Undergraduate Thesis

FRONT-END INTERFACE FOR A DISTRIBUTED STORAGE SYSTEM USING LARAVEL

Antreas Panagi

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2025

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Front-End Interface for a Distributed Storage System Using Laravel

Antreas Panagi

Supervising Professor
Dr. Chryssis Georgiou

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor's in Computer Science from the University of Cyprus

May 2025

Acknowledgments

I would like to express my deep gratitude to those who supported me in the eteraty of the project. Foremost amongst those to whom I am thankful are my family. They have given me their constant encouragement and understanding during the most trying moments of my work. I am especially grateful to my supervising professor, Dr. Chryssis Georgiou, for his expert guidance, insightful feedback, and continuous motivation. His constructive advice shaped my approach to problem solving and pushed me to maintain high academic standards. I would also like to thank Algolysis, the company that provided the real-world context and requirements for this platform. Their cooperation provided me a rare chance to get theoretical and technical knowledge to better understand the nature of software development challenges in industrial contexts. Finally, I would like to acknowledge my friends and fellow students who offered helpful insights and a supportive environment throughout this process. The willingness of these people to discuss ideas and share experiences greatly contributed to getting this project completed.

Abstract

This thesis focuses on the development of a web-based platform with Laravel devoted to simplifying file operations and collaboration in multi-user environments. Developed through collaboration with Algolysis, a Cypriot-based SME company, the system solves real-world issues such as secure file sharing, user-specific permissions for folder, files, and simple folder and file operations along with a smooth user interface.

The platform was constructed using the Laravel framework. The modular structure allows for maintainable development. Additionally, it works perfectly with MySQL through the use of strong Object Relational Mapping (ORM) tools and migration mechanisms. It is equipped with a responsive dashboard to list files and folders, detailed access permissions, dynamic shared folders, and file uploads for multiple-user collaboration scenarios. Further, grid and list view options were developed, providing users with choices for navigation.

When finally completed, testing was done to ascertain the platform's robustness under various workloads. The system met the goals for stability and usability. This document ends with a discussion on the limitations of the platform, possible future enhancements, and considerations for embracing the Laravel ecosystem for similar large-scale deployments.

Contents

Chapter **1** **Introduction**………………………………………………………….. ..**1**

1.1 Motivation and Purpose of the Work 1

1.2 Development Methodology and Timeline 2

1.3 Organization of the Document 5

Chapter **2** **Distributed Shared Memory (DSM)**…………………………………..**6**

2.1 Overview of Distributed Shared Memory and Consistency 6

2.2 Dynamic Membership and Reconfiguration Protocols 7

2.3 Large Objects and Fragmentation in Shared Memory 8

2.4 Combining Dynamic Reconfiguration with Fragmentation 11

Chapter **3** **Platform Description** ……………………………………….... **13**

3.1 Requirements and General Overview 13

3.1.1 Functional Requirements 14

3.1.2 Non-Functional Requirements 15

3.2 User Categories 16

3.3 Platform Constraints 17

3.4 Architectural Overview 19

3.5 Development Tools 21

3.5.1 Laravel (PHP Framework) 21

3.5.2 Bootstrap (Front-End Framework) 23

3.5.3 Alpine.js (JavaScript Library for Interactivity) 24

3.5.4 MySQL (Database Management System) 25

3.5.5 XAMPP (Development Server Environment) 26

3.6 Security Considerations 27

3.7 Database Structure 28

Chapter **4** **Implemented Platform** ……………………………………………... **32**

4.1 Existing Development 32

4.1.1 Login 33

4.1.2 Dashboard 35

4.1.3 User Profile 36

4.2 Main Page 37

4.2.1 List View 38
4.2.2 Grid View 39
4.3 Uploading Files 39
4.4 Actions 43
4.4.1 Rename 44
4.4.2 Delete 45
4.4.3 Download 46
4.4.4 Share 46
4.4.5 Info 49
4.5 Navigation 50
4.6 Messages 52
Chapter **5** **Implementation**………………………………………………….. **53**
5.1 Project Folder Layout 53
5.2 Key Libraries and Dependencies 54
5.3 Illustrative Code Snippets 54
5.3.1 Routing 54
5.3.2 Controllers and Eloquent Models 55
5.3.3 MySQL and Database Migrations 57
5.3.4 Laravel Query Builder 58
5.3.5 Bootstrap and Alpine.js for Front End Interactions 58
Chapter **6** **Conclusion**……………………………………………………… **60**
6.1 Summary of Achievements 60
6.2 Lessons Learned 61
6.3 Limitations and Future Work 62
6.4 Final Remarks 63

R e f e r e n c e s ………………………………………………………….. **64**

A p e n d i x A……………………..…………………………….………… **A-1**
-
A p e n d i x B……………………..…………………………….………… **A-5**

Chapter 1

Introduction

1.1 Motivation and Purpose of the Work 1

1.2 Development Methodology and Timeline 2

1.3 Organization of the Document 5

1.1 Motivation and Purpose of the Work

Generation and storage of data take place across a mountain range of platforms in the current digital era. It is estimated that the world will generate about 181 ZB of data in 2025 [17]. This growth of digital information indeed holds the need for quick systems that store, administer, and retrieve that very data. Organizations are acquiring an increasing number of documents, presentations, and all sorts of files that need to be handled securely and systematically. The interest for this work arises from the challenges brought on by this digital proliferation. Without an established management system, files become scattered on personal systems or cloud services, thereby increasing the efforts to maintain consistency, avoid data loss, and ensure that only authorized users can easily obtain the information they desire.

The storage system under examination in this thesis is designed to execute completely in a distributed mode, consistent with the principles of Distributed Shared Memory (DSM) [12]. As opposed to usual centralized systems relying on a single point of control, within this architecture storage responsibility is divided across different nodes, benefiting from enhanced resilience, scalability, and fault tolerance. One of the advantages of distributed storage is eliminating single points of failure, data can still be accessed even if one or more nodes go down. The decentralized structure also improves security by spreading out the attack surface and allows for more flexible maintenance, as updates can be installed in increments to nodes.

The foundation of this project is the collaborative research between the research team of Professor C. Georgiou and the Algolysis team to develop a backend service for Distributed Shared Memory (DSM). The backend service provides strong consistency and high concurrency for file operations.

The main motivation for this thesis was to develop a frontend platform that brings the benefits of distributed storage to a usable interface. The interface should be as intuitive as other popular cloud-based programs, so simple users could easily use the main features of the platform, files navigation, uploading, renaming, sharing and storing files without needing to understand the complexities of the distributed backend system.

To achieve this, Laravel [15] was selected as the web application framework due to its solid combination of security features, Model-View-Controller (MVC) pattern, built-in user authentication, flexible routing, middleware support, and Blade templating engine. Laravel offers an ideal environment to develop scalable, secure, and easy to maintain web applications with the performance and degree of customization required to integrate into the distributed backend.

The objective of this thesis, therefore, was to develop and implement a complete front-end interface on Laravel that was easy to use, which interacts with DSM backend using APIs flawlessly.

1.2 Development Methodology and Timeline

The process of development of the file management platform was very structured, balancing planning with iterative refinement. The very first step involved requirements analysis, both functional and non-functional. Functional requirements denote the main platform features, such as user authentication, uploading and downloading files, and creation and organisation of folders. Non-functional requirements specify that qualities to expect of the system, such as security, usability, performance, and scalability. These were gathered through discussion with the company.

With the requirements ready, the project proceeded along an iterative path to development strategy, inspired by Agile processes [13]. Rather than adopting strict Waterfall methodology whose phases must be completed fully before another can be commenced, Agile process considers developing the system incrementally in cycles, continually revisiting and refining the work. These iterations have been defined such that each should deliver only a subset of features that may be reviewed and tested, and this enabled feedback response to be done very early in time. This methodology ensured we were always open to accommodating any changes to requirements or select areas of the design that can be improved considering how the team came to view them during the development.

Project planning and tool selection were quite critical to the development procedure. The source-code version control was given into use with Git [18], allowing multiple programmers to work together on the same set of code and make changes in it while tracking them with time. This feature proved to be extremely useful in the iteration process when multiple new features were added, conflicts with existing features would arise, and in such cases, version-control history would help resolve the conflicts by allowing one to locate the last changed file or code.

Testing and quality assurance were a constant in the methodology. While a new feature was developed during each development iteration, it was tested at various levels. Unit testing would take place to confirm that fine-grained pieces work as intended: for example, tests to store a newly uploaded file for various file types and sizes or tests to ensure the function performing permission checks properly prevents unauthorized access to files. Besides unit testing, integration tests would verify that various system components interact with one another, an example here would be uploading a file through the web interface, then checking that the file is listed for the user with correct metadata in the database, and the file can be downloaded intact.

The project timeline stretched over the period of an academic year and was divided into separate phases and milestones. The first phase, from September through November, involved learning Laravel and all supporting technologies: Livewire [19], Alpine.js [8], and Bootstrap [7]. The period also covered research into Database design practices as well as User Interface development. In October and November, requirements engineering were done along with the study and review of all the technologies. It covered aspects like the features to be in the system, user roles of the system, technical constraints where feedback was received from the company. The next step in December and January related to prototyping. A functional prototype was created that supports the basic authentication, navigating directories, and uploading files as a proof of concept of the proposed system architecture.

Key activities from January to March opened with developments in expanding the platform's functionality. Core features were established in this period, namely, grid and list views for the flexible browsing of files, sharing options where permissions rights can be effectively managed and extensive user-interface enhancements. In the period of development, the testing continued, ensuring stability and usability. However, a final exhaustive testing and quality assurance were performed in April, including debugging and performance tuning, and Cross-Site Request Forgery (CSRF) protection verification as well. The other phase in April and May was cantered on the composition of the thesis document.

Through this phased schedule combining iterative development, preliminary testing, and progressive feature integration, the project was able to produce a stable and feature-packed Laravel-based file management system within the expected timeline. Below in Figure 1.1 is a Gantt Chart demonstrating the project timeline.

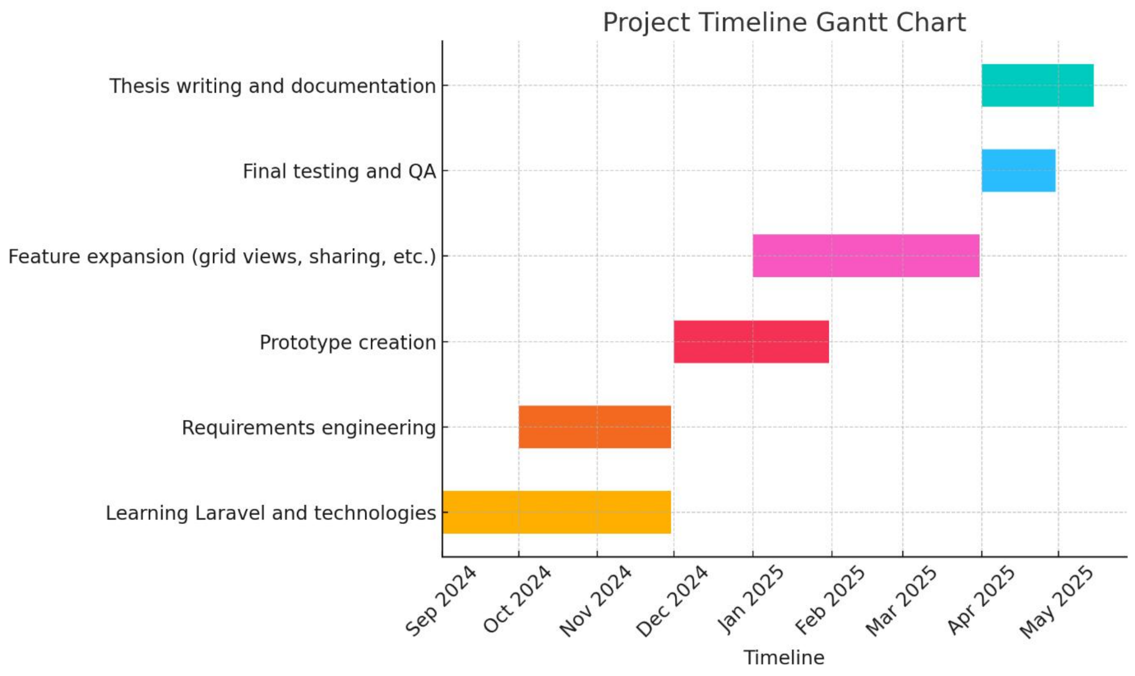


Figure 1.1 – Timeline Gantt Chart

1.3 Organization of the Document

This thesis consists of six chapters, including this one, with each chapter discussing an essential element of the research, design, and development journey of the project.

Chapter 2 discusses the theoretical foundation of Distributed Shared Memory (DSM) systems, encompassing static and dynamic DSM protocols, consistency models, and large fragmented data object handling in a distributed environment. This background sets the stage for the research context in the design of the platform.

Chapter 3 describes the requirements and design of the file management platform, user roles, system constraints, architectural design, development tools, and database schema.

Chapter 4 introduces the user interface of the implemented platform, explicating primary features such as login, dashboard, file upload, file/folder actions, and navigation.

Chapter 5 provides details about the underlying technology stack and the implementation style. It describes how the back end was developed using Laravel, the front end was crafted with Bootstrap and Alpine.js and illustrates some coding approaches encountered in key areas.

Lastly, Chapter 6 summarizes the outcomes of the project, discusses some lessons learned, evaluates the limitations of the results, and suggests potential future works to extend or enhance the current platform

The thesis concludes with references and a comprehensive appendix, including user documentation and a technical dictionary.

Chapter 2:

Distributed Shared Memory

2.1 Overview of Distributed Shared Memory and Consistency 6

2.2 Dynamic Membership and Reconfiguration Protocols 7

2.3 Large Objects and Fragmentation in Shared Memory 8

2.4 Combining Dynamic Reconfiguration with Fragmentation 11

2.1 Overview of Distributed Shared Memory and Consistency

Distributed Shared Memory (DSM) [12] provides the abstraction of a single coherent memory space to applications running on multiple networked nodes. Each data object in a DSM is replicated on a collection of servers, and clients perform reads and writes on these objects as if they were single-copy memory locations. The challenge is to keep all replicas consistent in the face of concurrent accesses and failures. The strongest consistency model typically sought is *atomic consistency*, also known as *linearizability* [14], which ensures that operations appear to execute indivisibly and in some global order consistent with real time. In practical terms, linearizability gives the illusion that there is a single copy of each object that all processes read and write, even though, behind the scenes, there may be many replicas.

One conventional approach to maintaining atomic consistency in DSM is the Lynch and Shvartsman algorithm [3], which uses a linearizable read/write register with quorums of replica servers. The algorithm ensures that every operation (either a read or write) involves contacting a quorum (usually a majority) of replicas. Each write operation receives a new, monotonically increasing timestamp (or tag) and is propagated with the new value to a quorum of servers. Each read operation then requests a quorum of servers for the value with the highest seen timestamp. Because any two quorum sets share at least one server, a read will always see the most recent write – the overlapping replica will yield the most current value if there was a newer write. This way, the Lynch and Shvartsman algorithm uses versioned writes and intersection of quorums to achieve a single-copy illusion consistency: all clients observe changes in one global order, regardless of failures or latencies of messages. The result is a strongly consistent, highly available memory service whose replicated data enjoys the properties of an atomic variable.

2.2 Dynamic Membership and Reconfiguration Protocols

A DSM that operates in a dynamic network must handle membership changes (servers joining or leaving) transparently, without sacrificing consistency. Naively, one could stop the world and use consensus to agree on each new membership configuration. However, stopping application operations during reconfiguration is undesirable, and repeatedly invoking consensus can become a bottleneck. Newer research explored how to reconfigure distributed shared memory efficiently, allowing reads and writes to progress concurrently with membership changes.

One notable approach is RAMBO, which stands for *Reconfigurable Atomic Memory for Basic Objects* [20]. RAMBO introduces the idea of dividing time into epochs or configurations, each with its own set of replicas and quorum system, and uses an agreement protocol to move from one configuration to the next in a controlled way [4]. Readers and writers in RAMBO must sometimes wait during reconfiguration events, since a new configuration must intersect with the old one to avoid losing the most recent writes. While RAMBO guarantees atomicity across configuration changes, it does so at the cost of coordination delays when installing a new configuration.

An alternative approach is exemplified by the DynaStore algorithm [21], which implements a dynamic multi-writer/multi-reader atomic memory by incorporating reconfiguration logic into the read and write protocols themselves. DynaStore organizes the sequence of configurations into a directed acyclic graph (DAG), where each node represents a configuration and edges represent transitions following reconfiguration proposals [4]. While DynaStore allows reads and writes to proceed without global consensus during membership changes, it assumes that the rate of reconfiguration is not unbounded to guarantee termination.

Ares is a more recent protocol designed to overcome these limitations and provide efficient reconfiguration in distributed shared memory. Ares (Adaptive, Reconfigurable, Erasure-coded, Atomic Storage) [6] distinguishes itself from earlier solutions in several key ways. First, unlike RAMBO and DynaStore which are purely replication-based (each server stores a full copy of the data), Ares employs *erasure coding* [2] to replicate data more efficiently, providing fault-tolerance equivalent to replication but at a fraction of the storage cost.

Secondly, Ares was the first algorithm to allow dynamic reconfiguration in an erasure-coded storage system. It separates the concerns of reconfiguration and read/write operations through a modular design. In fact, Ares' architecture consists of three components: (i) a reconfiguration protocol, (ii) a read/write protocol, and (iii) a set of data access primitives for interacting with the coded fragments. This modular approach means that the system can introduce new configurations concurrently with ongoing reads and writes, without service interruption. Ares ensures that configuration changes are applied in a linear order, maintaining a prefix property on the configuration sequence. Every new configuration is conceptually an extension of the previous configuration sequence, which prevents the kind of incomplete or diverging reconfiguration histories that could occur in earlier systems. This property simplifies the reconfiguration process and guarantees that the system state converges smoothly as membership changes

2.3 Large Objects and Fragmentation in Shared Memory

Early DSM algorithms assumed that each data object is a small unit (for example a single word or database record). If an application needs to share a large file or a huge array, one straightforward approach is to treat the entire file as one atomic object. This, however, is extremely inefficient. Every update would send the entire file over the network, and concurrency would suffer because two clients cannot update different parts of the file independently each update concerns the whole object. A logical step is to fragment large objects into smaller pieces that can be updated and stored separately. For example, a 10 MB file might be split into 100 blocks of 100 KB each, distributed across different servers. This raises the question: *can we allow operations on individual blocks while still providing a strong consistency guarantee on the file as a whole?*

Simply chopping a file into independent blocks and treating each block as a separate atomic register does not automatically yield a consistent file object in the linearizability sense. The difficulty is in preserving a meaningful order of operations on the entire file when different blocks may be written concurrently. If we simply make each block write linearizable on its own, we still need to define what it means for the whole file operation history to be linearizable. There needs to be a single global ordering of file level operations consistent with the ordering constraints of block operations. This leads to the notion of fragmented consistency.

Fragmented linearizability is a consistency criterion defined to capture exactly this situation [1]. It extends linearizability to composite objects composed of multiple fragments. In a nutshell, a fragmented object can be seen as a list (or set) of smaller objects (fragments), and an operation on the fragmented object may involve one or many of its fragments. Fragmented linearizability requires that there is a global order of all operations on all fragments such that each fragment's sub-sequence is consistent with that fragment's linearizability, and moreover, operations that logically pertain to the same high-level operation on the whole object are grouped appropriately in the order. Intuitively, if two writes happen on different blocks of the file, a fragmented linearizability guarantee will ensure that from the perspective of the file, it appears as if those two writes happened in some order (even if concurrent) and that any read of the whole file will see the result of either both writes or neither (depending on timing), not a partial mix that violates a sensible version of the file.

In practice, implementing a fragmented linearizable object often means adding a form of coordination or versioning at the higher level. One approach is to treat a write to the file as a transactional group of writes to blocks, which must all be serialized as one atomic operation relative to other file writes. However, that can reintroduce the serialization bottleneck we tried to avoid. A more flexible approach, introduced by Fernández Anta *et al.* [1], is to allow truly independent block operations while carefully restricting what a "read" of the whole file can observe. Specifically, they introduce coverable fragmented objects, which combine the idea of fragmentation with a newer consistency concept called *coverability*.

Coverability is a consistency guarantee that augments linearizability for versioned objects [5]. In a coverable object, every value is associated with a version number (or timestamp). Reads return the latest version and value, and writes are required to specify the version they intend to extend. A write in a coverable object will only take effect if it is writing "on top of" the current version of the object, if the object has advanced to a newer version in the meantime, the write is aborted. This mechanism prevents a write from overwriting a value that it did not know about effectively disallowing blind writes that would cover a more recent update. In the context of a file, coverability ensures value continuity: if a file has evolved to version 5, any write that was based on version 4 will be refused once version 5 is in place, so we never accidentally revert or lose the data from version 5. Instead, that write would realize it was stale and could be retried or treated as a no-op. Coverability thus suits use-cases like files where we prefer to append or update in order, rather than allow older writes to clobber newer content.

By combining fragmentation and coverability, we can attain a very robust model for large objects. Each block of a file can be stored as an atomic, linearizable register that is also versioned (coverable). The file as a whole is a linked structure of blocks. A write to the file may result in creating a new block or updating an existing block; this will be done with knowledge of the current file version. If another client concurrently writes elsewhere in the file, the two sets of block operations can occur in parallel. Each will either succeed entirely, advancing the file to a new version, or will be prevented from interfering with a concurrent update that beat it. The result is that the file's consistency (in terms of keeping all its pieces in sync as one object) is preserved, while allowing a high degree of concurrency on different fragments. Recent research prototypes have demonstrated that this approach can significantly boost throughput for operations on large objects. For instance, Anta *et al.* [1] implement a distributed file by representing it as a linked list of coverable blocks and show that concurrent writes to different blocks proceed without blocking each other, which reduces overall latency and improves bandwidth utilization.

A concrete system embodying these ideas is CoBFS (Coverable Block File System) [1]. CoBFS is a prototype distributed file system where each file is implemented as a fragmented object: an ordered list of blocks, each block being a coverable linearizable sub-object. CoBFS employs a modular architecture with two layers. The upper layer is the Fragmentation Module (FM), which handles dividing a file into blocks, naming them, and reassembling them for reads. The FM uses techniques like rolling hashes and sequence matching to identify how to split file data and to detect changes efficiently. The lower layer is the Distributed Shared Memory Module (DSMM), which provides the actual read/write operations on individual blocks, treating each block as a tiny object in a distributed shared memory. The DSMM in the original CoBFS was built on a static set of servers running an optimized variant of the ABD protocol for coverable objects. This yields a coverable**,** linearizable fragmented object implementation: the file can be read or written and will behave like a single versioned object that clients modify in a linearizable fashion.

2.4 Combining Dynamic Reconfiguration with Fragmentation

The two lines of research discussed above, dynamic membership and large-object fragmentation, were largely developed in parallel but combining them is essential for real-world DSM systems. A storage service in a volatile environment must reconfigure to replace failing servers or to incorporate new ones. At the same time, it must handle increasingly large data with high throughput. The state-of-the-art solution that unifies these aspects is presented by Georgiou *et al.* [2], which shows an algorithm to integrate a dynamic reconfigurable storage layer with a fragmented, coverable object layer. This effort builds upon the previously described CoBFS and the Ares dynamic atomic storage algorithm to create a new system sometimes referred to as Fragmented Ares or CoAresF.

In the design of Georgiou *et al.*, the DSM module is extended to support reconfigurations on the fly, like Ares does, but now each operation could involve multiple blocks. One of the main challenges was ensuring that the versioning logic (coverability) and the fragmentation ordering work correctly even as servers are added or removed. In their dynamic system, called CoAresF, they maintain the property that every new configuration of servers knows about the latest version of each block from the previous configuration (achieved by quorum overlaps and transfer of metadata). Additionally, they modify the Ares algorithm to handle versioned writes. The result is a coverable atomic memory that is reconfigurable and that supports large, fragmented objects.

By integrating fragmentation with reconfiguration, CoAresF addresses practical hurdles. For example, consider a file being actively appended by many clients while simultaneously some storage servers crash and new ones are brought in. With only fragmentation the system could handle the concurrent appends but would halt if enough servers crashed. With only dynamic reconfiguration the system could survive crashes but would serialize writes to the file or transfer entire file contents on each write, becoming a bottleneck. The combined approach can do both: seamlessly migrate blocks to new servers while allowing different parts of the file to be updated and read in parallel.

In summary, a Dynamic Fragmented DSM combines the strengths of dynamic reconfiguration protocols with the advantages of fragmented, coverable objects. The key ideas of Ares have been married with the key ideas of CoBFS splitting objects for concurrency and using versioned writes for safety. This makes distributed shared memory practical for modern systems, which demand both fault-tolerance in dynamic environments and high throughput for large data. Notably, CoAresF is utilized in the backend of our platform, providing a robust and efficient foundation for handling dynamic, large-scale distributed shared memory operations.

Chapter 3

Platform Description

3.1 Requirements and General Overview 13

3.1.1 Functional Requirements 14

3.1.2 Non-Functional Requirements 15

3.2 User Categories 16

3.3 Platform Constraints 17

3.4 Architectural Overview 19

3.5 Development Tools 21

3.5.1 Laravel (PHP Framework) 21

3.5.2 Bootstrap (Front-End Framework) 23

3.5.3 Alpine.js (JavaScript Library for Interactivity) 24

3.5.4 MySQL (Database Management System) 25

3.5.5 XAMPP (Development Server Environment) 26

3.6 Security Considerations 27

3.7 Database Structure 28

This chapter provides an overview of the front-end system requirements, user classes, and technical basis. We begin with enumerating the functional and non-functional requirements, assumptions and constraints, followed by the description of how various classes of users utilize the system. We then delve into the top-level architecture, describing the client side as well as server-side features. Lastly, we look at the core tools and technologies used and we explain how each contributes to security, efficiency and to creating a manageable solution.

3.1 Requirements and General Overview

In this section we list the requirements of the desire system and describe its general functioning and design. The system is designed as a web-based environment where end-users are able to perform necessary operations with a browser. To meet these requirements, the application is built with cutting-edge web technologies: front-end development using the Laravel framework (a PHP web framework) [15], user interface is built using Bootstrap for responsive UI [7], and client-side interaction is supported by Alpine.js [8]. Data storage takes place in a MySQL relational database [9], and the application is built and tested in a local server environment with XAMPP [10]. By leveraging this technology stack, the design should fulfill all functional requirements and possess a great degree of security, performance, and usability.

3.1.1 Functional Requirements

Functional requirements of the system specify the particular things users should be able to do within the web-based interface. The platform will be a front-end web application built on Laravel and should present a secure, responsive, and user-friendly interface for managing files and folders.

Users should be able to login using correct credentials and, once validated, see their own personal directory of files. The user interface should allow users to upload new files, make new directories, access nested directories, rename existing directories or files, and delete them. These should send with the backend so updates are always reflected in all nodes participating in the distributed system.

The system needs to have two browsing modes to be shown: a list view with information details about each file, and a grid view that shows items in a tile format. The system needs to provide the ability to switch between the two modes without causing any inconvenience to the user.

With regards to file sharing, the application must provide the users with the ability to share folders or files. Users should have the ability to select other registered users and grant them read-only or read-and-write access.

Feedback upon user activity is crucial. When the user uploads a file, creates a directory, or performs any other operation, the system must respond with the appropriate message of success or failure of the operation. If there is an issue, like uploading a file that already exists or doesn't have permissions, the platform needs to specify the issue clearly through alerts or modal dialogs. The app should also guide users through steps of conflict resolution, such as giving users the choice to rename or overwrite when duplicates are found.

The site should also handle user sessions correctly. Authenticated users alone should be able to access the system, and permissions should be checked before a file operation should be allowed. Additionally, file properties such as file size, last modification date, and ownership details should be available in the user interface.

In short, the platform must offer an end-to-end, sophisticated file management experience through a Laravel interface. It must enable secure login, easy file browsing and editing, access-sharing with roles, clear and consistent feedback, and transparent integration with the distributed storage backend. All these aspects must be implemented in such a way that it provides ease of use even to those who possess minimal technical knowledge.

3.1.2 Non-Functional Requirements

Apart from the above features, the system also needs to fulfill various non-functional requirements to ensure that the system is robust, secure, and friendly for the users. Among these, security is an extremely critical non-functional requirement. The platform must protect user data from unauthorized access. The system must safely store all user credentials and ensure secure authentication and session handling. Data validation needs to be carried out at all times during the application in order to prevent malicious input, and the system needs to be designed so that typical web threats such as injection attacks and cross-site scripting are circumvented. These requirements are discussed further in more detail in Section 3.6, but it is emphasized at the requirements level that the application must adhere to best practice so that data is kept confidential and integrity is preserved.

Another significant non-functional requirement is usability and responsiveness. The system's interface should be simple and easy to use for users. This is accomplished by having a straightforward layout and displaying content in a logical manner. Bootstrap helps meet this requirement by providing a responsive design [15]. Design consistency and accessibility are also considered, and the app becomes easy to use for a large number of users. Responsiveness of the platform isn't only layout, but performance as well. Pages need to load within a reasonable time period, and the interface needs to respond quickly to user input. Using Alpine.js for small interactive behaviour assists with giving a responsive user experience without the overhead of a heavy front-end framework [8].

Performance and scalability constitute another non-functional requirement. The application must handle numerous simultaneous users acting on the system without unacceptable loss of response time. The load is kept low for now but the design is scalable. The Laravel architecture and the concurrent transaction feature of MySQL guarantee that the system would be able to take increases in data volume or user base to some degree. For instance, Laravel's efficient database abstraction (ORM) and query optimization together with proper indexing in the MySQL database ensure the performance is tolerable for usual queries. In case of a need, the system may be hosted on a more sophisticated server or horizontally scaled. Therefore, although end-of-scaleability is not an initial requirement, the design makes it possible in the future.

Lastly, extensibility and maintainability are also crucial considerations. The system is designed upon the principles of the Model-View-Controller architecture (as described in Section 3.4), which intrinsically supports separation of concerns. This means that any developers that come afterward could modify or add to one area of the system without frightening the rest of the system-this makes it easier to maintain. It should be well documented, and strict adherence to a coding standard should be enforced so that it is easy to understand. Since these are popular frameworks and tools, a new developer having a good grasp on these technologies will take only some time to be effective. In essence, the non-functional requirements like, security, usability, performance, and maintainability-have heavily influenced the system design decisions, so that the delivered system is not merely functional but one that gives room for reliability and efficiency when it comes to operation.

3.2 User Categories

A crucial element in shaping this platform was the decision to designate specific user types based on each individual's relationship to any given file or folder. In the system's simplest form, there are two primary categories of users: the owner and the shared user. Defining these roles at the outset helped standardize access control, making it intuitive for the owner to control distribution and for the shared user to understand their privileges.

The first category is the *owner*, who is the individual responsible for uploading or creating a file or folder. By default, the owner holds full permissions over that item, including the ability to rename, delete, download, or share it with other users. In practice, the owner acts as the ultimate authority for that content. The system ensures that owners can manage their materials effectively without interference. Any subsequent activities, such as renaming or removing the file, must pass through checks confirming that the request is issued by the recognized owner.

The second category is the *shared user*, who has been granted access to view or edit a file that belongs to someone else. In actual practice, the owner can share an item either with read only or read write privileges, depending on the needs of the collaboration. A shared user with read only rights can see and download the content but cannot rename or delete it. By contrast, a shared user with read write access can partake in more comprehensive editing actions like renaming files. The rationale behind this distinction was rooted in real world scenarios, where certain files might need open collaboration like a shared text document while others must remain protected from change, such as archived data or official documents.

Having a pre-established set of user roles and permissions benefited the platform in several ways. First, it offered an open hierarchy of ownership so that there is no ambiguity about who really "controls" any resource. Second, it limited disruptive behavior to a small set of authorized users. Shared users lacking write access cannot inadvertently delete or corrupt data beyond retrieval, at the same time, owners remain free to grant more robust permissions where close collaboration is desired. Third, this architecture kept the system easy enough for typical use cases uploading and sharing sets of documents with colleagues or team members while still adhering to stricter access restrictions.

Finally, this arrangement promoted a more fluid user experience. Owners viewing their own content see the full suite of management tools at their disposal rename, move, delete, share whereas shared users only see the features their assigned privilege level permits. This contextual interface design makes it less likely that a user misinterprets their capabilities. By distinguishing owner privileges from those of shared users up front, the platform preserves clarity in how each item can be used or distributed.

3.3 Platform Constraints

Every software system must operate under some platform limitations that include the environment requirements, compatibility matters, and other restrictions by virtue of technologies employed. Platform limitations for this web-based program are primarily driven by the requirement to host a Laravel program and a MySQL database. First and foremost, the server environment must include support for PHP (the language upon which Laravel is founded) at the required version. XAMPP development logically means that the system is run on an Apache web server with PHP and MariaDB/MySQL installed [9]. A limitation thus is that the deployment environment should have an Apache (or similar) web server and MySQL or MariaDB [16] database installed. Should the system be moved from the development machine to a production server, a LAMP stack (Linux, Apache, MySQL, PHP) or WAMP stack (Windows, Apache, MySQL, PHP) would need to be used to duplicate the XAMPP environment.

Resource constraints are also considered. Since the application is run on a web server, the server must have sufficient resources CPU, memory, and disk space to run the application logic and the data stored. Since Laravel is a full-featured framework, it uses more memory than a plain PHP script, so the server must have sufficient RAM to run the framework overhead as well as concurrent usage. The database will store perhaps all persistent data so disk space is a constraint based on expected data size. If the system will be storing images or files uploaded by users, then additional storage must be planned for, or an external storage service must be integrated. For now the data size is small, so a standard development machine with a few gigabytes of free space is sufficient. However, the design anticipates that if usage grows, the site can switch to a higher capacity server or scale out through load balancing.

Another platform limitation is related to network connectivity. In XAMPP development, the application will be locally addressable (via http://localhost/). To use productively, the server must be network-visible to the target user group this could mean hosting on an intranet or on the public internet. Performance and response times experienced by users will be at the mercy of network conditions. If deployed publicly, the system should be accessed over HTTPS for security, meaning there will be a need to configure SSL/TLS certificates on the server side.

Finally, compatibility limitations can include having specific versions of frameworks or libraries included. For example, the version of Laravel being utilized might have to be compatible with the PHP version on the server. Similarly, the MySQL version should be a version that is compatible with Laravel. XAMPP's package comes with MariaDB which is an alternative to MySQL, and Laravel treats it precisely the same as MySQL. So, one of the constraints implied is that the database system, if not MySQL, must be MariaDB or another supported SQL database. It would not be feasible to utilize a completely different type of database without altering the design, as the system relies on a relational schema. In general, building within these constraints has affected the implementation and ensures that the system will operate consistently in its intended environment.

3.4 Architectural Overview

The system is deployed with a multi-tier design that separates the client interface, server-side application logic, and database. At a top-level perspective, it follows the classic three-tier web application pattern: the presentation tier (front-end user interface within the browser), the application tier (backend logic on the web server), and the data tier (database). Particularly, this architecture is implemented on the server side by using the Laravel framework, which enforces the Model-View-Controller (MVC) architectural pattern. MVC is a software design pattern that organizes program logic into three interacting components the Model, the View, and the Controller in order to isolate concerns and enable modular development [11]. By employing MVC in Laravel, the system retains data handling separation, UI, and control flow tidily within the codebase.

In MVC architecture, Controllers are the pieces of the system responsible for dealing with incoming requests from users. Whenever a user takes some action on the system that triggers an HTTP request towards the application within Laravel, an inbuilt router within Laravel delegates every request URL to a specific controller and action. The controller is where the request enters on the server, it processes the input of the user, applies the necessary business rules or logic, and then determines the response. One of the responsibilities of the controller is that it has to talk to the Model layer. In Laravel, Models are representations of data entities. In such a case, a model includes the data as well as business rules or logic regarding that data. For instance, there exists a User model for users of the system. The controller can ask the model to retrieve certain records, or to insert/update records by virtue of user requests. Laravel's ORM (Eloquent) simplifies these interactions by translating model operations into underlying database queries automatically.

While the View component is responsible for showing data to the user. Views, in a web application, are usually HTML templates filled with dynamic information. A templating engine (Blade) is used by Laravel to help produce HTML pages. Once data is gathered by a controller, it passes such data to a view. The view template now integrates the data into a presentation that the client can understand usually HTML, CSS, and JavaScript that will be sent back to the user's web browser. The separation between these points here is significant: models don't concern themselves with how data is represented, and views don't concern themselves with fetching and processing data each does what it needs to do.

By adhering to the MVC pattern, the system's architecture is more maintainable and scalable. One advantage is that multiple developers are able to work on different components simultaneously with little overlap. In addition, if how data is displayed needs to change that can be done within the view layer without modifying the underlying data retrieval in the model or controller. Conversely, if we alter the database schema or rules of business, the changes occur in models and controllers, and the views are comparatively unchanged. Modularity of this type is a simple benefit of the MVC design [11].

Another note on the architectural overview is where client-side script (Alpine.js) comes into play. The HTML views rendered often include JavaScript to add more interactivity after page load. This isn't a violation of the MVC pattern rather, it remains within the View layer since client-side scripting is presentation logic. An example of this is a view with an Alpine.js component for a collapsible menu or dynamically changing form fields. Alpine.js is executing within the browser to handle these interactive elements, but from the server's perspective, it's just a component of the view being presented. Server-side controllers are still the primary handlers for any significant actions. When asynchronous interactions are necessary, the architecture can employ AJAX requests or Laravel's API routes that still reach controllers and maybe output JSON data, which front-end JavaScript can subsequently use to refresh the page without a full reload. This is a variation of MVC as a client-server interaction where client-side script functions as a miniature controller on the client, yet any persistent updates go through controllers and models in the server.

Security and session management are included in the structure. Laravel includes middleware (an encapsulating layer around controllers) which can be considered part of the architectural scheme. Middleware can trap requests before they hit a controller, for example, there can be one that authenticates a user or checks if the user is an administrator before allowing access to certain controllers. This means that the architecture also contains layered security checking: requests go through authentication checks, then controllers, then possibly models, and finally render a view. Sessions and authentication status are handled in the server-side. This maintains MVC flow in sync with the user login status and role. For instance, an admin controller might be protected by an "admin" middleware so that if a non-admin user somehow attempts to access it, the request gets denied early.

In short, the system's architectural description is that of a healthy MVC web application. This is heavily influenced by the utilization of Laravel, as it provides a climate where architecture is controllers, models, and views. The layered approach provides a clear flow: requests from the browser reach routes and controllers, controllers invoke models to handle data, and controllers send views back to the browser. The database is abstracted by models, and the front-end is served cleanly via views. This design not only meets the current needs but also serves a good basis for the implementation of future changes or system extensions.

3.5 Development Tools

The development of the system utilized a collection of modern tools and frameworks that were all chosen for their suitability to deliver what the project required and simplify the implementation. It is here that the key technologies and pieces of software used are described, and how they all collaborate to finish up the system. Through the use of established libraries and tools such as Laravel, Bootstrap, Alpine.js, MySQL, and XAMPP, the project takes advantage of proven solutions rather than rewriting fundamentals from scratch. Every tool or framework is discussed below along with the justification for why it has been utilized in this project.

3.5.1 Laravel (PHP Framework)

Laravel was used as the primary back-end framework of this project. Laravel is an open-source PHP web application framework featuring expressive syntax and an extensive collection of features that speed up the development of web systems by far [6]. Laravel is an implementation of MVC and provides beautiful separation of concerns as described above. Laravel accommodates most activities in web applications such as routing (mapping from code to URLs), form handling and validation, session management, and user authentication into it. This premium set of functionality resulted in much boilerplate code not having to be written from scratch, for example, setting up a secure authentication system was more a matter of an configuring Laravel's authentication module rather than coding all the authentication code by hand.

One of the strongest arguments in favour of Laravel is its Eloquent ORM for database interactions. Eloquent allows developers to interact with database records using model classes in PHP instead of writing raw SQL queries. Not only does this speed up development, but it also reduces the chance of SQL errors and adds security by design. In Eloquent, Laravel itself handles parameter binding in queries, which keeps SQL injection attacks under control. The package also has a database migration system, which was used to define the database schema in PHP and manage its versioning. It made incrementally developing the database design over the course of development and quickly rebuilding the database structure easy to do.

Laravel's focus on developers paid off with this project, as it kept the code clean and readable. For example, routing is simply defined and in a natural manner, and models and controllers have convention so that code becomes self-documenting. Documentation for the framework is complete and up-to-date, which proved to be beneficial whenever implementation assistance was required. A further justification why Laravel suited our needs is because it features an extensive ecosystem and community. There are many packages that can be utilized to enhance functionality in Laravel, something that could be called on if needed. While these weren't necessarily used throughout this project, knowing that this capability was open to providing additional functionality brought reassurance that the framework would be able to accommodate future requirements as well.

Laravel was the foundation of the server-side application due to its robust infrastructure, built-in features, security benefits, and extensive community. It offered the capability of compliance with industry best practices standards as a box. Laravel's usage reduced the development time for complex features to a large degree and assisted in maintaining the application code clean and maintainable.

3.5.2 Bootstrap (Front-End Framework)

Bootstrap has been utilized as the front-end CSS framework in order to design a responsive and consistent user interface. Bootstrap is the most used HTML, CSS, and JS framework to develop responsive projects on the web [7]. It was initially created by Twitter, providing a complete set of pre-styled components (e.g., navigation bars, buttons, forms, modals.) and a grid layout. Bootstrap was employed as responsive design was required and it was desired to speed up the process of CSS designing. Rather than defining all the styles manually, the pre-defined styles offered by Bootstrap gave the application a new and standardized look with reduced efforts.

The use of Bootstrap actually improved the usability and accessibility of the interface. Bootstrap comes with in-built styles that give sufficient contrast and readable font, which assisted in putting in place basic accessibility guidelines. The components were utilized wherever possible to ensure that there is a consistent look and feel across the application.

Another advantage of Bootstrap is that it is cross-browser compatible. Web browsers would process HTML/CSS in certain situations differently, yet Bootstrap styles get normalized and get tested on prominent browsers. That reduced the odds of encountering browse specific layout errors while coding. Furthermore, Bootstrap being standard means there were ample templates and samples already present for reference. During development, if there was a particular interface pattern that needed to be achieved, there would be examples that one could look up in the Bootstrap documentation or community forums that could be used and adapted to our application. This ultimately saved much front-end development time as well as the assurance that the UI follows established patterns.

It's also interesting that Bootstrap's class and component usage is fully compatible with Laravel's Blade templating. We could create dynamic content on the server side and wrap it in Bootstrap classes without any conflict. Bootstrap is so flexible that we can extend or override some styles via a different CSS file to customize appearances, but Bootstrap met most requirements. In short, Bootstrap was a necessary front-end component of the application that gave a professional appearance and good responsive quality to the application while being developer-time-saving friendly because of its good pre-defined styles and components.

3.5.3 Alpine.js (JavaScript Library for Interactivity)

Adding client-side interactivity Alpine.js was utilized. Alpine.js is a lightweight JavaScript framework enabling developers to create interactive behaviors directly in the HTML markup [8]. One may consider it a very minimalistic alternative to some bulkier frameworks that provide reactivity and component-based functionality, albeit with the least amount of resources. The thought behind choosing Alpine.js was the kind of interactions required for this project: pretty simple UI behaviors (show/hide element, toggle class, slightly enhancing forms) that do not call for the full-blown single-page application framework.

It does this by adding special HTML attributes to various elements that declare behaviors. In this case, Laravel would be rendering pages on the server (Blade templates), and Alpine would be taking over to make those pages interactive as soon as they are loaded in the browser. The potential experience gain comes when you avoid a full page reload for a little interaction and also when you give immediate feedback to a user action.

One of the uses of Alpine.js here was for form input and validation feedback on the client side. While Laravel can handle solid validation on the server side, Alpine enables the client-side validation mechanism that provides users with instant feedback while they are filling out the form. In addition, it is easy to keep the code in the HTML-template, which is convenient for Alpine. There is also a usage for dynamic UI components such as modals.

Alpine.js is not interfering with Laravel or Bootstrap, rather, it serves as an excellent compliment for them. Alpine is scoped to the browser and impacts primarily the View layer of our MVC architecture. It is so minuscule in footprint that it neither significantly slows down page loads nor puts a strain on the client's device. This was important because we wanted the client side to be as light as possible with most of the logic existing on the server while offering a few cool JS-aided front-end features to the user. Moreover, Alpine.js has no requirement for a build step, it can simply be dropped into the <script> tag anywhere and used right away.

To sum up, Alpine.js turned out to be a great solution for delivering interactivity in the app. It offered the required dynamism to the user interface and was just simple enough and light enough for us. The alternative of heavier front-end frameworks would have felt quite out of place and out of scope with the actual project, and seemingly contrary to the desire for clarity and lean simplicity in the codebase.

3.5.4 MySQL (Database Management System)

MySQL was chosen as the database management system for storing data of this project. MySQL, in fact, is a widely used open-source RDBMS based on which almost all successful web applications are dependent on [9]. MySQL tables store all of the persistent data for the system such as user data, content records, and any transactional logs in the system. Of course, with structured data needing a relational path (foreign key) binding two or more entities, practically a relational kind of database is the most natural choice to make. MySQL fits in quite well with Laravel, thanks to the Eloquent ORM and database query builder previously mentioned.

One such reason for choosing MySQL has been its maturity and support: Being in existence for decades, it has passed the test of time in numerous production instances, including big ones. These imply that elements of common requirements such as indexing, complex querying, transactions, and concurrency are all taken care of in a robust way. For example, the system uses MySQL to enforce data integrity via constraints. These relational features, which MySQL offers, ensure data consistency in the database even when multiple operations are being performed. Also, given that the project might involve very frequent read operations of data, MySQL is well optimized in terms of performance for both read-heavy and write-heavy operations-a fact contributed to by the query engine and the use of indexes.

Weighing expertise and tooling, too, went into the decision for MySQL. Various tools can be used to manage the MySQL database, from command-line clients to GUI management applications such as phpMyAdmin (a popular case, since it comes bundled with XAMPP). Throughout the development phase, it helped to inspect the database, run queries, and undertake basic administrative tasks such as backups. Laravel migrations and query logging ensured the MySQL interactions were correct, but it was very good to be able to verify data in MySQL independently.

To conclude, MySQL served for data persistence within the project. Being known as "the world's most popular open source database," this title is truly deserved, It offered a stable, fast, and secure repository for our application data [9]. Also, using MySQL meant aligning ourselves with industry standards for web application development, supporting database scaling or future maintenance through community and enterprise know-how. Laravel and MySQL made transitioning from data models in code to actual database records very smooth.

3.5.5 XAMPP (Development Server Environment)

XAMPP was used as the development server environment for installing and testing the system. XAMPP is a free and open-source cross-platform web server solution stack package that comes preconfigured with the Apache HTTP Server, the PHP interpreter, and MariaDB, accompanied by other tools such as Perl and phpMyAdmin [10]. Putting it simply, XAMPP took care of installing all the server components required for running a web application right onto the development machines with a single click, locally creating a platform for the web application.

The ease of XAMPP got everything running fast. Just one installation set up Apache, PHP, and MySQL with consistent configuration instead of installing each separately and manually configuring them. It has a control panel from which all services can be started and stopped simply. The control panel was used to run Apache and MySQL, which basically hosted the Laravel application and database. Since XAMPP is cross-platform, it could be run on Windows, Linux, or macOS, making it work on any machine.

The interface, phpMyAdmin, being a MySQL interface running within a web browser, also came bundled with XAMPP, it was helpful during the development phase having manual inspection of tables, test queries, and import/export in existence. For example, with phpMyAdmin, after migrations via Laravel created the schema, the presence of the tables and columns was checked to be set properly. It also allowed an indirect way of debugging, able to peek at the data stored after certain operations. Even though Laravel comes with database seeding tools, sometimes we wanted to get a bunch of sample data solidly in one go just so you can test a page and do so a lot faster via phpMyAdmin.

Though most often seen as a development tool, a XAMPP setup gives you an experience near production. It should be noted that XAMPP is not meant to be run standalone as production, if set so, it would defeat the openness that was intended with development. However, in development, XAMPP is great because of simplicity and being feature complete. In the end, XAMPP set up the environment to develop and test on an integrated fashion by bundling all needed services, tools, and so on. It allowed us to concentrate on writing the application rather than environment setup and ensured that testing conditions locally were as close as possible to target on deployment, thereby increasing the precision of our development and testing efforts.

3.6 Security Considerations

Security has been a key priority and protection techniques were developed to defend the application and its data from threats. Some of the major security issues such as authentication, authorization, data security, and common web vulnerabilities are discussed with the aim of maintaining confidentiality, integrity, and availability of the system and the relevant information.

The system requires users to log into it for authentication and authorization for most functions, particularly for any that modify data. For safe login and session management, Laravel's built-in authentication module was used. User passwords are never stored in plain text, rather, before being stored, the password hashes are created using a strong one-way hashing algorithm called bcrypt. Bcrypt is recommended because it is computationally expensive to invert, and it includes a salt as well as a configurable "work factor" that can be adjusted over time to maintain security with the advent of faster hardware. Therefore, even in the unlikely event of an attacker reading the user database, the attacker cannot easily ascertain the actual password. Moreover, Laravel's authentication system uses encrypted cookies to maintain the session in a secure fashion and also uses the "remember me" functionality securely.

Other ways we have prevented against some common Web-Based Vulnerabilities were the following. On SQL Injection, the use of prepared statements and Eloquent ORM inherently guards against SQL injection by escaping inputs properly. Therefore, nowhere in the application do we concatenate unchecked user input into raw SQL queries. To be sure, the controllers do not directly work with raw queries but call only Eloquent model functions or the query builder provided by Laravel, both of whom parameterize the queries under the hood. Hence, the user-supplied data is never cast to execution inside the database, thus avoiding injection attacks. By default, Laravel's Blade templates escape output variables to HTML, meaning if a user were to insert a script tag as an input, it would render as text on the page instead of executing. Steps were taken to guarantee that our code does not circumvent this built-in escaping. This method also prevents malicious content uploaded by one user from impacting the view of another user.

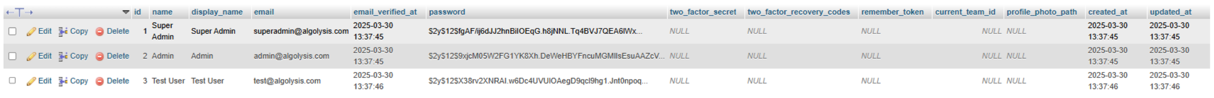
Cross-site Request Forgery (CSRF) was a further issue considered, but luckily Laravel automatically provides CSRF protection for forms and routes. Every form generated for user actions contains a hidden CSRF token field, and Laravel's middleware rejects all POST, PUT, and DELETE requests that do not provide a valid token. This token is bound to a session of a user so that no valid request could be forged by an attacker claiming to be an authenticated user without token knowledge. By this defense, the system prevents the exploitation of CSRF attacks wherein attackers may convince a logged-in user's browser to execute further unintended requests. Only forms from our application carrying this token will be considered valid on the server.

In the end, security considerations were taken into account during system design. The system handles risks of a wide variety by trusting widely accepted frameworks, following best practices in password hashing, and input validation. We understand that security entails continuous evolution with new threats, so the system's architecture and implementation are thought of as a solid base for secure web applications.

3.7 Database Structure

The system's database structure, designed using the relational model, reflects the logical organization of data as required by the application. The schema was normalized to the highest possible level of normalization using MySQL RDBMS to avoid redundant data and to enforce referential integrity between distinct entities. The design process first considers the main entities (tables) to be involved in satisfying the system's requirements, considers the relationships between these entities (one-to-one, one-to-many, many-to-many as appropriate), and finally defines the attributes (columns) of each entity along with their data types and constraints.

On a more high-level, higher importance was given to the tables representative of key domain concepts: for example, a users table exists to hold user account details, while another table named dsm_directory_items stores the files and folders that users create and interact with within. The users table has fields such as id, which is unique for each user (primary key), name, display_name, email, passwordHash, and creation/update timestamps for the account. Assigning unique identifiers to users would give other tables the ability to reference those users by foreign key, and this is essential in the relationship of data. Figure 3.1 shows the table of users, along with all of its values and fields, as represented in phpMyAdmin.

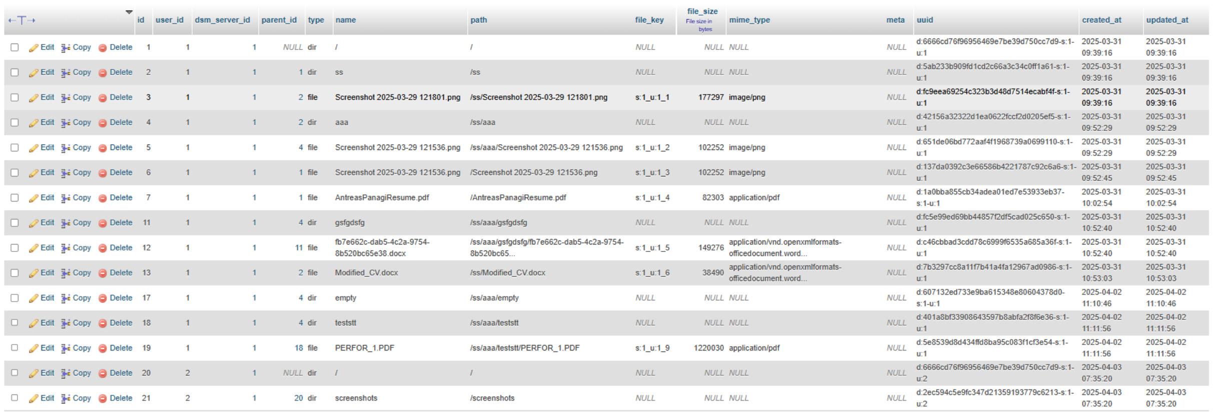


	id	name	display_name	email	email_verified_at	password	two_factor_secret	two_factor_recovery_codes	remember_token	current_team_id	profile_photo_path	created_at	updated_at
Edit Copy Delete	1	Super Admin	Super Admin	superadmin@algolysis.com	2025-03-30 13:37:45	$2y$12$fgAF/ij6dJJ2hnBilOEqG.h8jNNL.Tq4BVJ7QEA6lWx...	NULL	NULL	NULL	NULL	NULL	2025-03-30 13:37:45	2025-03-30 13:37:45
Edit Copy Delete	2	Admin	Admin	admin@algolysis.com	2025-03-30 13:37:45	$2y$12$9xjcM05W2FG1YK8Xh.DeWeHBYFncuMGMllsEsuAAZcV...	NULL	NULL	NULL	NULL	NULL	2025-03-30 13:37:45	2025-03-30 13:37:45
Edit Copy Delete	3	Test User	Test User	test@algolysis.com	2025-03-30 13:37:46	$2y$12$X38rv2XNRAI.w6Dc4UVUIOAegD9qcl9hg1.Jnt0npoq...	NULL	NULL	NULL	NULL	NULL	2025-03-30 13:37:46	2025-03-30 13:37:46

Figure 3.1 – users Table

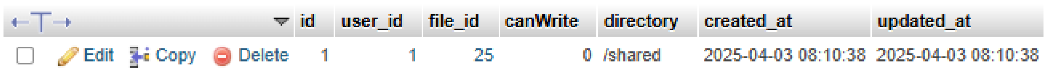
The dsm_directory_items table holds the primary data that the application manages. Each item has its own id (primary key) and fields representing its content or properties. Importantly, it contains a foreign key that associates it with a user in the users table (user_id). This establishes a one-to-many relationship: one user can have many items, but each record is tied to a single user. We enforce this relationship using a foreign key constraint in MySQL, so that a record cannot exist without a valid user. If a user were deleted, the foreign key constraint would cascade to delete the related records. Additionally, all items have a parent_id which point to the id of the folder that contains the file or folder depending of the type of the item, the root directory of each user has a value of null. There is also a type field which can have either "dir" or "file" as its values. Also there is a name field and a path that displays the entire directory path to the item. For files there is a file_size field that shows the size of the file in bytes and a mime_type field that shows the type of file. Lastly as all other tables timestamp columns (created_at and updated_at) are included, a convention that Laravel supports automatically. These timestamps are useful for both application logic (showing when something was last modified) and for maintenance (debugging and auditing changes over time). In Figure 3.2 the table dsm_directory_items is visible with all its fields and some values.

In addition to the main two tables, the schema also includes a pivot table named user_file which is useful for the sharing functionality of the platform. The table's fields include a unique id(primary key) and 2 foreign keys: user_id and file_id from users and dsm_directory_items respectively. There is also a Boolean field canWrite which decides if the shared user will have read only or read-write access. Lastly, there is a directory field which always has the value /shared and timestamps. Using this table the platform can decide to whom to display the shared file (user_id), which file is it (file_id) and the permission that the user has (canWrite). The user will be able to locate this file in the root/shared directory. The table user_file is shown below in Figure 3.3.



	id	user_id	dsm_server_id	parent_id	type	name	path	file_key	file_size File size in bytes	mime_type	meta	uuid	created_at	updated_at
Edit Copy Delete	1	1	1	NULL	dir	/	/	NULL	NULL	NULL	NULL	d:6666cd76f96956469e7be39d750cc7d9-s:1-u:1	2025-03-31 09:39:16	2025-03-31 09:39:16
Edit Copy Delete	2	1	1	1	dir	ss	/ss	NULL	NULL	NULL	NULL	d:5ab233b909fd1cd2c66a3c34c0ff1a61-s:1-u:1	2025-03-31 09:39:16	2025-03-31 09:39:16
Edit Copy Delete	3	1	1	2	file	Screenshot 2025-03-29 121801.png	/ss/Screenshot 2025-03-29 121801.png	s:1_u:1_1	177297	image/png	NULL	d:fc9eea69254c323b3d48d7514ecabf4f-s:1-u:1	2025-03-31 09:39:16	2025-03-31 09:39:16
Edit Copy Delete	4	1	1	2	dir	aaa	/ss/aaa	NULL	NULL	NULL	NULL	d:42156a32322d1ea0622fccf2d0205ef5-s:1-u:1	2025-03-31 09:52:29	2025-03-31 09:52:29
Edit Copy Delete	5	1	1	4	file	Screenshot 2025-03-29 121536.png	/ss/aaa/Screenshot 2025-03-29 121536.png	s:1_u:1_2	102252	image/png	NULL	d:651de06bd772aaf4f1968739a0699110-s:1-u:1	2025-03-31 09:52:29	2025-03-31 09:52:29
Edit Copy Delete	6	1	1	1	file	Screenshot 2025-03-29 121536.png	/Screenshot 2025-03-29 121536.png	s:1_u:1_3	102252	image/png	NULL	d:137da0392c3e66586b4221787c92c6a6-s:1-u:1	2025-03-31 09:52:45	2025-03-31 09:52:45
Edit Copy Delete	7	1	1	1	file	AntreasPanagiResume.pdf	/AntreasPanagiResume.pdf	s:1_u:1_4	82303	application/pdf	NULL	d:1a0bba855cb34adea01ed7e53933eb37-s:1-u:1	2025-03-31 10:02:54	2025-03-31 10:02:54
Edit Copy Delete	11	1	1	4	dir	gsfgdsfg	/ss/aaa/gsfgdsfg	NULL	NULL	NULL	NULL	d:fc5e99ed69bb44857f2df5cad025c650-s:1-u:1	2025-03-31 10:52:40	2025-03-31 10:52:40
Edit Copy Delete	12	1	1	11	file	fb7e662c-dab5-4c2a-9754-8b520bc65e38.docx	/ss/aaa/gsfgdsfg/fb7e662c-dab5-4c2a-9754-8b520bc65...	s:1_u:1_5	149276	application/vnd.openxmlformats-officedocument.word...	NULL	d:c46cbbad3cdd78c6999f6535a685a36f-s:1-u:1	2025-03-31 10:52:40	2025-03-31 10:52:40
Edit Copy Delete	13	1	1	2	file	Modified_CV.docx	/ss/Modified_CV.docx	s:1_u:1_6	38490	application/vnd.openxmlformats-officedocument.word...	NULL	d:7b3297cc8a11f7b41a4fa12967ad0986-s:1-u:1	2025-03-31 10:53:03	2025-03-31 10:53:03
Edit Copy Delete	17	1	1	4	dir	empty	/ss/aaa/empty	NULL	NULL	NULL	NULL	d:607132ed733e9ba615348e80604378d0-s:1-u:1	2025-04-02 11:10:46	2025-04-02 11:10:46
Edit Copy Delete	18	1	1	4	dir	teststt	/ss/aaa/teststt	NULL	NULL	NULL	NULL	d:401a8bf33908643597b8abfa2f8f6e36-s:1-u:1	2025-04-02 11:11:56	2025-04-02 11:11:56
Edit Copy Delete	19	1	1	18	file	PERFOR_1.PDF	/ss/aaa/teststt/PERFOR_1.PDF	s:1_u:1_9	1220030	application/pdf	NULL	d:5e8539d8d434ffd8ba95c083f1cf3e54-s:1-u:1	2025-04-02 11:11:56	2025-04-02 11:11:56
Edit Copy Delete	20	2	1	NULL	dir	/	/	NULL	NULL	NULL	NULL	d:6666cd76f96956469e7be39d750cc7d9-s:1-u:2	2025-04-03 07:35:20	2025-04-03 07:35:20
Edit Copy Delete	21	2	1	20	dir	screenshots	/screenshots	NULL	NULL	NULL	NULL	d:2ec594c5e9fc347d21359193779c6213-s:1-u:2	2025-04-03 07:35:20	2025-04-03 07:35:20

Figure 3.2 – dsm_directory_items Table



	id	user_id	file_id	canWrite	directory	created_at	updated_at
Edit Copy Delete	1	1	25	0	/shared	2025-04-03 08:10:38	2025-04-03 08:10:38

Figure 3.3 – user_file Table

To ensure data integrity and valid data, various constraints are placed at different levels: NOT NULL fields can not be left blank when it is a required field; UNIQUE makes sure no two records exist for accounts or other unique entities.

The database schema was implemented using Laravel's migration files, which literally are executable blueprints of schema. This approach enables the schema definition to be versioned alongside the code and move into different environments or rolled back if needed. Each migration creates a table or modifies an existing one according to the design intentions.

In short, the database design of the system is a relational schema that is shaped to the data requirements of the application while focusing on referential integrity, consistency of data, and supporting queries of the application. Thus, the distribution favors the application in storing, retrieving, and managing the data while keeping the operations consistent in terms of accurate data. The end result is a schema that is sturdy and forward-looking, capable of extending with another set of tables or even relationships should the system grow, with no need to undertake a major redesign of the existing structure.

Chapter 4

Implemented Platform

4.1 Existing Development 32

4.1.1 Login 33

4.1.2 Dashboard 35

4.1.3 User Profile 36

4.2 Main Page 37

4.2.1 List View 38

4.2.2 Grid View 39

4.3 Uploading Files 39

4.4 Actions 43

4.4.1 Rename 44

4.4.2 Delete 45

4.4.3 Download 46

4.4.4 Share 46

4.4.5 Info 49

4.5 Navigation 50

4.6 Messages 52

This chapter presents the developed platform with all of its functionality. After a comprehensive design process and careful requirements analysis, the implementation of the file management system took the form of a Laravel web application with a distributed shared memory backend. The following chapter shows the various pages of the platform, like the login and dashboard page, navigation and all of the possible actions that the user can complete.

4.1 Existing Development

The following section presents all of the functionality that was already implemented by the Algolysis team and required no further additional development. Basically, everything that has to do with user authentication, the dashboard page and the user profile page were implemented by the Algolysis team using Laravel.

4.1.1 Login

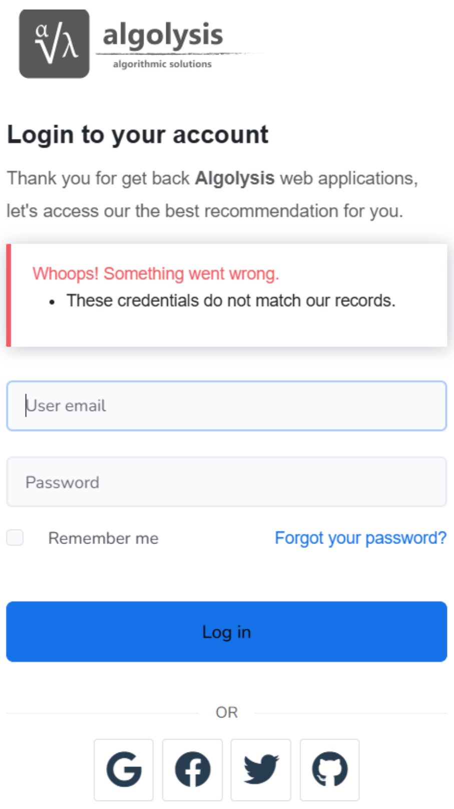
When users first arrive at the platform’s web address, they encounter a Login Page featuring a simple form requesting a Username and Password. Upon pressing “Login,” the platform verifies these credentials against the internal database. If valid, the user is redirected to the Main Page; otherwise, an error message indicates that the credentials are incorrect or missing. Once the system confirms the username and password match, an active user session is created. This session tracks the user’s identification (e.g., user ID), ensuring that subsequent navigation and requests occur under the appropriate permissions (owner, shared access, etc.). This login mechanism is managed by the application’s authentication layer in Laravel, which securely stores session data and rejects unauthorized attempts to access the platform.

If the username is unrecognized or the password does not match, an error banner appears near the top of the login form, prompting the user to recheck the entered data, as shown in Figure 4.1.

When either the username or password field is blank, a short instructional message asks the user to fill in the missing input before attempting to log in again. Figure 4.2 shows the instructional message when the email field while a similar message appears for when the password field is empty.

When a user presses the “Forgot your password?” link, the website redirects them to a new page to reset their password (Figure 4.3).

Users that don’t have an account can press the “Create an Account” option located at the bottom of the sign in page, there the users will have to fill a form that requires their full name, email, password and a repetition of the password, also there are two checkboxes that ask the users if they want to receive community emails which is not mandatory and to agree to the terms, conditions and fees. (Figure 4.4)

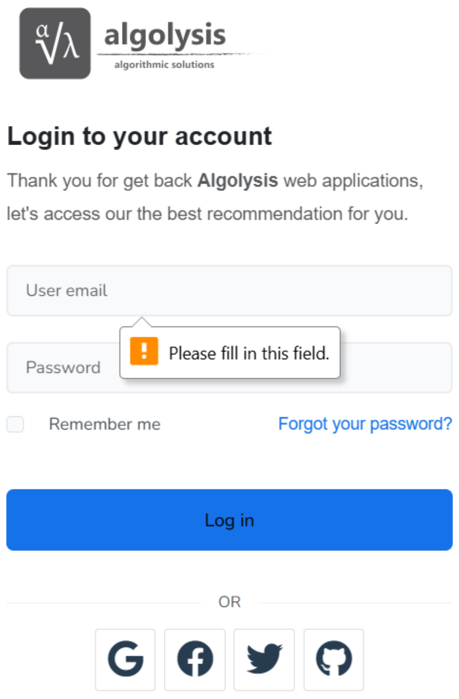

Login to your account

Thank you for get back **Algolysis** web applications, let's access our the best recommendation for you.

Whoops! Something went wrong.

- These credentials do not match our records.

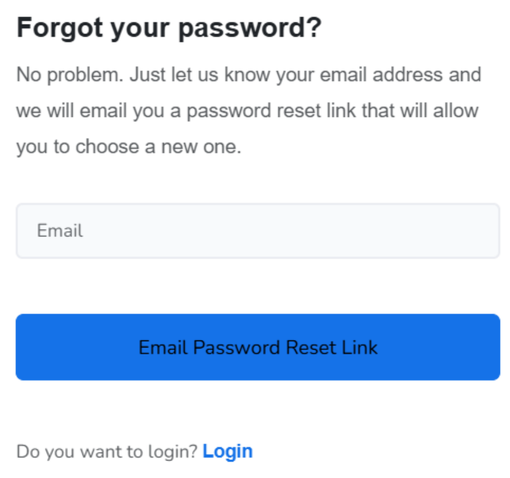
Figure 4.1 – Invalid Credentials Response


Login to your account

Thank you for get back **Algolysis** web applications, let's access our the best recommendation for you.

Please fill in this field.

Figure 4.2 – Missing Email Response



Forgot your password?

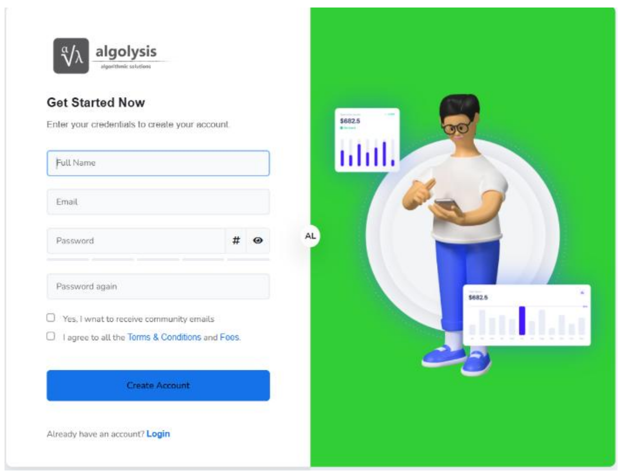
No problem. Just let us know your email address and we will email you a password reset link that will allow you to choose a new one.

Email

Email Password Reset Link

Do you want to login? Login

Figure 4.3 – Forgot Password Page



algolysis

Get Started Now

Enter your credentials to create your account

Full Name

Email

Password

Password again

Yes, I wnat to receive community emails

I agree to all the Terms & Conditions and Fees.

Create Account

Already have an account? Login

Figure 4.4 – Create an Account Page

4.1.2 Dashboard

When the user presses the “dashboard” button, shown in Figure 4.5, it redirects them to a simple page with a welcoming message, displayed in Figure 4.6.

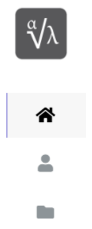


Figure 4.5 – Dashboard Button

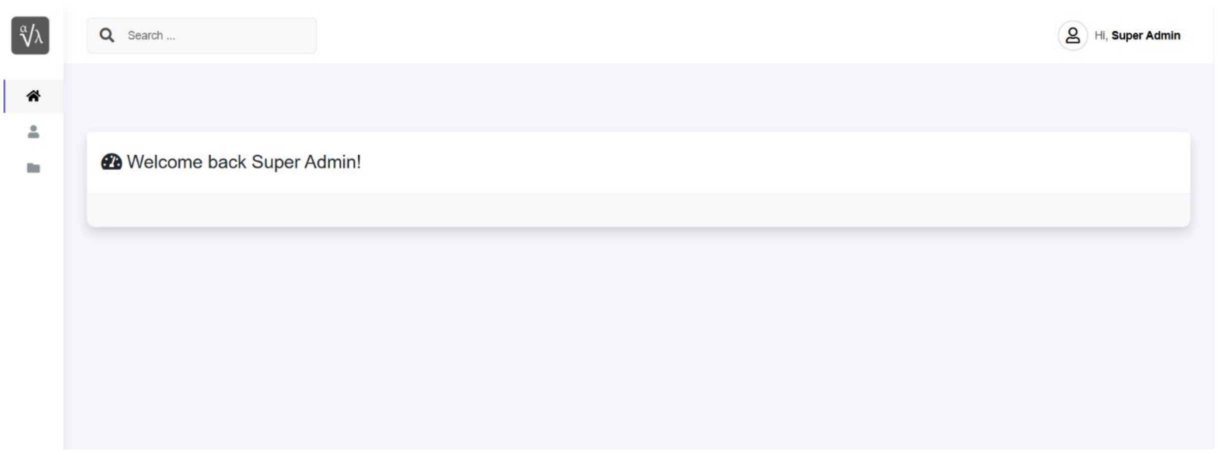


Figure 4.6 – Dashboard Page

4.1.3 User Profile

When the user presses the “profile”, shown in Figure 4.7 button they are redirected in a page where they can see and modify their information, update their password or manage their browser session.

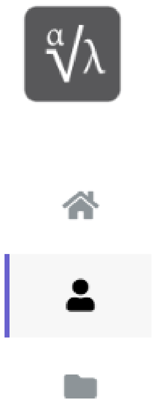
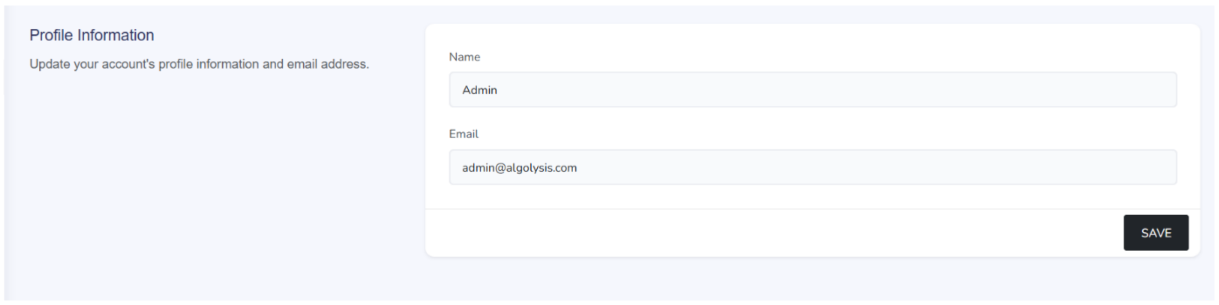


Figure 4.7 – Profile Button

The first section that appears is the “Profile Information” section (Figure 4.8), which allows the user to see and update their account's profile information like name and email address.


Profile Information

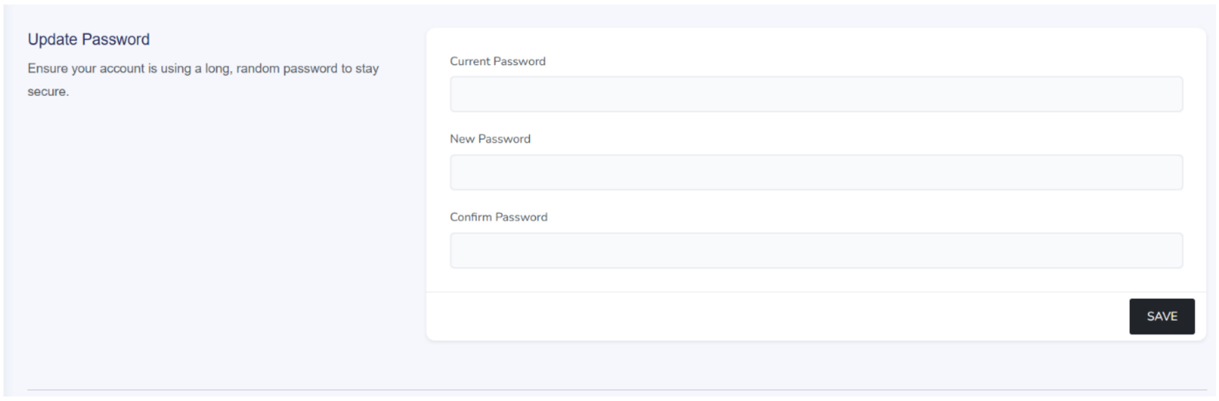
Update your account's profile information and email address.

Name: Admin

Email: admin@algolysis.com

Figure 4.8 – Profile Information Section

The next section is the "Update Password" section (Figure 4.9), which allows the user to update their password by inserting their old password, new password and reconfirming their new password. There is also a message that recommends the user to use a long and random password for security.


Update Password

Ensure your account is using a long, random password to stay secure.

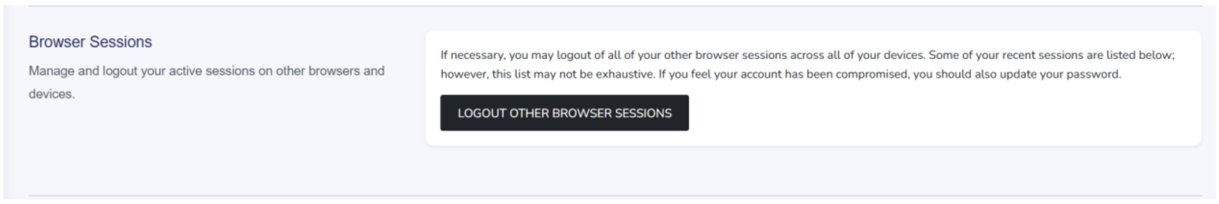
Current Password

New Password

Confirm Password

Figure 4.9 – Update Password Section

Lastly, the "Browser Sessions" section appears (Figure 4.10). This field allows the user to manage and logout their active sessions on other browsers and devices, once clicked the platform requires the user to enter their password for security reasons.


Browser Sessions

Manage and logout your active sessions on other browsers and devices.

If necessary, you may logout of all of your other browser sessions across all of your devices. Some of your recent sessions are listed below; however, this list may not be exhaustive. If you feel your account has been compromised, you should also update your password.

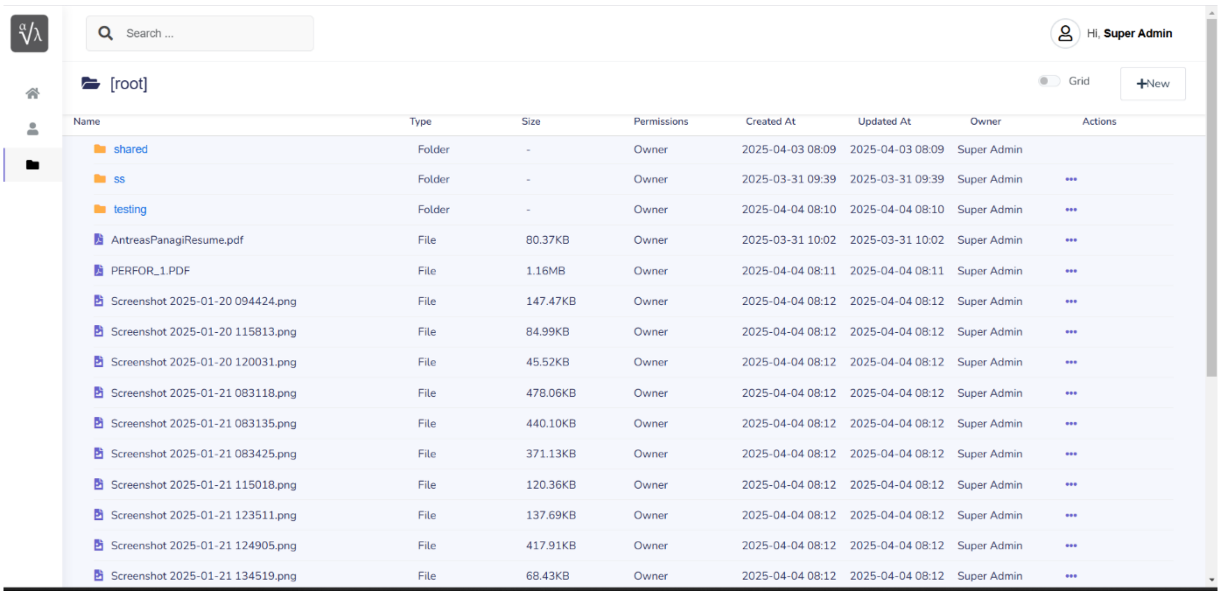
Figure 4.10 – Browser Sessions Section

4.2 Main Page

The following functionality, starting from this section until the end of the chapter was not yet implemented by the Algolysis team and they required our help to develop it. After successful authentication, users arrive at the Main Page. This central interface includes a consistent, horizontal header placed at the top. The header includes navigation tools, a toggle to switch between List View and Grid View and a “New button that lets users upload files and create folders. Below the navigation bar, the folder or file items in the current directory are displayed. Depending on the chosen layout, items appear in List or Grid format. Folders are denoted by folder icons, while files typically show type specific icons.

4.2.1 List View

In List View, each entry (folder or file) occupies one row within a table like format, displaying the following, Name which shows an icon and the name of the item, for folders, clicking the name navigates inside that folder. Type which indicates whether the item is a Folder or File. Size which is left blank for folders and files show their size. Permissions that show the user’s current access level (owner, read write, read only). Also Created At and Updated At display the Date time stamps for the item’s creation and most recent update. Lastly, Actions that display 3 dots which when clicked a dropped down appears that provides operations like rename, delete, download and share. In Figure 4.11 appears the root directory of a user in list view.



Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
shared	Folder	-	Owner	2025-04-03 08:09	2025-04-03 08:09	Super Admin	
ss	Folder	-	Owner	2025-03-31 09:39	2025-03-31 09:39	Super Admin	•••
testing	Folder	-	Owner	2025-04-04 08:10	2025-04-04 08:10	Super Admin	•••
AntreasPanagiResume.pdf	File	80.37KB	Owner	2025-03-31 10:02	2025-03-31 10:02	Super Admin	•••
PERFOR_1.PDF	File	1.16MB	Owner	2025-04-04 08:11	2025-04-04 08:11	Super Admin	•••
Screenshot 2025-01-20 094424.png	File	147.47KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-20 115813.png	File	84.99KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-20 120031.png	File	45.52KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 083118.png	File	478.06KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 083135.png	File	440.10KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 083425.png	File	371.13KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 115018.png	File	120.36KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 123511.png	File	137.69KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 124905.png	File	417.91KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••
Screenshot 2025-01-21 134519.png	File	68.43KB	Owner	2025-04-04 08:12	2025-04-04 08:12	Super Admin	•••

Figure 4.11 – List View

4.2.2 Grid View

In Grid View, items are displayed as separate "tiles" each containing a folder or type specific file icon near the top centre, the item's name is located under the icon, truncated if long. There is a 3-dot button in the tile's corner, which when pressed reveals a context menu (rename, download, share, delete and info). Grid View can be more visually appealing and a more intuitive approach for users who prefer icons over a table layout. In Figure 4.12 appears the root directory of a user in grid view. The files and folders in grid and list view are fetched from the server through the *Storage::disk('dsm')->directories($currentPath, false, true)* API which brings all the files and folders in a specific directory, in turn allowing us to update the database with the latest items.

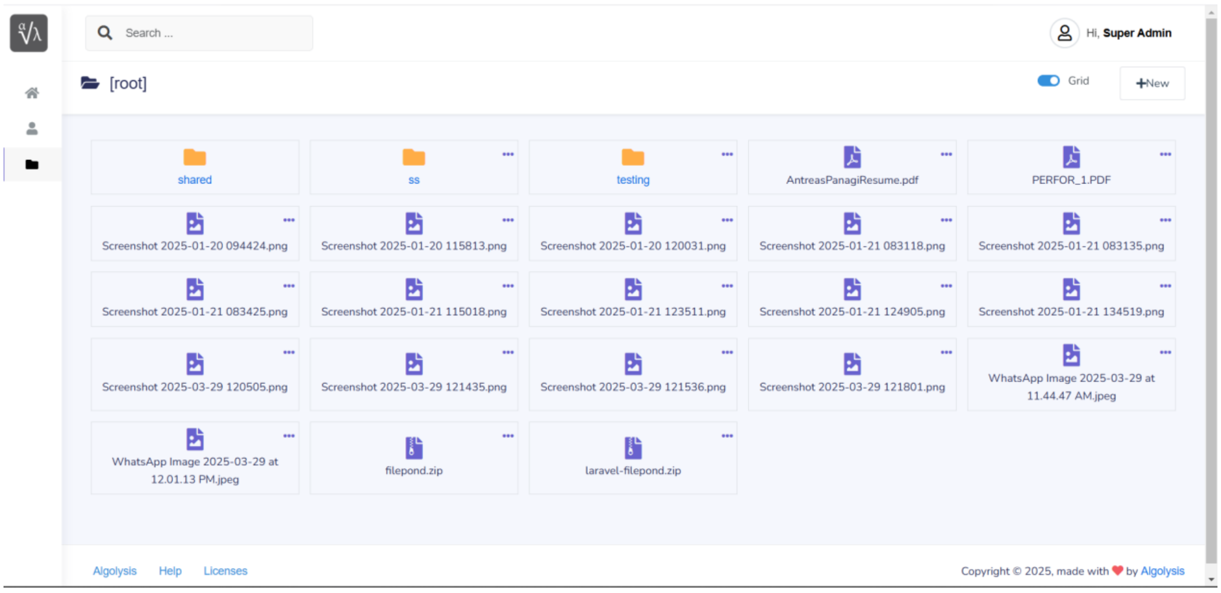

Search ...
Hi, Super Admin
[root]
Grid
+New
shared
ss
testing
AntreasPanagiResume.pdf
PERFOR_1.PDF
Screenshot 2025-01-20 094424.png
Screenshot 2025-01-20 115813.png
Screenshot 2025-01-20 120031.png
Screenshot 2025-01-21 083118.png
Screenshot 2025-01-21 083135.png
Screenshot 2025-01-21 083425.png
Screenshot 2025-01-21 115018.png
Screenshot 2025-01-21 123511.png
Screenshot 2025-01-21 124905.png
Screenshot 2025-01-21 134519.png
Screenshot 2025-03-29 120505.png
Screenshot 2025-03-29 121435.png
Screenshot 2025-03-29 121536.png
Screenshot 2025-03-29 121801.png
WhatsApp Image 2025-03-29 at 11.44.47 AM.jpeg
WhatsApp Image 2025-03-29 at 12.01.13 PM.jpeg
filepond.zip
laravel-filepond.zip
Algolysis
Help
Licenses
Copyright © 2025, made with ♥ by Algolysis

Figure 4.12 – Grid View

4.3 Uploading Files

Selecting the "New" button from the navigation bar triggers a modal window where users can select files from their local machine (Figure 4.13). The modal contains a section showing the current directory that the user is located, the uploaded files and folders will be created automatically inside that directory, unless the user decides to change the directory which can be done by clicking that section and choosing the desired directory path. Underneath it there is the "New Folder" section in which the user can type the desired name of the folder or path that the user wants the files to be located. The uploaded file is store in the DSM through the use of APIs that allow the front-end to interact with the back-end. More specifically, for uploading files we use the following API in the code

Storage::disk('dsm')->put($path, $file) that allows a specific file (*$file*) to be stored in a specific directory *($path*) in the servers. Additionally if we want to create a new folder or directory we use the *Storage::disk('dsm')->makeDirectory($newDir)* API which creates a new directory inside the servers.

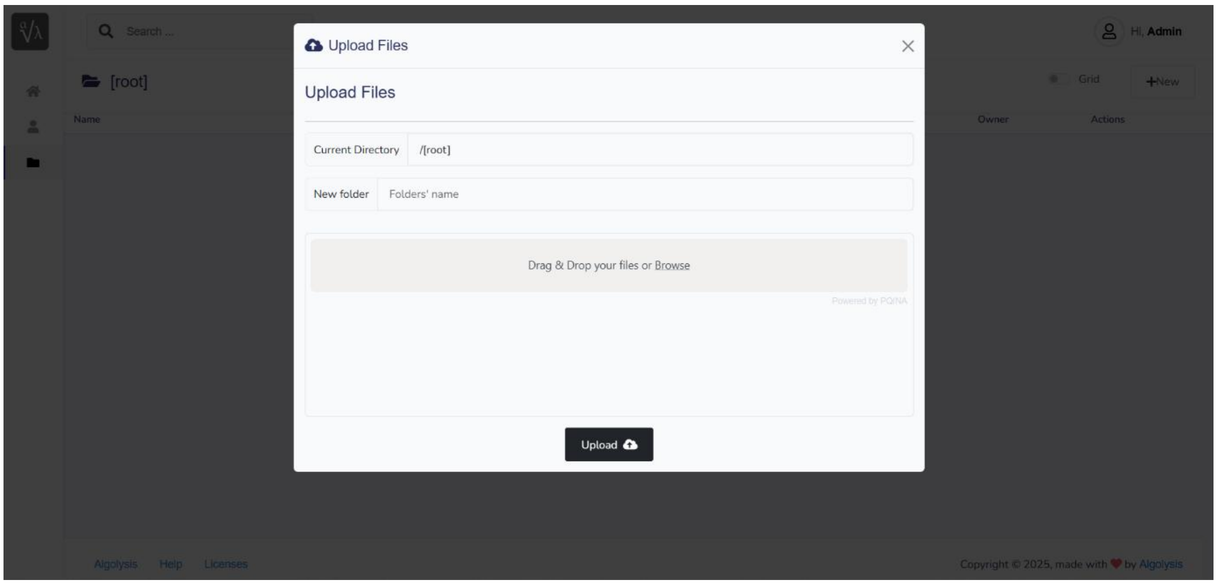


Figure 4.13 – Upload Modal

In the centre of the modal there is a drag and drop section to insert the files or alternatively the user can click on it, which will open the local file explorer and the user can decide which files to upload, as shown in Figure 4.14. After selecting the desired files, they will appear under the drag and drop section and by filling also the folder name, a completely filled modal will look like Figure 4.15.

At the bottom of the modal there is an Upload button which will upload the selected files in the desired directory. If the user chooses to create an Empty Folder, they can just write the name of the folder inside the "New Folder" section and simply just press upload without selecting any files, as shown in Figure 4.16. In Figure 4.17 we can see the user's empty root directory before uploading any files, while in Figure 4.18 we observe the result of uploading the previous files in the root directory and in Figure 4.19 inside */screenshots/favourite.*

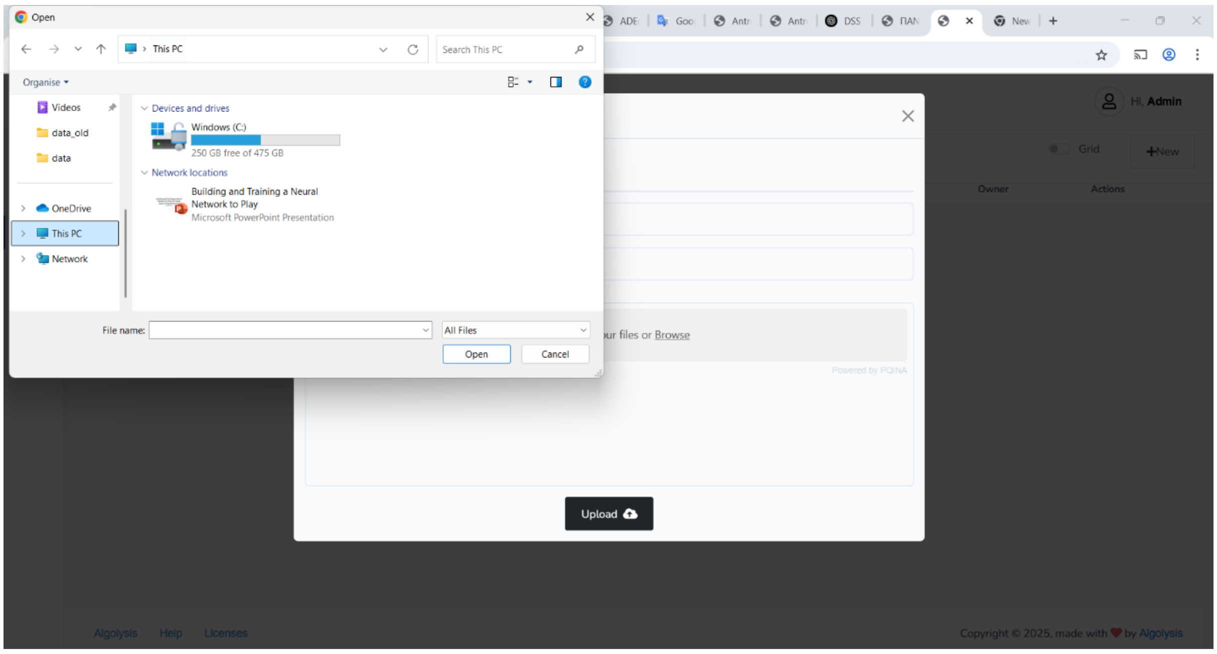
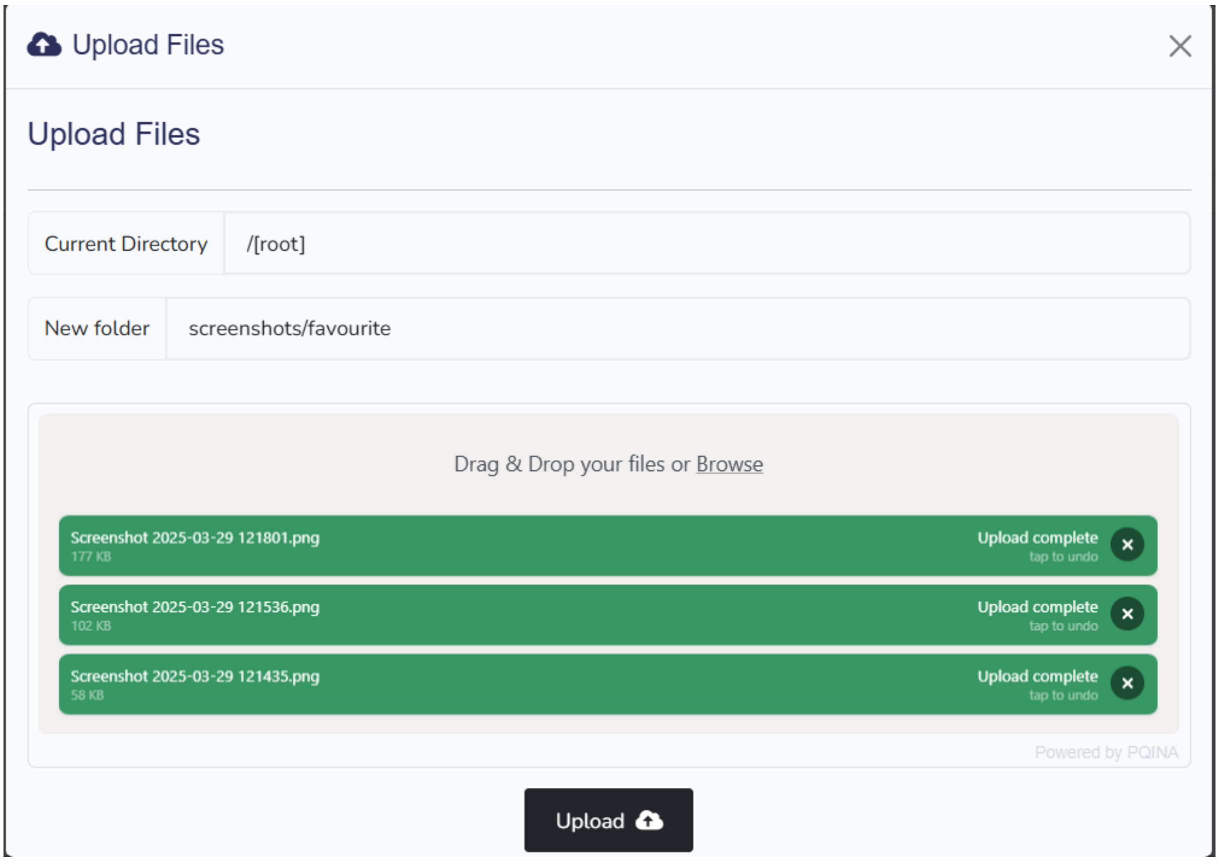


Figure 4.14 – File Explorer Pop Up



Upload Files

Upload Files

Current Directory	/[root]
New folder	screenshots/favourite

Drag & Drop your files or Browse

Screenshot 2025-03-29 121801.png
177 KB
Upload complete
tap to undo

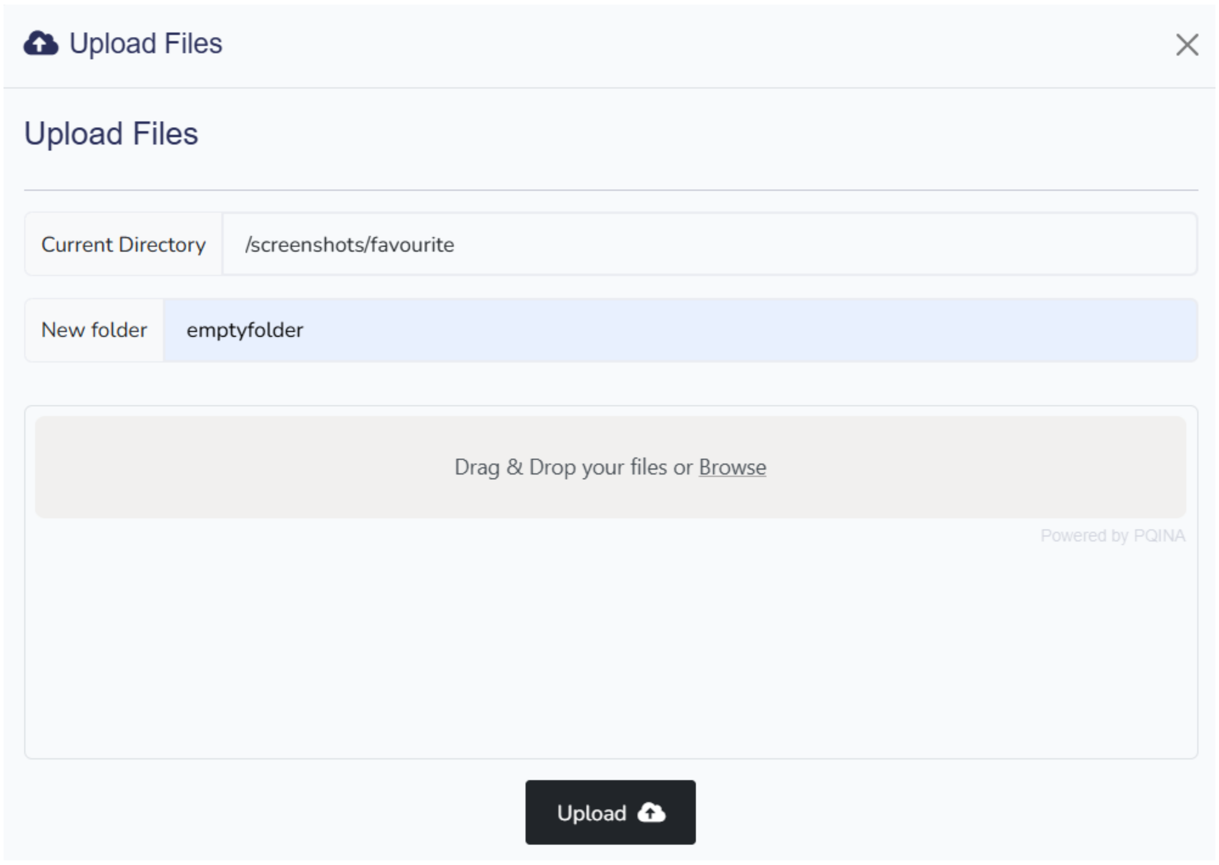
Screenshot 2025-03-29 121536.png
102 KB
Upload complete
tap to undo

Screenshot 2025-03-29 121435.png
58 KB
Upload complete
tap to undo

Powered by PQINA

Upload

Figure 4.15 – Filled Modal


Upload Files

Upload Files

Current Directory /screenshots/favourite

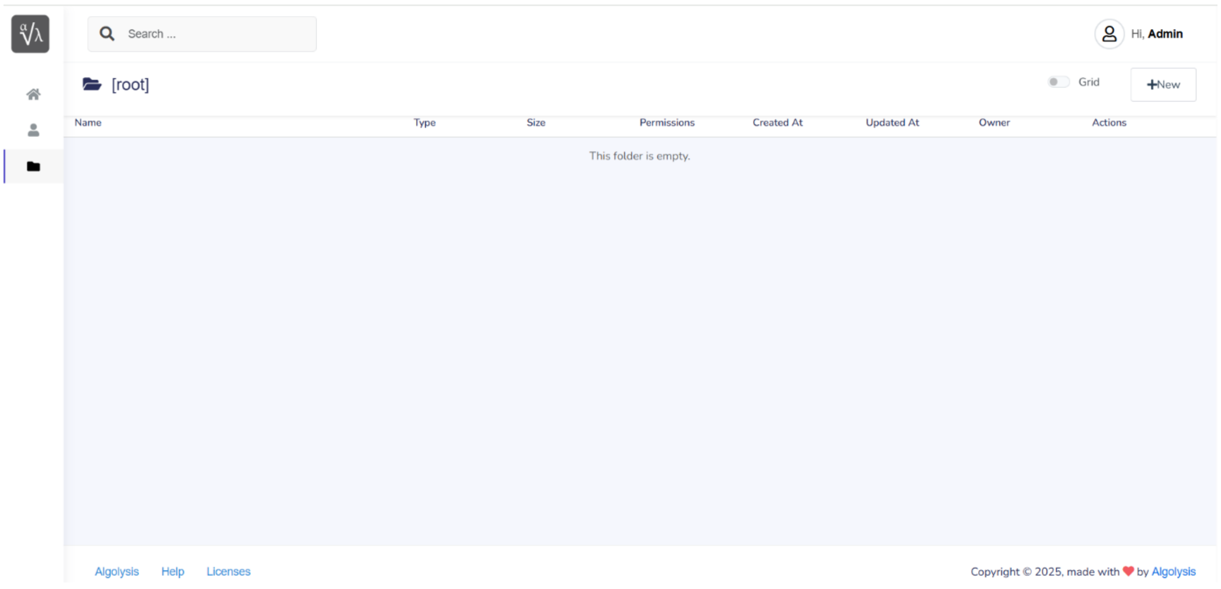
New folder emptyfolder

Drag & Drop your files or Browse

Powered by PQINA

Upload

Figure 4.16 – Creation Of Empty Folder


Search ...

Hi, Admin

[root]

Grid

+New

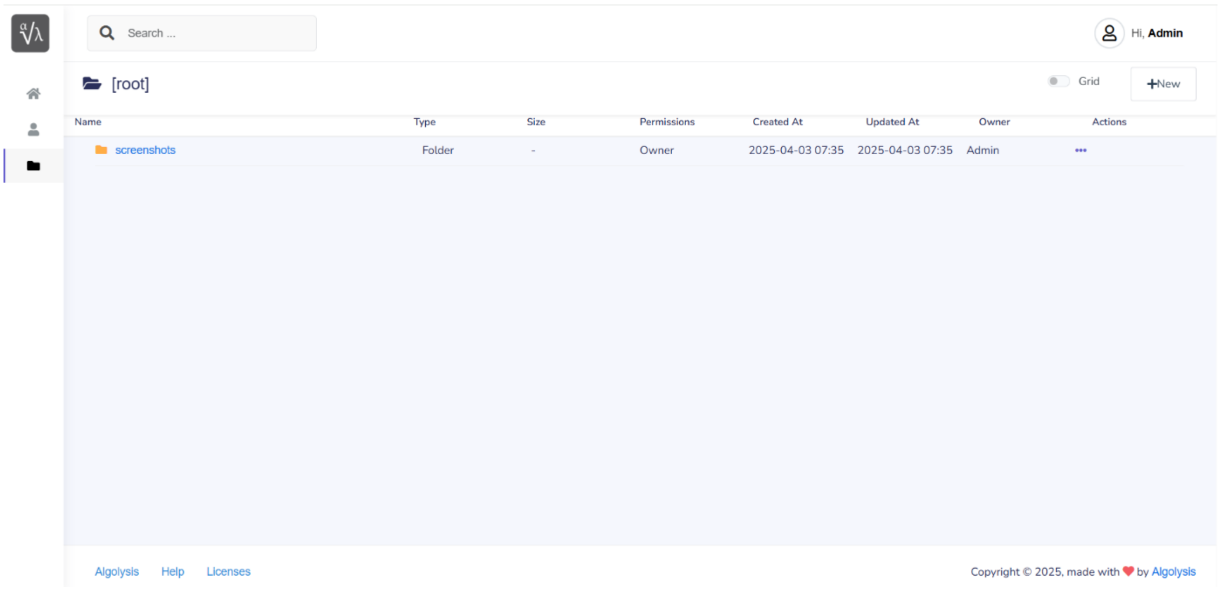
Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions

This folder is empty.

Algolysis Help Licenses

Copyright © 2025, made with ❤ by Algolysis

Figure 4.17 – Before Uploading



Search ...

Hi, **Admin**

[root]

Grid

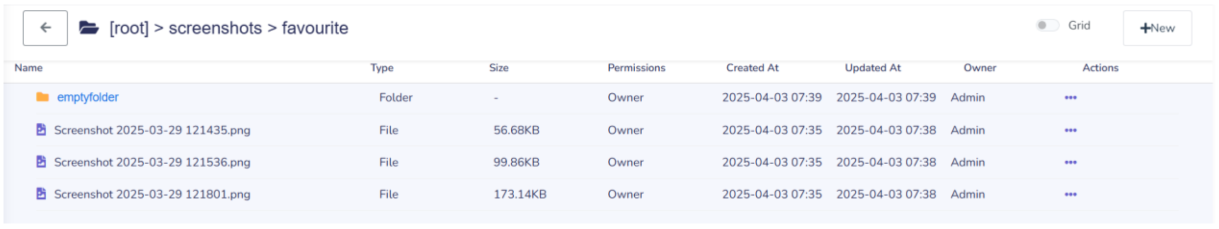
+New

Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
screenshots	Folder	-	Owner	2025-04-03 07:35	2025-04-03 07:35	Admin	•••

Algolysis Help Licenses

Copyright © 2025, made with ❤ by Algolysis

Figure 4.18 – After Uploading Root Directory



[root] > screenshots > favourite

Grid

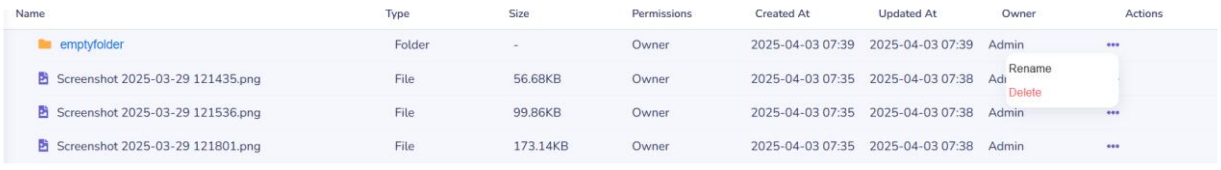
+New

Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
emptyfolder	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:39	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121801.png	File	173.14KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.19 – After Uploading Inside Folder

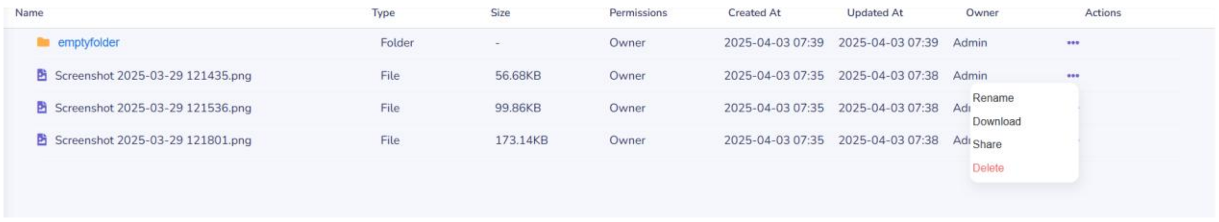
4.4 Actions

A major component of the platform is the ability to perform various management actions on files and folders. These features ensure users can handle daily file related tasks smoothly. In Figure 4.20 we can see the two actions for folders in list view, Rename and Delete and in Figure 4.21 we can see the actions for files in list view, which are, Rename, Download, Share and Delete.



Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
emptyfolder	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:39	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Adı	
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121801.png	File	173.14KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.20 – Folder Actions

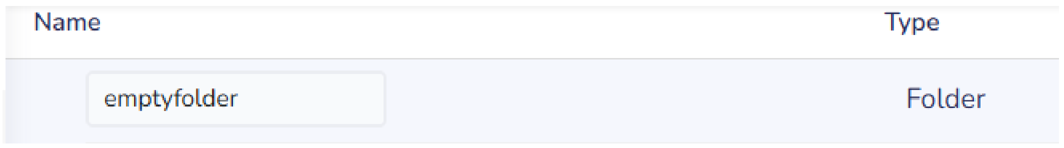


Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
emptyfolder	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:39	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Ad	
Screenshot 2025-03-29 121801.png	File	173.14KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Ad	

Figure 4.21 – File Actions

4.4.1 Rename

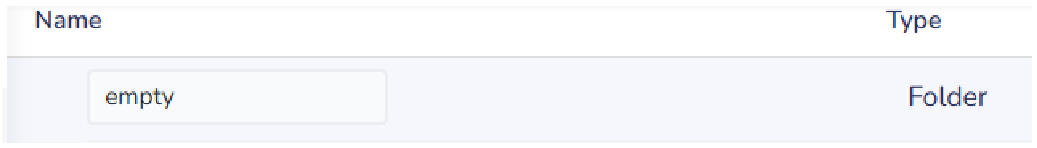
The "Rename" option is found under the Actions dropdown in each row. Selecting it replaces the item's name cell with an inline textbox (Figure 4.22).



Name	Type
emptyfolder	Folder

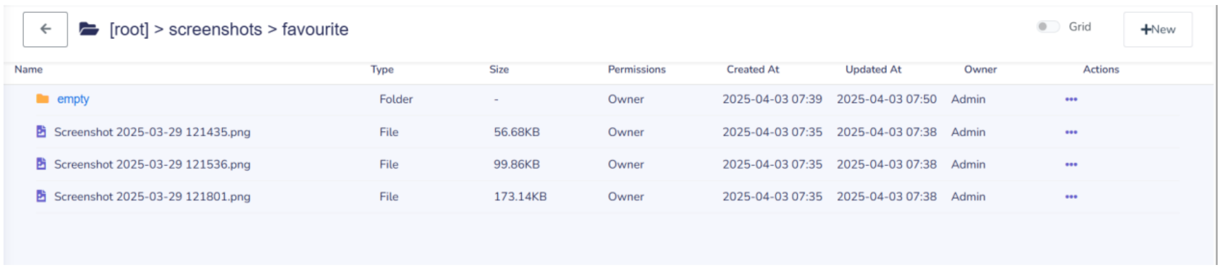
Figure 4.22 – Inline Textbox

The users can then change the name of the item and on pressing Enter or clicking away, the system updates the name on the server, and the row reverts to normal display. Figure 4.23 shows a user that changed the name of a folder from *emptyfolder* to *empty* and when the user presses the enter button, the result appears in Figure 4.24. The rename action is achieve through the use of the following API *Storage::disk('dsm')->rename($request->input('old_path'), $request->input('new_name'))* that notifies the servers to change the path and the name of the file.



Name	Type
empty	Folder

Figure 4.23 – Renaming Of Folder


[root] > screenshots > favourite

Grid

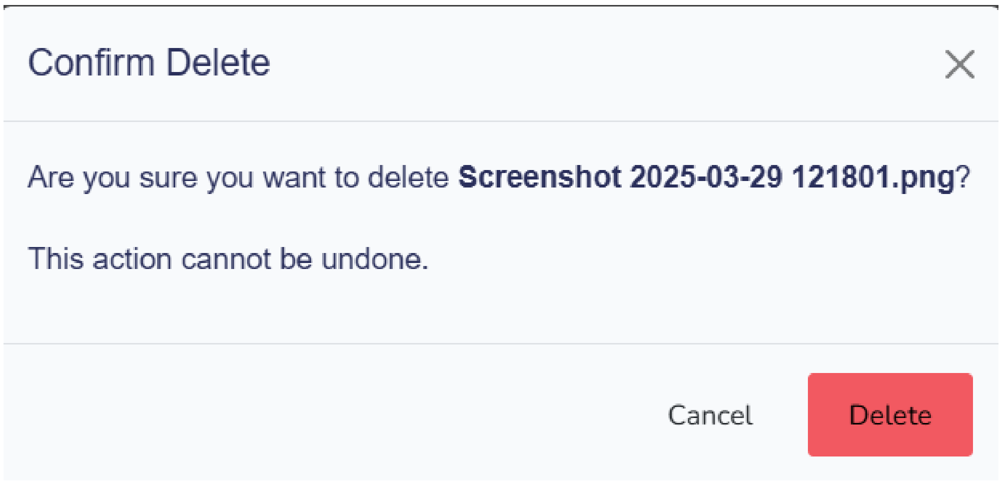
+New

Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
empty	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:50	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121801.png	File	173.14KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.24 – Result of Rename Action

4.4.2 Delete

Users can delete a file or folder via the “Delete” entry in the item’s action menu. Once pressed, a confirmation popup (Figure 4.25) ensures the user wants to proceed.

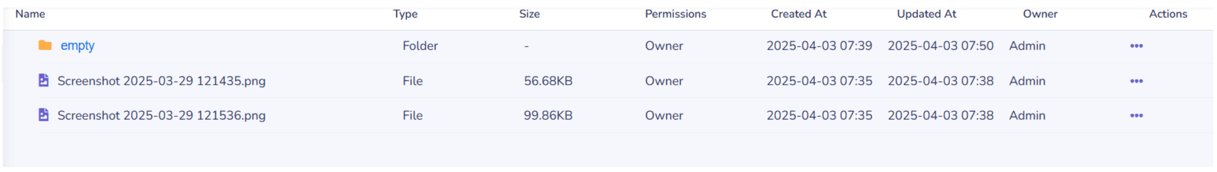

Confirm Delete

Are you sure you want to delete **Screenshot 2025-03-29 121801.png**?

This action cannot be undone.

Figure 4.25 – Delete Confirmation Modal

Confirming removal, updates the view instantly, removing the item from the interface and the server. The results of the delete action appear in Figure 4.26. It also, updates the servers to remove the item through the use of Storage::disk('dsm')->deleteDirectory($path) or Storage::disk('dsm')->delete($path) API which delete either an entire directory or a specific file.

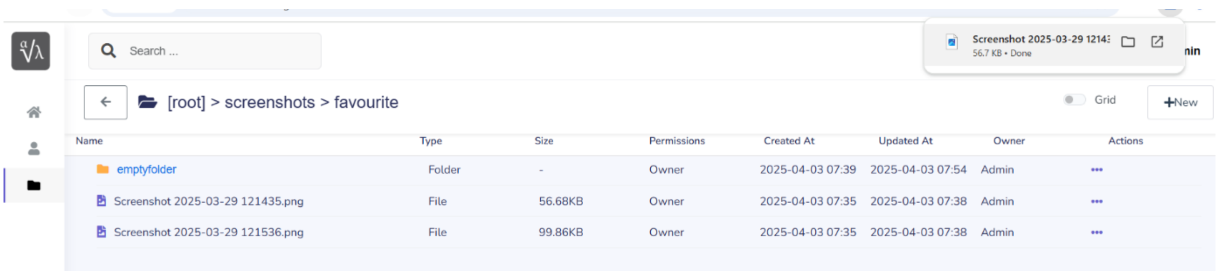


Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
empty	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:50	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.26 – Result Of Deletion

4.4.3 Download

Download is available for files only, when this option is pressed it triggers the browser's default download mechanism. Allowing the users to store the file inside their machine. The result of pressing the download button appears in Figure 4.27. This is achieved through the *Storage::disk('dsm')->download($fullPath)* API that requests from the servers the specific file in order for the user to be able to download it.



Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
emptyfolder	Folder	-	Owner	2025-04-03 07:39	2025-04-03 07:54	Admin	•••
Screenshot 2025-03-29 121435.png	File	56.68KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

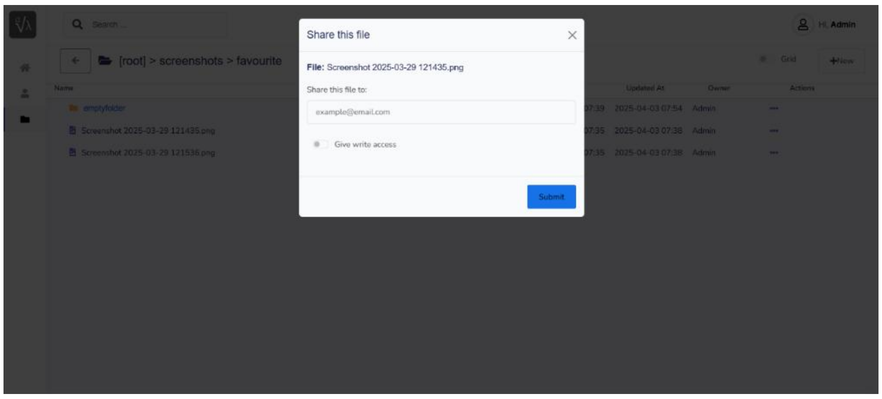
Figure 4.27 – Result Of Downloading

4.4.4 Share

Owners can share files with other users. When Pressing "Share" a modal opens (Figure 4.28) and the sharer may enter the recipient's email. The owner can also decide which permissions to allow the other user. Having read-write access will allow the user to have the ability to rename or download the file while read access gives them the ability to only download the file. In Figure 4.29 we observe a filled in modal with the email field set as superadmin@algolysis.com that has been givem write access.

Once shared, the item shows up under the recipient's "Shared" folder, which if it doesn't exist is created automatically and appears at the top of the list or grid view. In Figure 4.30 we see the root directory of a user before having a file shared to them, while in Figure 4.31 we see the root directory right after having a file shared to them, a shared folder has appeared at the top of the page. Once clicked it redirects the user inside the shared directory, where all of their shared files will be located. In Figure 4.32 we can see an example of a shared file with read/write permissions where as in Figure 4.33 the exact same file has only read permission. This changes the permissible actions from Rename and Download to only Download. Sharing does not require the use of APIs since the front-end doesn't need to interact with the back-end, it only needs to update the database, more specifically the *user_file* pivot table, which stores the id of the files being shared, the id of the users that the files are being shared to and the permissions of said users, either read-only or read-write.

When the user presses the share button again on the same file, the same modal appears again but at the lower part of the modal a list of all the users that the current file is being shared to appears, displaying their email and their permissions (Figure 4.34).



emptyfolder 07:39 2025-04-03 07:54 Admin
Screenshot 2025-03-29 121435.png 07:35 2025-04-03 07:38 Admin
Screenshot 2025-03-29 121536.png 07:35 2025-04-03 07:38 Admin

Share this file

File: Screenshot 2025-03-29 121435.png

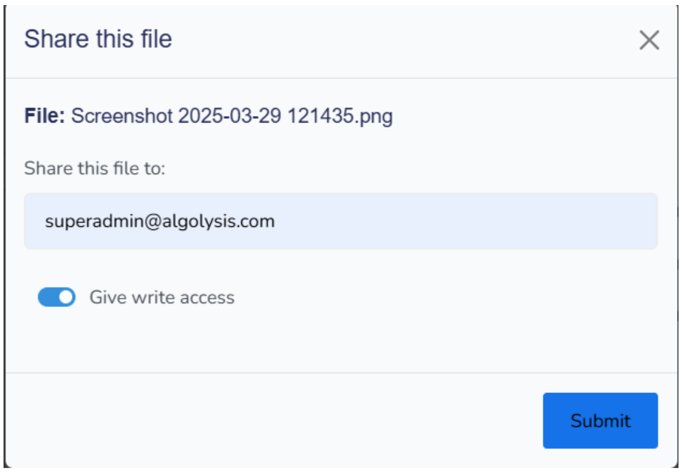
Share this file to:

example@email.com

Give write access

Submit

Figure 4.28 – Empty Share Modal



Share this file

File: Screenshot 2025-03-29 121435.png

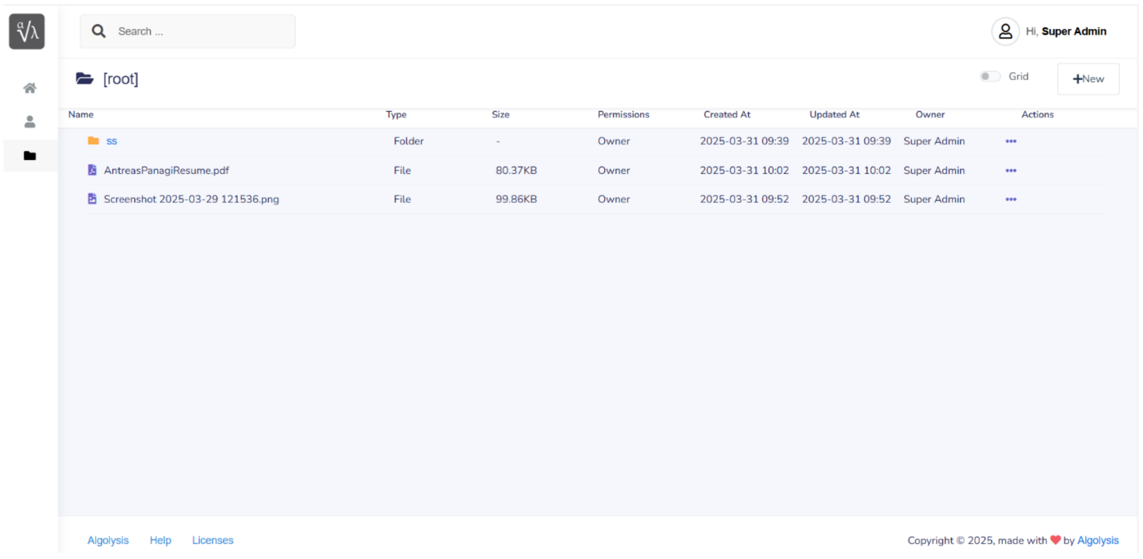
Share this file to:

superadmin@algolysis.com

Give write access

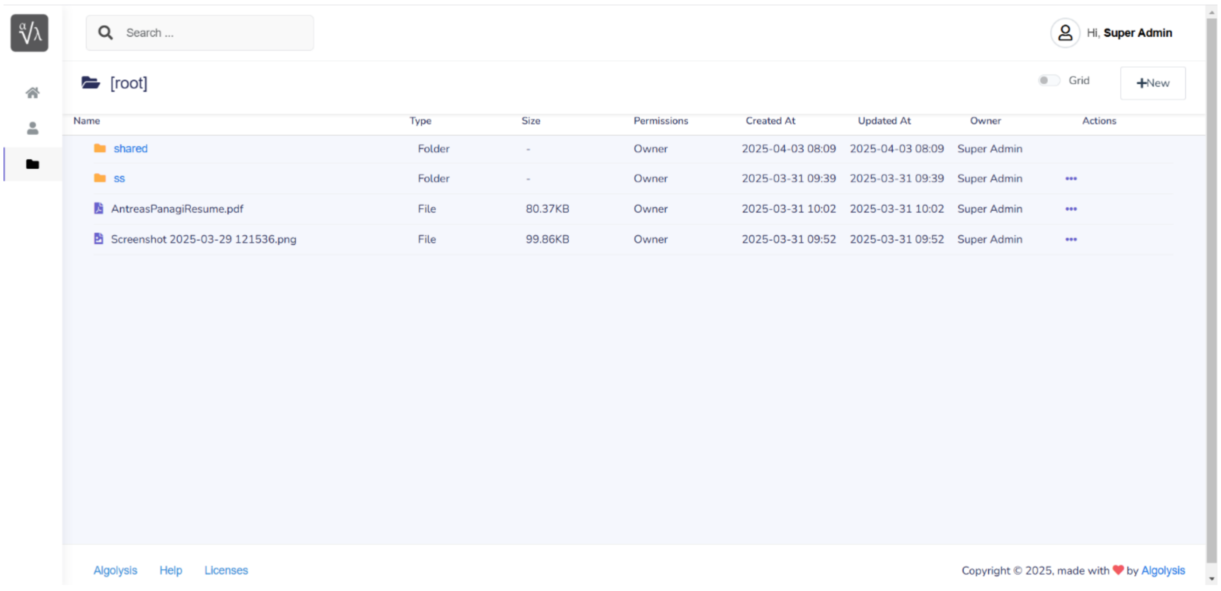
Submit

Figure 4.29 – Filled Shared Modal



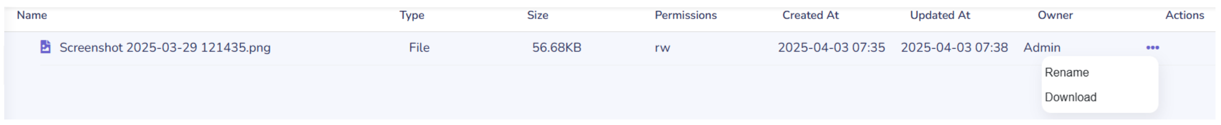
Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
ss	Folder	-	Owner	2025-03-31 09:39	2025-03-31 09:39	Super Admin	•••
AntreasPanagiResume.pdf	File	80.37KB	Owner	2025-03-31 10:02	2025-03-31 10:02	Super Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-03-31 09:52	2025-03-31 09:52	Super Admin	•••

Figure 4.30 – Recipient Before Sharing



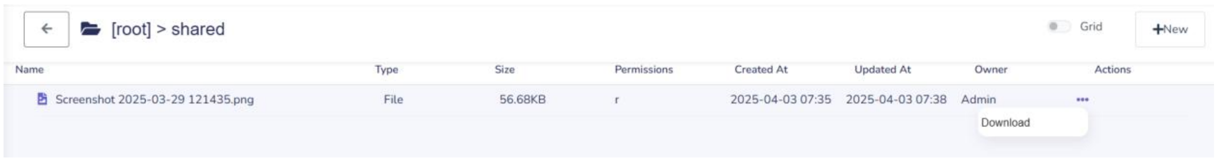
Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
shared	Folder	-	Owner	2025-04-03 08:09	2025-04-03 08:09	Super Admin	
ss	Folder	-	Owner	2025-03-31 09:39	2025-03-31 09:39	Super Admin	•••
AntreasPanagiResume.pdf	File	80.37KB	Owner	2025-03-31 10:02	2025-03-31 10:02	Super Admin	•••
Screenshot 2025-03-29 121536.png	File	99.86KB	Owner	2025-03-31 09:52	2025-03-31 09:52	Super Admin	•••

Figure 4.31 – Recipient After Sharing



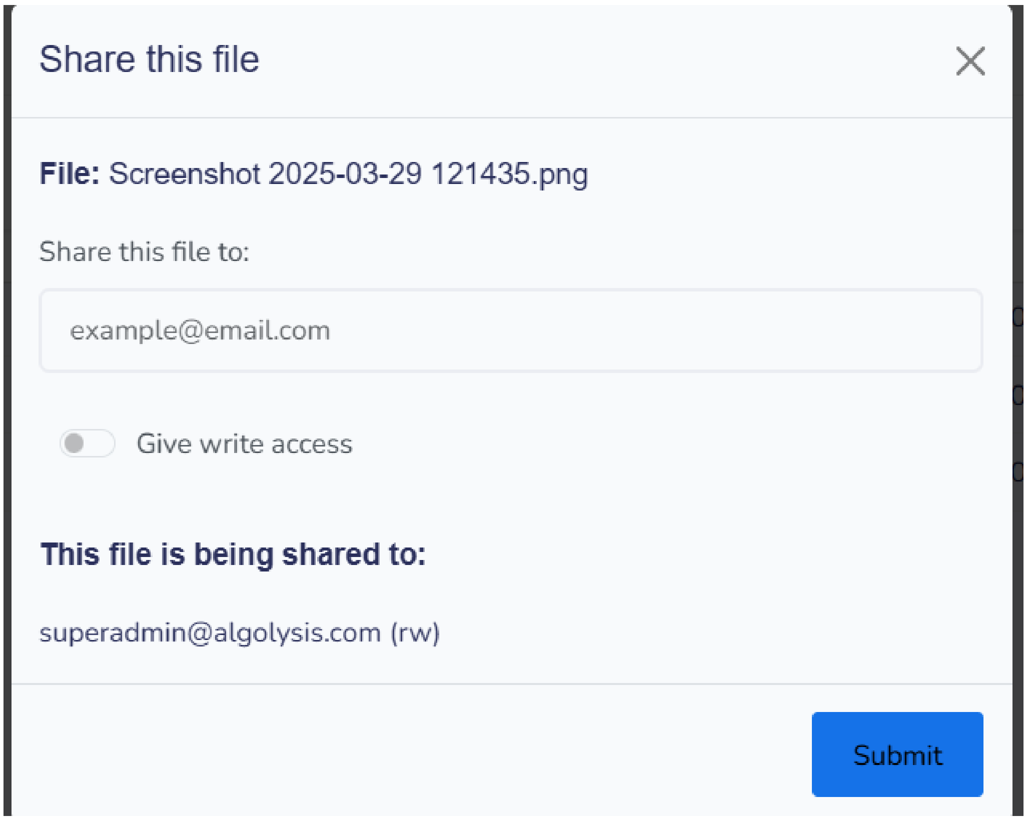
Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
Screenshot 2025-03-29 121435.png	File	56.68KB	rw	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.32 – Inside Shared Folder With Read Write Permission



Name	Type	Size	Permissions	Created At	Updated At	Owner	Actions
Screenshot 2025-03-29 121435.png	File	56.68KB	r	2025-04-03 07:35	2025-04-03 07:38	Admin	•••

Figure 4.33 – Inside Shared Folder With Read Permission


Share this file

File: Screenshot 2025-03-29 121435.png

Share this file to:

example@email.com

Give write access

This file is being shared to:

superadmin@algolysis.com (rw)

Figure 4.34 – Share Modal After Sharing

4.4.5 Info

In Grid View, the action button has the exact same actions as in List View with one addition, the “Info” option (Figure 4.35) which provides a brief overview of the item’s properties, such as file size, type, creation date, last update date, and owner identity, basically the same information which appears in the List View’s columns. This information appears in a pop up without needing to navigate away from the current folder (Figure 4.36).

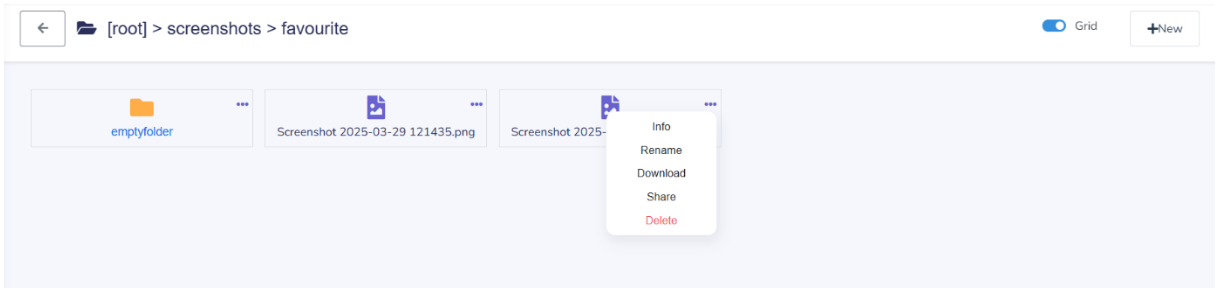
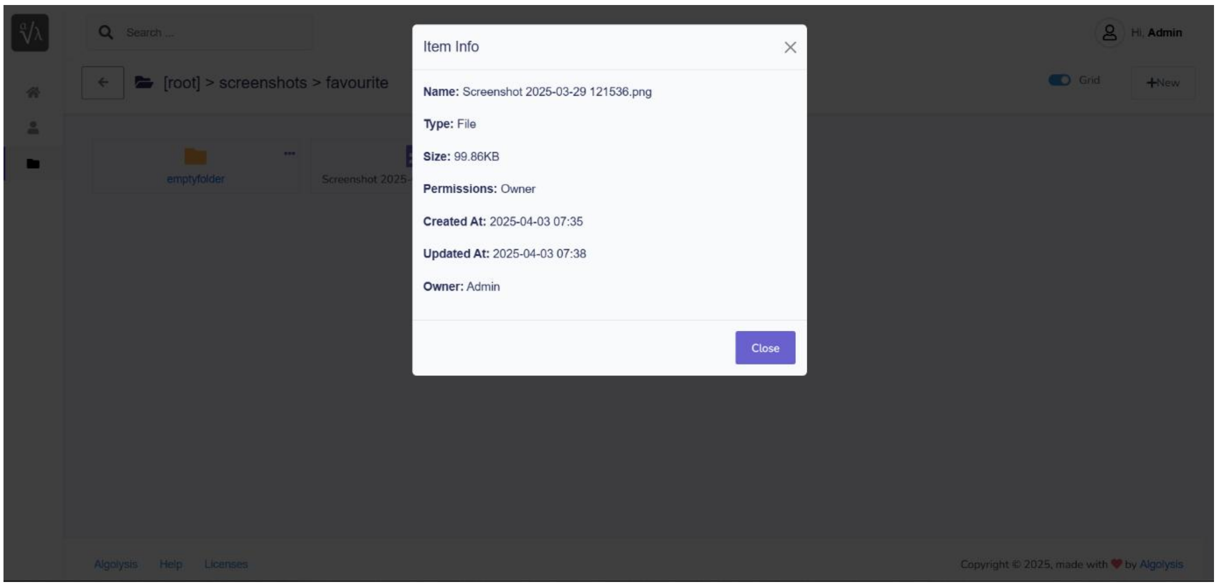

[root] > screenshots > favourite
Grid
+New
emptyfolder
Screenshot 2025-03-29 121435.png
Screenshot 2025-
Info
Rename
Download
Share
Delete

Figure 4.35 – Action Button In Grid View


Item Info

Name: Screenshot 2025-03-29 121536.png

Type: File

Size: 99.86KB

Permissions: Owner

Created At: 2025-04-03 07:35

Updated At: 2025-04-03 07:38

Owner: Admin

Close
Search ...
Hi, Admin
[root] > screenshots > favourite
Grid
+New
emptyfolder
Algolysis
Help
Licenses
Copyright © 2025, made with ❤ by Algolysis

Figure 4.36 – Info Modal

4.5 Navigation

Multiple methods of navigation help users efficiently move through the directory structure and quickly locate their files. Firstly, there is a Back Arrow button in the header that takes the user up one level in the directory hierarchy, effectively returning them to the folder where they came from (Figure 4.37).

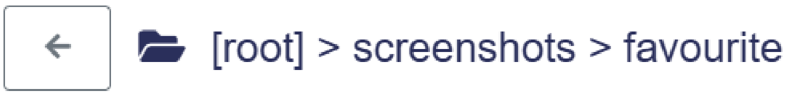


Figure 4.37 – Back Arrow

There is also a Breadcrumb Trail where a series of clickable links ([root] > screenshots > favourite > emptyfolder) appears near the top of the page. Each link corresponds to an ancestor folder in the path. This approach enables direct jumping to any parent folder rather than requiring repeated presses of the back arrow. In Figure 4.38 appears a snapshot of a user being navigated to the root directory by pressing it, while being in the "favourite" directory.

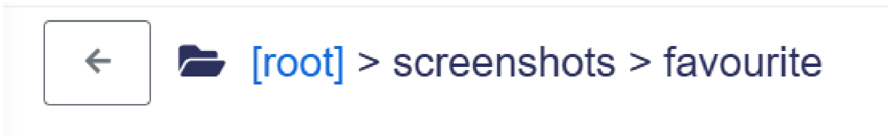


Figure 4.38 – Breadcrumb Trail pressed

To navigate deeper into a folder in both List and Grid views, the user must press the folder name or icon and the displayed contents then update to reflect the chosen subdirectory, and the breadcrumb adapts accordingly. In figure 4.39 we can see a folder with a blue colour name, indicating a link, also when the mouse hovers over the name it appears clickable which makes it easier for user to understand how to use the platform.

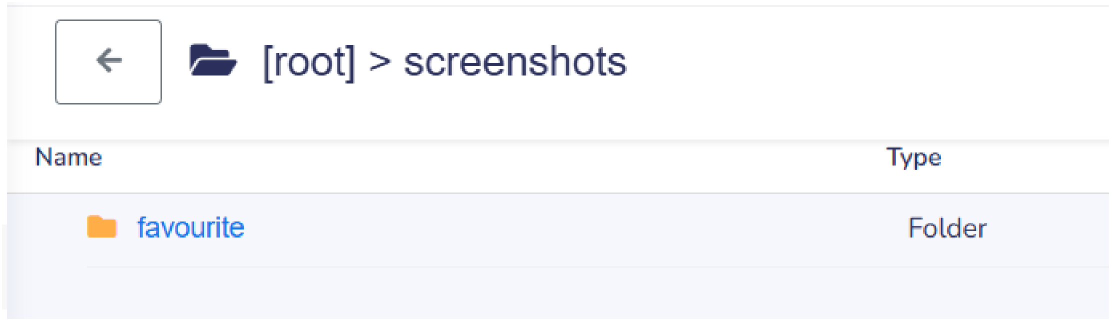


Figure 4.39 – Folders Are Linked To Their Directories

Lastly there is a file explorer button on the side bar on the left (Figure 4.40) that immediately brings the user to their top level directory, saving time. These navigation techniques, combined with clear icons and layout, provide a user-friendly environment that makes file browsing, uploading, and management straightforward, even for those less experienced with web based storage solutions.

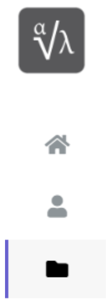
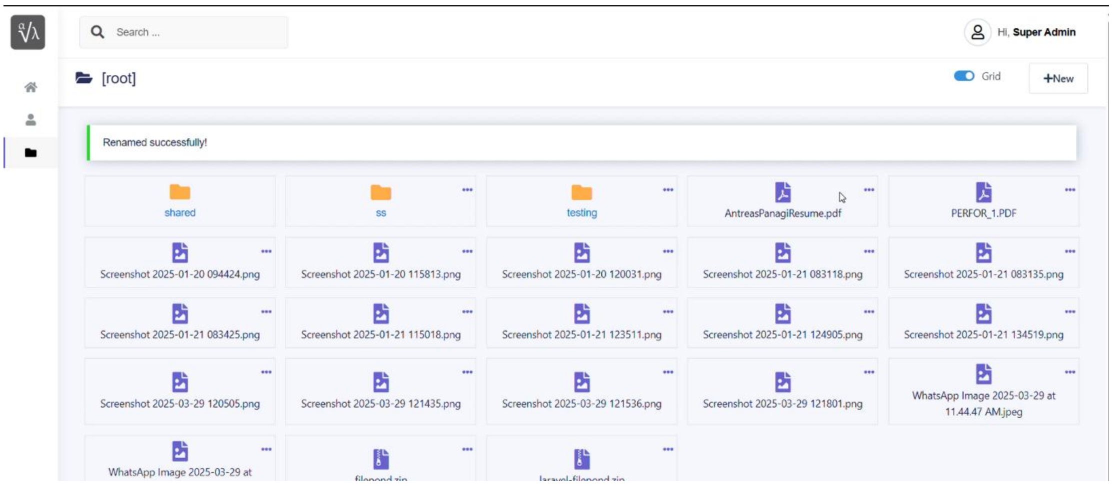


Figure 4.40 – File Explorer Button

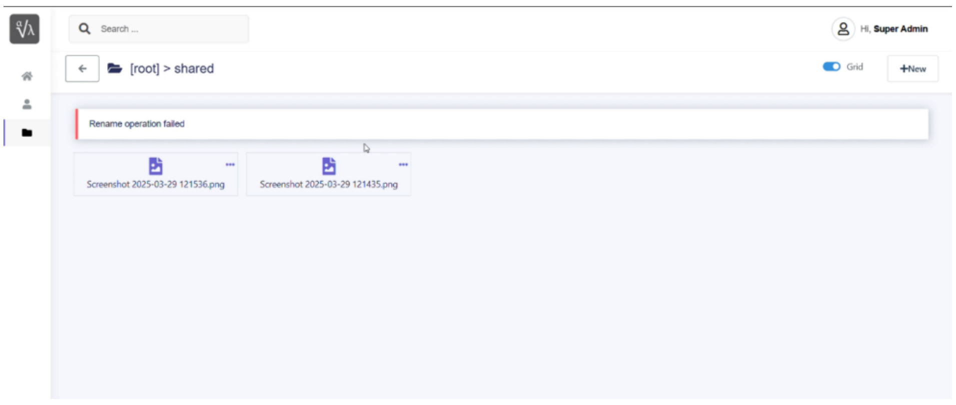
4.6 Messages

When a user performs an action, the user receives a message at the top of the platform informing the user whether the action was completed successful or whether something went wrong with the server and the action could not be completed. In the following example, we have a user that tried to rename a file. In Figure 4.41 is the success response message that the user receives when everything goes smoothly. While, in Figure 4.42 is the fail response message when something goes wrong.


Search ...
Hi, Super Admin
[root]
Grid
+New
shared
ss
testing
AntreasPanagiResume.pdf
PERFOR_1.PDF
Screenshot 2025-01-20 094424.png
Screenshot 2025-01-20 115813.png
Screenshot 2025-01-20 120031.png
Screenshot 2025-01-21 083118.png
Screenshot 2025-01-21 083135.png
Screenshot 2025-01-21 083425.png
Screenshot 2025-01-21 115018.png
Screenshot 2025-01-21 123511.png
Screenshot 2025-01-21 124905.png
Screenshot 2025-01-21 134519.png
Screenshot 2025-03-29 120505.png
Screenshot 2025-03-29 121435.png
Screenshot 2025-03-29 121536.png
Screenshot 2025-03-29 121801.png
WhatsApp Image 2025-03-29 at 11.44.47 AM.jpeg
WhatsApp Image 2025-03-29 at

Renamed successfully!

Figure 4.41 – Success Response Message


Search ...
Hi, Super Admin
[root] > shared
Grid
+New
Screenshot 2025-03-29 121536.png
Screenshot 2025-03-29 121435.png

Rename operation failed

Figure 4.41 – Fail Response Message

Chapter 5

Implementation

5.1 Project Folder Layout 53

5.2 Key Libraries and Dependencies 54

5.3 Illustrative Code Snippets 54

5.3.1 Routing 54

5.3.2 Controllers and Eloquent Models 55

5.3.3 MySQL and Database Migrations 57

5.3.4 Laravel Query Builder 58

5.3.5 Bootstrap and Alpine.js for Front End Interactions 58

Following the overall platform overview, this chapter deals with the details of the system development process. It outlines how the proposed features and designs were turned into code. Special focus is given to how the application of Laravel's MVC framework enabled modularity, reusability, and maintainability during implementation. We discuss how routing, models and controllers were implemented, showing also the database migrations that enabled the platform's specific structure and how Laravel Query Builder helped us interact with said database through the code. Finally, we talk about Bootstrap and Alpine.js that helped make the platform more responsive and easier to use.

5.1 Project Folder Layout

The application keeps the classic Laravel directory tree, so anyone who has opened a Laravel project before will feel at home:

- **app** – Holds the application's "brains":
    - Http (controllers, middleware, form-request classes)
    - Models (the Eloquent models that "speak" to the database)
    - Providers (service-provider classes that bootstrap features such as events, policies, queues, etc.).
- **bootstrap** – A couple of small files that spin the framework up and wire every dependency together.

- **config** – One file per concern. Changing a value here tweaks behaviour without touching code.
- **database** – Migration scripts (schema changes) and seeders (sample or mandatory data).
- **public** – The only folder the web server can see. index.php is the single entry point; everything else (images, compiled CSS/JS, favicons) lives here too.
- **resources** – Uncompiled assets: Blade views, language strings, and the raw SCSS / JS that will later be compiled.
- **routes** – All URL definitions. Each route simply maps an HTTP verb + URI to a controller action.

Laravel, like most modern PHP frameworks, enforces **MVC** (Model–View–Controller). The model represents data and business rules, the *view* renders HTML (or JSON, etc.) for the user, and the controller sits in the middle translating incoming requests into model operations and selecting the right view for the response.

5.2 Key Libraries and Dependencies

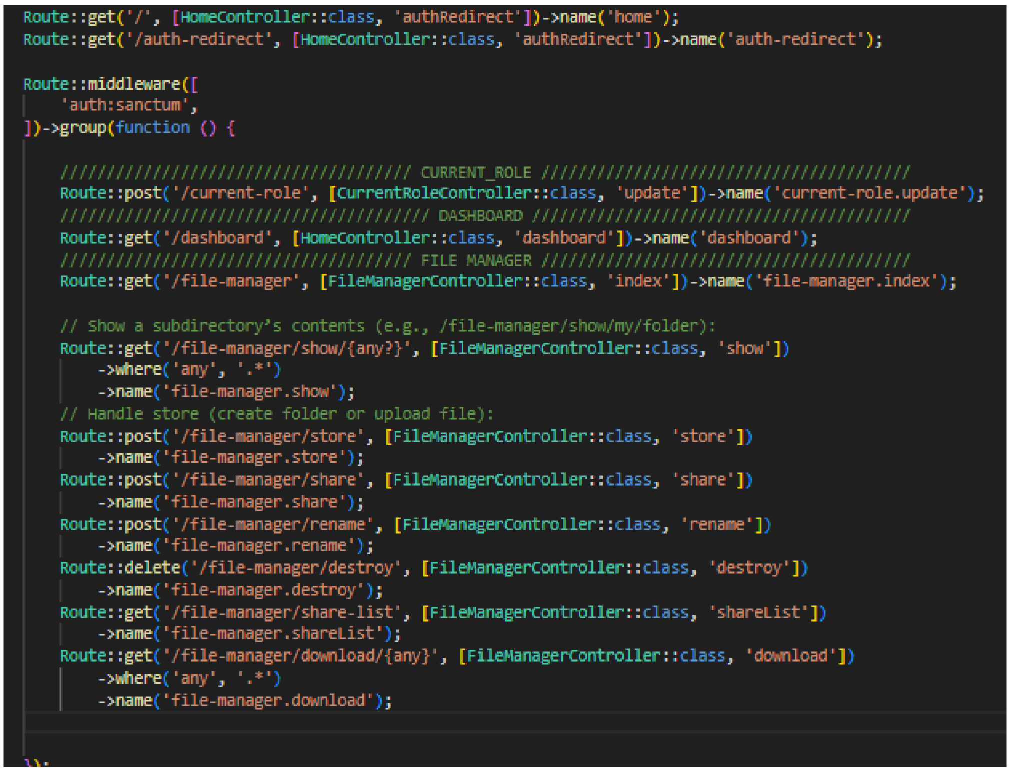
Laravel ships with – and encourages – a set of helper libraries that make development faster, safer, and cleaner:

- **Blade** – Laravel's ultra-light template engine. It lets you embed plain PHP inside HTML. At runtime every Blade file is compiled down to vanilla PHP, so there is almost zero overhead.
- **Middleware** – Mini-filters that intercept every request or response. Perfect place for auth checks, CORS headers, logging, or tweaking the request before it reaches the real controller.
- **Laravel Query Builder** – A fluent, secure, chainable way to build SQL that is resistant to SQL-injection and easy to read.

5.3 Illustrative Code Snippets

5.3.1 Routing

One of the fundamental components provided by Laravel is its routing system, which maps URL endpoints to the code that handles them. The routing mechanism ensures that incoming requests are directed to the correct controller or component along with any required parameters, in the most efficient manner. All web routes are defined in a central file (routes/web.php), which Laravel automatically loads via the App\Providers\RouteServiceProvider. The route definitions clearly list the HTTP method, the URI, and the controller (or action) responsible for each application feature. Following in Figure 5.1 some routes used for by the platform are displayed.

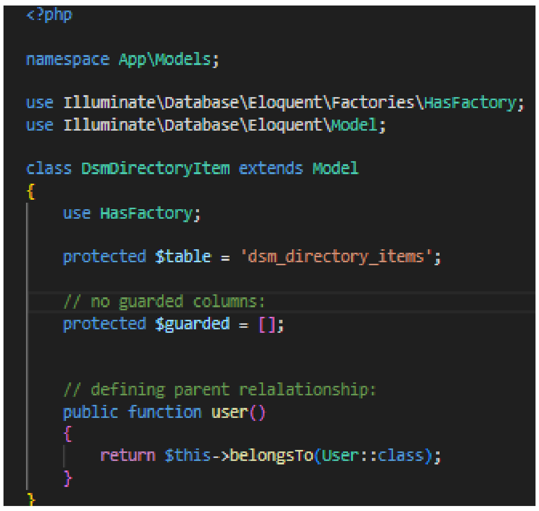


```php
Route::get('/', [HomeController::class, 'authRedirect'])->name('home');
Route::get('/auth-redirect', [HomeController::class, 'authRedirect'])->name('auth-redirect');

Route::middleware([
    'auth:sanctum',
])->group(function () {

    ////////////////////////////////////// CURRENT_ROLE ////////////////////////////////////////
    Route::post('/current-role', [CurrentRoleController::class, 'update'])->name('current-role.update');
    //////////////////////////////////////// DASHBOARD ///////////////////////////////////////////
    Route::get('/dashboard', [HomeController::class, 'dashboard'])->name('dashboard');
    ////////////////////////////////////// FILE MANAGER ////////////////////////////////////////
    Route::get('/file-manager', [FileManagerController::class, 'index'])->name('file-manager.index');

    // Show a subdirectory’s contents (e.g., /file-manager/show/my/folder):
    Route::get('/file-manager/show/{any?}', [FileManagerController::class, 'show'])
        ->where('any', '.*')
        ->name('file-manager.show');
    // Handle store (create folder or upload file):
    Route::post('/file-manager/store', [FileManagerController::class, 'store'])
        ->name('file-manager.store');
    Route::post('/file-manager/share', [FileManagerController::class, 'share'])
        ->name('file-manager.share');
    Route::post('/file-manager/rename', [FileManagerController::class, 'rename'])
        ->name('file-manager.rename');
    Route::delete('/file-manager/destroy', [FileManagerController::class, 'destroy'])
        ->name('file-manager.destroy');
    Route::get('/file-manager/share-list', [FileManagerController::class, 'shareList'])
        ->name('file-manager.shareList');
    Route::get('/file-manager/download/{any}', [FileManagerController::class, 'download'])
        ->where('any', '.*')
        ->name('file-manager.download');


});
```

Figure 5.1 – web.php Code

5.3.2 Controllers and Eloquent Models

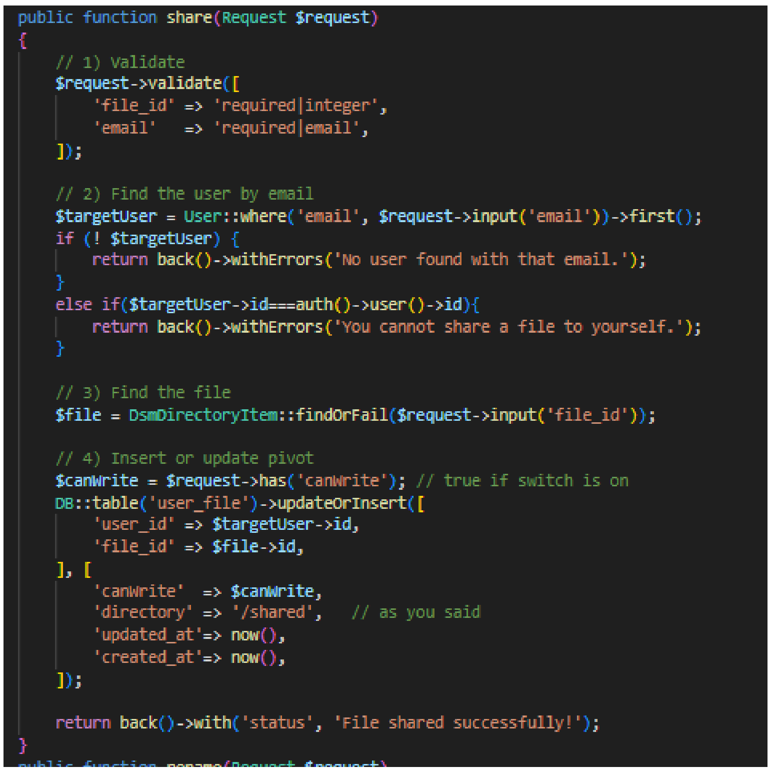
At the center of the platform's data layer is Laravel's Eloquent ORM, which presents a high-level, object-oriented abstraction of the backing relational database (MySQL [13] in this implementation). With Eloquent, every database table is mirrored in the Laravel application by a Model class, which wraps not just the table schema but also any business logic or relationships inherent in that data. This means that developers interact with database records through model objects and methods rather than direct SQL queries. The Model classes define the data structure and enforce relationships and constraints at the application level. For example, in Figure 5.2 is what DsmDirectoryItem looks like, which is a model for the items (folder and files):



```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class DsmDirectoryItem extends Model
{
    use HasFactory;

    protected $table = 'dsm_directory_items';

    // no guarded columns:
    protected $guarded = [];


    // defining parent relalationship:
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Figure 5.2 – DsmDirectoryItem Code

The FileManagerController orchestrates the logic for listing, sharing, deleting, and renaming items, returning a Blade view or JSON response. For instance, in Figure 5.3 we observe the code to share a file:



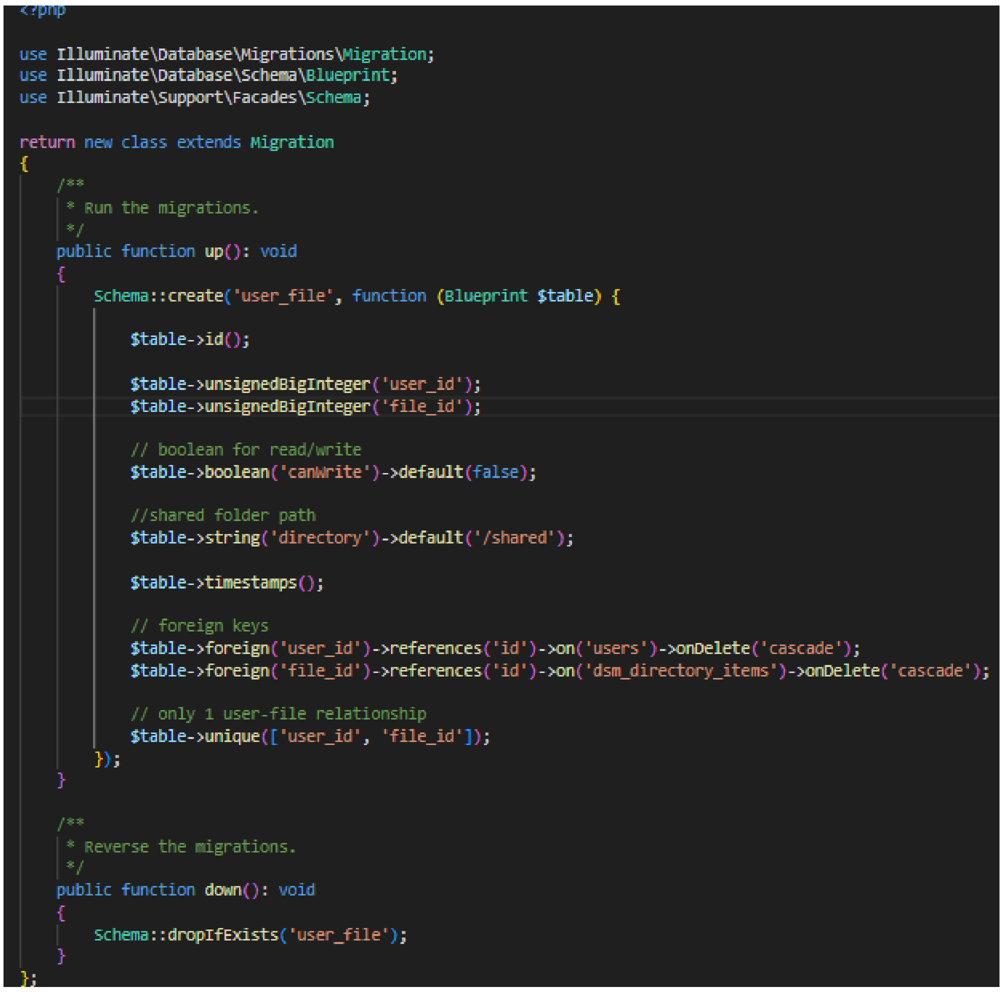
```php
public function share(Request $request)
{
    // 1) Validate
    $request->validate([
        'file_id' => 'required|integer',
        'email'   => 'required|email',
    ]);

    // 2) Find the user by email
    $targetUser = User::where('email', $request->input('email'))->first();
    if (! $targetUser) {
        return back()->withErrors('No user found with that email.');
    }
    else if($targetUser->id===auth()->user()->id){
        return back()->withErrors('You cannot share a file to yourself.');
    }

    // 3) Find the file
    $file = DsmDirectoryItem::findOrFail($request->input('file_id'));

    // 4) Insert or update pivot
    $canWrite = $request->has('canWrite'); // true if switch is on
    DB::table('user_file')->updateOrInsert([
        'user_id' => $targetUser->id,
        'file_id' => $file->id,
    ], [
        'canWrite'  => $canWrite,
        'directory' => '/shared',    // as you said
        'updated_at'=> now(),
        'created_at'=> now(),
    ]);

    return back()->with('status', 'File shared successfully!');
}
public function rename(Request $request)
```

Figure 5.3 – FileManagerController Code For Sharing Functionality

The code validates that the user has given the email to whom the user wants to share a file with and that it has correctly fetched the file_id of the file that will be shared. Then through the User table it finds the desired User by using the users email. Then it finds the file in the DsmDirectoryItem table by using the file's file_id and it checks wether the user has been given read or read/write access. Then we create an entry in the user_file table with the user_id and file_id, we also set the variable canWrite to the permissions of the user, always set the directory to the shared folder and add the time values for updated_at and created_at.

5.3.3 MySQL and Database Migrations

Laravel's migration system, was used to define the database schema in PHP and version control it. For instance, a table to store the pivot table user_file for sharing appears as follows (Figure 5.4).



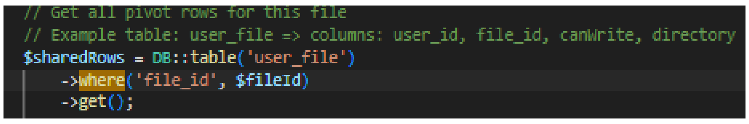
```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('user_file', function (Blueprint $table) {

            $table->id();

            $table->unsignedBigInteger('user_id');
            $table->unsignedBigInteger('file_id');

            // boolean for read/write
            $table->boolean('canWrite')->default(false);

            //shared folder path
            $table->string('directory')->default('/shared');

            $table->timestamps();

            // foreign keys
            $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade');
            $table->foreign('file_id')->references('id')->on('dsm_directory_items')->onDelete('cascade');

            // only 1 user-file relationship
            $table->unique(['user_id', 'file_id']);
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::dropIfExists('user_file');
    }
};
```

Figure 5.4 – user_file Pivot Table Migration Code

Here when we push the migration it creates a table with the following values: user_id, file_id, canWrite, directory and we have foreign keys that link to users and dsm_directory_items tables.

5.3.4 Laravel Query Builder

Following in Figure 5.5 an Eloquent example showing a code snippet that queries all files that are in the user_file pivot table, returning an Eloquent collection for further processing. This will be used to identify which files are currently being shared.

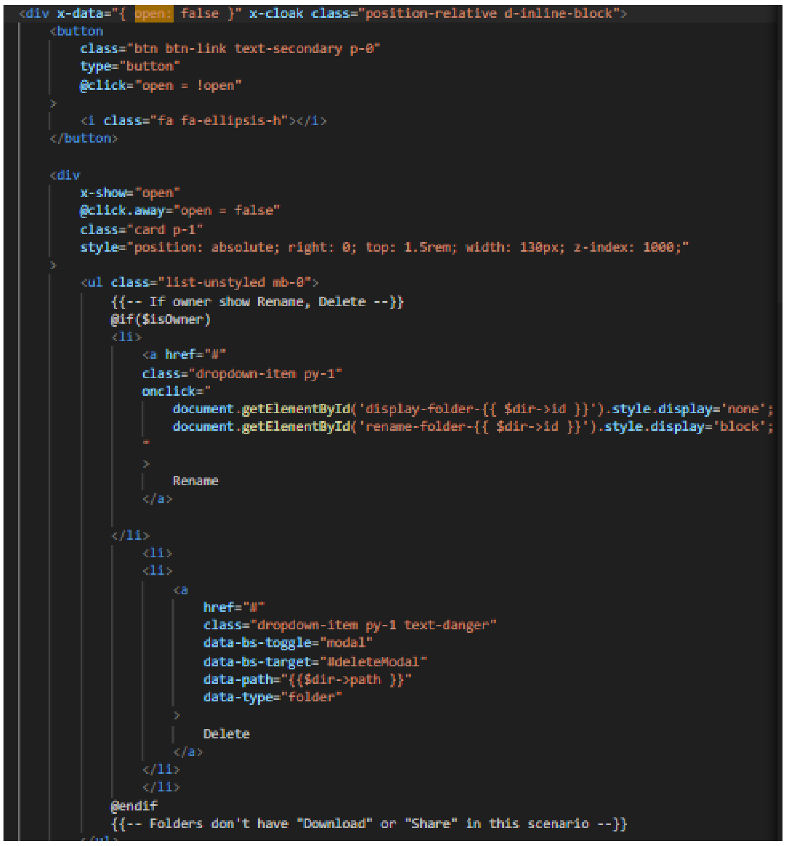


```
// Get all pivot rows for this file
// Example table: user_file => columns: user_id, file_id, canWrite, directory
$sharedRows = DB::table('user_file')
    ->where('file_id', $fileId)
    ->get();
```

Figure 5.5 – Sample Eloquent Code

5.3.5 Bootstrap and Alpine.js for Front End Interactions

For front-end behaviour and presentation, the platform leverages Bootstrap and Alpine.js, two technologies that complement the Laravel ecosystem by providing UI consistency and client-side lightweight interactivity. Bootstrap is a widely used CSS framework that includes a rich collection of pre-styled styles and components for web interfaces. The reason to adopt Bootstrap is the need for a responsive, consistent look-and-feel throughout the application with not much effort going into custom CSS. Bootstrap handles the presentation layer, and Alpine.js is added to handle client-side behavior in a lightweight way. Alpine.js is a "rugged, minimal" JavaScript library that allows developers to add reactive and declarative behavior directly within their HTML markup. It can be thought of as a lightweight framework, with much less overhead cost and complexity than other frameworks. The motivation for using Alpine.js in this platform is to leave a gap for small-scale interactivity that does not need a complete round-trip to the server via Livewire. After Figure 5.9 a code snippet showing the actions toggle button which on clicking shows all the available actions for folders.



```
<div x-data="{ open: false }" x-cloak class="position-relative d-inline-block">
    <button
        class="btn btn-link text-secondary p-0"
        type="button"
        @click="open = !open"
    >
        <i class="fa fa-ellipsis-h"></i>
    </button>

    <div
        x-show="open"
        @click.away="open = false"
        class="card p-1"
        style="position: absolute; right: 0; top: 1.5rem; width: 130px; z-index: 1000;"
    >
        <ul class="list-unstyled mb-0">
            {{-- If owner show Rename, Delete --}}
            @if($isOwner)
            <li>
                <a href="#"
                class="dropdown-item py-1"
                onclick="
                    document.getElementById('display-folder-{{ $dir->id }}').style.display='none';
                    document.getElementById('rename-folder-{{ $dir->id }}').style.display='block';
                "
                >
                    Rename
                </a>

            </li>
                <li>
                <li>
                    <a
                        href="#"
                        class="dropdown-item py-1 text-danger"
                        data-bs-toggle="modal"
                        data-bs-target="#deleteModal"
                        data-path="{{$dir->path }}"
                        data-type="folder"
                    >
                        Delete
                    </a>
                </li>
                </li>
            @endif
            {{-- Folders don't have "Download" or "Share" in this scenario --}}
```

Figure 5.9 – Code Snippet For Bootstrap and Alpine.js

Chapter 6

Conclusion

6.1 Summary of Achievements 60
6.2 Lessons Learned 61
6.3 Limitations and Future Work 62
6.4 Final Remarks 63

6.1 Summary of Achievements

This thesis set out to conceptualize, develop, and evaluate a distributed web platform that would manage files and directories and emphasize user experience, security, and scalability. The technology foundation revolved around the Laravel framework and was supplemented with concepts borrowed from distributed shared memory (DSM) systems and object-based storage (CoBFS). These paradigms were reflected in both the software architecture and concurrency, access control, and data persistence strategy of the system.

The process of development centred around a user-focused design philosophy. Much time was spent building an intuitive and interactive user interface that allowed seamless directory and subdirectory navigation. The users were provided with the ability to upload new files and transfer existing files, rename existing items, delete existing data, and retrieve comprehensive metadata for folders and files. Since it was understood that user interests vary greatly, the platform brought in both list and grid views to navigate through. This visual variability accommodated different workflows and accessibility requirements, maximizing the platform's flexibility to accommodate different groups of users.

Our primary concern was that the core functionality should be robust and scalable. The choice for the Model-View-Controller pattern in Laravel was made on architectural beauty as well as on maturity of the existing framework that allows reusing solutions to a variety of commonly seen problems in web applications-such as session management: where stateful user interactions and authentications come into play. These became the grounds for more complicated flows, whereby files and directories can be renamed, downloaded, deleted, or shared, always with very high integrity and traceability on the data.

Security and dependability had also been augmented through the embracing of certain best practices and technologies. CSRF tokens were applied against forms and state-changing requests to prevent abuse of commands. User credentials were hashed in some secure manner to minimize password exposure or brute-force attacks. Read and write access was very finely locked down throughout the platform, and shared resources respected the level of access that the owners had defined. The distributed system concepts from DSM and object-based storage in particular informed the design for how to keep consistency in the presence of concurrent user actions and hence indicate a path towards horizontal scale-out across numerous nodes in the future.

One other of the key achievements is the transparent integration between the frontend based on Laravel and the backend built by Algolysis' research team that offers a DSM service. The frontend interacts with the backend only through an array of APIs so that all the operations of a user, whether browsing directories, downloading or uploading files, renaming objects, or deleting can be routed to the backend through the APIs. This architecture maintains the frontend lightweight and user-friendly while offloading the heavyweight distributed consistency and concurrency control to the backend engine. Clearly defined API endpoints enable modularity, scalability, and a clear separation of concerns, thus enhancing maintainability and supporting future growth. Regardless of whether it's dealing with a single node or a distributed multi-node environment, the frontend is kept unaware of the data storage logic behind it, using these APIs to execute operations securely and effectively.

Together, these accomplishments provided a working prototype that effectively demonstrates how existing concepts in distributed computing can be put into alignment with a high-end web application suite. The platform design and implementation compromise on ease of use versus technical sophistication to deliver a safe, extensible mechanism for shared directory and file management.

6.2 Lessons Learned

The development of a distributed platform to facilitate concurrent access and sharing of resources came with an abundance of valuable lessons, from issues technical to more broad principles of software engineering.

Making Laravel the starting point was the most obvious of decisions. As it followed the MVC principle, it was able to separate concerns cleanly and encapsulate business logic.

The introduction of Eloquent ORM eliminated the necessity to write raw SQL which is cumbersome and error-prone and instead gave a cleaner way of dealing with data access. Middleware security policies and HTTP request preprocessing made for an elegant implementation, while addressing the entire session and authentication needed for rapid development of secure, stateful UIs.

A second significant takeaway represented the essential value in maintaining good, adaptive user interfaces. The list view and grid view implementations illustrated the technical side of conditional rendering and state management, while user interface design-related factors influence productivity and satisfaction, hence the platform was able to entertain more interaction types and usage models with multiple modes of interaction supported.

Moreover, the iterative and cooperative style of development was a significant factor in the project's success. Regular communication with the company, along with continuous code review and testing, ensured a culture of continuous improvement. With an agile philosophy of incrementally adding features, testing them with user feedback, and iterating on the design, there was guaranteed alignment of the platform to the company's needs.

6.3 Limitations and Future Work

Some built-in limitations were encountered along the way, either during development or marketed as points of improvement down the line. The system has mostly been benchmarked with small-size datasets. While touted as theoretically scalable in design, the actual deployment into a high-traffic, multi-node production setup would call for extra complexity layers. The areas ripe for further enhancements include distributed caching mechanisms, more advanced load-balancing solutions, and data replication strategies for fault tolerance and availability at scale.

In the security realm, well-defined for the current scale, there lays a big opportunity for higher-order augmentation. Complete RBAC, SSO support, enterprise-grade auditing, and logging facilities have all been cited as being high-value additions. These improvements not only enhance the platform's security position but also start to force the platform into levels of compliance and regulation demanded on the enterprise level.

Another constraint involves a lack of offline use and caching. Some existing distributed applications benefit from having the capability to queue operations or cache data locally if the user's network connection is unstable or does not exist. Offering offline synchronization, in addition to advanced conflict resolution strategies, would be greatly enhanced for usability in real-world scenarios where connectivity cannot be guaranteed.

Future evolution could also introduce additional features such as data versioning, which would allow users to revert back to previous states when inadvertently altering or deleting files. Increased monitoring and telemetry would provide real-time insight into usage patterns, performance bottlenecks, and concurrency hotspots, which would inform both users and administrators. Finally, the application of machine learning algorithms could make tagging and classifying uploaded content automatic, allowing for more advanced search and organization features.

6.4 Final Remarks

The platforms construction and deployment represent a telling case of the convergence of novel web frameworks, distributed storage methods, and people-oriented design towards the development of a usable, secure, and scalable application. The facility to compose Laravel's flexible toolbox with familiar distributed computing concepts is a definitive sign of the viability of merging popular web development technologies and frontier systems design.

The approach of iterative prototyping, comprehensive testing, and high modularity focus taken in this project has not only yielded a viable prototype but also a flexible template for further development. With ongoing changes in the distributed computing environment, particularly with cloud computing and decentralized paradigms, this platform is set for further refinement and development.

Hence, this project serves as a reminder that achieving the goal of an efficient distributed system centred around the user requires more than just technical know-how. It requires a deliberate balancing act among software engineering best practices, sound knowledge of distributed systems theory, and constant interaction with and feedback from the users. Keeping these in mind, the platform serves as the basis over which scalable, fault-tolerant, and user-friendly solutions can be implemented to address the pressing data management problems in today's digital landscape.

References

[1] A. F. Anta, C. Georgiou, T. Hadjistasi, N. Nicolaou "Fragmented Objects: Boosting Concurrency of Shared Large Objects," *Proc. of the 28th International Colloquium on Structural Information and Communication Complexity (SIROCCO 2021)*, pp. 106-126, Wroclaw, Poland, Online, 2021.

[2] C. Georgiou, N. Nicolaou, and A. Trigeorgi, "Fragmented ARES: Dynamic Storage for Large Objects", *Proc. of the 36th International Symposium on Distributed Computing (DISC 2022)*, pp. 25:1-25:24, Augusta, GA, USA, 2022.

[3] Lynch, N. and Shvartsman, A. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In Symposium on Fault-Tolerant Computing. IEEE, 1997.

[4] P. Musial, N. Nicolaou, and A. A. Shvartsman, "Implementing Distributed Shared Memory for Dynamic Networks," *Commun. ACM*, 2014.

[5] N. Nicolaou, A. Fernandez Anta, and C. Georgiou. Coverability: Consistent versioning in asynchronous, fail-prone, message-passing environments. In Proc. of IEEE NCA 2016.

[6] Nicolas Nicolaou, Viveck Cadambe, N. Prakash, Andria Trigeorgi, Kishori M. Konwar, Muriel Medard, and Nancy Lynch. ARES: Adaptive, Reconfigurable, Erasure coded, Atomic Storage. ACM Transactions on Programming Languages and Systems (TOPLAS), 2022.

[7] Bootstrap Documentation – *Introduction to Bootstrap (Responsive Front-end Framework)*. [Online]. Available: https://getbootstrap.com/docs.

[8] Alpine.js – *Alpine.js Official Website* (Lightweight JavaScript Framework). [Online]. Available: https://alpinejs.dev.

[9] Oracle Corporation, "MySQL: Understanding What It Is and How It's Used," Oracle.com, 2023. [Online]. Available: https://www.oracle.com/mysql/what-is-mysql/.

[10] "XAMPP," *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/XAMPP.

[11] "Model–View–Controller (MVC)," *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Model–view–controller.

[12] "Distributed shared memory", *Wikipedia, The Free Encyclopedia*. [Online]. Available: https://en.wikipedia.org/wiki/Distributed_shared_memory.

[13] "Agile software development", *Wikipedia, The Free Encyclopedia*. [Online].
Available: https://en.wikipedia.org/wiki/Agile_software_development

[14] "Linearizability", *JEPSEN.* [Online].
Available: https://jepsen.io/consistency/models/linearizable

[15] Laravel Documentation – *Laravel: The PHP Framework for Web Artisans*. [Online]. Available: https://laravel.com/docs.

[16] "MariaDB", *Wikipedia, The Free Encyclopedia.* [Online].
Available: https://en.wikipedia.org/wiki/MariaDB

[17] "Amount of Data Created Daily (2025) ", Exploding Topics**.** [Online].
Available: https://explodingtopics.com/blog/data-generated-per-day

[18] Git Documentation – *Git Official Website* [Online].
Available: https://git-scm.com/doc

[19] Livewire Documentation – *Laravel-Livewire Official Website* [Online].
Available: https://laravel-livewire.com/docs/2.x/quickstart

[20] Gilbert, S., Lynch, N. and Shvartsman, A. RAMBO: A robust, reconfigurable atomic memory service for dynamic networks. Distributed Computing 23, 4, (Dec. 2010), 225–272.

[21] Aguilera, M.K., Keidar, I., Malkhi, D. and Shraer, A. Dynamic atomic storage without consensus. JACM 58 (Apr. 2011), 7:1–7:32

Annex A

This appendix presents the user manual for the distributed file management platform.

Login

To access the web platform, the user must open a modern web browser and navigate to the designated URL provided by the organization or administrator. The login page is the first screen presented. Users must possess valid credentials issued by the administrator to gain access.

1. Enter your email and password in the respective fields.
2. Click the Login button to proceed.

Navigation Menu

The navigation menu is located at the top left of every page within the web platform. It provides quick access to the primary sections:

- Dashboard
- My Profile
- File Manager

The navigation bar remains consistent across all views to ensure an intuitive and seamless user experience.

Home Page

Upon successful login, users are directed to the Dashboard page. Here, a brief overview of the system and recent updates may be displayed (depending on administrator configuration). Users can then proceed to the File Manager to begin interacting with files and directories.

File Manager

The File Manager serves as the core of the platform, allowing users to manage their files and directories efficiently.

Browsing and Navigating

The main panel displays the contents of the current directory, including subfolders and files. Users can navigate into subdirectories by clicking on folder names. A breadcrumb navigation system appears at the top, allowing users to move back to parent directories easily.

Users can choose between List View and Grid View by clicking a toggle button on the top right. The List View provides detailed metadata for each item (such as size, permissions, and creation date), while the Grid View presents a visual, tile-based layout.

Uploading Files

To upload files:

1. Click the New button located in the upper right corner of the File Manager interface.
2. Select the desired files using the provided file picker or drag-and-drop them into the upload area.
3. Optionally, specify the target directory or create a new folder.
4. Click Upload to initiate the process.

Creating Folders

To create a new folder:

1. In the upload modal, enter the desired folder name in the “New Folder” section.
2. Optionally, select any files that you want to add to the folder.
3. Click Upload.

Renaming Items

To rename a file or folder:

1. Click the three dots menu on the right side of the item.
2. Select Rename.
3. Enter the new name and press enter.

Deleting Items

To delete a file or folder:

1. Click the three dots on the right side of the item.
2. Select Delete.
3. Confirm the deletion in the pop-up modal.

Please note that deleted items cannot be recovered.

Download Files

To download a folder:

1. Click the three dots on the right side of the item.
2. Select Download.

The file will automatically get downloaded on your device.

Sharing Files

Users can share files with other registered users:

1. Click the three dots menu next to the file.
2. Select Share.
3. In the sharing modal, enter the recipient’s email.
4. Optionally, grant write permissions by enabling the "Can Write" switch, otherwise the user will only have read access.
5. Click Submit.

The recipient will immediately gain access to the shared file in their Shared directory.

Viewing Item Information

For detailed information about a file or folder in the Grid view:

1. Click the three dots menu next to the item.
2. Select Info.
3. A modal will appear showing the item's name, type, size, permissions, creation date, last updated date etc.

Account Settings

To change your name or email:

1. Click on the “My Profile” option in the navigation bar.
2. Go to the “Profile Information” section put your desired information on the Name and Email fields.
3. Click Save.

To change your password

1. Click on the “My Profile” option in the navigation bar.
2. Go to the “Update Password” section.
3. Enter your current password, your new password and reconfirm your new password.
4. Click Save.

Annex B

This appendix presents a dictionary of some technical terms found in the paper.

Agile

A software development approach that emphasizes incremental delivery, collaboration, and flexibility in responding to changes.

Alpine.js

A lightweight JavaScript library used to add interactive features to web pages without requiring large frameworks.

API (Application Programming Interface)

A set of rules and tools that allows different software systems to communicate with each other.

Apache

An open-source web server software that delivers web content to users' browsers.

Authentication

The process of verifying the identity of a user or system.

Authorization

The process of determining what actions or resources a verified user is allowed to access.

Back-end

The server-side part of a web application where data processing, business logic, and database interactions occur.

Blade

Laravel’s built-in templating engine used to create dynamic web pages.

Bootstrap

A popular front-end framework that helps design responsive, mobile-friendly web pages.

Client-side

Refers to operations that are performed in the user's web browser rather than on the server.

COBFS (Coverable Block File System)

A system that manages data storage in blocks, supporting version control and consistency.

Controller

In MVC architecture, the component that processes user requests, works with models, and selects the correct views.

CRUD (Create, Read, Update, Delete)

The four basic operations for managing data in an application.

CSRF (Cross-Site Request Forgery)

A type of web security vulnerability where unauthorized commands are transmitted from a user that the web app trusts.

Database Migration

The process of managing database schema changes over time using version control.

DAG (Directed Acyclic Graph)

A graph data structure used to represent processes or data that flow in one direction without cycles.

Data Integrity

Ensuring the accuracy and consistency of data over its lifecycle.

Distributed Shared Memory (DSM)

A method for sharing memory across multiple networked computers, making them appear as a single memory space.

Dynamic Reconfiguration

The ability of a system to change its structure or behaviour without stopping operations.

Eloquent

Laravel's built-in ORM (Object-Relational Mapping) system for working with databases using PHP objects.

Encryption

The process of converting data into a coded format to prevent unauthorized access.

Front-end

The part of a web application that users interact with, typically built using HTML, CSS, and JavaScript.

Git

A version control system used to track changes in code during software development.

HTTPS (Hypertext Transfer Protocol Secure)

A secure version of HTTP, which encrypts data exchanged between a user's browser and the server.

JSON (JavaScript Object Notation)
A lightweight data format commonly used for exchanging data between servers and web applications.
Laravel
An open-source PHP framework designed for building web applications using the MVC architecture.
Livewire
A Laravel package that allows creating dynamic interfaces without writing much JavaScript.
MariaDB
An open-source relational database management system, often used as a drop-in replacement for MySQL.
Metadata
Data that provides information about other data, such as file size, creation date, or permissions.
Middleware
Software that acts as a bridge between different parts of an application, often used for handling requests and responses.
Model
In MVC architecture, the component that handles data and business logic.
MVC (Model-View-Controller)
A software design pattern that separates an application into three interconnected components: Model, View, and Controller.
MySQL
A widely used relational database management system for storing and managing data.
ORM (Object-Relational Mapping)
A technique that allows developers to interact with databases using object-oriented programming concepts.
PHP (Hypertext Preprocessor)
A popular scripting language used for web development and creating dynamic web pages.
phpMyAdmin
A web-based tool for managing MySQL and MariaDB databases.

RBAC (Role-Based Access Control)

A method for restricting system access based on a user's role.

Responsive Design

An approach to web design that ensures web pages look good on all devices, from phones to desktops.

Router

A system that maps incoming web requests to specific code or controller actions.

Scalability

The ability of a system to handle increasing amounts of work or data without performance loss.

Session Management

The process of managing user sessions in web applications, including login status and activity tracking.

SQL (Structured Query Language)

A language used to communicate with relational databases.

SSL/TLS (Secure Sockets Layer/Transport Layer Security)

Protocols that provide secure communication over a computer network.

Unit Testing

A software testing method where individual components of an application are tested in isolation.

User Interface (UI)

The part of a software application that users interact with directly.

Version Control

A system for managing changes to software code over time.

View

In MVC architecture, the component responsible for presenting data to the user.

XAMPP

An open-source package that provides an easy-to-install Apache server, PHP, and MySQL environment for development.