many features that are needed in order for the users to keep and maintain track of their new and old files. The interface has the ability to store and let the users have access on their files based on permissions. The main functionality of the system is the dashboard that includes adding new files, delete existing ones, read and download files, manage user roles and permissions, and download files. All of these can happen in real time and the system has the ability to receive post and get requests from COBFS without needing to login in the system with the use of APIs. Also, the system is straight forward to help a new user to create an account and login to his/her new account.

For example, when a user visits the register page and enters the email account, then, the system will prompt the user to verify the account through his/her email inbox. There are two kinds of users, the administrator and the normal user. The administrator has more access to the system and can change a lot of information that can influence all of the users, i.e. like changing the database tables where all of the accounts and file information is saved. When a user logs in the system will prompt him/her to the dashboard where all the files of the user are there. The user can change/read the existing files or add new files from the drag and drop box. The files that are shown to a user are the files that the user has access to based on the permissions. The permissions of each file can be changed only from the owner of the file. The owner of the file is the user that created the file by uploading it to the user interface.

4.2.2 System Requirements and Specifications

The requirements of the system are very important and many meetings with the team researchers were needed in order to specify all the functional requirements of the system. The functional requirements are for user roles that were considered in the study. The main requirements for the system are:

- **User account:** Create an account, write new permissions, delete a password by replacing it with new, rename username.
- **User files:** Create, write, delete, rename, upload new files, change permissions.
- **Database Table Records Management:** Create records, edit records, delete records, add new permissions.
- **Admin Access:** dashboard access, database access where all the tables are shown.

4.2.3 System User Specifications

It is critical to accurately specify the system's users while defining them. To ensure that the system is created appropriately, it is critical to consider aspects such as the users' educational level and the duties they will have in the system.

The two user roles are:

- Normal User
- Administrator

Normal Users

Users can create new accounts and assign roles to other users or themselves using this user interface. With administrators having more control over the database and system settings, this enables different levels of access to the system.

Users have the ability to create new files, add material to them, remove or change already existing files after logging in. The system is built to disperse the storage of the files, making them available from different servers and reducing the possibility of data loss.

The ability for users to manage file permissions is a crucial component of your user interface. To regulate who can view, update, or delete a file, users who have access to a particular file can grant or revoke access to other users. This aids in preserving the safety and privacy of the system's stored files.

Administrator

The administrator has more rights than the normal user. The administrator has access to capabilities like altering user roles and changing the database. This enables more control and customization over the system, ensuring that it satisfies the requirements of the business or person utilizing it.

Each role has its own set of rights, however each user role may have distinct privileges like the permissions on different files or accesses to different files. All the roles are important because they are designed to fulfill their purpose.

4.3 System Architecture

The client-server architecture (Client-Server) is used in this system. In this design, the client asks information or an action from the server (COBFS), which then utilizes the database to carry out the requested action. The client then analyses the server's response to show the results on the client's screen. Figure 4.1 shows exactly how this architecture works.

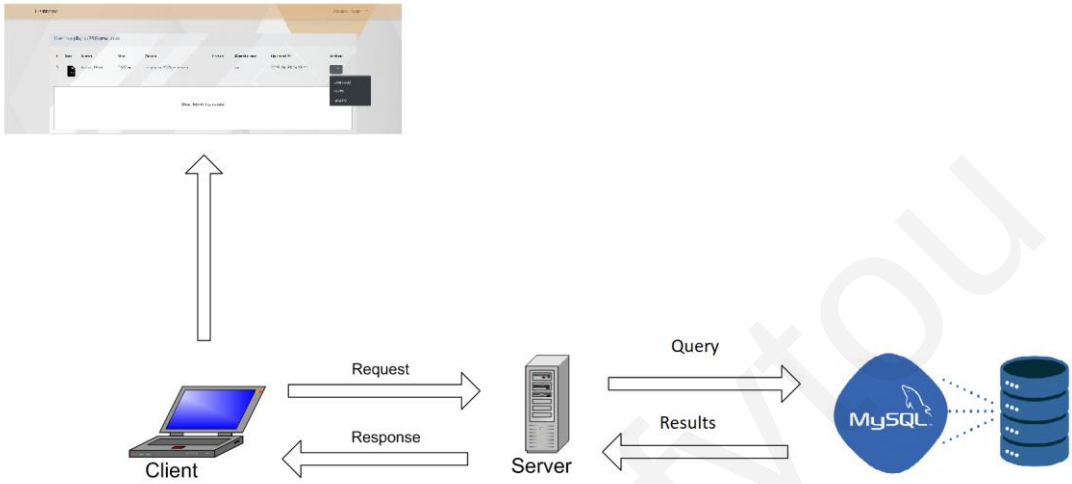

Figure 4.1 System Architecture

4.4 Laravel Framework as a Tool to Implement the User Interface

Building web applications, including distributed storage systems, may be immensely helpful when using the well-liked and potent PHP framework Laravel [6]. The following are some components of Laravel that assisted this study in designing the user interface for COBFS [6]:

Blade templating engine: Laravel has an effective templating engine called Blade that makes it simple to construct reusable templates for your user interface. Thus, Someone, can create a consistent and eye-catching user interface for a system rapidly with Blade. This was used in our study by creating a lot of blades (views) that worked as the main pages of the user interface. For example, the dashboard view of the user interface.

Eloquent ORM: Eloquent ORM from Laravel offers a simple and intuitive approach to interact with the database. Eloquent makes it simpler to access and store data from a database, which makes it simpler to create dynamic user interfaces that react to user input. For example, accessing the database a lot of times in the Page Controller code and with a simple one-line code someone we could retrieve and store newly added data to the database tables of the user interface.

Routing: The routing system in Laravel offers a versatile means of handling user requests and directing them to the proper controller method. For this user interface the Web.php file is used that helped to create user-friendly URLs for the system (get and post routes), which enhance the user experience.

Artisan CLI: The Artisan command-line interface (CLI) from Laravel offers a robust collection of tools for controlling the application. It helped in this study to give the ability to produce code, carry out database migrations, and do other operations with Artisan that assisted in developing a better user experience. For example, when wanting to delete everything from the database and start fresh, the command ‘php

artisan migrate: refresh' is called that basically recreates the database tables. Also, for this study it was used to create new tables that assisted in the user interface.

Community Support: Laravel has a sizable and vibrant developer community that contributes to the framework and offers assistance through forums, blogs, and other platforms. When creating a user interface, this can be incredibly beneficial because you can learn from the mistakes of others and come up with fixes for problems that are frequently encountered. Very useful websites were [7] and [8] that both helped to overcome a lot of issues and errors or even new features that were included in the interface. For anyone using the Laravel framework for web development, community support is a crucial component. There is a sizable and vibrant community of Laravel developers who work on the framework, support it through forums, blogs, and other platforms, and impart their wisdom to others. For developers who are new to Laravel or are having difficulties with their projects, this community support can be immensely helpful. Developers who have access to a helpful community can get answers to their questions, discover new methods and best practices, and immediately get assistance with problems. This can help them prevent errors and save time, which will result in quicker and more effective development initiatives.

Security: In terms of security Laravel is emphasizing on offering robust security features right out of the box. Laravel is widely recognized as a very secure framework that helped us protect the interface from many possible attacks. Below, the most important features of security that were used in this study are explained. Among Laravel's primary security features are the following that our research team used them in this study. The way these features are used is explained below:

- Sensitive information is safeguarded both in transit and at rest thanks to Laravel's robust encryption capabilities for user data.

- Secure password hashing: By default, Laravel employs the bcrypt algorithm, which is regarded as one of the safest password hashing techniques available. When creating a user, the bcrypt function is used in order to hash the password of each user. With this way all the user's passwords are protected by encryption in the user interface.

- Cross-site request forgery (CSRF) protection: Laravel has built-in CSRF defense that aids in thwarting malicious attempts that try to forge requests to the application. For example, all the post requests have to be authenticated before our interface executes that post request, otherwise it blocks that specific post request.

- Preventing SQL injection attacks via prepared statements and parameter binding is a feature of Laravel. This kind of attacks are prevented by allowing only the administrator having access to the MySQL Database.

- The built-in authentication and authorization features offered by Laravel make it simple to create secure user authentication and access management. A user has to log in our interface to access important data (files).

- Two-factor authentication (2FA): Two-factor authentication, which adds an extra layer of protection to user accounts, is also included into Laravel. All the user accounts have to be authenticated and verified by the email of the user.

- Automated security updates are provided by Laravel, helping to keep the framework up to current with the most recent security patches and fixes. So, with this way the interface will be up to date in terms of security.

In conclusion, Laravel offers a variety of features and tools that assisted in designing the user interface for COBFS. A user can create a new account in the interface that is both aesthetically pleasing and simple to use with Laravel's aid thanks to its template engine, ORM, routing system, CLI, and community support.

4.5 MVC

The MVC (Model-View-Controller) [9] architectural pattern is used in the Laravel framework to divide application logic into Models, Views, and Controllers. Because of the separation of concerns, the program can be maintained more easily in the long run and has superior scalability and code structure. Here is an explanation of each component's duties:

Views (Blades): The Views component is in charge of showing the user the application's user interface. Views in Laravel are usually created using the Blade templating engine. Views should be as straightforward and devoid of business logic as feasible. Views receive data from Controllers and organize and visually appealingly display it to the user. Figure 4.2 shows all the views of the interface that we have built.

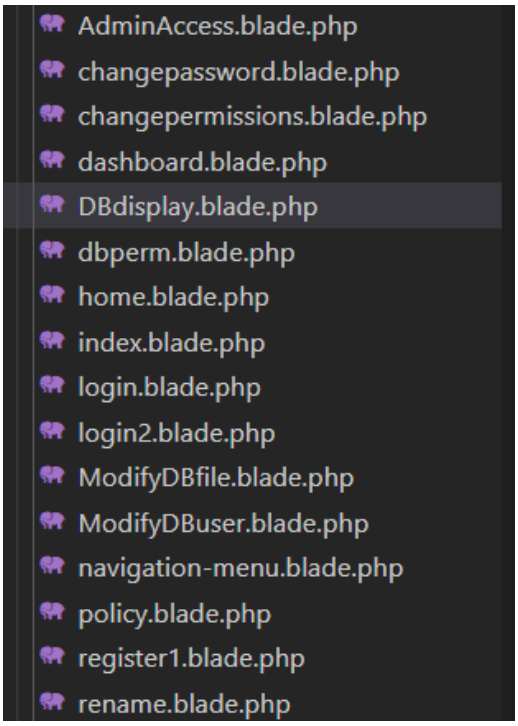

Figure 4.2 Views of User Interface

Layout Blades: Layout blades are views that are crucial elements in Laravel that make it simpler to develop reusable views and templates. An HTML page's layout is its basic structure and includes a header, footer, and other widely used components. On the other side, blades are the templates that have dynamic content injected into them.

Blade templates can be used to construct layouts in Laravel. With the use of blade templates, developers may create clear, simple, and reusable code. Blade templates' main advantage is that they let programmers isolate presentation logic from application logic. As the application develops, this makes it simpler to maintain and adjust the layout.

Blade templates provide a straightforward syntax that is simple to learn and comprehend. Curly braces are used to indicate variables, and the @ sign is used to indicate directives. Developers can incorporate conditional logic, loops, and other programming elements into a template by using directives, which are control structures.

The @extends directive can be used to extend blade templates. Developers can specify a parent template that contains the page's fundamental structure and child templates that describe the content of the page using the @extends directive. Using the @section directive, child templates can take sections from the parent template.

The @include directive can also be used to add blade templates. Developers can reuse common HTML components in various templates by using the @include directive. This makes it simpler to maintain an application-wide look and feel that is consistent.

In conclusion, the two crucial elements of Laravel that enable developers to produce reusable views and templates are layouts and blades. Developers may more easily maintain and adjust the layout as the application matures by separating the presentation logic from the application code using Blade templates. Developers can construct dynamic, responsive web pages with Blade templates that are simple to read, edit, and manage.

Below, is the app.blade (Figure 4.3) which is a layout view in the Dashboard and in the header of the Dashboard appear buttons like Logout, change password, Database and even a refresh button called Dashboard that reloads the page of the dashboard (Figure 4.4).

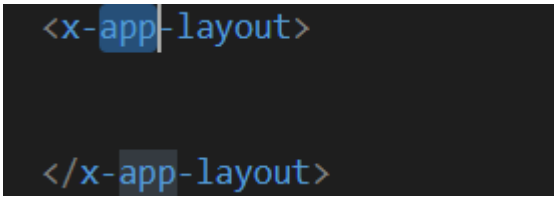

Figure 4.3 The App Layout blade file

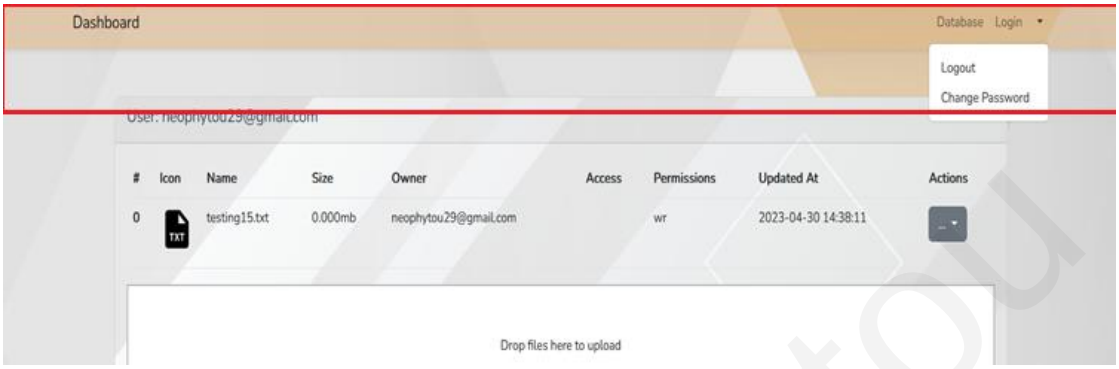

Figure 4.4 Dashboard

In this study all of these buttons are defined in the header, in the app.blade as seen in Figure 4.5:

```
<li class="nav-item dropdown">
    <a id="navbarDropdown" class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown"

    </a>

    <div class="dropdown-menu dropdown-menu-end" aria-labelledby="navbarDropdown">
        <a class="dropdown-item" href="{{ route('login') }}"
           onclick="event.preventDefault();
                    document.getElementById('logout-form').submit();">
            {{ __('Logout') }}
        </a>


        <form id="logout-form" action="{{ route('login') }}" method="GET" class="d-none">
            @csrf
        </form>

        <a class="dropdown-item" href="{{ route('change') }}"
           onclick="event.preventDefault();
                    document.getElementById('change-form').submit();">
            {{ __('Change Password') }}
        </a>

        <form id="change-form" action="{{ route('change') }}" method="GET" class="d-none">
            @csrf
```

Figure 4.5 App Layout

Controllers: The Controllers component manages application logic and responds to user queries. Users provide input to controllers, which are then responsible for validating it and using it to communicate with models to retrieve or change data. Views then display this information to the user after being passed by controllers. Controllers should manage application flow and business logic rather than include any HTML code. Figure 4.6 the main Controller file PageController.php is shown.

```
class PageController extends Controller {

public function users()
{
```

Figure 4.6 Controller File

Routes: The Routes component specifies how the application should respond to user queries. The routes/web.php file in Laravel defines routes. Each route links a URL to a particular Controller method, which manages the request. The HTTP method (GET, POST, PUT, or DELETE) necessary to reach a certain resource can also be specified using routes. In this project the post and the get routes were used. The information in [11] and [12] were very helpful to create these routes. In Figure 4.7 the file of the routes web.php is shown.

```
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\PageController;
/*
|--------------------------------------------------------------------------
| Web Routes
|--------------------------------------------------------------------------
|
| Here is where you can register web routes for your application. These
| routes are loaded by the RouteServiceProvider within a group which
| contains the "web" middleware group. Now create something great!
|
*/
```

Figure 4.7 Routes file

GET and POST are two HTTP methods that are frequently used in the Laravel framework to communicate with a web server. These two HTTP methods differ mostly in the following ways:

GET technique:

- Through the URL query string, data is transmitted.
- The maximum length of a URL determines the amount of data that can be delivered.
- The same outcome will be produced by several identical GET requests because these requests are idempotent.
- The browser has the ability to cache GET requests.
- When requesting information from a server, such as to fetch a web page, GET requests should be used.
- Views are in charge of showing data to the user, Controllers are in charge of taking user input and controlling application logic, and Routes specify how the application should respond to requests. Developers may build cleaner, more organized code that is simpler to maintain and scale over time by separating these roles.

POST approach:

- The request's body contains the data.
- The HTTP protocol does not place any restrictions on the maximum quantity of data that can be transferred.
- POST requests are not idempotent, which means that different outcomes may be obtained from multiple requests that are similar.
- The browser cannot save POST requests in its cache.
- When sending information to the server, such as when submitting a form, POST requests should be used.

Overall, POST requests are used to submit data to the server, whilst GET requests are used to get data from the server. POST requests have no such restrictions and cannot be cached, whereas GET requests have a limit on the amount of data they can transmit and can. To achieve the best performance and security, it's crucial to select the correct HTTP method for the particular activity at hand.

4.6 Sanctum Security

Laravel Sanctum is a popular authentication package that provides a simple and safe solution to deploy token-based authentication in Laravel applications. Positive and significant aspects of Laravel Sanctum include:

Simple and intuitive interface: Laravel Sanctum makes it simple for developers to start using the package by offering a simple and intuitive interface for enabling token-based authentication in Laravel apps.

Lightweight and adaptable: Laravel Sanctum is an authentication package that is adaptable and lightweight, and it can be simply configured to meet the unique requirements of an application.

Secure: Laravel Sanctum employs secure token-based authentication, which aids in the prevention of typical security risks such as cross-site scripting (XSS) and cross-site request forgery (CSRF).

Cross-platform compatibility: Laravel Sanctum is made to function with both classic server-rendered apps and cutting-edge single-page applications (SPAs) without a hitch, offering a constant authentication experience across platforms.

Revocation features are included into Laravel Sanctum, enabling developers to quickly revoke users' access to particular tokens as needed.

Integration with other Laravel features: Built on top of Laravel's robust authentication and authorization tools, Laravel Sanctum offers easy integration with other Laravel elements like routes, middleware, and policies.

Laravel Sanctum is a strong and flexible authentication package that provides a safe and user-friendly solution for token-based authentication in Laravel applications. Developers that require a dependable and adaptable authentication solution for their

apps should use Laravel Sanctum because of its lightweight design, built-in security features, and seamless connection with other Laravel components.

For this project the auth:sanctum middleware was used to protect the routes. When a user signs in then the current user has access to certain routes that are only allowed to authenticated users and these routes have to be protected. These routes are the access to files, for example download, delete, upload, rename or read. Password change or user's permissions change to files and access to table of the permissions route, changes in the whole database which is only allowed to the administrator. Other important routes that can change files or data in user interface have to be inside the group protected by the auth:sanctum middleware. Figure 4.8 shows the part of code that auth:sanctum middleware, protects the routes in the user interface.

Figure 4.8 Sanctum in Routes file

4.7 Bootstrap

It is a collection of fully-made website components and functionalities (tables, buttons, bars, menus, headers) which are open source and easy to be used by any developer. Bootstrap was used in this study to help the presentation design of the user interface that saved a lot of time and effort during this study. One important example, are the files that are displayed in the dashboard of the UI. Bootstrap was able to help to present the files in a table design, making them more easy to use and more understandable for the users.

4.8 Database in Laravel

In Laravel, databases are essential to the development of online applications. Working with databases is made simpler and more intuitive with Laravel's expressive and practical database abstraction layer. Figure 4.9 is a part of the .env file where it shows how the database was created by calling Laravel to built it.

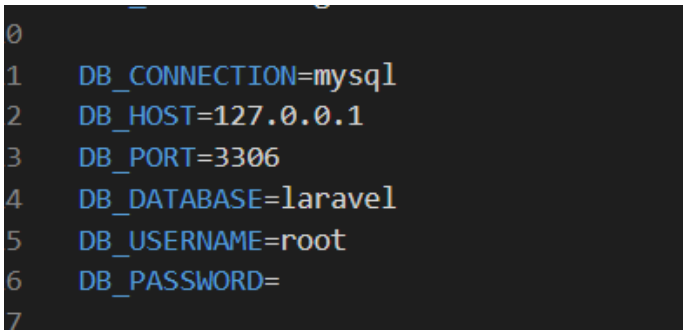

Figure 4.9 ENV file

Numerous databases, including MySQL, PostgreSQL, SQLite, and SQL Server, are supported by Laravel. The database layer of the framework offers a straightforward and standardized API to interact with various databases. Additionally, the database layer of Laravel includes tools like migrations, eloquent ORM, and query builders.

With Laravel's query builder capability, programmers can construct expressive and straightforward database queries. This feature facilitates the creation of complicated queries while assisting in the prevention of SQL injection threats. Developers can also connect together many query clauses using the query builder to create more complicated inquiries.

Laravel's query builder feature is a potent tool for creating database queries with an easy-to-use syntax. It offers a fluid API for building SQL queries that may be used with various database systems.

4.8.1 Entity Relationship Diagram

In Figure 4.10 the Database entity relationship diagram is presented.

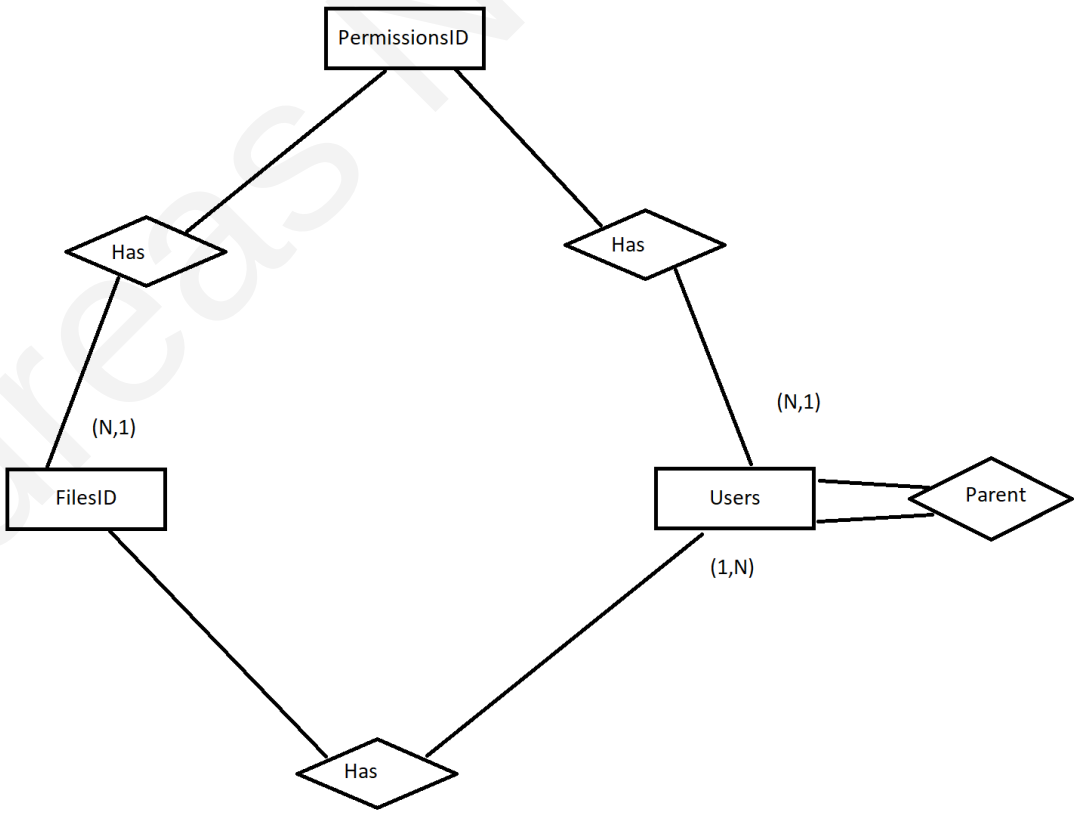

Figure 4.10 Entity Relationship Diagram

Each User:

- Can have many files and each file ids can have only one owner as user.
- Can have many permissions and each permission id can have only one user.

Each Permission:

- Can have only one file and many files can have many permissions.
- Can have only one User and each user can have many permission ids.

Each File:

- Can have one user as owner and each user can have many files.
- Can have many permissions and each permission can have one file.

4.8.2 Tables

Users Table

In this Table all the information about a user is stored.

#	Name	Type
1	ID	int
2	Password	String
3	Username	String
4	Email	String
5	Token	String
6	Email Verified At	Date

Table 1 User Table

Permissions Table

In this Table each User and File have their permission type. With this way the interface checks the permission of each user to each file.

#	Name	Type
1	ID	int
2	User ID	String
3	File ID	String
4	Permission	String

Table 2 Permissions Table

Files Table

In this table all the information for each file is stored. All the records have their unique ids where with this way the interface identifies the files. The field “updated at”

is when the file was updated and the “created at” when the file was created. Both are dates.

#	Name	Type
1	ID	int
2	Name	String
3	Size	String
4	Owner	String
5	Access	String
6	Updated At	Date
7	Created At	Date

Table 3 Files Table

In the user interface the Database consists of 3 tables, the User database which consists of 'username', 'email', 'password', 'remember_token' which is the token of the user when authenticated, 'name' which is the username and 'email_verified_at' which is when the user has been authenticated. The File id table which consists of 'name', 'owner', 'size', 'access' which are all the users that have access on the specific file. Lastly, the PermissionsID table where it consists of 'userid', 'fileid', 'permissions'.

The way that the tables work is that all the information of the user is saved in a new record in the Users table. Then, if the user creates a new file, then all of the information of the file is contained in the FileID table. Also, when the user, owner of a file adds new permissions to new users to access the file, the file id and the user id with the permissions w(write), r(read), wr(write/read) are saved to a new record in the PermissionsID table.

4.8.3 Laravel's Query Builder

Utilizing Laravel's query builder has a number of advantages, one of which is that it reduces the risk of SQL injection attacks. Input from users is automatically escaped by Laravel, making it considerably more challenging for hackers to insert dangerous SQL code.

The query builder offers considerably simpler and more user-friendly syntax for creating complex SQL queries in addition to its security advantages. The query builder, for instance, enables you to chain methods together to create queries in a way that is more readable and maintainable than manually concatenating text.

Figure 4.11 is an illustration of how to create a straightforward SQL query in Laravel using the query builder from my User Interface code:

```
    //database
if(!file_exists("C:/xampp2/htdocs/cms/public/files/".''.$file->getClientOriginalName())){

    $db=FileID::create([
        'owner'=>auth()->user()->username,
        'name'=>  $file->getClientOriginalName(),
        'permissions'=> 'wr',
        'size' => strval(number_format(filesize($file)/pow(1024,2),3)).''.'mb',
        'access'=>''

    ]);
```

Figure 4.11 Record creation in FileID

As you can see the SQL create feature is very easy to use and very dynamic and it allows the flexibility to developers to make fast records in databases. Basically, in the above code when a new file is uploaded the upload function is called in the controller file script and the record is created in the FileID table of the database.

Furthermore, another example that was used in this project and was very useful in a lot of evaluations and if statements is Figure 4.12 and Figure 4.13:

```
DB::table('fileid')->where('name',$file->getClientOriginalName(),)->delete();
```

Figure 4.12 Delete Record

```
public function MDBFile(Request $request)
{

    $f= FileID::where('name',$request->input('name'))->first();


    if($request->input('name')!=null)
        $f->fill([
            'name' => $request->input('name'),
      ])->save();

      if($request->input('owner')!=null)
      $f->fill([
          'owner' => $request->input('owner'),
    ])->save();

    if($request->input('permissions')!=null)
    $f->fill([
        'permissions' => $request->input('permissions'),
  ])->save();

  if($request->input('access')!=null)
  $f->fill([
      'access' => $request->input('access'),
])->save();
```

Figure 4.13 FILL function Laravel Databases

So, Laravel has the ability to find the table and run a query just by saying the word like where, order by and select. In the above example the code finds a specific file name in the 'name' column and deletes the specific record. The other ability is the fill() function where it overwrites a specific record in a column and replaces the data with the new one. The first() basically captures the first record that is found using the where clause.

Another Laravel tool that enables developers to change the database schema in an organized manner is migrations. Database tables, columns, indexes, and foreign keys may all be defined using Laravel's simple and elegant syntax. Because migrations are version-controlled, developers can quickly roll back or forward their modifications.

Eloquent, Laravel's ORM, offers a straightforward and expressive syntax for working with database records. Database tables can be defined as PHP classes by developers, who can then utilize Eloquent to communicate with the database. Eloquent has a number of capabilities, including query scopes, soft deletes, and model relationships.

Laravel offers a reliable and practical database layer that makes working with databases easier for developers. Eloquent ORM with features like the query builder and migrations make it simple for developers to create complicated web apps.

4.9 HTTP Requests using GUZZLE in Laravel

Popular PHP HTTP client Guzzle is simple to incorporate into Laravel projects. It supports features like HTTP/2, middleware, and authentication and offers a straightforward and user-friendly API for sending HTTP requests to external services or APIs.

Guzzle may be set up and utilized in Laravel with Composer, PHP's package management. After installation, Guzzle can be used to quickly send HTTP queries to external services or APIs. To use Guzzle the following procedure is followed.

- First, launch a fresh instance of the Guzzle client as follows:

```
$client = new GuzzleHttp\Client();
```

- Next, send a GET request to an external API using the get() method:

```
$response = $client->get('https://api.example.com/users');
```

- With this, a GET request will be made to the specified URL, returning a response object from which we can get the response content, the status code, and other details:

```
$body = $response->getBody();
```

$status = $response->getStatusCode();

Guzzle has a wide range of functionality, including request parameters, middleware, and authentication, in addition to supporting more HTTP methods like POST, PUT, and DELETE.

Guzzle, a component of Laravel, can be used to communicate with third-party APIs or services, like payment processors, social media APIs, and cloud services. It can also be used to send HTTP requests to a microservice or another internal API in order to communicate with other components of our application.

Figure 4.14 and Figure 4.15 are showing real parts of code that were used to send post requests to COBFS the 'create' and the 'read' of a file:

```
if(!file_exists("C:/xampp2/htdocs/cms/public/files/".''.$file->getClientOriginalName())){

    $db=FileID::create([
        'owner'=>auth()->user()->username,
        'name'=>  $file->getClientOriginalName(),
        'permissions'=> 'wr',
        'size' => strval(number_format(filesize($file)/pow(1024,2),3)).''.'mb',
        'access'=>''

    ]);

    $client = new Client();
    $res = $client->request('POST', 'http://controller.setup.collaborate.emulab.net:8000', [
        'json' => [
            "type" => "create",
            "clientID"=>DB::table('users')->where('username',auth()->user()->username)->value('id'),
            "fID"=> strval(DB::table('fileid')->where('name',$file->getClientOriginalName())->value('id')),
            "path"=> "C:/xampp2/htdocs/cms/public/files/".''.$file->getClientOriginalName(),
            // "dest_path"=> "test1",
            "value"=>base64_encode( file_get_contents($request->file('file'))),
        ]
    ]);
```

Figure 4.14 HTTP Request example 1

```
$client = new Client();
    $res = $client->request('POST', 'http://controller.setup.collaborate.emulab.net:8000', [
        'json' => [
            "type" => "read",
            "clientID"=>DB::table('users')->where('username',DB::table('fileid')->where('name',$request)->value('owner'))->value('id'),
            "fID"=> strval(DB::table('fileid')->where('name',$request)->value('id')),
        ]
    ]);
```

Figure 4.15 HTTP Request example 2

Overall, Guzzle is a strong and adaptable HTTP client that is simple to incorporate into Laravel applications. It is a wonderful option for developing contemporary online applications because of its support for features like HTTP/2, middleware, and authentication, as well as its straightforward and user-friendly API that makes it simple to perform HTTP calls to external services or APIs.

4.10 Creation of APIs to Communicate with Servers using POST and GET

In order for COBFS to communicate with user interface we constructed multiple APIs to the user interface for COBFS. These APIs were created to allow COBFS request information from the user interface, such as file renaming, file information for a specific user, login token requests, and file ID requests. COBFS is able to deliver a more streamlined and efficient data by using these APIs of the user interface, allowing COBFS to quickly manage and interact with the files stored in the system. All of these APIs were tested and the procedure followed is explained in this chapter.

GetFiles using get method API: The url of this method is http://localhost:8000/feestype/getFiles. The technique developed in this study takes the user's token from COBFS as input and outputs all of the files owned by that user. This technique allows the server to access the files quickly and easily by merely entering their token. Details such as the file name, file size, file type, and the date of the last update are supplied. This strategy simplifies file management for the user by allowing them to rapidly view all of their files' information in one spot. Furthermore, it minimizes the time and effort necessary for the server to discover a certain file since the technique allows the server to easily search through the files using the information offered by the approach.

In Figure 4.16 the code shows the input that the function gets from the server and returns back the name, updated at, possible permissions, size and id of each file owned by a particular user. Otherwise, it gives error message back.

```
}


public function getFiles(Request $request)
{
    $name='';

    $token= $request->bearerToken();
    $name=DB::table('users')->where('remember_token',$token)->value('username');


$result= FileID::select('name','updated_at','size','permissions','id')->where('owner',$name)->get();


if(count($result) != 0 )
return  response()->json(['status' => 'Okay', 'files' => $result]);
else
return  response()->json(['status' => 'Error', 'files' => $result]);
}
```

Figure 4.16 API Request example 1

RenameRequest post method API: The url of this API is http://192.168.0.102:8000/feestype/renameRequest. The "rename file" technique in COBFS user interface takes two inputs: the file ID and the new name that the server wants to assign to the file. This approach is intended to allow the server to simply

rename files without having to explore the user interface file structure. The technique waits for these inputs from COBFS and then performs the renaming action. The procedure then returns to COBFS the file's new information, including the updated information for the renamed file. This functionality allows the server to easily manage the files and quickly rename the files.

Figure 4.17 shows the method that the rename function gets as input the id of the file, finds the file from the database and then changes the file name from both the database and the file itself. Lastly, it returns a success message and the new name of the file. Otherwise, the api returns an error message.

```
public function renameRequest(Request $request)
{

    $f= FileID::where('id',$request->input('id'))->first();
    if($f=='')
    return  response()->json(['status' => 'Error', 'id' => 'Not found']);
    $section = DB::table('fileid')->where('id',$request->input('id'))->value('name');
    $name=$request->input('name');

    $newval=$name;


   if($section[0]!='C' && $section[1]!=':'){
    $v='C:/xampp2/htdocs/cms/public/files/''.$name;
    rename('C:/xampp2/htdocs/cms/public/files'.'/'.$section,$v);
    $f->fill([
        'name' => $name,
  ])->save();
    }
    else
    {

        $onlyBool=false;
        $onlypath='';
        $ch=FALSE;
        $path='';
        $reversed=strrev($section);
    foreach (str_split($reversed) as $char)
    {
        if((($char == '.') && $ch==FALSE) || $ch==TRUE){


            $ch=TRUE;
            if($char== chr(92))
            $onlyBool=true;
```

Figure 4.17 API Request example 2

RequestToken get method API: The url of this api is http://localhost:8000/feestype/requestToken. The "request token" method in COBFS user interface is intended to wait for system input in the form of a user's login and password. When this input is received, the method checks the user's credentials against the system's database to see if the user already has a token. The function returns the token to the system if the user has one. Otherwise, the user interface gives back an error message meaning the certain user does not exist in the database.

Figure 4.18 shows the part of code where the user interface tries to find from the database the user token and id. If the user exists the output of the result is not null and the method returns the information. Otherwise, the error message status, as mentioned in the paragraph above, is returned.

```
}

//api get requests
public function requestToken(Request $request)//from now on there are the requests
{

    $name='';
    $name=$request->username;
    $name.=' ';
    $name.=$request->password;

    $result = DB::table('users')->where('username',$request->username)->where('unencryptedpass',$request->password)->value('remember_tok
    $result2=DB::table('users')->where('username',$request->username)->where('unencryptedpass',$request->password)->value('id');

    if(strlen($result) != 0 )
    return  response()->json(['status' => 'Okay','id' =>$result2, 'token' => $result]);
    else
    return  response()->json(['status' => 'Error']);




}
```

Figure 4.18 API Request example 3

Get File ID post method API: The "get file id" API is intended to take a file name and token of a user from COBFS and produce a new ID for that file. This API generates a new record in the user interface's database to hold the file details. When a user asks information on a file, the user interface will access the database using the file ID. The user interface may successfully handle all of the files stored in COBFS by producing a new record for each file. This approach is an important part of the user interface since it allows COBFS to quickly access and manage files by assigning a unique identity that can be used to get file information. Furthermore, this API is critical for preserving the database integrity of the user interface, ensuring that each file is displayed appropriately and without duplication.

Figure 4.19 shows that the api creates a new record in the database with the username of the token that COBFS gave and adds the name of the file in the record. This api responds back to the server the new id of the file with a success respond message.

```
public function getFileID(Request $request)
{

   $token= $request->bearerToken();
    $name='';
    $name=$request->input('name');

  if($name[0]=='C' && $name[1]==':')
  {
    $str1=$name;
    $strnew=str_split($str1);


      for($i=0;$i< count($strnew)-1;$i++)
      {
          if($strnew[$i]==chr(92) && $strnew[$i+1]==chr(92))
          {

            $strnew[$i]='';
          }

      }
      $str1=implode($strnew);
     $name=$str1;
  }

    $result2=FileID::create([
        'owner'=> DB::table('users')->where('remember_token',$token)->value('username'),
        'name'=> $name,
        'permissions'=> 'wr',
        'size' => '',
        'access'=>''

    ]);

$result='111';
if(strlen($result) != 0 )
```

Figure 4.19 API Request example 4

4.11 EMAIL Verification in Laravel using Mailtrap as a Testing Tool

When registering users for web applications, email verification is an essential step. The implementation of email verification in Laravel is rather simple, and using Mailtrap makes testing and debugging much simpler.

Without sending the emails to actual users, Mailtrap is a program that simulates sending and receiving emails in a testing or development environment. It can be used to test email-based functions like user email verification in web applications.

Developers must first set up the Laravel application to utilize Mailtrap as the mail driver in order to enable email verification using Mailtrap in Laravel. This entails adding Mailtrap login information to your Laravel project's .env file.

This study used the following methods below, and implements the email verification after setting up Laravel to use Mailtrap:

a) First use the make:auth command of the php artisan script to create a fresh Laravel authentication scaffolding.

b) Add a verified attribute and a sendEmailVerificationNotification method by editing the User model. The sendEmailVerificationNotification method sends the email verification link to the user's email address, and the verified attribute indicates if a user's email address has been verified.
c) Make a new notification class that will deliver the user's email address the email verification link. The php artisan make:notification command can be used to accomplish this.
d) Add a toMail method that generates the email verification message by editing the VerifyEmail class.
e) Create a new route to handle requests for email verification.
f) To display a success message when the user has validated their email address, update the verification.blade.php file.

These procedures helped to test and debug Laravel's Mailtrap email verification. A link will be provided to the user's email address for email verification, once he/she registers for the app. The user can click the link to validate his/hers email address, update attribute for verified status, and gain access to the application.

Figure 4.20 shows the User Model where it is imported the email verification.

```php
class User extends Authenticatable implements MustVerifyEmail
{
    use HasApiTokens;
    use HasFactory;
    use HasProfilePhoto;
    use Notifiable;
    use TwoFactorAuthenticatable;
    /**
     * Route notifications for the mail channel.
     *
     * @param  \Illuminate\Notifications\Notification  $notification
     * @return array|string
     */
```

Figure 4.20 User Model with Email Verification

Also, in order to make certain routes use email verification, then the middleware (['auth', 'verified']) has to be added in order for the interface to ask for email verification. Figure 4.21 and Figure 4.22 show the protected routes using the authentication protection.

```php
Route::get('feestype/{feesType}/logout',[PageController::class,'login2'])->name('feestype.logout')->middleware(['verified']);
Route::post('feestype/{feesType}/logout',[PageController::class,'login2'])->name('feestype.logout')->middleware(['verified']);
```

Figure 4.21 Routes with middleware verified 1

```
Auth::routes([

    'verify'=>true

]);
```

Figure 4.22 Routes with middleware verified 2

Chapter 5

Program Implementation

It is crucial for this project to comprehend how the user interface functions and what each button accomplishes. Users will engage with COBFS primarily through the User Interface, therefore it must be clear and simple to use. This project contains a number of buttons that carry out different functions, knowing what each one will help the user utilize the application more efficiently.

5.1 Login Page

The first view when someone opens the interface is the Login page. Figure 5.1 shows the Login page which is the first page that a user comes across. The Login page view has a good UI, easy for the User to understand and navigate through it. It has the functionality to inform the user for bad credentials with an alert warning, rules that act as validations when the user clicks submit and a button for the user to register

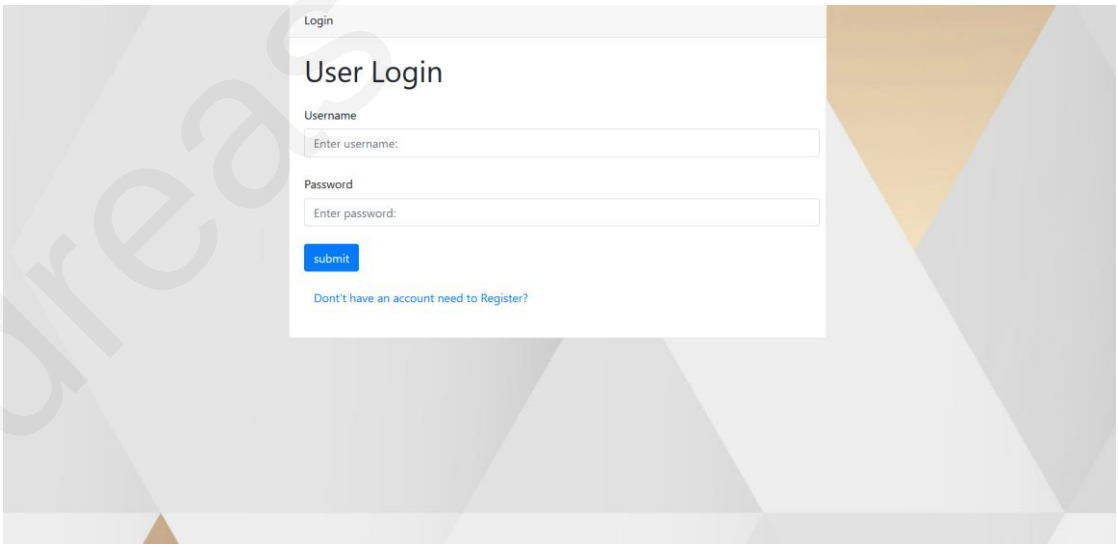

Figure 5.1 Login

Having a number of built-in tools and functionalities that make it simple to establish reliable user authentication and authorization procedures is one of the main advantages of using Laravel for the login page. This includes functions that can help to guarantee that user data is safe and secure, such as secure password hashing, two-factor authentication, and multi-factor authentication. Additionally, Laravel offers a straightforward and understandable syntax for building forms that can be used to build unique login pages that are catered to the particular requirements of the application. By offering a simpler and more user-friendly login process, this can aid in enhancing the user experience. The login page's security is further improved by Laravel's strong

routing and middleware features like sanctum that was used to secure our routes to only logged in Users.

5.2 Dashboard

The Dashboard view as shown in Figure 5.2 is activated when the user enters the correct credentials and logs in to his account after he/she presses the button submit. The Dashboard has the most functionalities of the interface. The most important functionality is the part where it shows all the files that are hosted from the server. A user can download, delete, rename, read or even change the permissions of a file or even upload a file. Other than the files, this page is self-synchronizing after a file is successfully uploaded, so if a user uploads a new file, then after the synchronization it will be shown to him/her. A user can also press the Dashboard text that will call again the dashboard and refresh the page (for synchronization purposes). There are also some buttons that help with navigation like the login that takes the user to the login page, logout that logs out the user, the change password that lets the user change a password. Also, there is the "User:" text box that shows who the user is that is logged in currently.

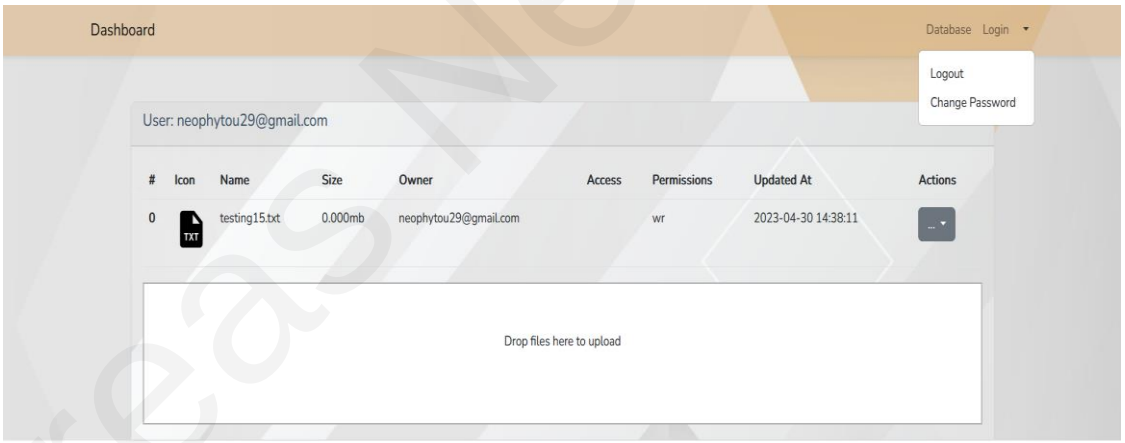

Figure 5.2 Dashboard

The upload part of the interface is also included in the dashboard page as discussed above. It is very user friendly allowing the user to drag and drop any file that he/she wants (.jpeg,.jpg,.png,.pdf,.doc,.docx, txt). There are two ways to upload a file by drag and drop or click the white box of the upload and find the file from the local computer. When the user inserts a file, a done sign will be shown to him/her or a failure mark if it is uploaded successfully or not.

There are functionalities for each icon of the file, for example when clicking the interface lets the user change the permissions of file or double click to open the permissions view in order to change the permissions.

There are two requests that can happen from the upload drag and drop page as shown in Figure 5.3. Firstly, the write operation where someone replaces the old file with the new file but the files have to be identical, and the upload which is the create request that the user(owner) uploads a new file. Both operations have to be firstly approved from the server before they can happen.

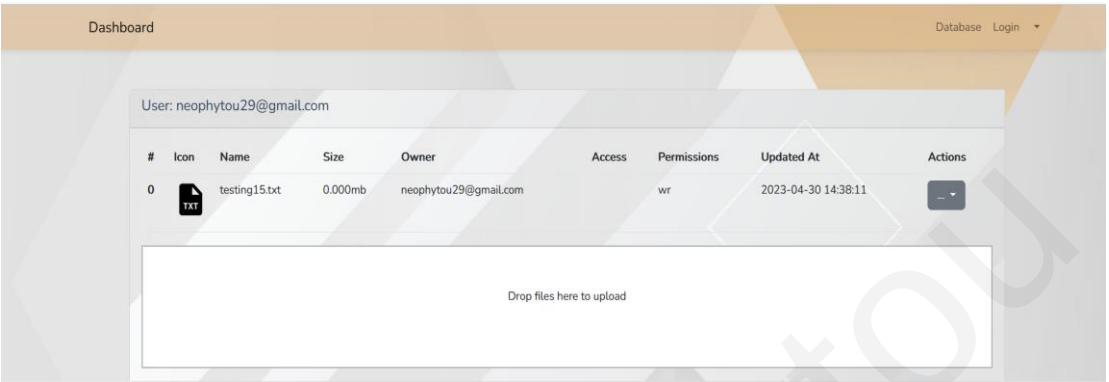

Figure 5.3 Dashboard

As shown in Figure 5.4, there are also the delete operation where it deletes the file which is a request delete to the server and has to be approved, the download operation (read operation) and the rename operation which is a write operation explained in the next sub chapter. Also, there is the read operation which is a double click on the file. All these operations have to be accepted by the server before they proceed to take action.

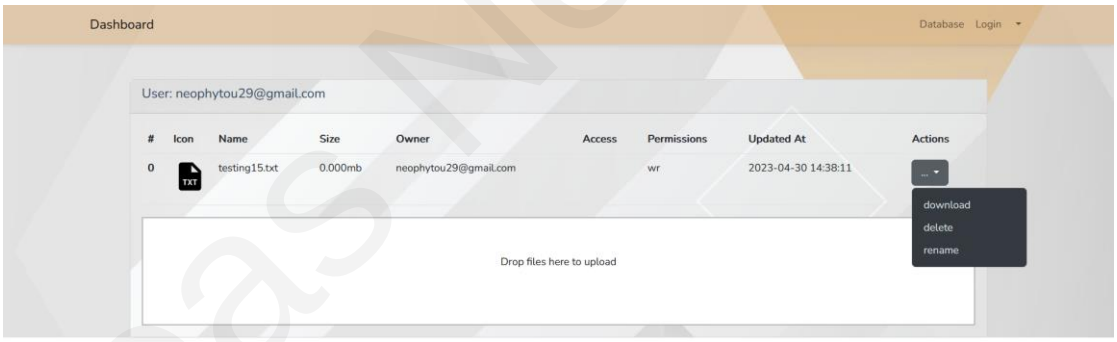

Figure 5.4 Dashboard 2

Any storage distributed system user interface must include a dashboard because it gives users a central location to access critical file information. It acts as a window into the system, enabling users to follow progress, view the status of their files, and easily access the locations they use the most. As the dashboard is frequently the first thing users see when accessing the system, it must be simple and simple to use. The dashboard can save users time and increase efficiency by giving them the information they need to make informed decisions about their files. Additionally, it enables users to spot problems or errors right away, reducing the chance of data loss or system outages. Any storage distributed system must have a well-designed dashboard, and it must be simple to use and intuitive so that users can efficiently manage their files and complete their tasks.

5.3 Change Password

The user has the ability to also change the password of his/her account. The change password button is in the right-up corner in the Dashboard view. Firstly, the user enters his/her username, his password for verification and the new password that wants to be changed. The system then finds the username from the database, verifies that the user is indeed himself/herself and changes only the password with the new

one. After the change happens when the user will need to log in again and he/she will use the new password that was entered. When the user presses the submit button the program navigates the user automatically to the login page where he can login again with the new credentials. Figure 5.5 shows the page where a user can change password.

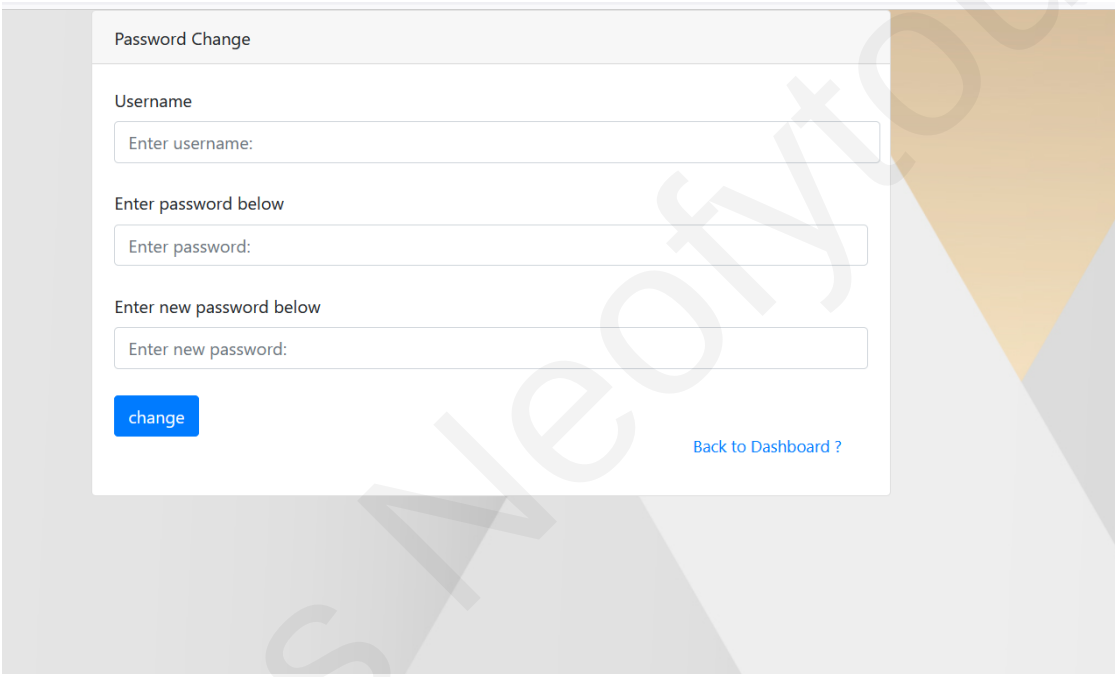

Figure 5.5 Password change

5.4 Rename File Page

Other than the permissions of a file, there is also the view about the rename of a file where the program finds the record of the file in the database and replaces the old name with the new name. The view can be accessed from the dashboard (Figure 5.6).

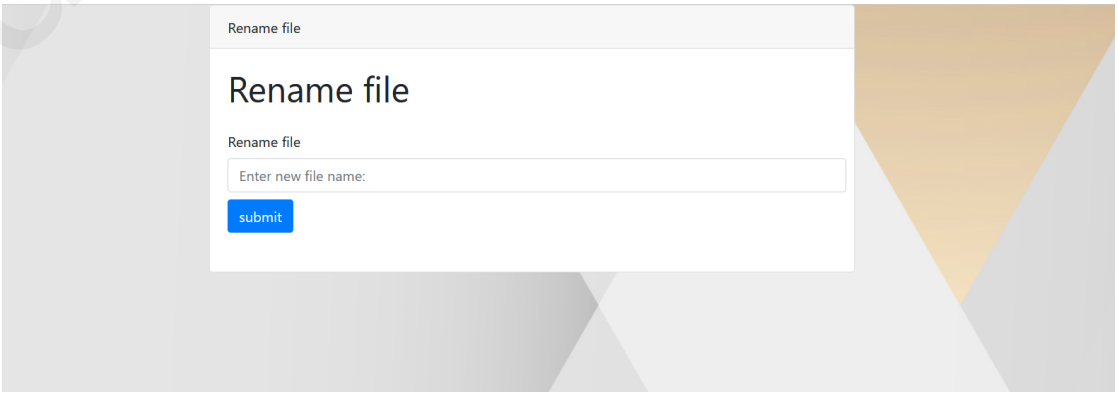

Figure 5.6 Rename File

Next to each file (Figure 5.6) there is a button ‘rename’ and with this way it makes it easier for users to understand the functionality of the button.

The rename feature is connected with the server and it needs a rename request to be accepted from the server before the rename is made.

Also, when a user renames a file, he/she has to enter the type of the file in other words the extension of the file (ex. Doc, docx, txt, pdf). This happens because the user can change the extension of the file by typing another extension. For example, if a doc file can be changed to a pdf file. But this feature is only acceptable to convertible files like doc/docx and pdf.

The rename page of files is a crucial function that is required in the user interface for a number of reasons. Here are a few examples:

Clarity and organization: In a user interface system, it might be challenging to maintain track of all the files a user owns. Users can label and organize their files with the use of a rename option, making it simpler to discover what they need right away.

Consistency: Any file management system should have a consistent naming convention. A rename feature can help make sure that all files have the same structure. This can be crucial when several people are collaborating on the same files.

Usefulness: The ability to rename files makes it easier for users to edit their content without having to delete and reupload the original files. Time is saved, and the chance of data loss by accident is decreased.

Flexibility: By renaming files, users can alter a file's name without changing its contents, making it simpler to use the same information in various circumstances.

Maintenance: A file system can be kept up with the use of a rename feature. For instance, files can be renamed to clearly indicate that they are no longer in use as they age or cease to be necessary.

5.5 Administrator Access Page

If a user is an administrator, then he/she has some more privileges. These are that the admin can access the database tables of users and files, there the user can modify or even delete records of users and files. This page is only accessible to administrators. In the testing phase, user "neophytou29@gmail.com", was used as the sole administrator in the project. Figure 5.7 shows the page where an administrator can access the dashboard or access the database for modifications.

A database's administrators are crucial to its management, hence they must have access to it. The stability, security, and performance of the database are the responsibility of the administrators, and access to it enables them to keep track of, troubleshoot, and improve its operations.

To carry out recurring maintenance procedures like recovery, patching, and upgrades, administrators need database access. In order to increase the database's effectiveness, they must also be able to track its performance indicators, identify problems, and

make the required adjustments. In order to manage users, give rights, and guarantee data security, so administrators require to have access to the database.

The availability, dependability, and security of the database are the responsibility of the database administrators. They need access to the database in order to manage users and permissions, monitor performance, diagnose problems, and execute normal maintenance. Without this access, the database's security and functioning may be jeopardized, potentially resulting in data loss, downtime, and other problems.

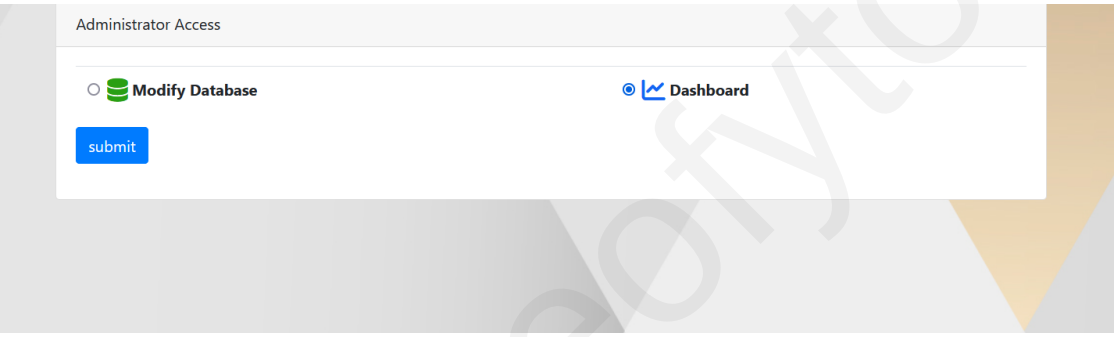

Figure 5.7 Administrator Access

5.6 The Database Page

The database page consists of two tables (Figure 5.8). The user table where each record holds the data of each user includes, the id, username, email, password, token when authenticated and date of email which is when the account is verified. Also, the files' table includes the id, name, owner permissions, accesses and size of the file. The admin can modify the record of the table by clicking the edit button(pencil). Then a new page comes up asking which value can be modified and if the user writes something it replaces the old with the new data or otherwise skips the empty input by keeping the old data. A user can also delete the record by easily clicking the trash button and the page redirects to this page again with the record deleted.

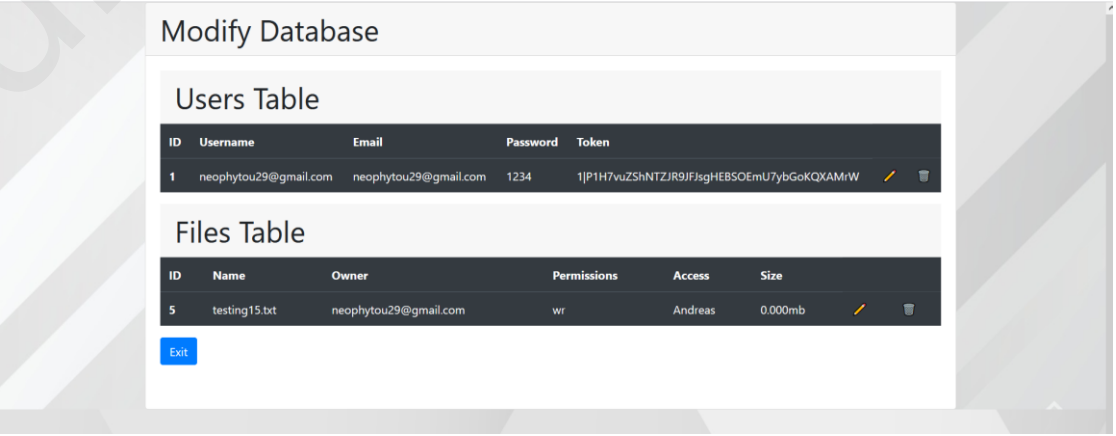

Figure 5.8 Database Access

Effective database management demands a user interface that enables database table modifications. Database administrators and other authorized users can easily edit the

tables' fields, structure, and data using a screen like this without having to directly access the database.

Users can carry out operations including adding or removing columns, altering data types, modifying constraints, and inserting or updating data when the database's tables provide from the user interface for table modification. Particularly for complex databases with several tables and fields, this capability may be time-saving.

The user interface by changing database tables can also assist in avoiding errors and data damage. By enforcing data validation criteria, the interface can stop users from entering inaccurate data. It can also offer auto-complete tools or suggestions to guarantee uniformity across the database.

A vital component for effective database management, the user interface by changing database tables can increase productivity, decrease errors, and improve data consistency.

How the page works:

1. First the admin has the ability to delete the record by pressing the trash button (Figure 5.9).

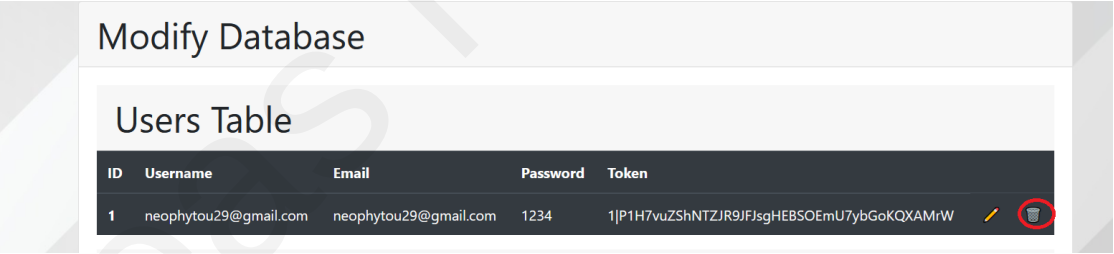

Figure 5.9 Database Access 2

2. Then the user is redirected back to the Dashboard page with the record missing (because it has been deleted).

3. The edit works by pressing the pencil button next to each record and the admin can change username, password or email of a record as shown in Figure 5.10. (The same with the files but for name, permissions, owner and accesses).

4. If the admin will not change anything the program detects it and ignores the null text boxes and leaves the old data in.

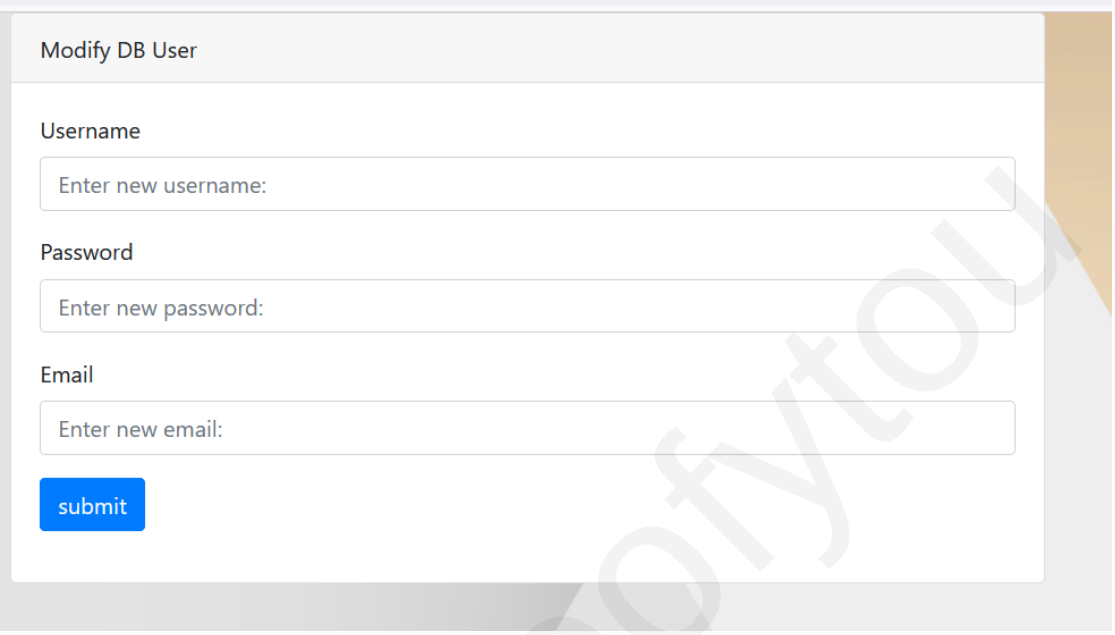

Figure 5.10 Modify Table

5.7 The Permissions Page

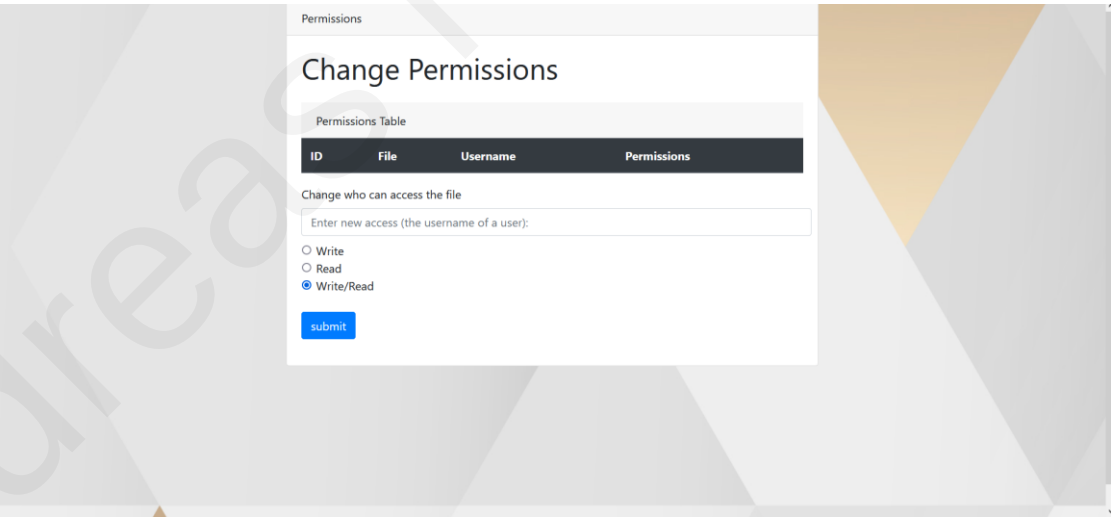

Figure 5.11 Modify Permissions

An effective method for controlling access to a file is a user interface that enables users to add new rights and accesses (Figure 5.11). Users can often provide the appropriate access level or permission type, such as read-only or read-write access, from the table of the database.

The user interface page shows a list of all users with access to the particular file along with their matching permission levels (r,w,wr) once the user submits the form or adjusts the permissions. It is simpler to handle access control when owners of the files can quickly and easily determine who has access to the file and at what level, thanks to this list.

The user interface enables users to add new rights and accesses is an efficient way to restrict access to a file. Through a form or fields on this interface, users can frequently specify the proper access level or permission type, such as read-only or read-write access.

Once the user submits the new access or changes the permissions, the user interface page displays a list of all users with access to the specific file along with their corresponding permission levels (Figure 5.12). When owners can quickly and easily determine who has access to the file and at what level, thanks to this list, it makes access control easier to manage. A column in the dashboard was also added that shows all the users that have access to the file to help all the users that have access to the file to know which other users can access the file.

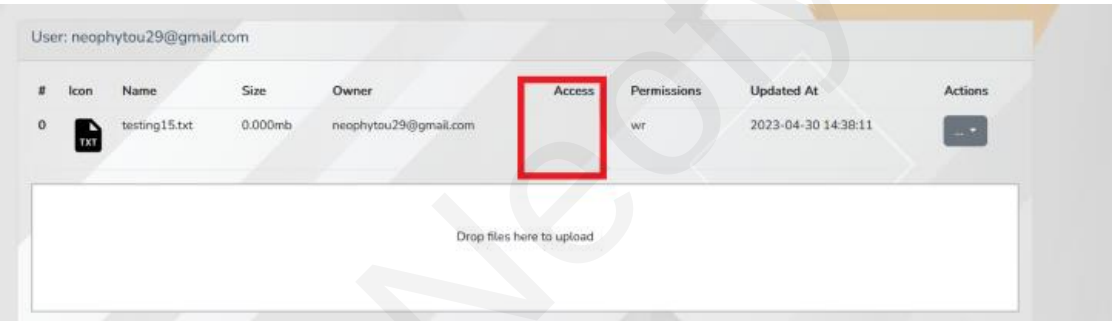

Figure 5.12 Access

How to create permissions:

1. First, in order to give permissions, you have to be the owner of the file. Press the icon of the file once and the permissions (Figure 5.13).

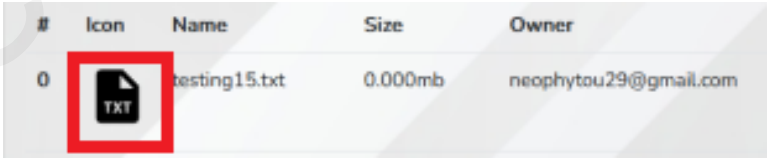

Figure 5.13 Icon of File

2. After the icon is clicked once (click twice is for read) the permissions view is brought up to the screen. The permissions screen shows the different permissions of users in a file and which user has access to the file (Figure 5.14).

3. After filling the username and clicking on what kind of permissions the owner of the file wants the user to have, the owner presses submit.

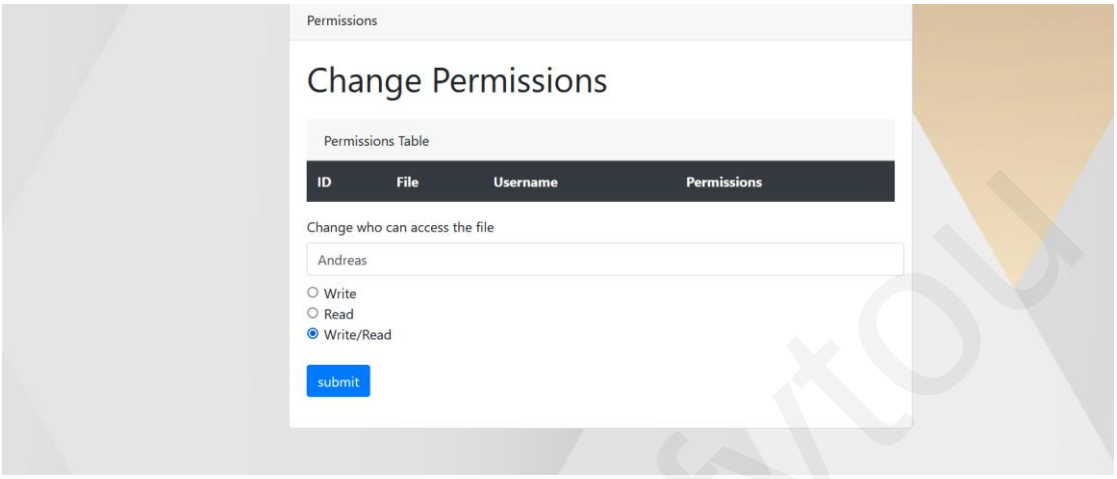

Figure 5.14 Modify Permissions 2

4. The new permission is added to the table as seen in the Figure 5.15.

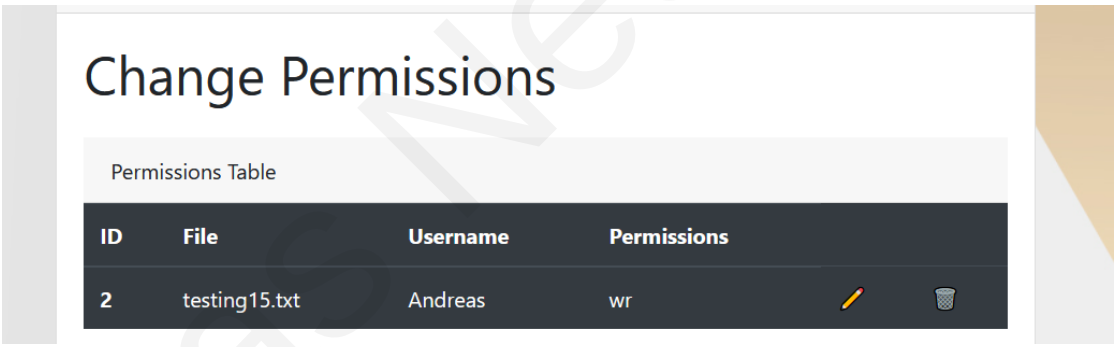

Figure 5.15 Permissions Table

5. If the pencil is clicked on the current record permission, then the user can change the permission as shown in Figure 5.16 or click the trash button to delete the whole permission record access.

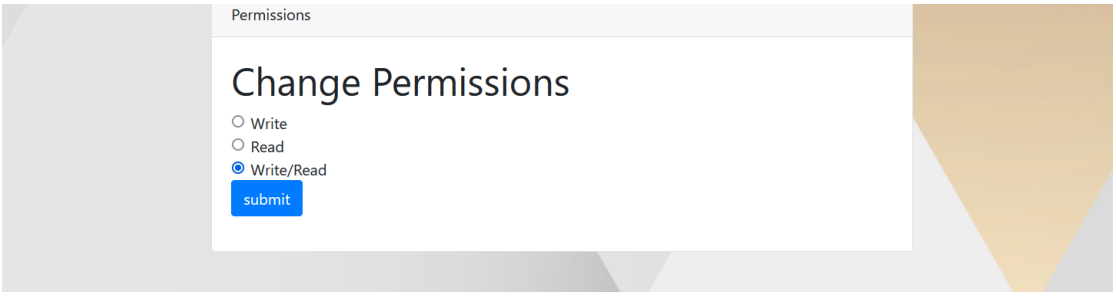

Figure 5.16 Modify Permissions 3

6. The acess colunm on the dashboard is updated with the new user that has permission (Figure 5.17).

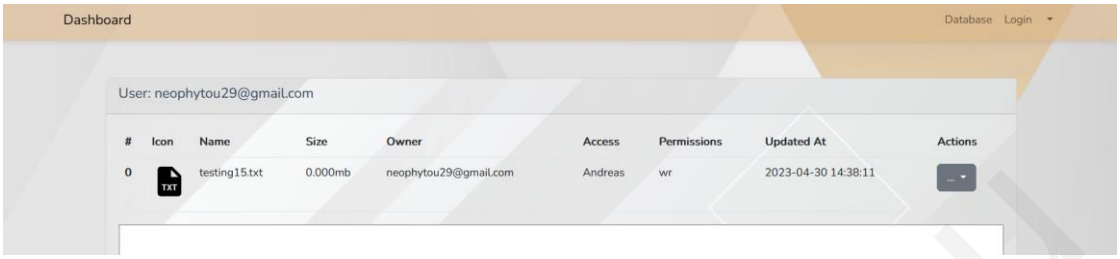

Figure 5.17 Modify Permissions

5.8 Register Page

Having a register functional page in the user interface is important for various reasons.

First, the registration page gives new users a way to register for an account and log into the system. Without this feature, system administrators would have to manually add users, which might be both time-consuming and ineffective.

The system can confirm the user's identity on the register page, which helps to ensure that only authorized users can access the system. This promotes system security and guards against illegal access to private information.

Thirdly, visitors can create an account and customize their preferences on the register page by adding their own password, username and email. As a result, users may have a better user experience and find the system more desirable.

Fourthly, the system can gather vital data about the user from the registration page, like their email address or username. If the users forget their login information, this information can be used to email them password reset links or to alert them to significant system upgrades or changes in the future.

Overall, a distributed storage system's user interface should have a register functional page since it gives new users a quick and secure way to log in and customize their account settings. By confirming the user's identification and gathering vital data about them, it also contributes to maintaining the security and integrity of the system.

How the page works:

The file distribution system registration page's user interface was made to appear uncomplicated in order to offer customers who want to create an account a hassle-free and smooth experience. Users may simply traverse the page and understand what information is required of them thanks to its straightforward and user-friendly design components.

Users are guided step-by-step through the registration process by the page layout, which includes labels and explicit directions for each field. There are few distractions or extraneous pieces of information on the interface, which is tidy and uncluttered.

The design places a strong emphasis on usability and simplicity in order to provide a satisfying user experience. Users are more likely to successfully register an account and interact with the system if the registration procedure is simplified and the user interface is friendly. COBFS will ultimately become more widely adopted and used, which is essential for any software application to succeed.

1. First the user adds his/her Username, email and password in the missing fields as shown in Figure 5.18.

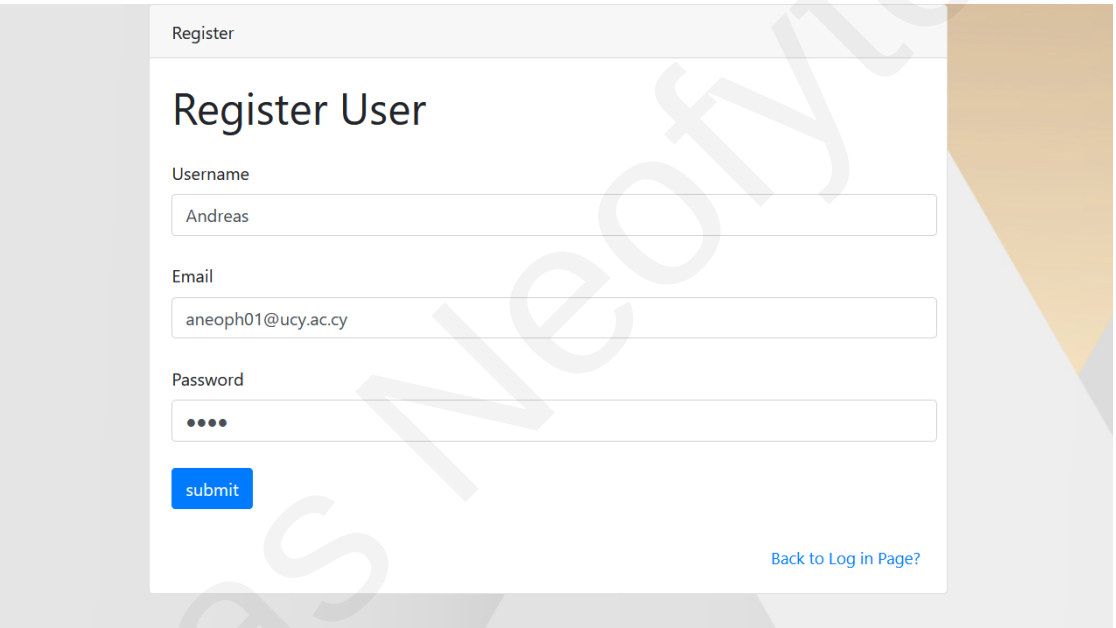

Figure 5.18 Register

2. Then the submit button is pressed and if the user credentials are accepted based on the rules, i.e., the password has to have length of four and above, or the username has to be unique, the account is created and added to the database.

3. Then a message is shown on the page informing the user that the account has been created (Figure 5.19).

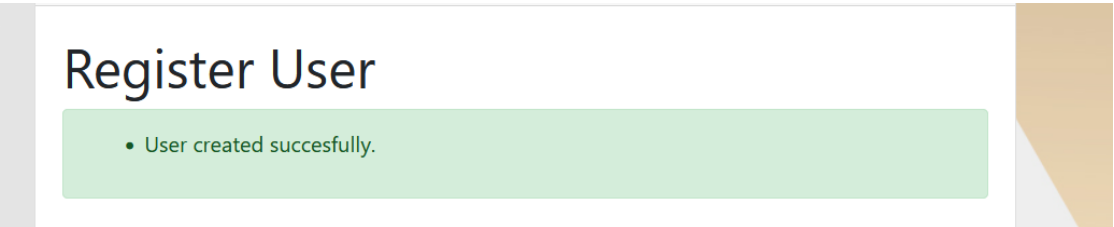

Figure 5.19 Success Registration

4. If the account already exists then a message is shown on the page and the account will not be created as shown in Figure 5.20.

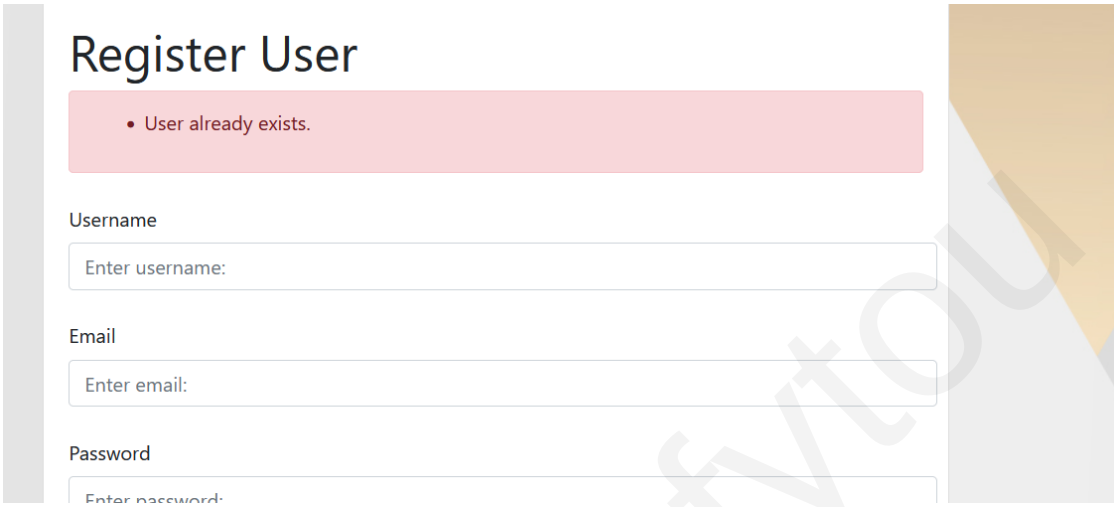

Figure 5.20 Failed Registration

5. After the return to log in page is pressed the user is prompt to verify his/her new account by informing the user that a verification link is sent (Figure 5.21).

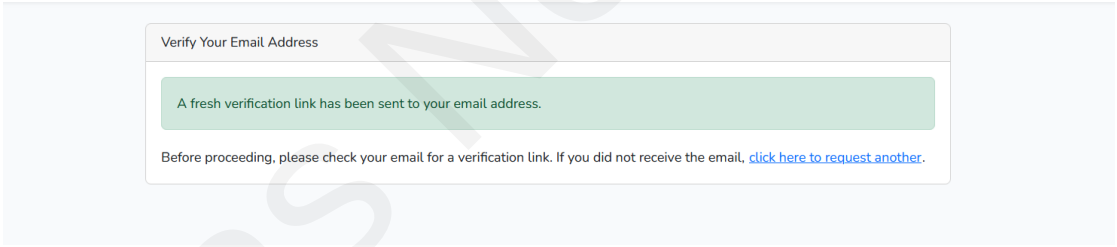

Figure 5.21 Email Verification

6. To verify the email laravel just send a verification email to the inbox (Figure 5.22 and Figure 5.23) of the entered email and the user just has to press verify email and he/she will be promt to the log in page to log in with the new account. From that point and on the account is verified.

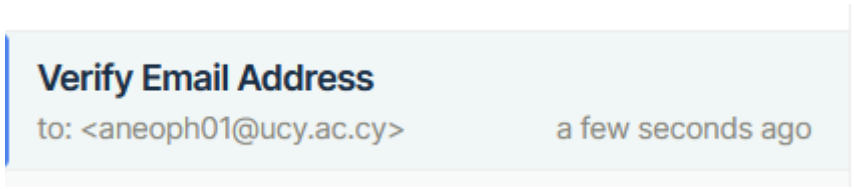

Figure 5.22 Mailtrap

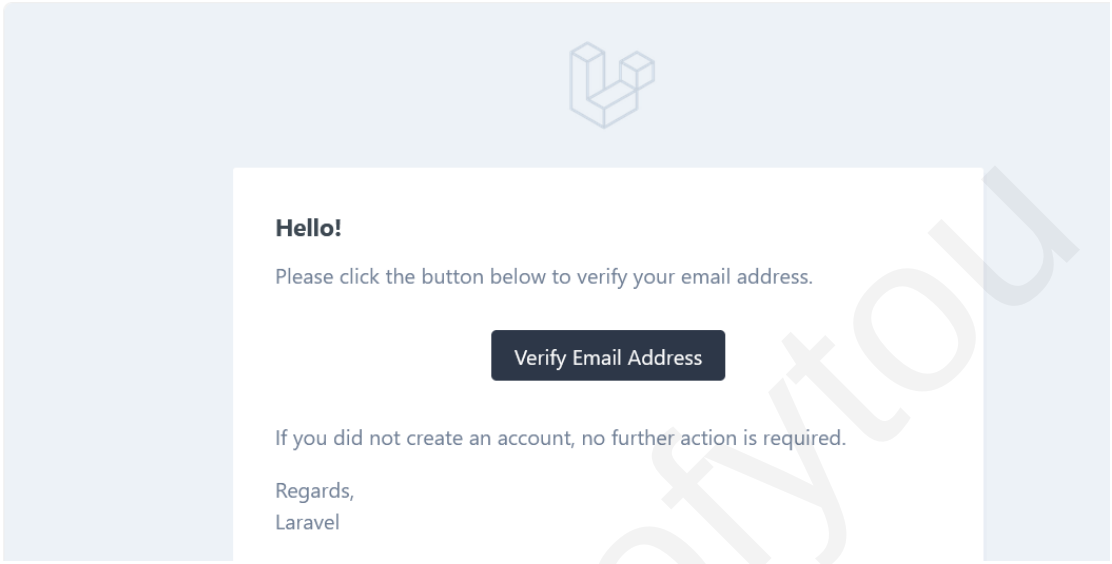

Figure 5.23 Mailtrap 2

7. Lastly, the user interface shows that the user is created succesfully and the user exists in the database as presented in Figure 5.24.

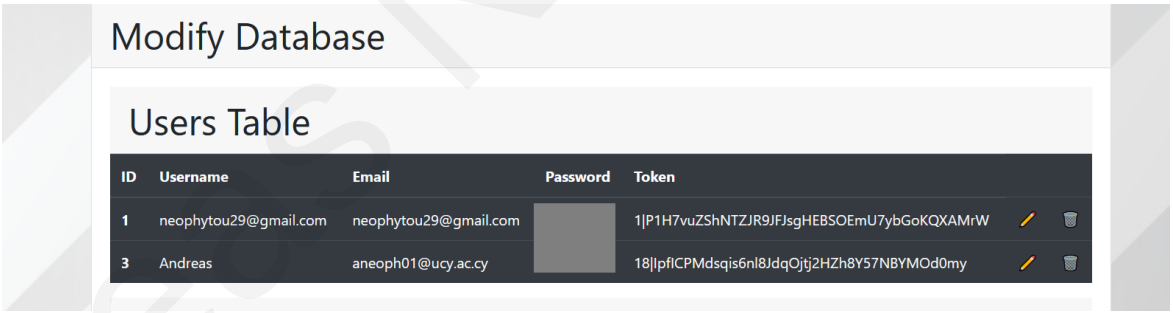

Figure 5.24 Database

Chapter 6

Conclusions

1.1 Summary

In conclusion, any online application, including those created with the Laravel framework and a distributed storage system, must have a user interface. The user experience, user engagement, and overall application performance can all be improved by a well-designed user interface.

During this study we have successfully built a user interface on Laravel that has the ability to let users interact with COBFS. Users for example are able to create files, open files, share files with other users, edit files, delete and rename files, interact with the system using their personal account, view the system file list continuously, which includes all the information users need to know, preview a file, and determine what permissions other users will have for the files they uploaded (owners of files). As a result, the system we created performs all of the duties that a user interface is intended to do in these situations.

Several problems were encountered throughout the development phase of the study on user interface for COBFS utilizing the Laravel framework. One of the difficulties was to design a user-friendly interface that consumers could readily understand and utilize. The user interface was built in a straight forward manner so that it is easy to understand and use its features.

Another problem was ensuring the system's security, especially when dealing with sensitive data such as user passwords and files. This was solved by enforcing strong password restrictions and adopting secure authentication and permission processes (sanctum), such as utilizing encrypted tokens. These are Laravel tools that we were able to import them in the user interface.

The system's performance was also an issue because it had to manage significant requests at the same time. This was handled by minimizing database queries, employing caching methods, and rewriting some functions using the help of Laravel framework's tools.

Integration with the distributed storage system utilizing COBFS was also a problem that was successfully overcome by building and testing APIs for different activities such as renaming a file, requesting a user token, listing all files, and obtaining the file id. The procedure for building and testing the APIs has been referred to in the specific chapter of this study. These APIs worked locally and remotely. They were tested with the local address by sending local requests. With the use of port forwarding we managed to test these APIs remotely. The system was able to receive requests from COBFS although it was located in a different area.

Upon completion of this thesis we were able to gain a lot of significant knowledge in various fields. This significant information includes getting familiar with all the tools that were used in this study like developing a web user interface, user authentication and more advanced programming knowledge.

In conclusion, developing a user interface is a crucial component of creating online applications, particularly this one that was created with the Laravel framework to connect to a distributed storage system. We tried to create a user interface that was both practical and aesthetically beautiful by adhering to UI development best practices and utilizing Laravel's broad UI development tools.

1.2 Future Work

Every system including this one has room for improvement. One important feature that can be included in the system is AI. For example, when a user last login exceeds a certain number of months, then the account should be deleted to free up space in the database of unused accounts. This will be done by the system because it will be able to recognize inactive accounts. Another important future work is about administrators. Now, the system has one administrator, so in the future the number of administrators can be increased by letting the main administrator (the current administrator that this user interface has) assign the privilege of admin rights to other users. This can be done by adding another column in the users table and naming it admin which will be Boolean. True if the user has admin rights and false if the user is just a normal user. The only one that will be able to change that column will be the main administrator.

Further additions for future work can include personalized themes for each user on the user interface so each user can have his/her own theme when logging in and a profile picture that will be saved in the database with the users' information. Furthermore, a new page for settings can be included in the future work that will let users customize important functionalities of the user interface and their accounts. Lastly, the user interface is accessible through the browser of a user. This can be enhanced in the future by letting mobile users access the UI from a mobile application in Android/IOS. In this way future users will be able to manage their files everywhere they are by simply using a mobile device.

Laravel apps like user interfaces are likely to see a lot of exciting new developments in the future. This is because Laravel has a lot of tools that can help developers and researchers add new features or make the UIs even better. These tools are upgraded by Laravel letting these applications up to date and were also used by us to build a lot of functions in the user interface like user authentication (Laravel sanctum). We expect in the future the tools we used in this study to become even more sophisticated and this can lead our user interface meet a higher standard.

In conclusion, by adding all of these future work designs to the application it will increase the productivity of the user interface and let users control more easily their accounts and manage their files in a more effective way.

Bibliography

1. [1] C. Georgiou, N. Nicolaou, and A. Trigeorgi, “Fragmented ARES: Dynamic Storage for Large Objects,” arXiv:2201.13292 [cs], Jan. 2022, Accessed: May 08, 2023. [Online]. Available: https://arxiv.org/abs/2201.13292

2. [2] A. Fernández Anta *et al.*, “Fragmented objects: Boosting concurrency of shared large objects,” *Structural Information and Communication Complexity*, pp. 106–126, 2021. doi:10.1007/978-3-030-79527-6_7

3. [3] Write, read, and server protocols of the ABD algorithm., https://www.researchgate.net/figure/Write-read-and-server-protocols-of-the-ABD-algorithm_fig2_279843747.

4. [4] “Distributed Storage: What’s Inside Amazon S3?,” Cloudian, 2011 https://cloudian.com/guides/data-backup/distributed storage/#:~:text=A%20distributed%20storage%20system%20is.

5. [5] Ijad Madisch, Sören Hofmayer, and Horst FickenscherResearchgate.net. [Online].Available: https://www.researchgate.net/figure/, 2008 Simulation-results-for-algorithms-COABD-and-COBFS_fig3_349620869.

6. [6] Taylor Otwell, “Laravel - the PHP framework for web artisans,” ”, June 9, 2011, Laravel.com. [Online]. Available: https://laravel.com/.

7. [7] Jeffrey Way, "Laracasts - The Best Laravel and PHP Screencasts”, 2013. https://laracasts.com/

8. [8] Stack Overflow, “Stack Overflow - Where Developers Learn, Share, & Build Careers,” Stack Overflow, 2022. https://stackoverflow.com/

9. [9] “How Laravel implements MVC and how to use it effectively | Pusher blog,” pusher.com. https://pusher.com/blog/laravel-mvc-use/#:~:text=Laravel%20is%20a%20PHP%2Dbased.

10. [10] Taylor Otwell , “Laravel - The PHP Framework For Web Artisans”, June 9, 2011. laravel.com. https://laravel.com/docs/10.x/blade

11. [11] Taylor Otwell, “Laravel - The PHP Framework For Web Artisans” , June 9, 2011. laravel.com laravel.com. https://laravel.com/docs/10.x/routing

12. [12] M. Satterfield, “PHP: Difference between laravel get and post route,” CopyProgramming, https://copyprogramming.com/howto/difference-between-laravel-get-and-post-route?utm_content=cmp-true.

13. M. I. Ali and M. D. Assaf, "Design and implementation of a web-based user interface for cloud storage systems," 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, Victoria, BC, 2014, pp. 128-133, doi: 10.1109/WAINA.2014.43.