University of Elmergib

Faculty of Information Technology

Implementation of Secured Cloud Storage using Node.js and MongoDB

Submitted By:

عبد الرؤوف مفتاح القاضي	1811058
عبد العزيز أبوراس	1711082
عماد خالد سعود	220181028

Supervised by:

أ. فوزي العربي

This graduation project is submitted to the Computer Network Department as partial fulfillment for the requirements for obtaining a Bachelor's degree in Computer Networks.

الملخص

الحوسبة السحابيه قامت بثورة في ادارة البيانات و التطبيقات عن طريق انشاء خدمات عبر الانترنت قابلة للتوسع و قادرة علي استيعاب الطلبات. الحوسبة السحابية تزيح الحاجة لامتلاك المكونات الماديه بحيث يستطيع المستخدمون للنظام السحابي الاستفادة من الموارد المتاحة مثل أجهزة التخزين عبر الانترنت و من أين ما كانو.

التخزين السحابي هو نموذج خدمة للحوسبة السحابية يتيح تخزين و ادارة و استرجاع البيانات من أي مكان مع ضمان أمان البيانات من الوصول الغير مصرح به. أنظمة التخزين السحابي هي جزء مهم للأنظمة السحابي بحيث يجب علي النظام السحابي أن يكون قادر علي تلبية العديد من الوظائف من تخزين و حماية ملفات الأفراد الي ادارة بيانات الشركات و المنظمات الضخمة.

يهدف هذا المشروع لانشاء نظام سحابي محمي يوفر تخزين و استرجاع الملفات بطريقة امنه و فعاله. تم بناء هذا التطبيق باستخدام Node.js و هو بيئة تشغيل للغة JavaScript, و MongoDB و هي قاعدة بيانات غير علائقية.

اضافة علي انشاء النظام السحابي قمنا بانشاء البيئة التشغيلية للنظام السحابي و التي تعتمد علي استخدام جدار ناري Sophos XG و قد تم استخدامه كواجهة gateway لنظام التشغيل الذي يحمل النظام السحابي. و لاضافة طبقة اخري للحماية في النظام السحابي قمنا لاستخدام أداة Suricata لكشف و منع التطفل. قمنا بتشغيل النظام السحابي علي نظام تشغيل Linux Mint و هو نظام بسيط و فعال مبني علي توزيعة Ubuntu.

في المجمل هذا المشروع يقدم تطوير و تشغيل ناجح و فعال لنظام التخزين السحابي باستخدام تقنيات متقدمة مثل Node.js و MongoDB اضافة الي طبقات حماية اخري لانشاء بيئة تشغيل محمية للنظام السحابي مثل Sophos XG و Suricata و Linux Mint.

Abstract

Cloud computing has revolutionized data and application management by enabling scalable and on-demand services across the internet. The technology eliminates the need for physical infrastructure so that businesses and users can access computing resources, storage, and other services via the internet from remote locations. Cloud storage is a service model of cloud computing that enables storing, maintaining, and retrieving data from any location while ensuring data availability, redundancy, and security. Cloud storage facilities are now fundamental components of information technology systems, handling any quantity of varied functions from individual file protection to company-wide data handling.

This project aims to create a cloud storage application that will offer a secure and efficient file storage and retrieval. The system was built utilizing Node.js, a JavaScript runtime environment, with MongoDB, a NoSQL database used for its flexibility and scalability.

For providing security and reliability to the application, the deployment environment was configured by utilizing Sophos XG as gateway firewall and Suricata for Intrusion Detection and Prevention Systems (IDS/IPS). The Sophos XG functions as the principal gateway delivering enhanced threat mitigation, web content filtering, and oversight of network traffic. In conjunction with this configuration, Suricata enhances security by providing real-time intrusion detection and prevention mechanisms. The system was implemented on Linux Mint, which is a simple and efficient Linux Operating system based on Ubuntu.

In conclusion, this project thus proves to be successful development and deployment of a cloud storage application with advanced technologies such as Node.js and MongoDB, with added security provided through a strong deployment environment supported by Sophos XG, Suricata, and Linux Mint. The outcome is therefore a scalable and secure cloud storage application.

TABLE OF CONTENTS

TABLE OF FIGURES

Chapter 1: Introduction

1.1 Introduction

In This Project we will implement Cloud Storage application with Node.js and MongoDB in a Virtual Machine running Linux Mint and protect the server traffic using Sophos XG firewall.

We decided to use Node.js for its event-driven architecture that makes it easier to handle concurrent requests efficiently. That would make it ideal for uploads and downloads of files. Its extensive open-source library and framework ecosystem, like Express and Mongoose, are also perfect for fast development of web applications; solid solutions to common problems in cloud storage are provided, especially for NodeJs.

We went with MongoDB because it is easily integrated into Node.js applications. As for the firewall, we used Sophos XG because it was easy to implement into this environment-it has built-in advanced malware, ransomware, and intrusion attempt protection.

1.2 Project Aim

Implement secured Cloud-based Storage web application that provides file uploads and downloads functionality to the users of the application. and use industry standards for authentication and authorization of users.

Besides implementation of the Cloud storage application we will use Sophos XG firewall to enhance the server security and prevent unauthorized access and use of our resources.

1.3 Problem Statement

This project will include basic upload and download of files, and we will implement secured authentication and authorization mechanisms through the implementation of JWT tokens.

We will ensure only the real owner of the file would have access to the file and that no other user is could download the file apart from him.

1.4 Project Objectives

- Learn about cloud based applications.
- Learn about client-server applications.
- Design the system architecture.
- Install the necessary tools for the development and the production of the application.
- Create the database models of the application resources.
- Create the controller functions for the middleware stack.
- Implement the API on the server side.
- Implement the view of the application.
- Configure the virtual network of VMware workstation Pro.
- Install and configure virtual Sophos firewall on VMware workstation Pro.

1.5 Project Scope

The project will cover the implementation of the cloud storage service with only the upload and download functionality so it's not possible for the user to delete or edit the files he already submitted to the server.

1.6 Technology Stack Overview

- **Node.js**: is a JavaScript runtime environment basically used to allow JavaScript to run outside of the browser, so Node.js makes it possible to use JavaScript in the server-side besides the client-side.
- **Express.js**: is a minimal and flexible web application framework for Node.js, designed to simplify the development of web and API applications and it provides powerful set of features for handling HTTP requests and responses, and it simplifies the development of the API with the use of routes and middleware functions.
- **MongoDB**: is an open source, non-relational database management system (DBMS) that uses flexible documents to process and store various forms of data.

- **Sophos XG**: is an advanced firewall solution designed to protect network traffic and secure data transmissions.

1.7 Project Outlines

Chapter 1: Introduction

Chapter 2: Cloud Computing Fundamentals

Chapter 3: System Design and Architecture

Chapter 4: Deployment Environment and Security measures

Chapter 5: Implementation

Chapter 6: Conclusion

Chapter 2: Cloud Computing Fundamentals

1.8 Introduction

The concept of cloud computing can seem complex. In this chapter, we will begin with a general overview of the cloud and its related concepts. We will then delve into the key factors driving organizations to adopt cloud technologies. Finally, we will address some of the challenges that limit broader cloud adoption.

The content and structure of this chapter are significantly informed by ***The Basics of Cloud Computing*** by **Derrick Rountree** and **Ileana Castrillo**, which offers an in-depth exploration of cloud computing principles.

What is the Cloud?

There has been a debate of what the cloud is, some people define it as a set of technologies but these technologies are not the core of the cloud, the cloud is primarily a set of services. This is partially that the cloud has been hard to define.

Initially, the cloud was viewed as a collection of combined services, technologies, and activities, with its internal workings hidden from users. This opacity is partly how the cloud earned its name. However, this definition has evolved over time. Providers have recognized that, while some users remain indifferent to the inner workings, many others are keenly interested. This growing interest has led providers to be more transparent about their operations. In many instances, customers are even given the ability to configure their own system monitoring solutions.

Like all services, the cloud and its offerings have evolved over time. Most services, particularly technology-related ones, adapt quickly to meet customer needs. Consider the services you use—how many have remained unchanged? Likely very few. Service providers must continually adjust and refine their offerings to stay relevant and valuable, and the cloud is no exception. This constant evolution has contributed to the confusion surrounding its definition. Every time someone proposed a solid definition, the services would change. Many believed that the National Institute of Standards and Technology (NIST)

would provide a definitive description of cloud computing. However, even the NIST has revised its definition over time.

Even with the changes, NIST definition is still the most preferred for most people, the NIST definition has three main components that we will discuss:

- Five key cloud characteristics
- Four cloud deployment models
- Three cloud Service models

1.9 Key Cloud Characteristics

Many companies and service providers have sought to capitalize on the cloud's popularity. However, some claim to offer cloud services without meeting the criteria for true cloud computing. Simply being web-based does not make an application is a cloud application. For an application and its surrounding service to qualify as a genuine cloud implementation, it must demonstrate specific characteristics. According to the NIST definition of cloud computing, these include on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service. All five characteristics must be present for an offering to be considered a true cloud service.

2.2.1 On-Demand Self-Service

One of the most important capabilities, on-demand self-service enables customers to request and receive a service offering without requiring much, if any, intervention from administrators or support staff. Additionally, the request and fulfillment are fully automated; this brings about major benefits to both service providers and customers alike.

The self-service of users enables them to obtain and access their required services much quicker, so it's highly in demand by the cloud. It speeds up the resource provisioning process to be quicker, easier, and simpler. In traditional environments, this was always destined to take some time, often days or weeks, finding delays in projects. In cloud environments, this is no longer an issue.

Adding self-service can be painful, but for cloud providers it is certainly worth the investment of time and resources. Typically, user self-service is provided through a user portal. Several out-of-the-box portals are available to provide the requisite functionality; occasionally a portal may need to be custom built for particular cases. On the front-end, users interact with a friendly user interface that captures the required information. On the back-end, the portal interacts with management APIs provided by the applications and services. This can be quite painful if the back-end systems do not have APIs, or some other way to automate operations seamlessly.

2.2.2 Broad Network Access

The services should be easily accessible with only a basic network connection for users to access applications or services. The connection is normally done via the internet. The quality of internet bandwidth is improving, but it is still a long way behind a LAN connection in terms of speed. In relation to this, service providers should ensure that their services do not demand much bandwidth on the part of users in order for them to use the services.

2.2.3 Resource Pooling

Resource pooling reduces the cost for the providers; it introduces flexibility. It is premised on the fact that the clients do not always require all the available resources. Resources used by one customer are available to others if they are not in use by that particular customer. This enables the providers to serve far more customers than they could if each customer required dedicated resources.

Resource pooling generally realizes from virtualization. Virtualization allows providers to increase the density of the systems. Virtualization allows running multiple virtual sessions on a single physical system. The resources of one physical machine are pooled and made available for use by multiple virtual systems in this environment.

2.2.4 Resource Elasticity

Rapid elasticity refers to the capability of a cloud environment to scale seamlessly to meet user demand. Cloud deployments should have the necessary infrastructure in place to expand service capacity when required. With a well-designed system, scaling may involve simply adding additional computing resources, storage, or similar components. The key is that these resources remain idle until needed, enabling providers to minimize consumption costs, such as power and cooling, when demand is low.

2.2.5 Measured Service

The cloud service should be measurable in terms of usage that can be attained by any metric-like time, bandwidth, or data consumed. It is this measured service characteristic feature that allows the 'pay-as-you-go' cloud computing model. Once a proper metric is decided, a rate is given which states how much customers will be charged for their consumption. This 'measured' approach will charge the client only when they utilize the resource, and if they do not consume it on any particular day then no charges will apply.

1.10 Cloud Deployment Models

Cloud computing can be applied in many different ways, and every organization is unique in the requirements of the services needed and the level of control over their environment. To better serve these diverse needs, there are several different cloud environment deployment models that an organization can use. Of course, each of these is associated with various requirements and benefits. The NIST definition of cloud computing recognizes four basic deployment models: public, private, community, and hybrid.

2.3.1 **Public**

In the public cloud service model, all the systems and resources that enable the service are hosted by an external service provider. This provider is responsible for oversight and administration of these systems in order to ensure their operational performance and maintenance. The client is only responsible for any installed software or client applications located on user-end devices. Public cloud services are usually made available through an Internet access.

2.3.2 **Private**

In a private cloud, the infrastructure and resources used to deliver the service are managed within the organization that is using them. The onus of management and maintenance of these systems, along with their software or client applications on end-user devices, lies with the organization. Private clouds are usually accessed via a local area network (LAN) or a wide area network (WAN).

2.3.3 **Community**

Community clouds are semi-public communities shared by a set of organizations for some particular purpose or mission. Since the community members would need greater assurance of privacy and security than that provided by a public cloud, they do not use the latter. They also do not depend on a single organization to handle the cloud operation but share it among themselves.

2.3.4 **Hybrid**

A hybrid cloud model is formed when two or more different cloud deployment models are combined. That means even though each cloud remains distinct, they are joined together to function as one system. Hybrid clouds do add more complexity to the environment; however, they provide added flexibility for differing organizational goals.

1.11 Cloud Service Models

The NIST definition of cloud computing outlines three basic service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

2.4.1 Infrastructure as a Service

Infrastructure as a Service –IaaS– provides the core infrastructure services to the clients, which include physical machines, virtual machines, network capabilities, storage options or any combination of these resources. IaaS serves as an alternative to traditional on-premises infrastructure, that requires an organization to maintain its own systems. IaaS allows both individuals and organizations to rent or buy resources as it eliminates the maintenance burden from them since this responsibility is taken care of by the cloud service provider.

2.4.2 Platform as a Service

PaaS is a cloud computing model where a flexible, elastic, and scalable platform for developing, deploying, running, and managing applications is delivered. With PaaS, developers get everything they may need for application development without having to care about hardware management or upgrading of operating systems and development tools. Instead, all this would be given out and kept by a third-party provider through the cloud.

2.4.3 Software as a Service

Software as a Service (SaaS) delivers applications and data services through the cloud. In this model, the service provider manages and provides the applications, data, and all underlying platforms and infrastructure. SaaS is the original cloud service model and remains the most popular, offering the widest range of provider options.

1.12 Cloud Drivers

The cloud offers people numerous new opportunities. In the past, rolling out new applications required significant upfront investments in infrastructure and staff training. Now, depending on the provider, those costs can be dramatically reduced. The cloud has played a key role in enabling a new era of consumerism. End users are no longer confined to applications that don't meet their preferences or needs. They can more easily transition to alternatives that better suit their requirements. While the process isn't entirely seamless, it is far simpler than it once was.

System Drivers

Several factors are driving organizations toward the adoption of cloud-based solutions. Often, organizations seek specific system characteristics that their existing infrastructure cannot support. Lack of expertise or economic resources may prohibit them from achieving these characteristics on their own, which encourages them to look to cloud service providers for support. The main attributes offered by cloud computing are agility, reliability, scalability, and performance.

2.5.1 Agility

Cloud environments are very agile; one can quickly move resources to wherever these resources are needed. Such flexibility allows you to shuffle your resources more to systems needing them, scaling down systems with lesser needs. This allows for the addition of new systems in order to scale up capacity.

Organizations are able to optimize their infrastructure resource usage in an internal cloud environment. Virtualization allows large increases in infrastructure density and utilization rates within a cloud environment. As such, this minimizes the possibility of the system's sitting idle apart from a few, thereby maximizing the utilization of the existing resources.

2.5.2 Reliability

Achieving reliability normally comes expensive in terms of your environment, either through the replication of several systems or across data centers. Included DR and continuity planning involving such complexity should be coupled with drills to ensure readiness against disruptions.

The fact is, most cloud providers already have infrastructure across multiple locations, so you can instantly increase the reliability of your environment just by leveraging their services. You might have to make a request to use multiple locations for your service, but having that option, which is readily available, greatly simplifies things for you.

2.5.3 Scalability

A cloud environment can automatically scale to accommodate customer demands. Additional resources can be dynamically allocated to handle increased usage, ensuring customer needs are consistently met.

2.5.4 Ease of Maintenance

Probably, the most considerable advantage of cloud computing is ease of maintenance, which changed the whole industry of cloud computing for many business entities. If a third-party provider is managing the infrastructure, hardware, or software, this big workload and complexity is taken from the customer. It means the enterprise can just focus on their major activities, like the development of applications, by shifting the deployments and infrastructure management tasks to a cloud service provider.

2.5.5 Cost

Ecosystems of cloud computing bring in big cost savings, mostly through the conversion of capital expenditures into operating expenses. Under the traditional model, organizations are forced to invest heavily in infrastructure and hardware at the outset, usually paid for out of their capital budget. In a cloud ecosystem, such large upfront investments are not needed. Instead,

organizations pay only for the services they use, making their expenses more controllable and predictable.

1.13 Summary

For a cloud application to be rightly tagged as a true cloud service, it has to meet the standards set by the definition of cloud; there are five characteristics including Resource pooling, Broad network access, resource elasticity, and measured service.

There are three models of services on the cloud: IaaS (Infrastructure as a Service), PaaS (Platform as a Service), and SaaS (Software as a Service).

There exist four cloud deployment models: Public, Private, Community, and Hybrid.

Chapter 3: System Design and Architecture

1.14 Chapter Overview

This chapter provides a substantive account of the design and architecture of the cloud storage application, showing how all components interact in view of the realization of a secure and scalable storage solution. Further, this section summarizes the system requirements, then gives a high-level overview of the architecture of the system, illustrating the core components: Node.js back-end, MongoDB database, and Sophos XG firewall.

In the following sections, we will explain in detail each of these components: the routing and middleware configuration of the Node.js server, the MongoDB database schema and indexing strategy, and how the Sophos XG firewall plays its part in securing the traffic on the server. Furthermore, we will outline the flow of data across the system to show how the different parts-user request, data management, and security measures-come together in a coherent solution. Finally, we will present the analysis of the system for scalability and expansion to handle growth in the future.

1.15 System Requirements

The development of this application was conducted on a computer running Windows 10 with the following specifications:

- CPU: Intel i12400-5f
- RAM: 16GB DDR4
- GPU: NVIDIA GTX 1660 super
- Storage: 512GB Nvme Gen 4 SSD

The local running of MongoDB requires the CPU to support AVX, Advanced Vector Extensions. In case the CPU in the production (deployment) environment is too old to support AVX, MongoDB will never work locally. In this case, it's highly recommended to host MongoDB at some third-party service, like AWS or Google Cloud, because of consistency and stability.

The following demands on the production environment ensure a reliable and efficient operation of the cloud storage application:

3.2.1 Hardware Requirements

- Server: Server; should be at least 16GB of RAM, at least a CPU with 6 cores, and at least 256GB SSD Storage to provide the high read/write speeds required.
- Network: High speed because of low latency assures smooth flow and less downtime.

3.2.2 Software Requirements

- Operating System: The Linux Operating System (e.g. Ubuntu, Mint) is preferred in order to host the server in a secured/stable environment.
- Back-end Framework: Node.js v20.16.0; The application works on the server side, routing, requesting, and providing middleware.
- Database: MongoDB v7.0.14 will store the user information and metadata of files for efficient retrieval and storing.

Additional Libraries and Tools

- **Express.js** (v4.21.1): Web Application Framework for Node.js that would make building the application quite easier because it puts in place a structured way of developing APIs. It provides, out of the box, support for routing, middleware management, and request handling, thus enabling developers to build APIs in an effective and scalable manner.

- **Mongoose** (v8.7.2): An ODM library for MongoDB provides enhanced functionality in modeling objects by schemas and models with respect to structuring and organizing your data with it. It manages your interactions with MongoDB. Mongoose simplifies the definition of relationships among data, validation of data, and complex queries.
- **jsonwebtoken** (v9.0.2): Library for generating and verifying JSON Web Tokens to authenticate users in web applications. JSON Web Token is a compact, URL-safe means of representing claims to be transferred between two parties.

1.16 High-Level System Architecture

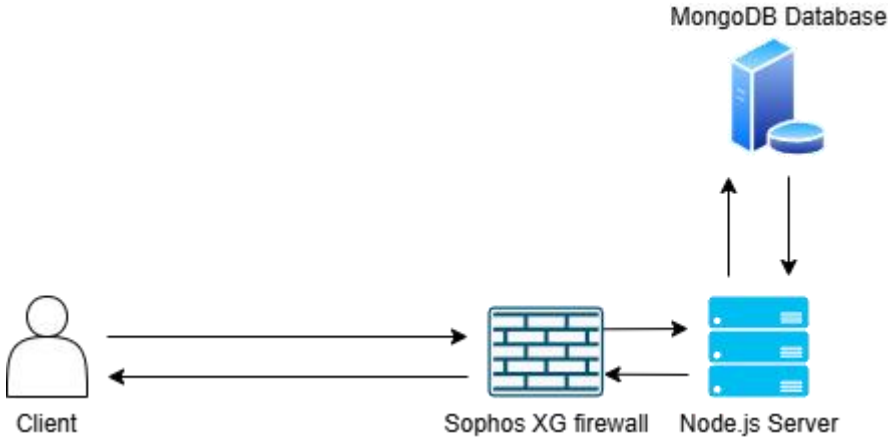

Fig.3.1 System Architecture

Before defining the architecture of the server and client applications, we are going to do some data modeling of the system. There are going to be two main resources in the application: Users and Files; each in a different MongoDB collection.

3.3.1 User Collection

The Users collection handles authentication and authorization; it provides control so that clients' files and data could not be accessed by unauthorized parties. It shall store user credentials and any linked information safely for identity verification.

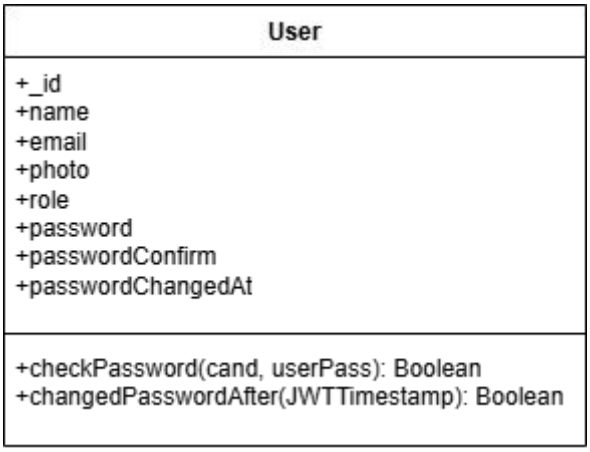

Fig.3.2 User Schema

The structure of the user document consists of several fields, each serving a critical purpose to ensure the proper functioning of our application:

- **_id**: A unique identifier, MongoDB assigns this to every document at the time of creation.
- **name**: The username of the user. This field stores a string representing the user's display name within the application.
- **email**: The user's email address used during sign-up, stored as a unique string to make sure there are no duplicate emails on the system.
- **photo**: A URL or path to the profile photo of the user. It could permit avatar functionality.
- **role**: Permissions inside an application that the user belongs to. 'admin', 'user', etc.-different roles might restrict or allow some functionality to the user.
- **password**: A hashed string serving as the user's login credential. It's kept hashed for security rather than in plain text.
- **passwordConfirm**: During registrations, it confirms the user's intended password. This field is never stored in the database; it's used for validation and then discarded.
- **passwordChangedAt**: Timestamp for when password was last reset. Used to invalidate JWT tokens that were issued prior to this time. User will be asked to log back in with a new password.

- **checkPassword**: This method takes a candidate password and compares it with the user's hashed password. We used bcrypt (hashing library) to validate the password.
- **changedPasswordAfter**: Takes a JWT timestamp and checks whether password was changed after the passed timestamp. If the password was changed, it invalidates the JWT token, which makes the user log in again to have a new token.

3.3.2 Files Collection

The Files collection will store the metadata of each uploaded file, along with the _id of the user who submitted it. This will help manage the files more effectively, and it will implement access control, given that every file is associated with a user.

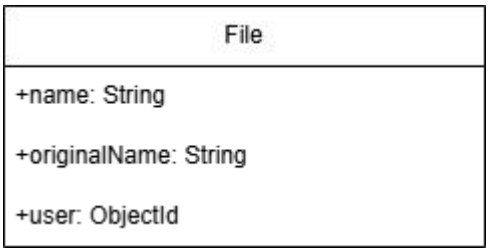

Fig.3.3 File Schema

The structure of the File document includes several fields that either contribute to or are important for the proper functioning of our application:

- **name**: This is the name under which the file will be saved on the server. It's generated based on a certain schema to avoid duplicates; hence, it includes all the key information: user-{User._id}-{current_timestamp}-{file.originalname}. This will let us track it more easily by user ID, time of upload, and original name without conflict.
- **originalName**: The original name of the file as it was sent from the client; this is kept for reference and display, which helps the user identify their files.

- **user**: This is the User who uploaded the file. Normally, it is an ObjectId that references the document of the user; this sets a relation to this exact file by the user who owns it.

The client-server architecture system hosts, delivers, and manages most of the resources and services which may be requested by a client. In this model, all the requests and services are delivered over the network; hence, it is also termed the networking computing model or client server network.

Client-server architecture, or often just client-server, describes a network application structured according to a division of tasks and workloads between clients and servers located on the same system or communicating over a computer network.

The general structure for a client-server architecture involves workstations, PCs, or anything that belongs to multiple users connected to a central server using an Internet connection or some network. The client requests access to certain data, and the server receives the request and responds to it by sending packets of data to the user requiring access.

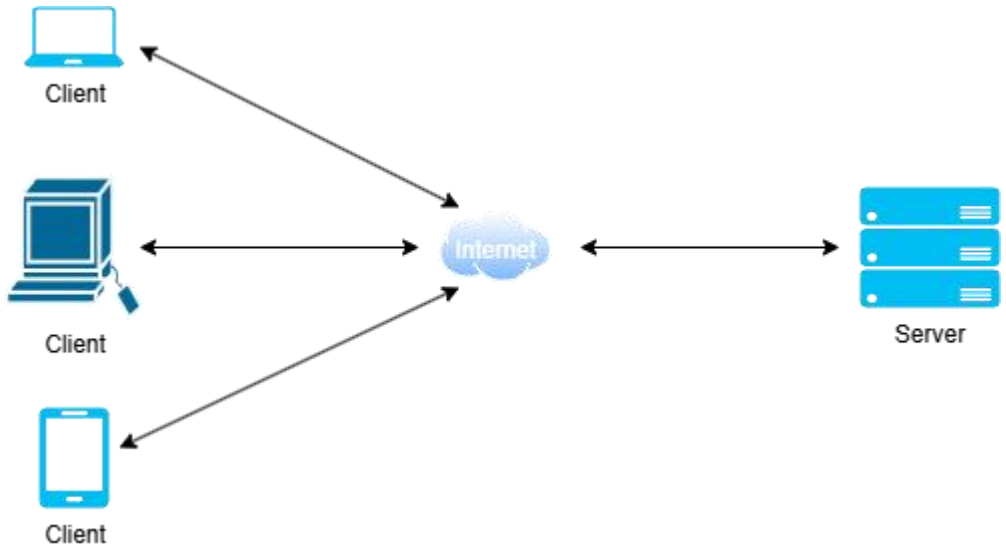

Fig.3.4 Client/Server architecture

Request-Response Cycle consists of three stages:

- The client sends a request to the server.
- The server accept and process the request.
- The server sends a response back to the client containing the requested data or information.

Since we used Express.js as the back-end framework for Node.js to implement the API, it is important to first explain how Express.js handles requests.

Express.js handles requests in a structured approach based on routes and middleware.

An overview of how Express.js works:

- **Incoming requests**: When a client sends a request to the server for something, say GET, POST, PATCH, DELETE, etc., Express.js catches it through predefined routes.
- **Routing**: Express.js defines which paths and methods—GET, POST, PATCH, DELETE, etc.—the server listens to, using middleware functions to process the requests and send a response. The route app.get('/api/users', ...route_handlers) will handle the GET request to the '/api/users' endpoint.
- **Middleware execution**: When a request is routed to a certain endpoint in the server, it goes through a middleware stack. A middleware stack is the ordered set of functions that may be applied to deal with and pre-process requests within the API. For example, we have seen a middleware function called protect, which checked whether a token existed inside an Authorization header or inside cookies. In such a case the middleware validated the token and fetched the corresponding user from the database to attach the user object to the request object. Following middleware functions in that stack could then use the user object to do other things.

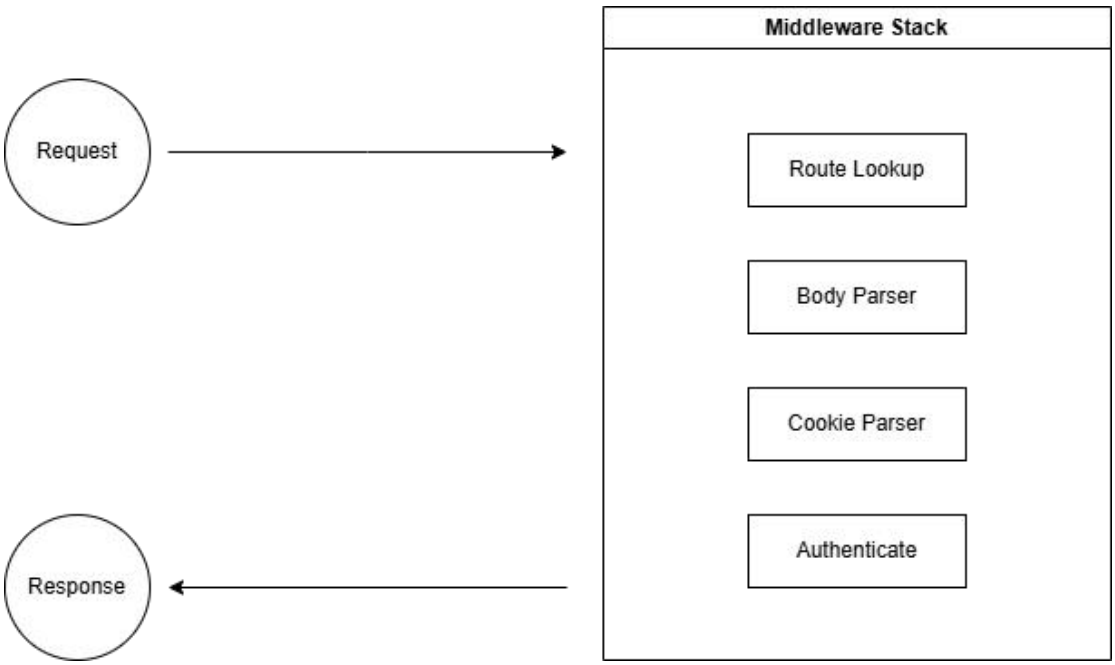

Fig.3.5 Middleware Stack

- **Route handler**: Once the chain of middleware processing is done, the request lands to the route handler. It's actually a function which fulfills the primary job of the request against your application, like querying database, returning responses, etc.
- **Response** :The server responds to the client, the response may be in the form of JSON data or HTML page.
- **Error handling**: We configured the system so it catches known error and sends back an appropriate response instead of crashing the system.

3.4.1 Back-end Server (Node.js)

The back-end of this application is implemented using Node.js, providing a scalable and efficient environment for handling HTTP requests and managing the main functionalities of the cloud storage system. The back-end server acts as an interface between the client-side interface and the database, ensuring secure and reliable communication.

- Key components:

1. **Routing and API endpoints**:

The back-end uses Express.js to define structured API routes. These endpoints handle user authentication, file uploads, and downloads ensuring seamless interaction between the users and the server.

2. **Middleware Integration**:

Middleware functions are utilized to handle application operations such as request body parsing, authentication, logging, file uploading. For instance, the ‘protect’ middleware we mentioned before secures endpoints by verifying tokens and authenticating users.

3. **Database Integration**:

The back-end uses MongoDB through the use of an ODM library called Mongoose. Mongoose allows a much more ordered structure of data, defining the schema and thereby carrying out database operations in a very effective manner to store user information and metadata about the files.

4. **Authentication and Security**:

The back-end is implemented using jsonwebtoken authentication. Tokens are thus issued and checked upon every subsequent request for every process of signing in and signing up to protect the users' data from unauthorized access.

5. **File Handling**:

he back-end supports file upload and download functionality by providing respective mechanisms for their storage and retrieval. Events metadata such as file name, original file name, and owner are kept in the database, whereas the file content is stored securely on the server side.

6. **Scalability and Performance**:

The back-end is built to scale, since Node.js asynchronous event-driven architecture leverages how the system deals with multiple requests all at once; it therefore stands ready for the days to come when the demand of users will increase.

3.4.2 User Interface (Front-End)

When a user accesses the application and navigates to the root page (/), the server first checks the cookies in the client's request. If a valid cookie is found, the server queries the user's file metadata and renders the overview page for the user, as showed in the following picture:

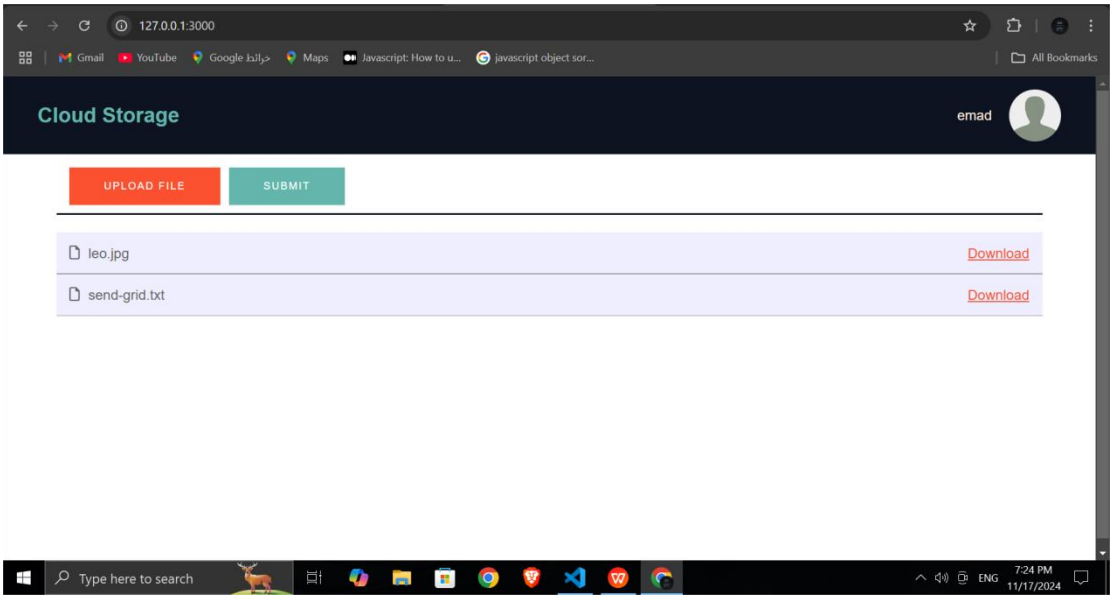

Fig.3.6 User Interface

The client will see a list of previously submitted files displayed in an unordered HTML list. Each list item includes the original file name and a download link. Above the file list, there is a form with two buttons:

- **Upload File**: Allows the client to select a file from their system and add it to the form.

- **Submit**: When pressed, this button sends a multipart/form-data request containing the selected file to the server.

If the server does not find a valid cookie, it redirects the user to the login page (/login). The login page appears as shown below:

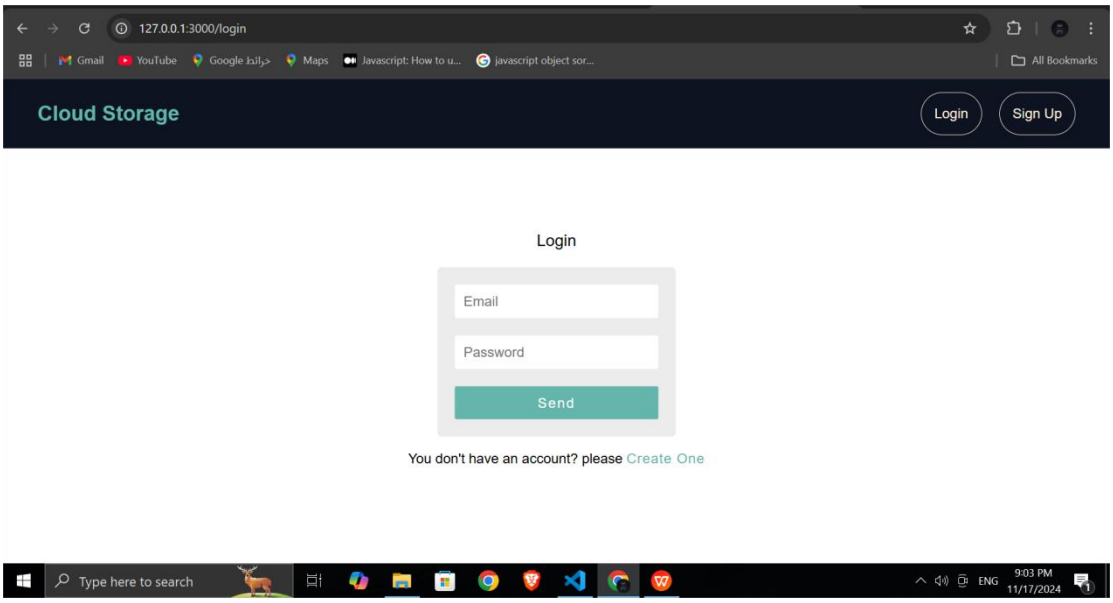

Fig.3.7 Login page Interface

The /login page provides a form with two input fields: Email and Password. Upon submission, the client sends a request containing the email and password to the /api/users/login API endpoint using Axios as a POST request.

- If the provided credentials are valid, the server responds with a JWT token, and a green alert displaying the message “Logged in successfully” is shown to the client.
- If the credentials are invalid, the server returns an error response with a 401 Unauthorized status code. A red alert with the message “Incorrect email or password” is displayed to the client.

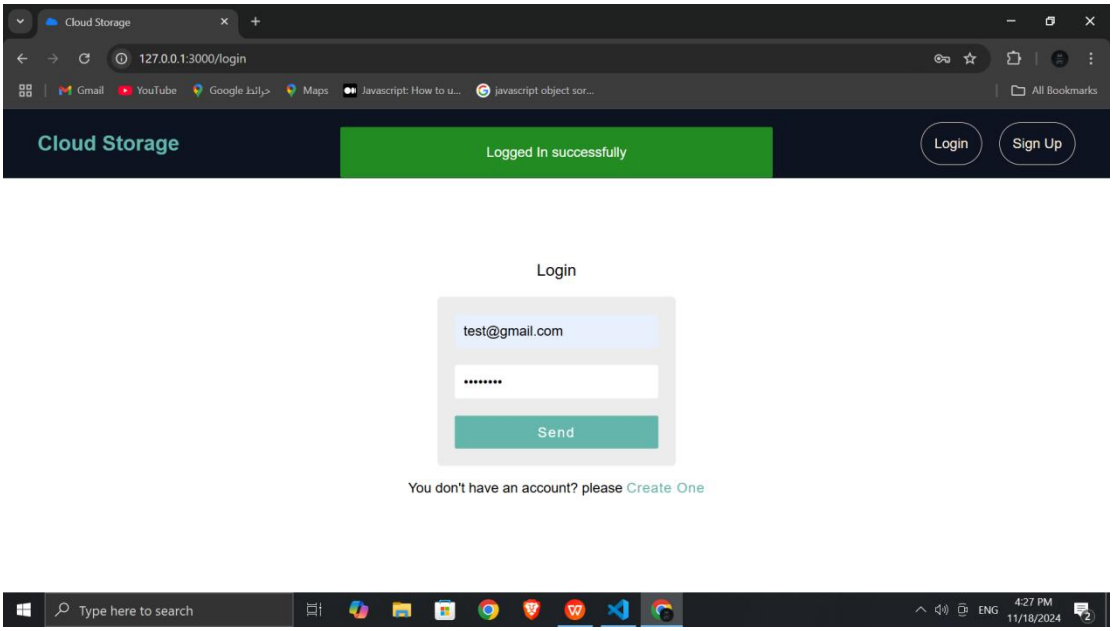

Fig.3.8 Successful Login Attempt Alert

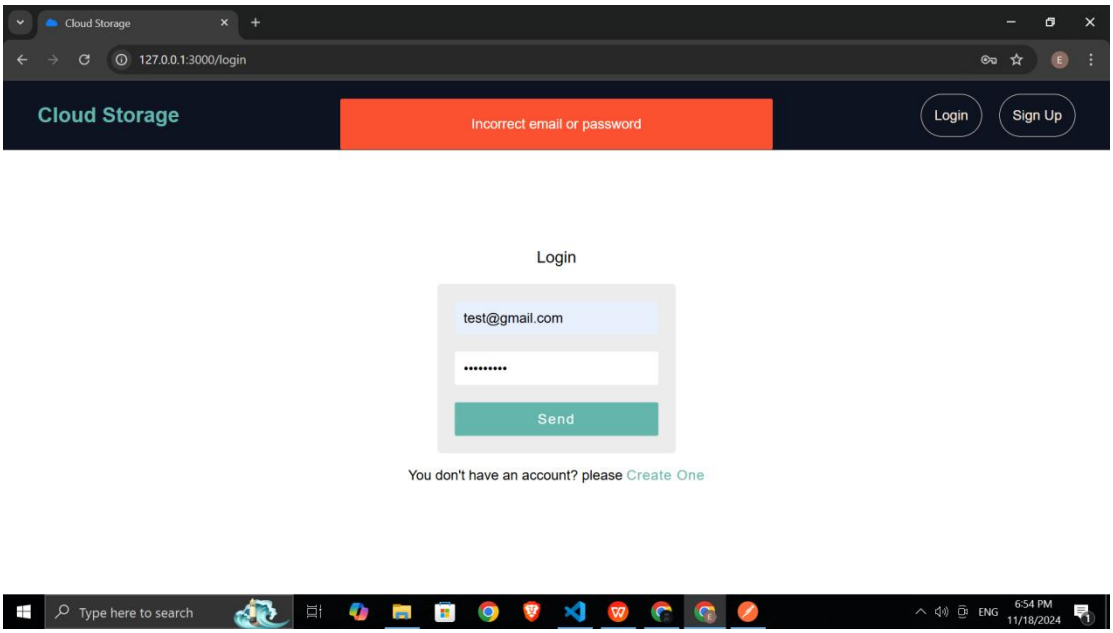

Fig.3.9 Failed Login Attempt Alert

Another route accessible to the client without verification is the sign-up page. This page features a form with four input fields: Name, Email, Password, and Password Confirm.

Upon submission, the client sends the data to the /api/users/sign-up API endpoint using Axios as a POST request.

- On successful sign-up, the server responds with a JWT token, and a green alert displaying the message “Signed up successfully” appears to the client.
- Upon a failed sign-up attempt, the server responds with an error message detailing the issue, such as missing information or incorrect data format, if the error is operational. For non-operational errors, the server returns a generic message: 'Something went wrong.' A red alert is then displayed to the client, showing the server's message.

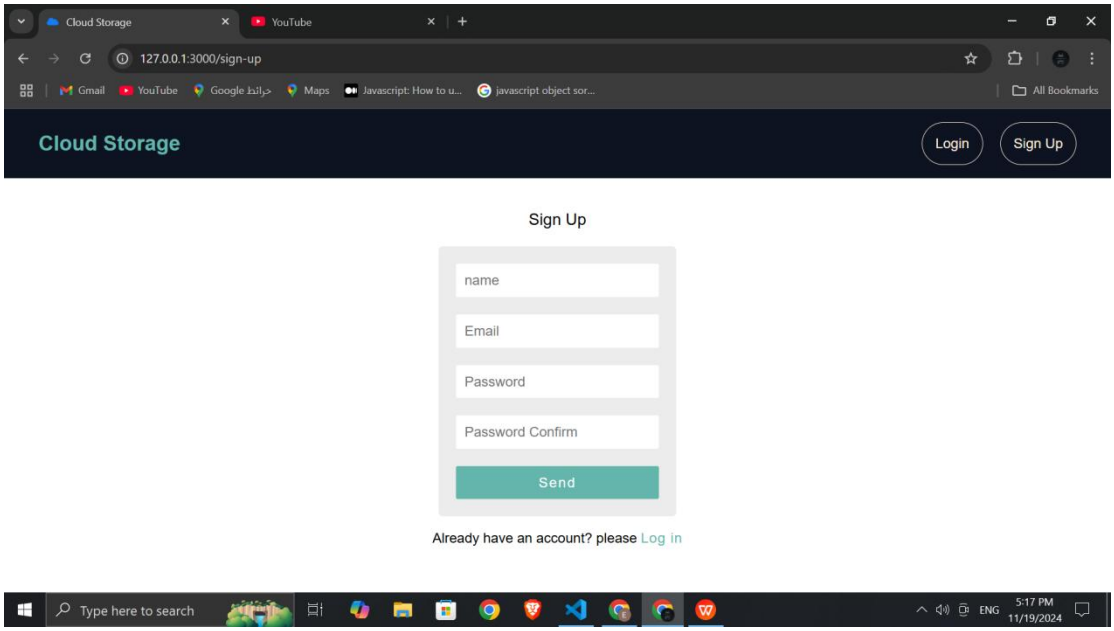

Fig.3.10 Sign-up Page Interface

NOTE: The alert messages styling, position, and color are the same for the sign-up, login, and file uploads and downloads.

1.17 Detailed Component Design

In this section, we provide a detailed breakdown of the core components compromising back-end of the cloud storage application. Each component is designed to execute specific tasks, such as user authentication, file management, and database interactions, working together to ensure a secure and seamless user experience.

The components that we will focus on in this section are:

- Authentication Module
- User Management
- File Management
- Security Features

3.5.1 Authentication Module

This application uses JWT tokens to verify user identity. Each time a user sends a request to the server, the token is included either in cookies or the authorization request header.

Each token have the Id of the user in it's payload, so when a request hits a protected API endpoint the server verifies the request and find the user with the Id is decoded from the token and perform the requested process if verification was successful.

NOTE: We set the expiration date of each JWT token to 90 days from the issue date. If a token is older than 90 days, the verification process will fail.

JWT is constructed from three main parts: header/payload/signature. And the process of validating JWT consists of four statements or operations:

- Check if the token is in right format: header.payload.signature
- Check if the signature is valid, valid signature means that both header and payload are valid too because signature is constructed of both header and payload and a secret key.
- Check if validation time is after nbf (Not Before).
- Check if validation time is before exp (expiration date).

The next diagram shows the operations to validate a JWT token.

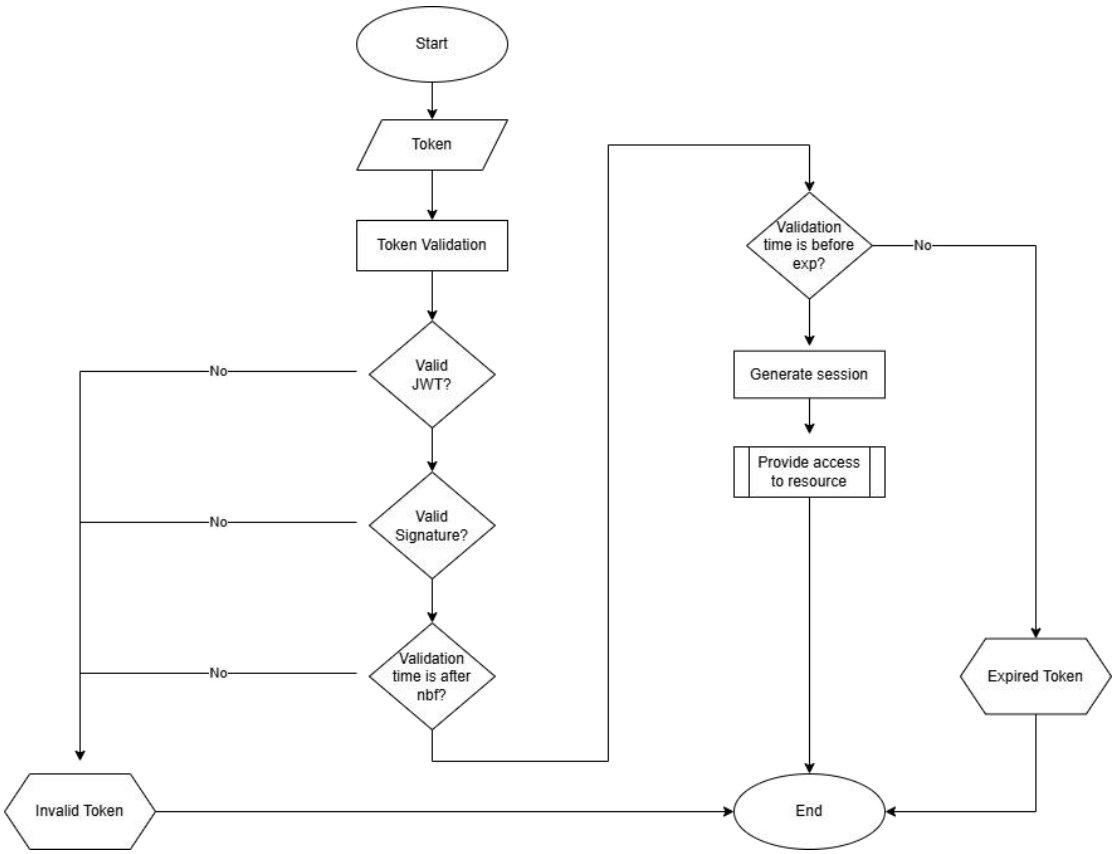

Fig.3.11 JWT Validation Diagram

JWT token consists of three parts: Header, Payload, Signature. And payload contains claims or statements about the identity of the user (In our application we only add the id of the user) and additional claims such as:

- **exp**: expiration date which is a claim that identifies the expiration time on or after which the JWT must not be accepted for processing.
- **nbf**: Not Before, which identifies the time before which JWT must not be accepted for processing.

If the JWT validation failed the server will respond with a 401 “unauthorized” status code and an error message indicating that the client doesn't have a valid token to verify his identity.

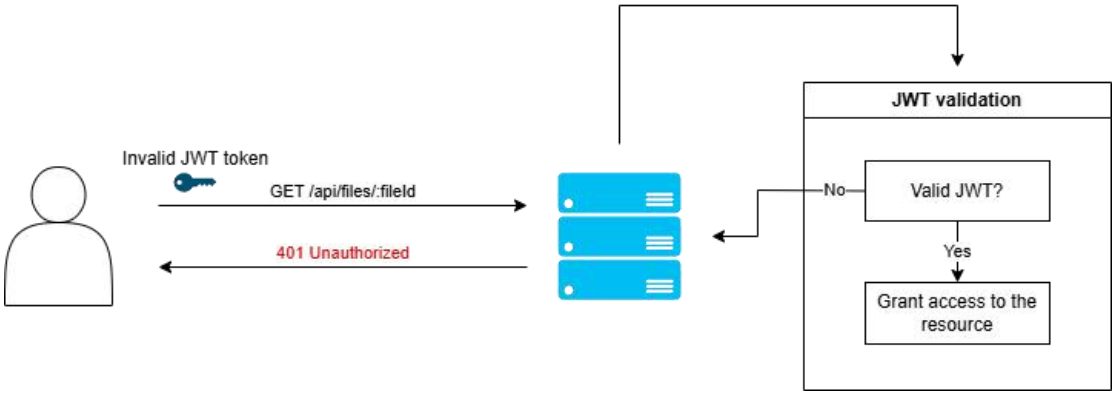

Fig.3.12 Invalid JWT Response

And If the JWT validation succeeded the client will be granted access to the resource and the server will respond with the requested resource.

NOTE: Even if the token is valid, the server will only provide access to the resource if the user has the necessary permissions. A valid token does not grant a user access to another user's data.

Authentication is handled through a single middleware to maintain a modular design for the application. This middleware is reused across multiple endpoints that require a valid JWT token to access resources.

The actual implementation of the middleware will be presented in a later chapter. In this chapter, we focus primarily on the theoretical aspects of the application.

3.5.2 User Management

The user management component is responsible for handling user-related operations, including registration, authentication, and authorization. It ensures only legitimate users can access the system, safeguarding user data and files.

This component ensures the integrity and confidentiality of user data by utilizing strong authentication mechanisms. It also provides the base for user-specific operations, such as accessing stored data.

As we illustrated the user schema in prior section in this chapter now we can dive deeper in the details of the user management component.

There is two main endpoints for the user component:

- **Sign-up**: This endpoint is used to create a user document in the MongoDB Users collection. Once registered, the user can access the application to upload and download their data.

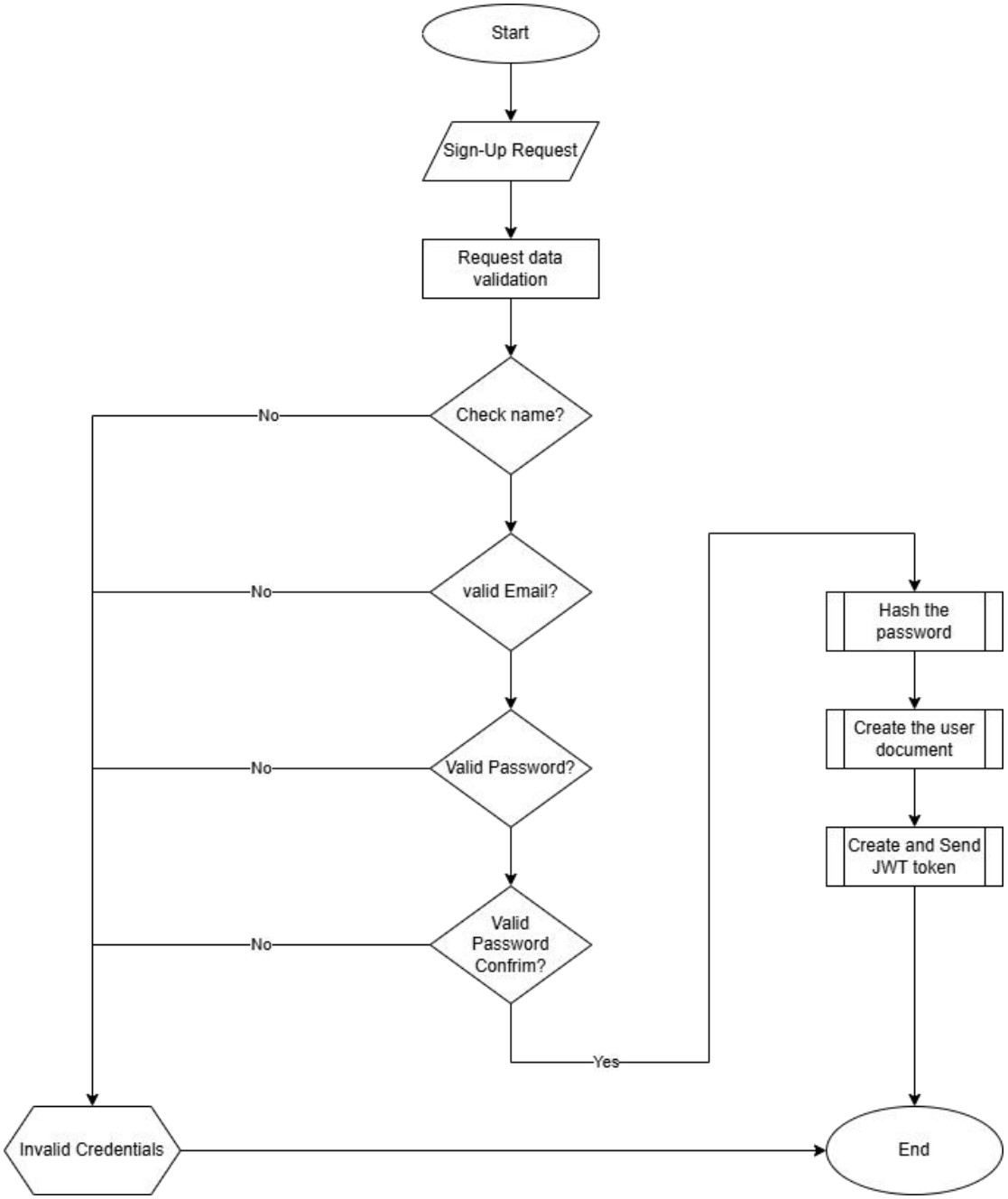

Fig.3.13 Sign-up Operation Diagram

- **Login**: used to issue JWT token for the client to be able to authenticate his identity and perform operations provided by our application.

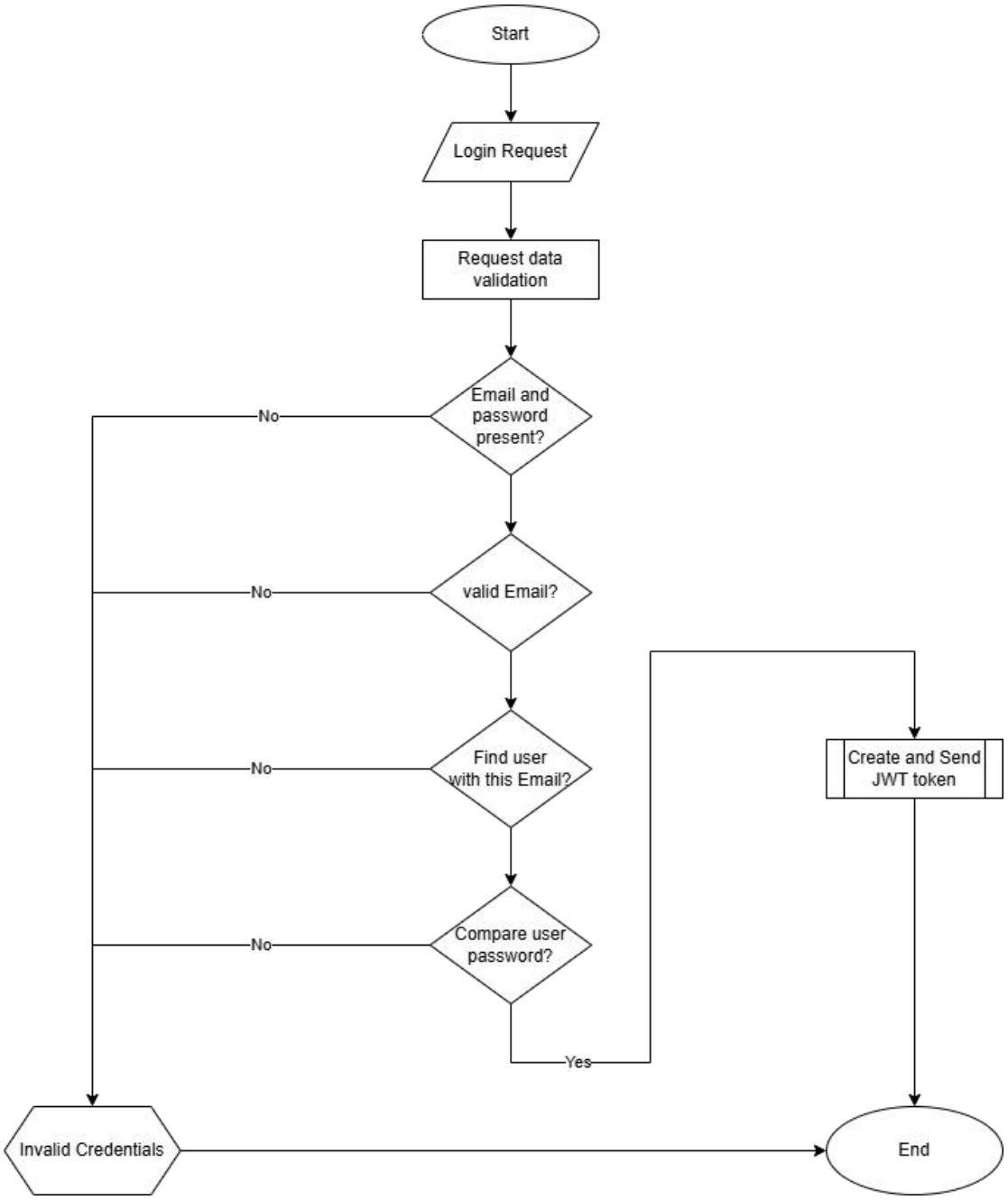

Fig.3.14 Login Operation Diagram

When a user document is created, the password is not stored in plain text. Instead, it is hashed using bcrypt. During login, the plain text password sent by the client is compared with the hashed password stored in the user's document.

If the verification is successful, the server generates and sends a JWT token to the client, granting access to the user's data.

3.5.3 File Management

For the file resource our application provides two endpoints for two separate operations:

- **/api/files/upload** : This endpoint allows the upload of a single file per request using multipart/form-data. It is a protected endpoint, meaning a valid JWT token is required to perform the file upload.
- **/api/files/:fileId** : This endpoint enables the retrieval or download of a single file per request. Like the upload endpoint, it is protected and requires a valid JWT token to access the file.

File Upload:

In this application, files are uploaded from the client to the server using the multipart/form-data format. After uploading these files are stored in a directory named storage, located in root directory of the project folder. Upon reaching the server, the files are processed by Multer, a middleware designed to handle file uploads, and are stored in the designated storage directory.

Once the upload is complete, the server proceeds with the next middleware in the stack. At this point, the file's metadata becomes accessible within the request object. Using this metadata, the server creates a file document in the files collection of the database.

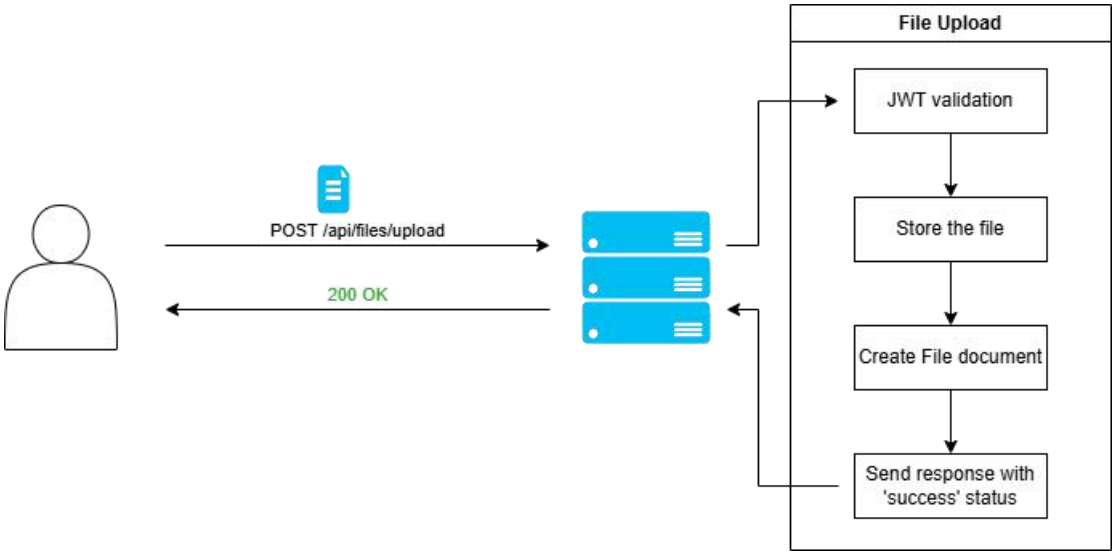

Fig.3.15 File Upload Operation Diagram

NOTE: If the client sends a request to the upload endpoint without including a file in the multipart/form-data payload, or omits a file entirely, the server will respond with a 400 “Bad Request” status code and an error message. Conversely, if the file document is successfully created, the server will respond with a 200 “OK” status code and a success message.

File Download:

In this application, files are retrieved by sending a GET request to the API endpoint /api/files/:fileId, where fileId corresponds to the _id field of the file stored in the MongoDB database.

Upon receiving the request, the server first validates the JWT token. If the token is valid, it searches the files collection in the database for a document with an _id matching the provided fileId and a userId matching the id in the JWT token payload. If a matching file document is found, the server responds by sending the corresponding file from the storage directory.

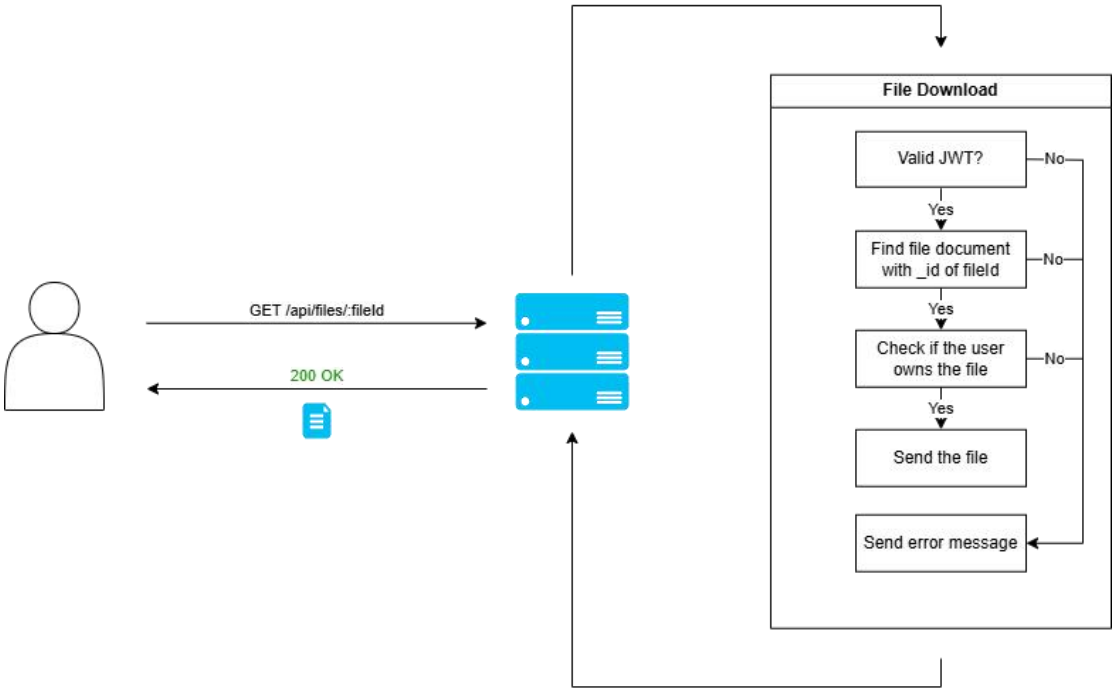

Fig.3.16 File Download Operatio Diagram

3.5.4 Security Features

Security is the topmost enabler of this cloud storage application, ensuring the data of users, files, and server resources are safe from any unauthorized access and possible attacks. Further, the section depicts various security measures within the system that are implied for strong protection.

Authentication and Authorization

We also used two separate middleware functions in handling user authentication and authorization for the implementation of good and secure cloud storage.

The first middleware is called 'protect' and serves to check the JWT token. This middleware, in function of the result of the verification, will continue the request to the following middleware in stack or send back an error response to the client saying they don't have permission to access that resource.

But here, the second middleware is restrictTo, which is actually a wrapper function taking only one argument, an array of roles. This middleware returns another middleware that checks if the user object inside the request has a role

present in the array of roles passed. If the user's role fits one of the allowed ones, he's allowed to pass this middleware to the next one in line. If his role is not part of the allowed roles of the called endpoint, he gets a response of an error in the form of a denied action.

NOTE: This is a very common restriction in most web applications, whereby normal users are not allowed to delete, for example, a document from the database, or fetch all the user records from an application.

The following figure presents the flow of the restrictTo middleware: This middleware here protects an endpoint that should be accessible only to administrators. Here, the regular user has a role of "user" and sends a GET request to the /api/users endpoint. That endpoint is supposed to fetch all user documents within the users' collection in our MongoDB database.

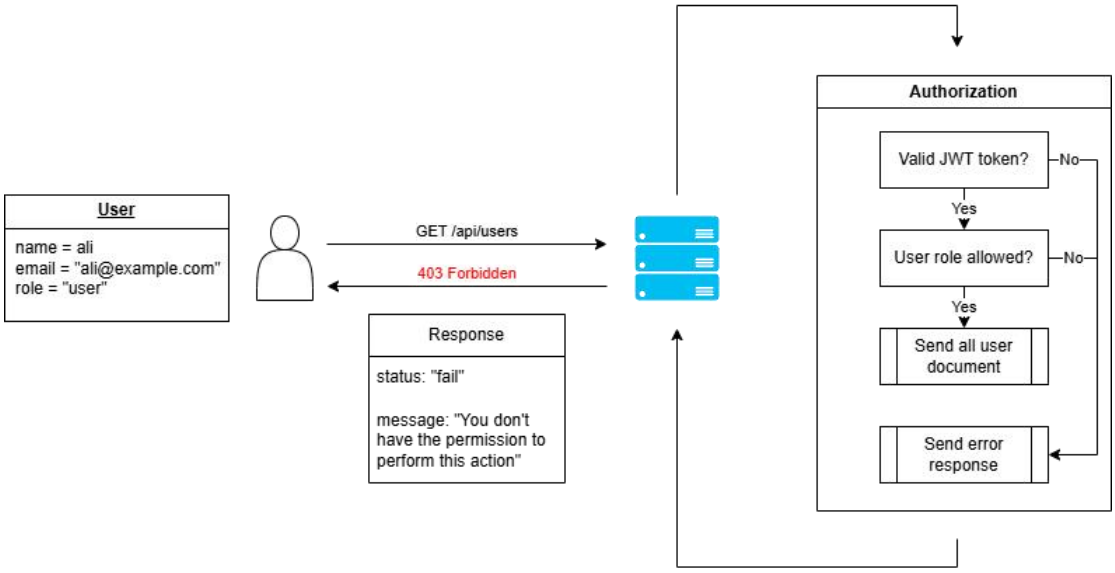

Fig.3.17 restrictTo Operation Workflow Diagram

NoSQL Injection

NoSQL injection is a security vulnerability that occurs when an attacker manipulates the queries an application sends to a NoSQL database. This type of attack can allow the attacker to:

- Bypass authentication or security mechanisms.
- Extract, modify, or delete data.
- Cause denial of service.
- Execute unauthorized code on the service.

For example, an attacker could manipulate the database query by injecting an operator to retrieve sensitive information. Consider the /api/users/login endpoint, where a legitimate user would typically send the request body in the following format:

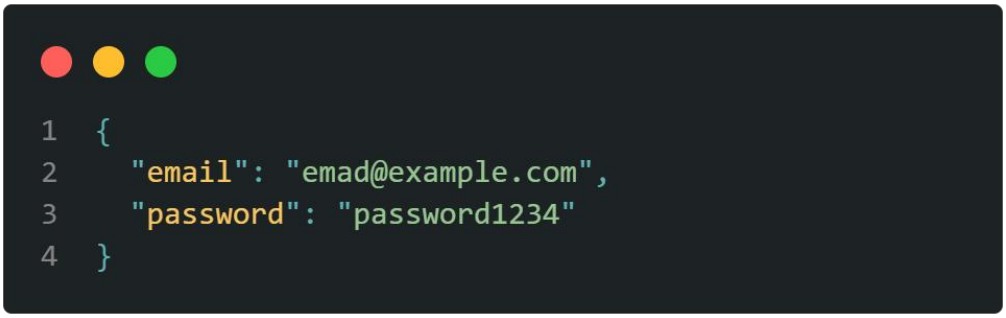

Fig.3.18 Normal Login input

But, as noted above, it's possible for an attacker to manipulate the query object that's sent to the database. Rather than sending through a valid email and password combo for the server to locate a matching combo, it could send a request body formatted like so:

Fig.3.19 Malicious Login input

In this scenario, the attacker exploits the server by crafting a query to search for an email and a password. However, instead of providing a legitimate password, the attacker injects the $ne operator (which means "not equal") along with an arbitrary incorrect password.

Here's what happens behind the scenes:

The server processes the query, looking for a document with the specified email and a password that is not equal to the injected string. For example, if the actual password is password1234 and the attacker injects 'wrong-password', MongoDB interprets the query as finding a user whose password is not 'wrong-password'. This matches the intended user document, allowing the attacker unauthorized access to the account.

To safeguard against this type of attack, we implemented the express-mongo-sanitize middleware. This function sanitizes user input before it is utilized in a database query. Specifically, it removes any keys starting with '$' sign from the input, effectively neutralizing potential NoSQL injection attempts.

Cross Site Scripting (XSS)

Cross-Site Scripting (XSS) attacks are a type of injection attack where malicious scripts are embedded into otherwise trusted and harmless websites. These attacks occur when an attacker exploits a web application to deliver harmful code—typically browser-side scripts—to an unsuspecting user.

For an example an attacker could inject JavaScript code to a request body property that is then used by the server to create a user document or file document. Consider the /api/users/sign-up which accepts four properties: name, email, password, passwordConfirm. A normal user will send the request body in this format:

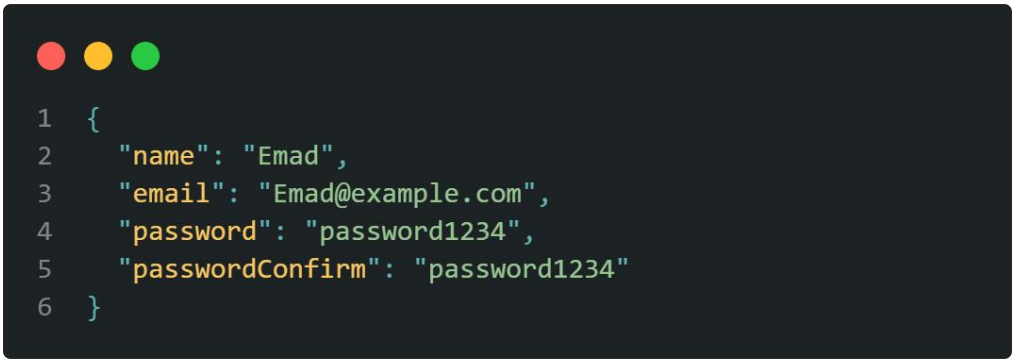

Fig.3.20 Normal Sign-Up input

So, an attacker injects a tag into the property of any request body and embeds malicious code that will run on access to this compromised resource by any other user.

The attacker's request body might look something like:

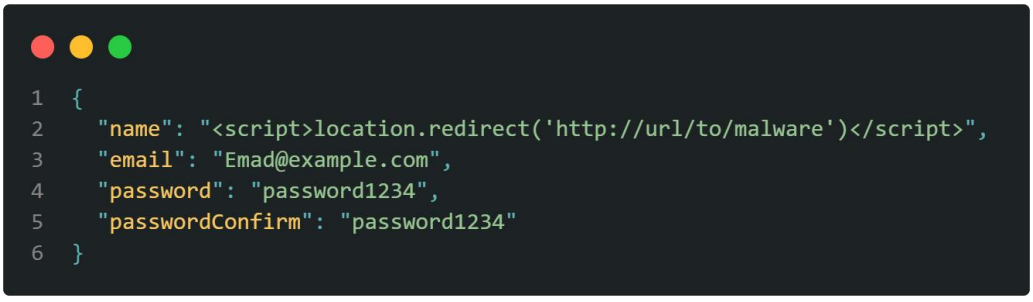

Fig.3.21 Malicious Sign-Up input

Here, an attacker has exploited the server in that a user document with a malicious value in the name field was created containing a <script> tag with embedded JavaScript code designed to perform a page redirect to a download link for a malicious file named “malware.exe”.

For that, once another user accesses the profile page of this attacker-created account, the content in the name field won't be rendered as plain text. So, in loading this page, it will execute the malicious <script> performing the unwanted download of the malicious file.

To protect from this type of attack, we have enabled the xss-clean middleware, which cleans user input to prevent the execution of malicious code. Actually, it removes the first < character in any script or HTML tag that is being sent to the server, making it harmless since it treats it like a normal string and does not execute it as code.

1.18 Summary

Two major resources form the basis for our cloud storage system: users and files; each of them has their own collection in our MongoDB database.

Such a user's collection manages user authentication and authorization, defining permissions to access certain resources or take specific actions. This resource supports two important operations: one is sign-up, for adding a new user, and the other one, called login, to log in a user, who will be able to access his information by means of a JWT token.

The files collection will be used to store metadata of files, like the name of the file, original name of the file, and who uploaded the file. Only two operations are allowed on the file resource: upload enables a user to upload a file to which only the user has access, while download enables users to download their own files.

Chapter 4: Deployment Environment and Security Measures

1.19 Introduction

The software, platforms, and infrastructures collectively known as the 'Cloud' have existed in some form since the late 1960s. However, cloud data storage raises significant privacy and security concerns, impacting the data itself, metadata about users and their data, and the transmission of data between devices and cloud storage hardware. In professional or research settings, you may store data belonging to others in the cloud. Even personal data can contain sensitive information about third parties.

Legal challenges to data privacy and security include cloud provider terms of use, legal warrants, and varying laws across jurisdictions. Additionally, cyber-criminals may target your data for purposes beyond identity theft. Sharing data in the cloud introduces further risks, such as ensuring trust among collaborators and effectively managing access control.

In this chapter, we will create the deployment environment for our application. This involves configuring a Linux virtual machine (VM) to run the application along with Suricata for intrusion detection and prevention. Additionally, we will set up another VM to run Sophos XG, providing enhanced security and network traffic monitoring for the application.

1.20 Deployment Environment

The deployment environment of our application relies on multiple tools to make it possible to monitor and secure the network traffic for the application, these tools are:

- **VMware workstation 17 pro**: virtualization software that enables users to run multiple machines in a single physical hardware.
- **Linux Mint**: It is a user-friendly and community-driven Linux distribution based on Ubuntu.
- **Sophos XG Firewall**: is a comprehensive network security solution designed to protect businesses of all sizes from wide range of cyber threats.

- **Suricata**: is a powerful open-source network security tool that functions as both an Intrusion Detection System (IDS) and Intrusion Prevention System (IPS).

The deployment environment is configured using Linux Mint as the operating system to host the application and Suricata as an Intrusion Detection System (IDS). Additionally, a separate virtual machine is set up with Sophos XG Firewall, serving as a gateway for Linux Mint to enhance network security and monitor traffic for the application.

First we will start with VMware workstation pro.

4.2.1 VMware workstation pro

We utilized VMware Workstation 17 Pro (version 17.5.2) as the virtualization software. At the time of writing this paper, submitted as partial fulfillment for a Bachelor's degree, VMware Workstation 17 Pro is available as free software for personal use and can be downloaded from the Broadcom website.

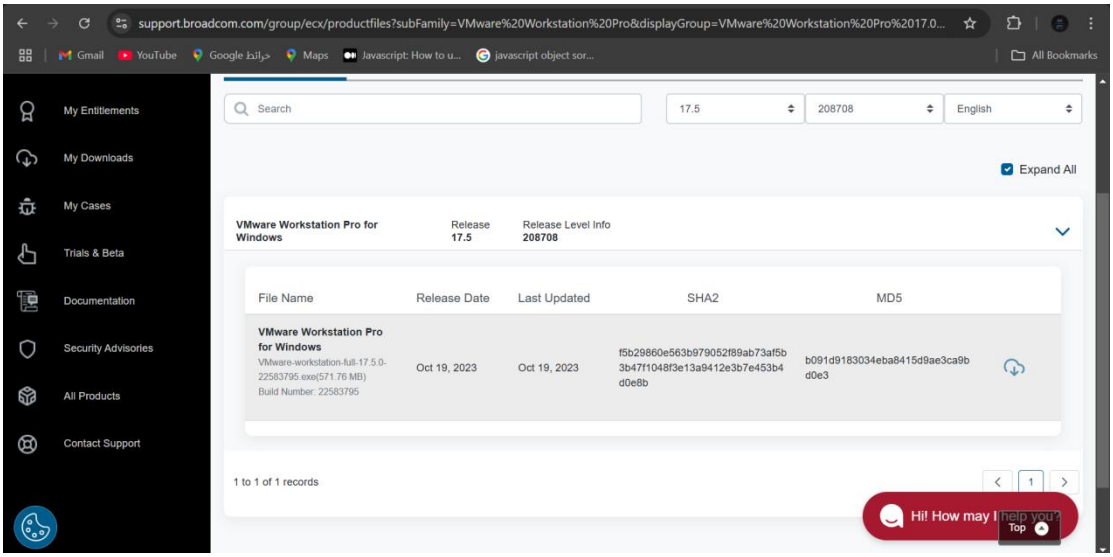

Fig.4.1 VMware Workstation 17 Pro Download Page

After downloading, a file named *VMware-workstation-full-17.5.0-22583795.exe* will appear in the Downloads folder. To set up VMware Workstation Pro on a Windows machine, execute this file to complete the installation process. Once installed, VMware can be used on the local machine. Before setting up Sophos XG and Linux Mint on VMware, we need to create a virtual network to route all Linux Mint traffic through the Sophos XG firewall. The virtual network is configured with these properties:

- **Name**: VMnet1
- **Type**: Host-only
- **Subnet IP**: 10.1.1.0
- **Subnet Mask**: 255.255.255.0

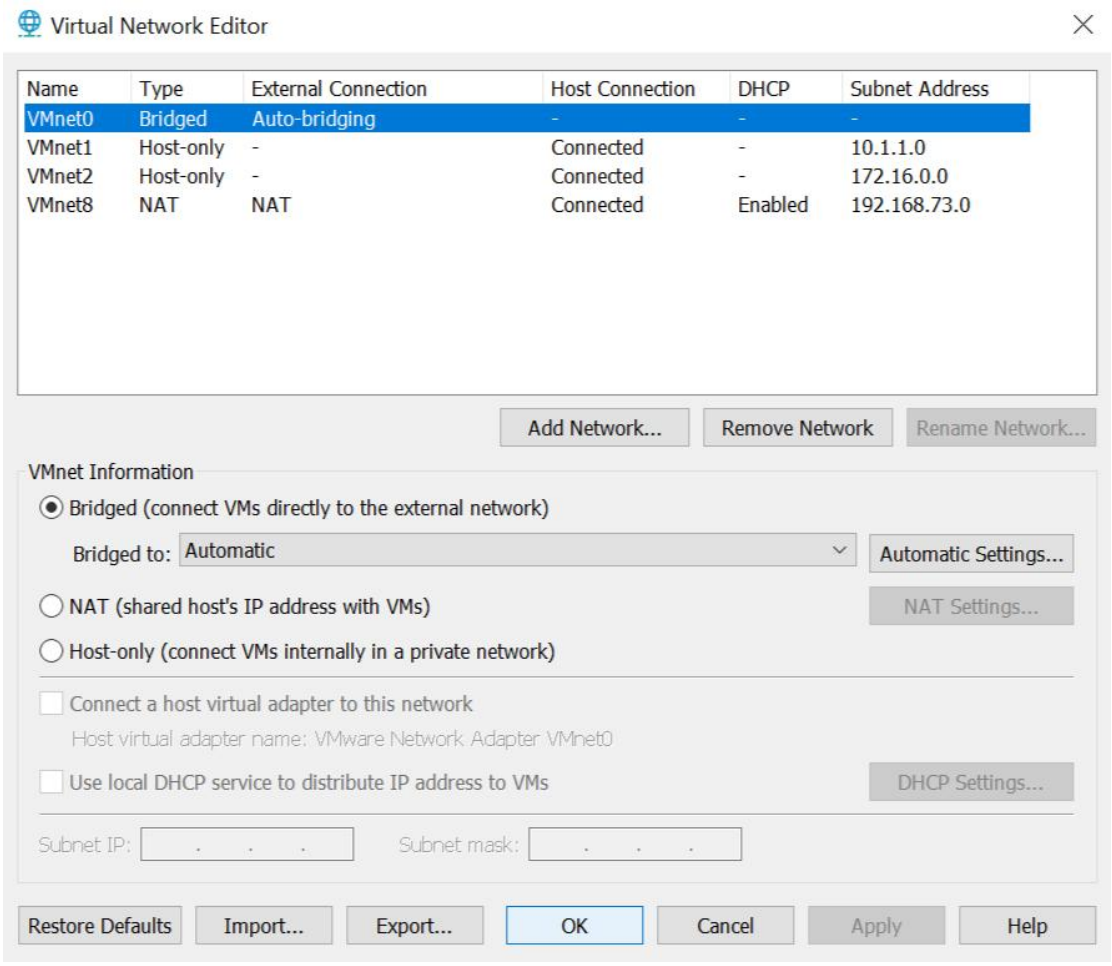

Fig.4.2 Virtual Network Configuration

4.2.2 Sophos XG Firewall

We deployed the virtual Sophos XG firewall in a virtual machine and used it as a gateway for the Linux system that is running the Cloud Storage application to enhance the security of the application traffic.

The setup process for Sophos XG starts with downloading the installation files from the official Sophos website. After downloading, the compressed folder must be extracted to access the necessary files for configuring the Sophos XG firewall virtual machine.

To set up the virtual machine, run VMware Workstation, and select the 'Open a Virtual Machine' option from the main screen.

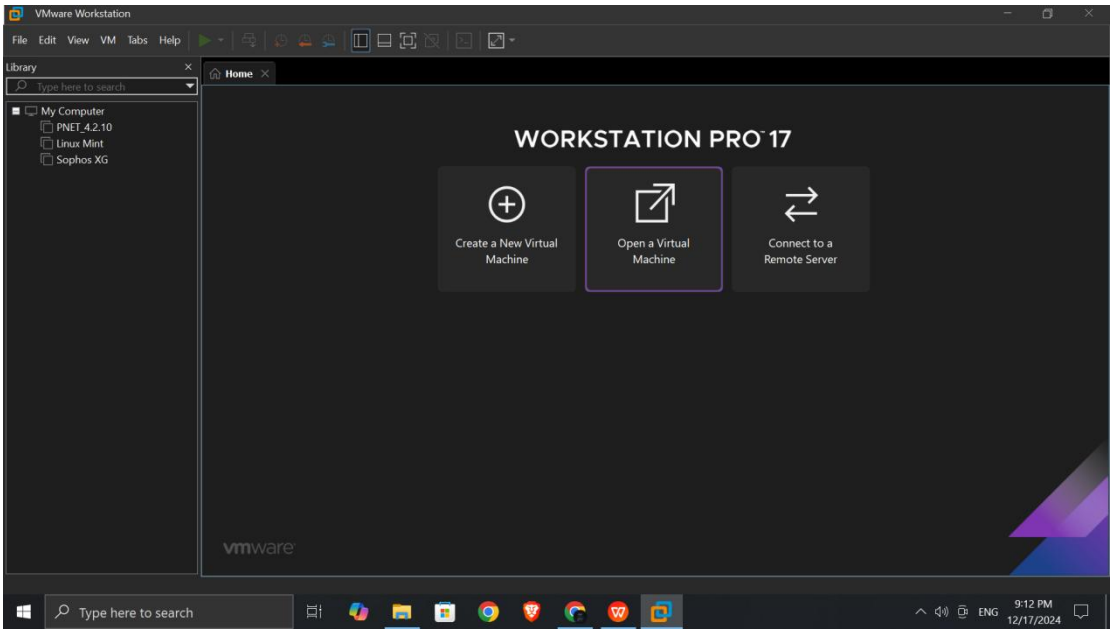

Fig.4.3 VMware Workstation Pro Interface

Next, select the *sf_virtual_vm8.ovf* file from the previously extracted folder.

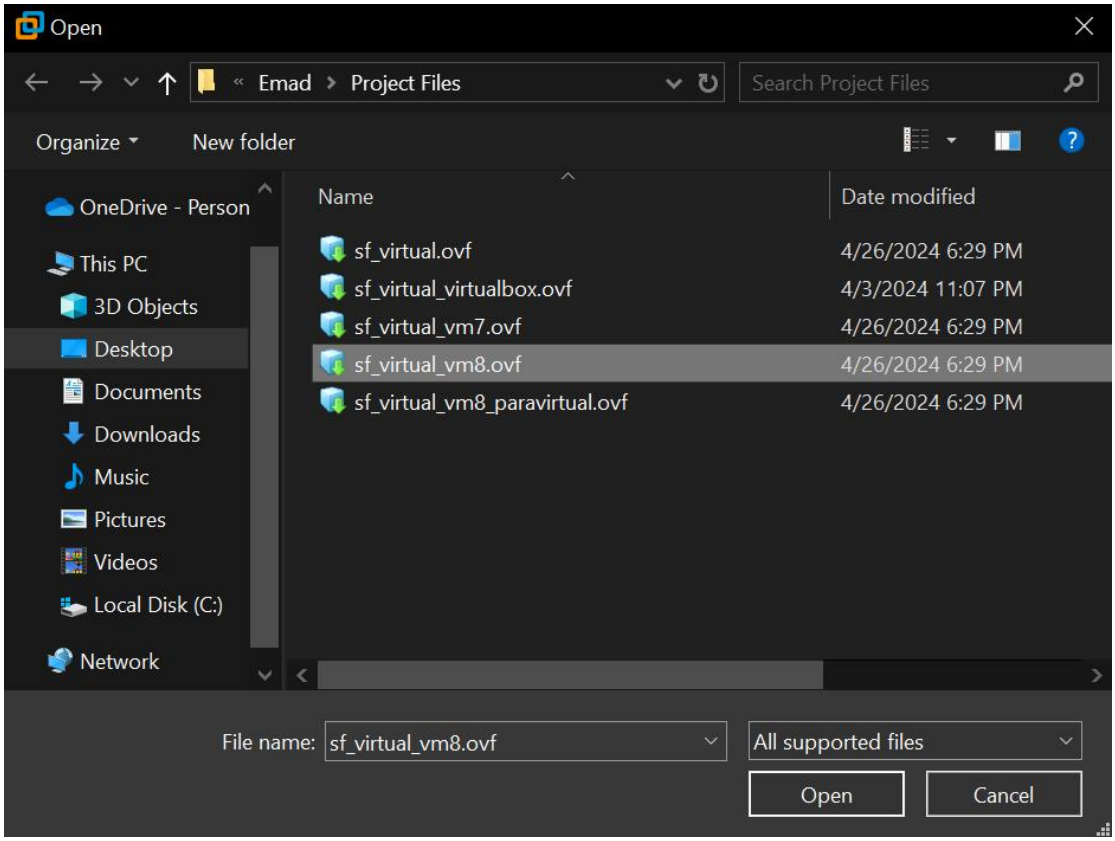

Fig.4.4 Sophos XG Virtual Machine Setup File

After the successful creation of the Sophos virtual machine, this needs to be configured to be talking with the Linux Mint virtual machine. This will require that both virtual machines are under the same virtual network. The network adapter settings for the Sophos XG virtual machine should be set to 'VMnet1' as earlier setup in the VMware virtual network settings.

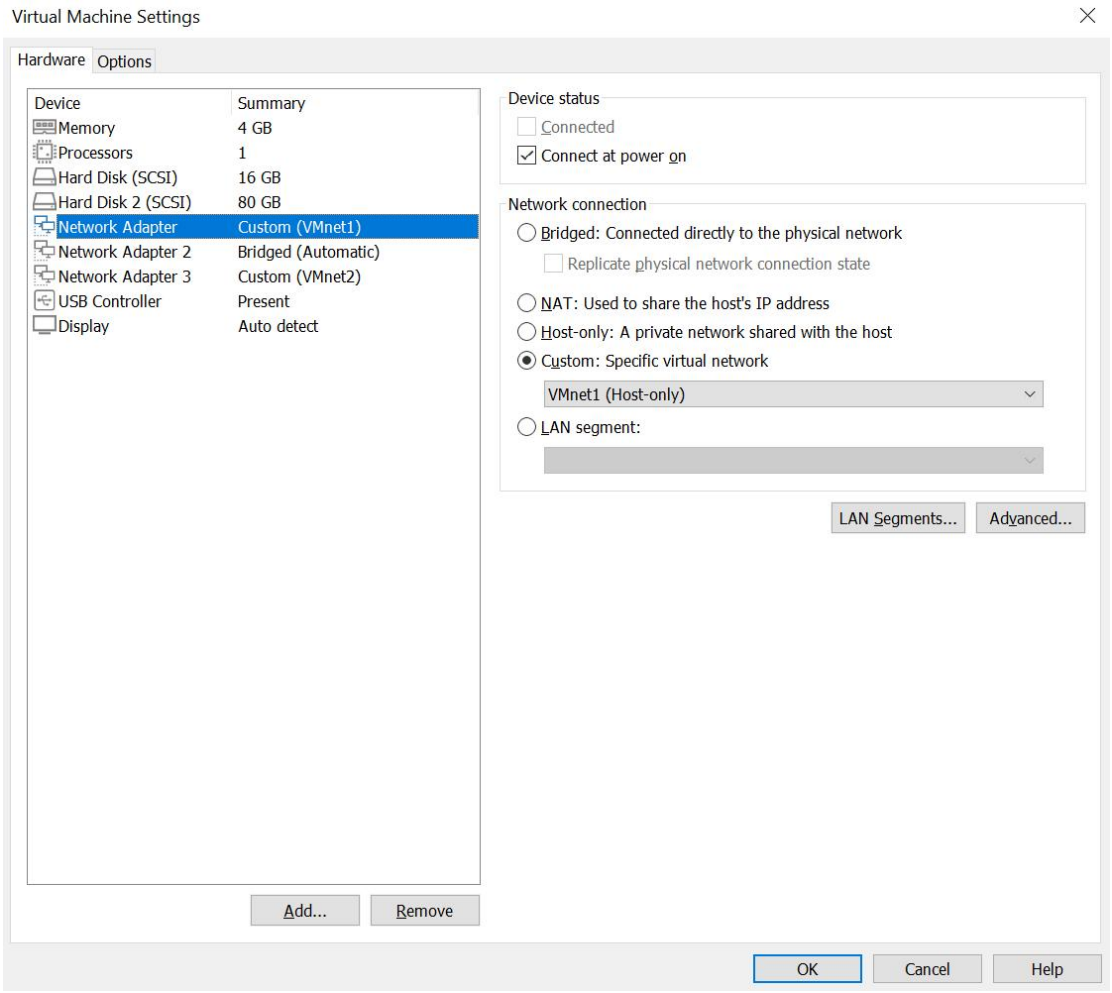

Fig.4.5 Sophos network adapter settings

Once the setup is complete, the firewall configuration can begin. After starting the Sophos XG virtual machine, the main menu will appear, allowing navigation through the firewall's settings and configuration options.

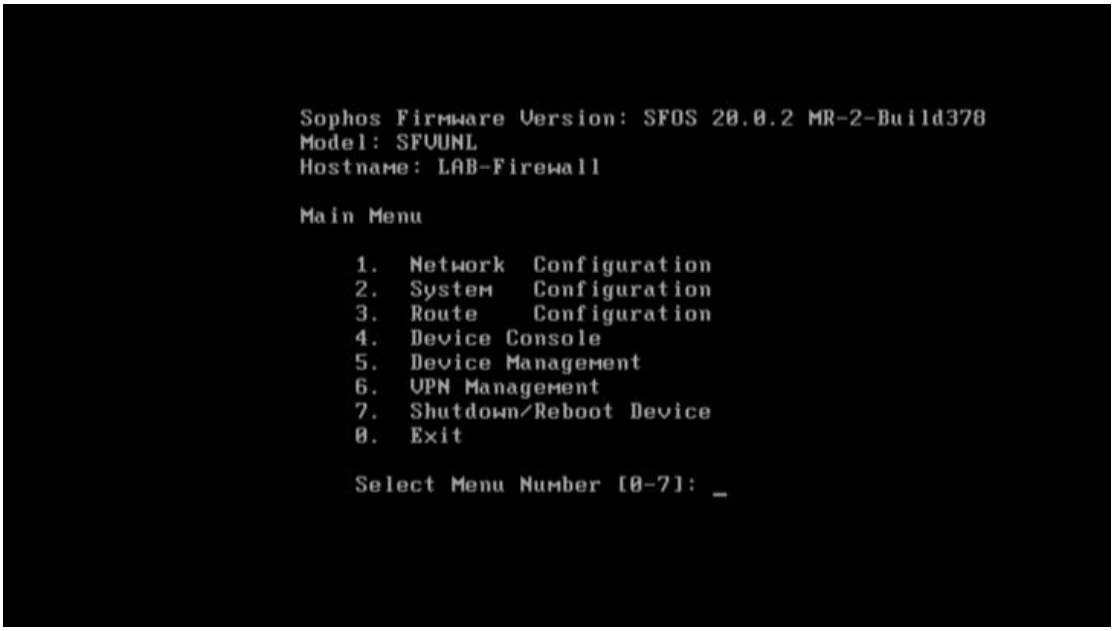

Fig.4.6 Sophos XG main menu

To configure the network address for the firewall's first port (PortA), navigate to **Network Configuration** -> **Interface Configuration**. Set the IPv4 address with the following properties:

- New IP Address: 10.1.1.200
- New Netmask: 255.255.255.0

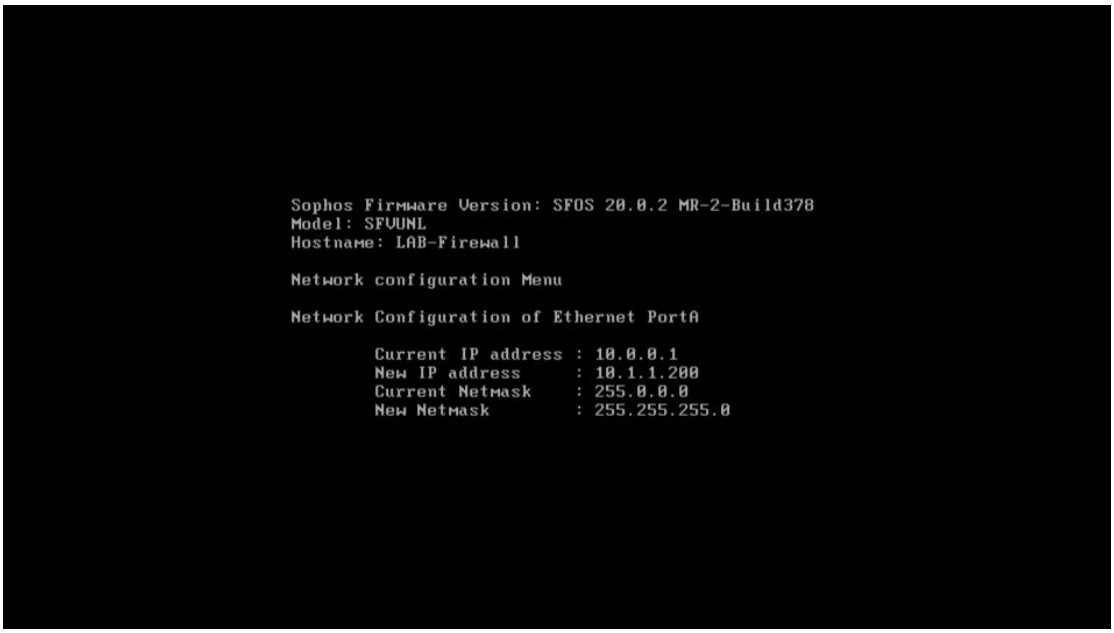

Fig.4.7 Sophos XG PortA Configuration

After configuring the new IP address for Port A of the firewall, the admin panel can be accessed via a web browser by navigating to 10.1.1.200:4444. Upon the first login, Sophos requires the creation of an account on the Sophos Central platform. This account grants access to the Sophos XG Firewall's free 30-day trial.

In the admin panel, firewall rules can be configured; however, the free tier of Sophos XG does not include Intrusion Detection System (IDS) and Intrusion Prevention System (IPS) features. To address this, we used Suricata, an alternative tool for Intrusion Detection and Prevention.

4.2.3 **Linux Mint**

For the ease of use and reliability, this system will be set up to host our Cloud Storage application and Suricata; thus, Linux Mint has been chosen. Setting this up would include downloading the ISO image off the official Linux Mint site, creating a new virtual machine on VMware, and placing the ISO image as the installer file.

To allow the Linux Mint Virtual Machine to communicate with the Sophos XG Virtual Machine, we configure the Network Adapter of the Linux Virtual Machine to use the same Virtual Network VMnet1 where the virtual machine Sophos XG has been joined. Following are the recommended settings for the Linux Mint Virtual Machine:

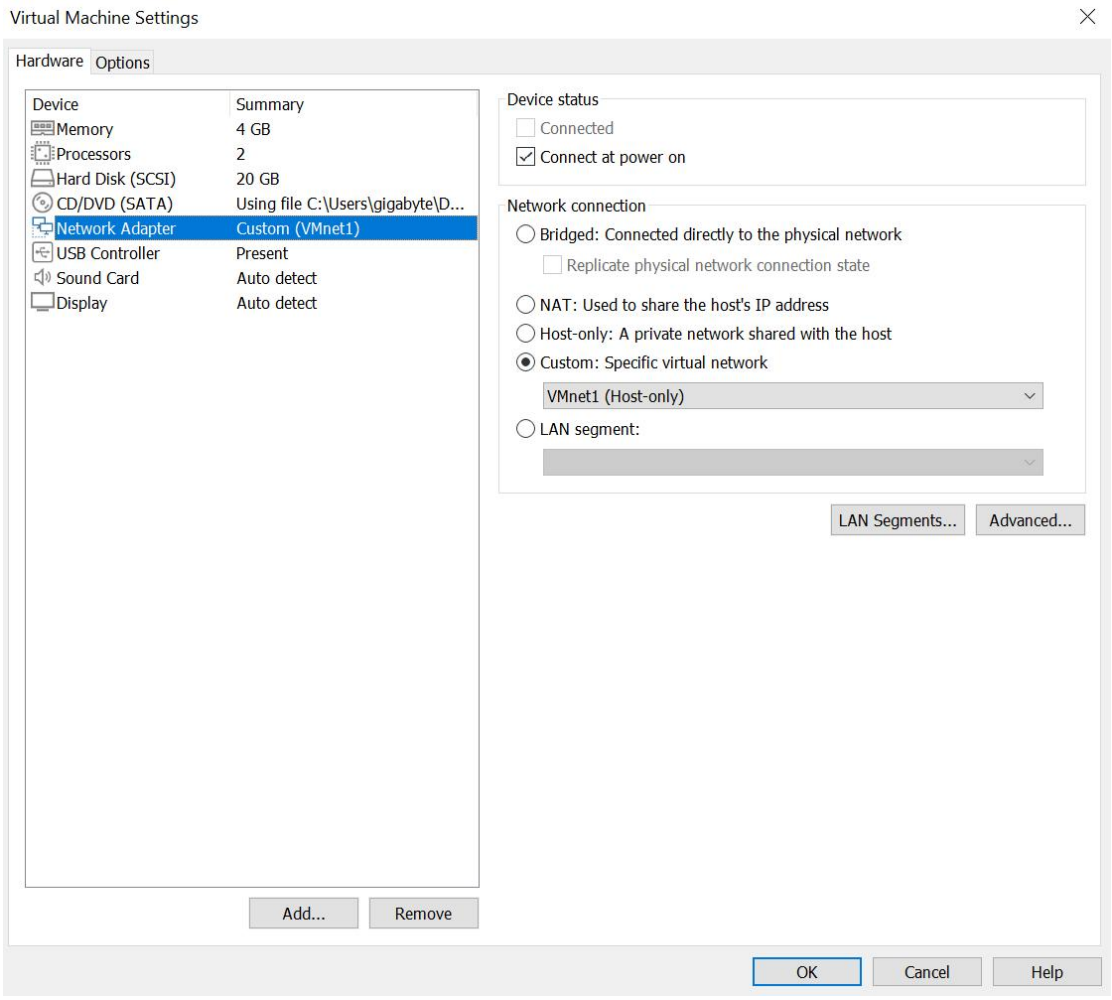

Fig.4.8 Linux Virtual Machine Settings

Once creation of the Linux Mint Virtual Machine is done, the next thing to be done was to set up the network and manually set the default gateway to the Sophos XG. Below is how the network settings of the Linux Mint virtual machine would look:

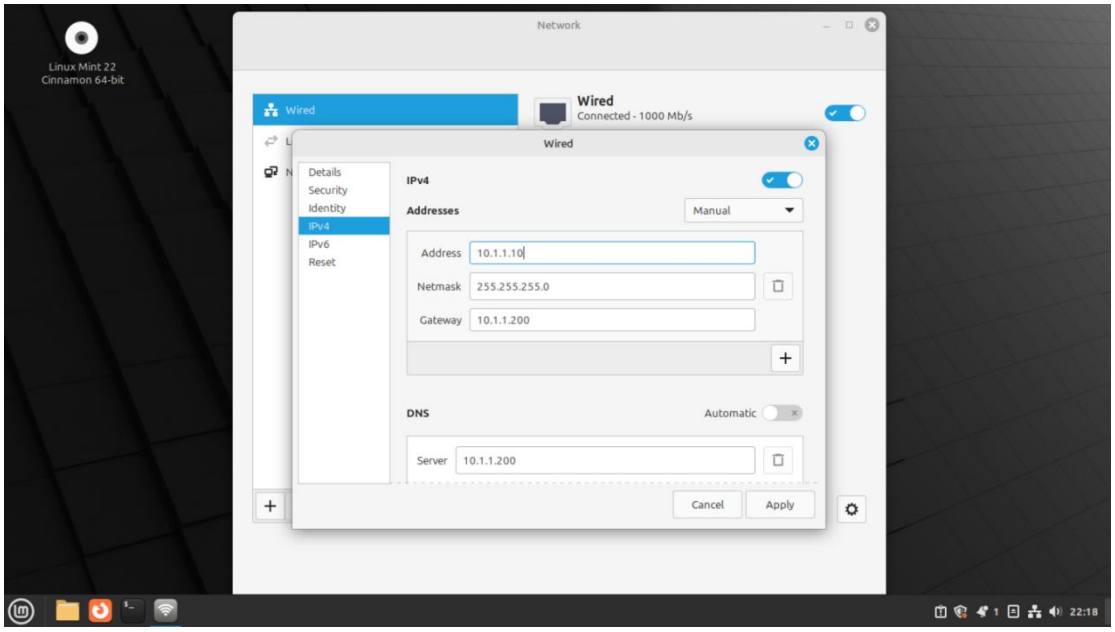

Fig.4.9 Linux virtual machine network settings

We will run the command **ip route** in the terminal to view the network routes to verify that the Linux VM uses the Sophos XG as its gateway.

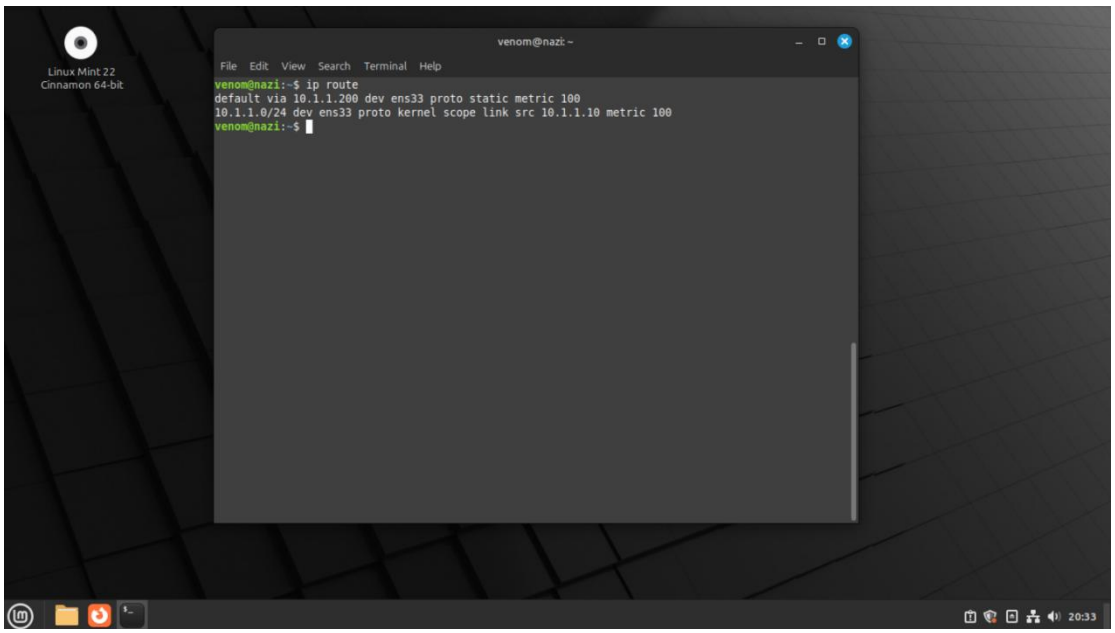

Fig.4.10 Ip route command

4.2.4 Suricata

We then deployed Suricata, an IDPS, to monitor our virtual network, VMnet1, for anomalous traffic and generally enhance the security of our Cloud Storage application. Installation of Suricata is pretty simple and can be installed using the following command: sudo apt install suricata.

As part of the Suricata setup, we edited the configuration file at /etc/suricata/suricata.yaml to change the variable **HOME_NET** to the address of the VMnet1 network, which is 10.1.1.0/24. This way, Suricata will observe the network's traffic.

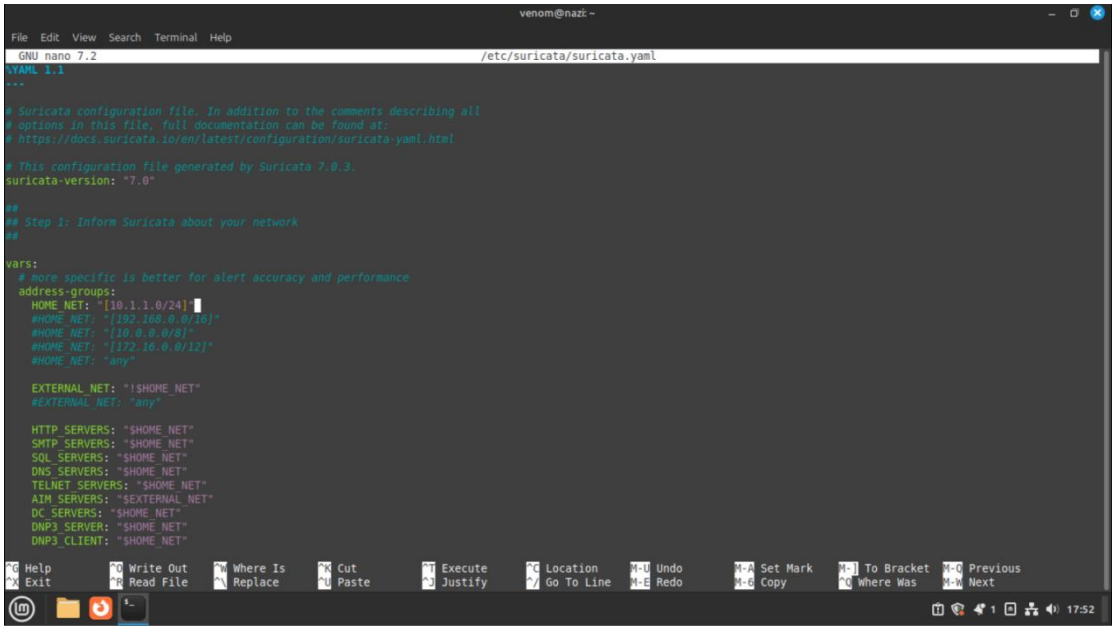

Fig.4.11 Suricata Network Range Scan

After defining the network scan range, the network interface (**ens33**) is configured for both af-packet and pcap modes:

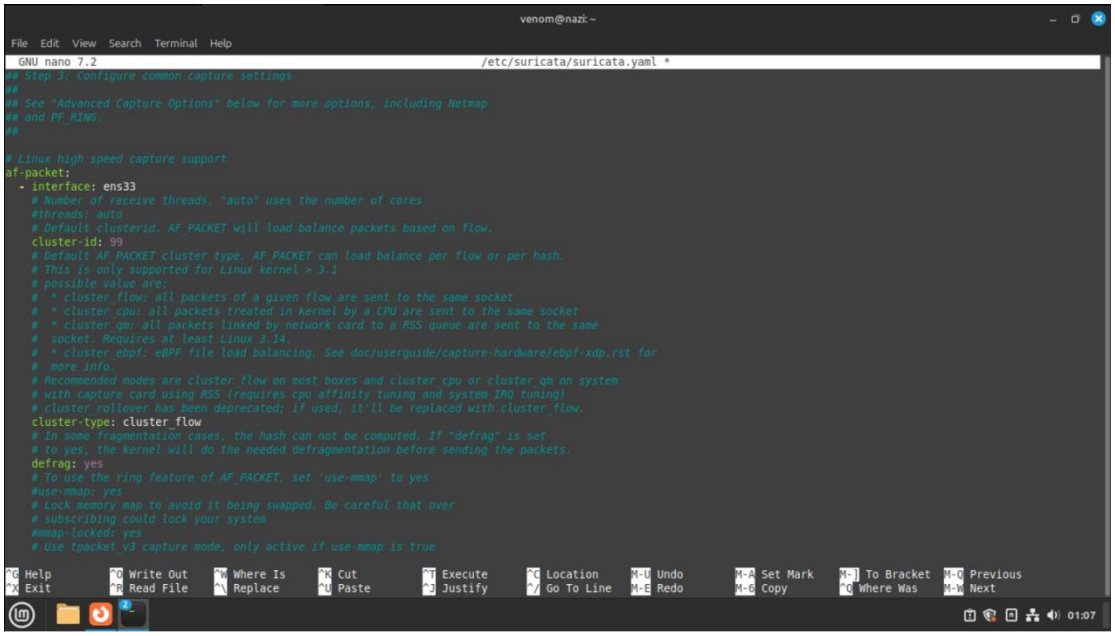

Fig.4.12 Suricata Network Interface (af-packet)

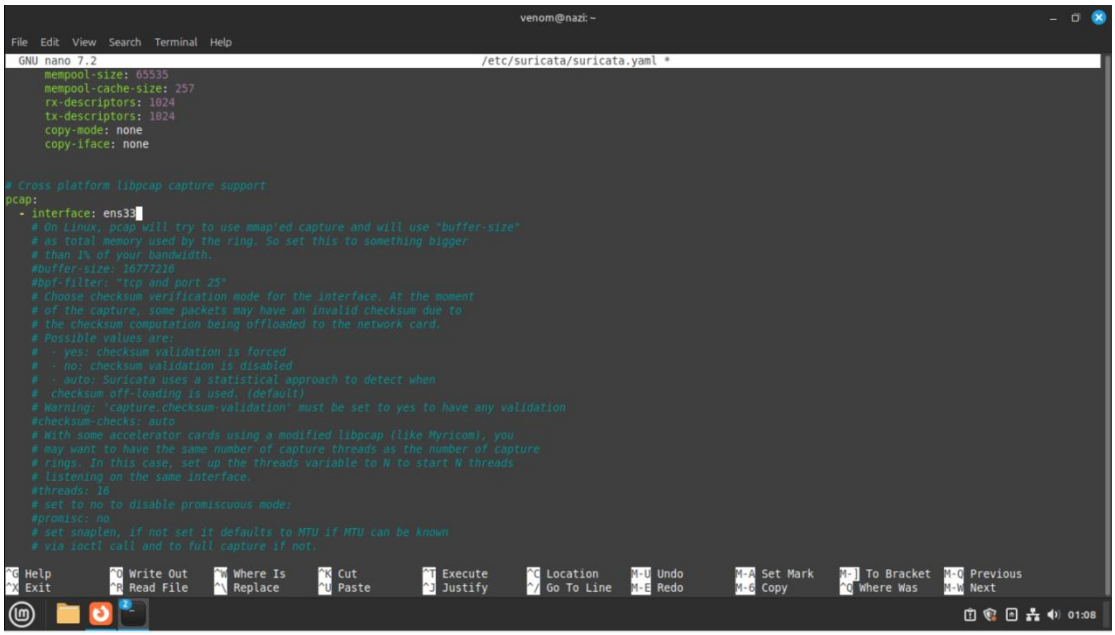

Fig.4.13 Suricata network interface (pcap)

After completing the configuration, we validate the Suricata setup using the command **suricata -T -c /etc/suricata/suricata.yaml**. Once validated, Suricata is started as a system service with the command **sudo systemctl start suricata.service**.

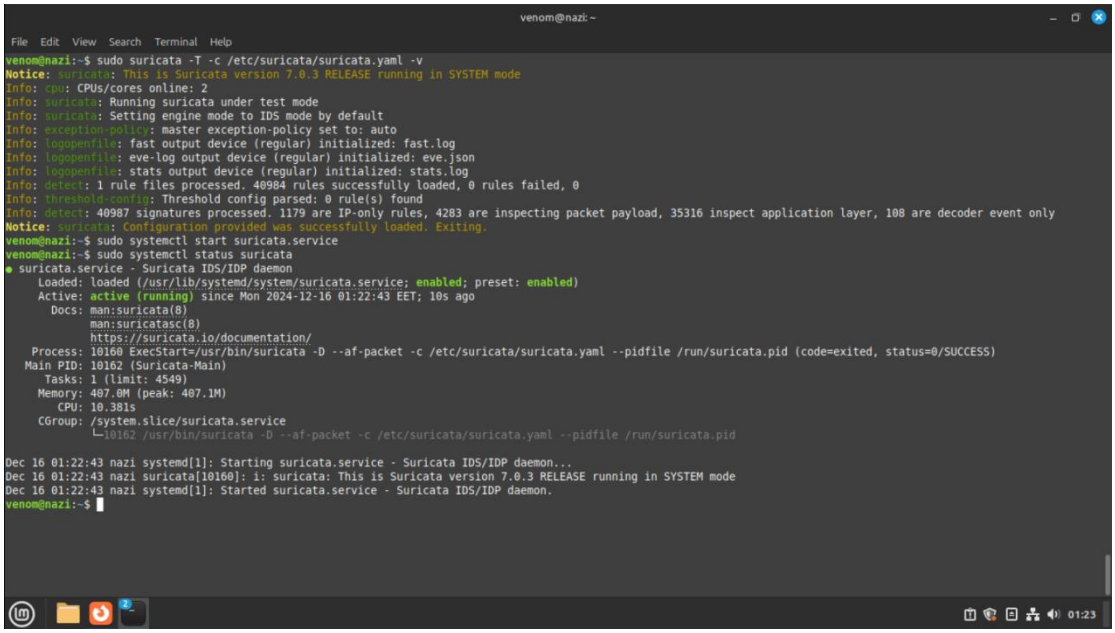

Fig.4.14 Suricata running and config validation

With Suricata running, its intrusion detection functionality can be tested by sending a request to **http://testmynids.org/index.htm.** To verify detection, the log file can be reviewed to know if suricata detected any suspicious behaviour.

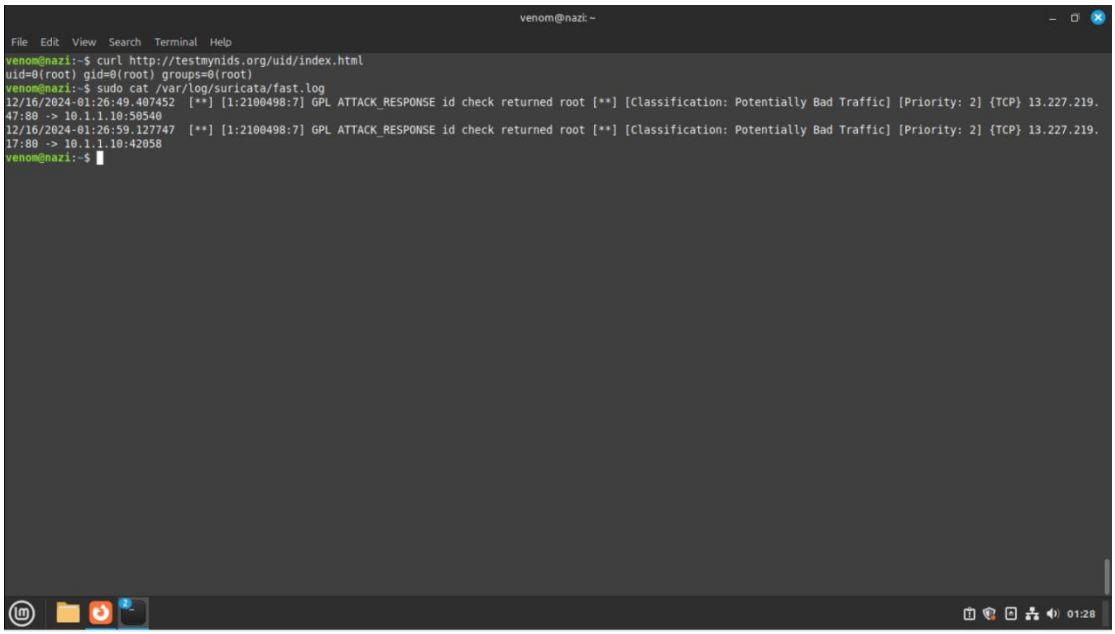

Fig.4.15 Suricata IDS testing

1.21 Summary

In this chapter, we explained the deployment environment of Cloud Storage that was basically composed of three important elements, namely: Sophos XG, Linux Mint, and Suricata.

- Sophos XG This will provide gateway services to the Linux computer while providing protection capabilities to the virtual network designed in VMware Workstation Pro.
- Linux Mint provides the operating system that hosts the Cloud Storage application alongside Suricata.
- Suricata is a high-class security tool that provides IDPS/intrusion prevention/detection system features, thus enhancing the general security of the deployment environment.

Chapter 5: Implementation

1.22 Introduction

The Implementation chapter is the third and the last chapter of the paper that focuses on the practical development of the cloud storage application and the steps and processes followed to create the system. It explains the development of back end server that controls the the main functionalities of the system that includes authentication, authorization, file management and development of secure API endpoints for implementing the major features of the system for the client.

Besides the back-end, in this chapter we will also briefly demonstrate the front-end and how the user interface with the application.

This chapter thus completes the link between the conceptual architecture and the actual application.

The implementation of the application is done using Model-View-Controller (MVC) architecture, so it's the first section of the project we will discuss.

1.23 Installation

Before we start the development stage of the cloud storage application, first we download Node.js and MongoDB on the deployment environment which is Linux Mint we configured previously. First we start with the process of downloading Node.js.

5.2.1 Node.js installation

The installation process of Node.js is straight forward and it's done with a single command performed on the terminal: **sudo apt install nodejs**

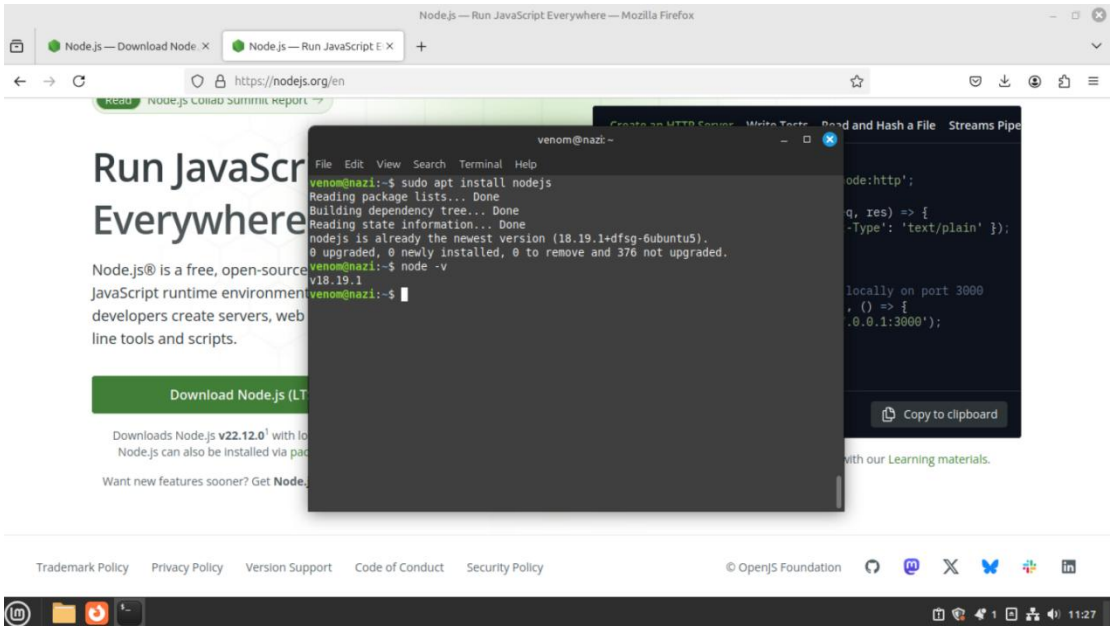

Fig.5.1 Node.js installation process on Linux

Once the installation is done, verify the installation process with the command: '**node -v**'

Which shows the version of Node.js that is installed on the machine, in case of a successful installation it will show the version of the Linux machine.

5.2.2 **MongoDB Installation**

The installation process of MongoDB is well documented on their official website and we used the same steps they presented on their official website. First we download curl and gnupg to import the public key.

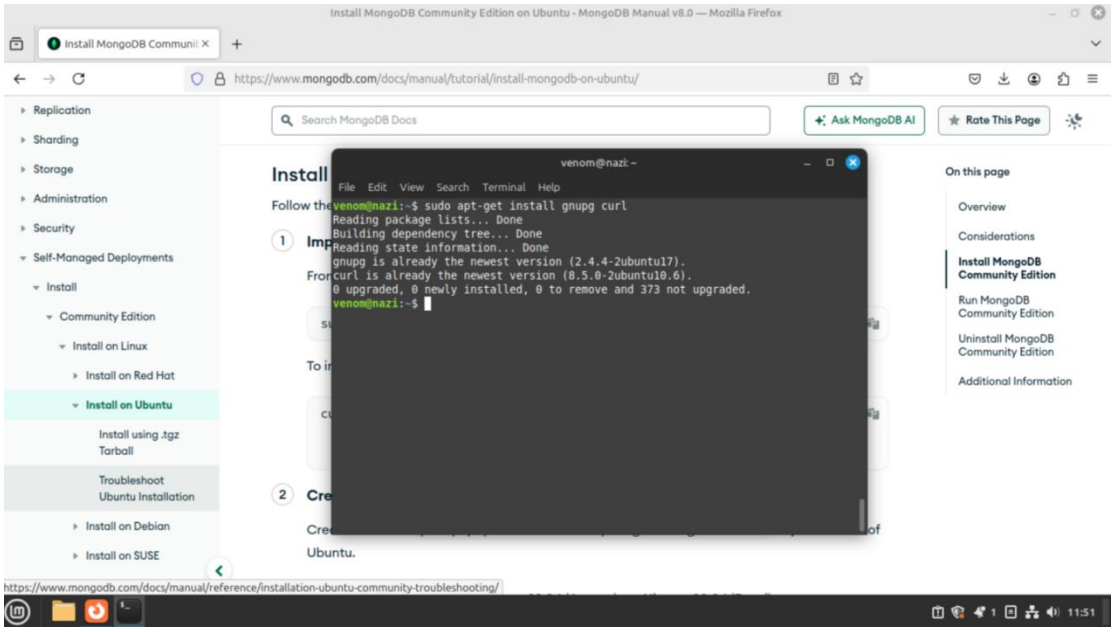

Fig.5.2 Curl and gnupg installation

Once curl and gnupg a MongoDB GPG public key is imported with the command shown in the next figure.

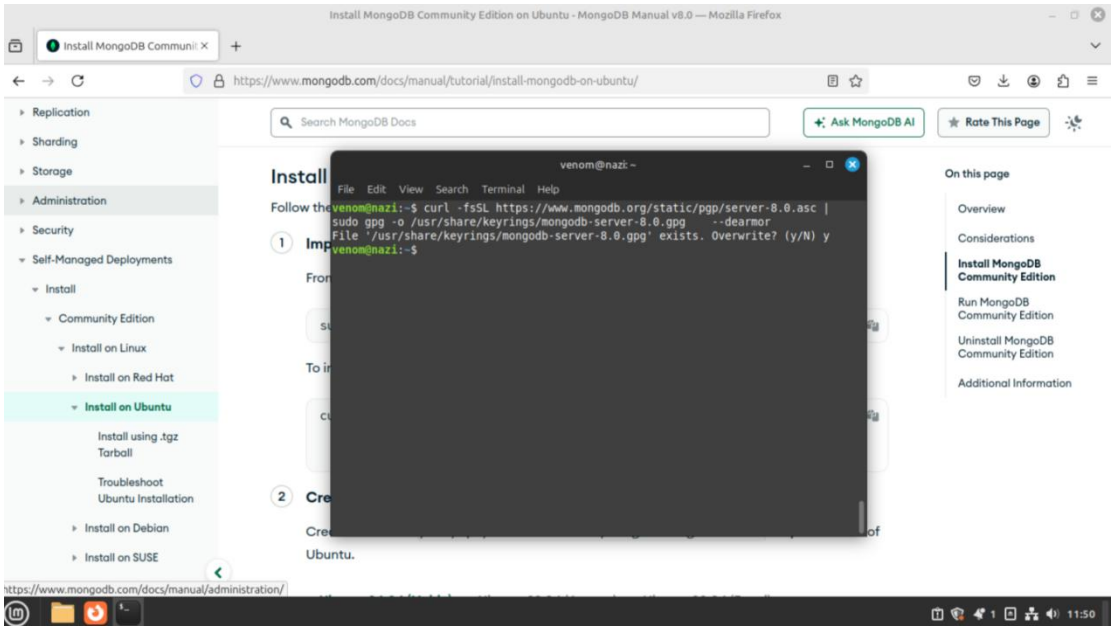

Fig.5.3 MongoDB GPG public key installation

Once MongoDB GPG public key is installed successfully, a list file must be created and depending on version of the operating system is used a specific

command for that OS is performed in the terminal. The next figure shows the command for creating the required list file for Linux Mint.

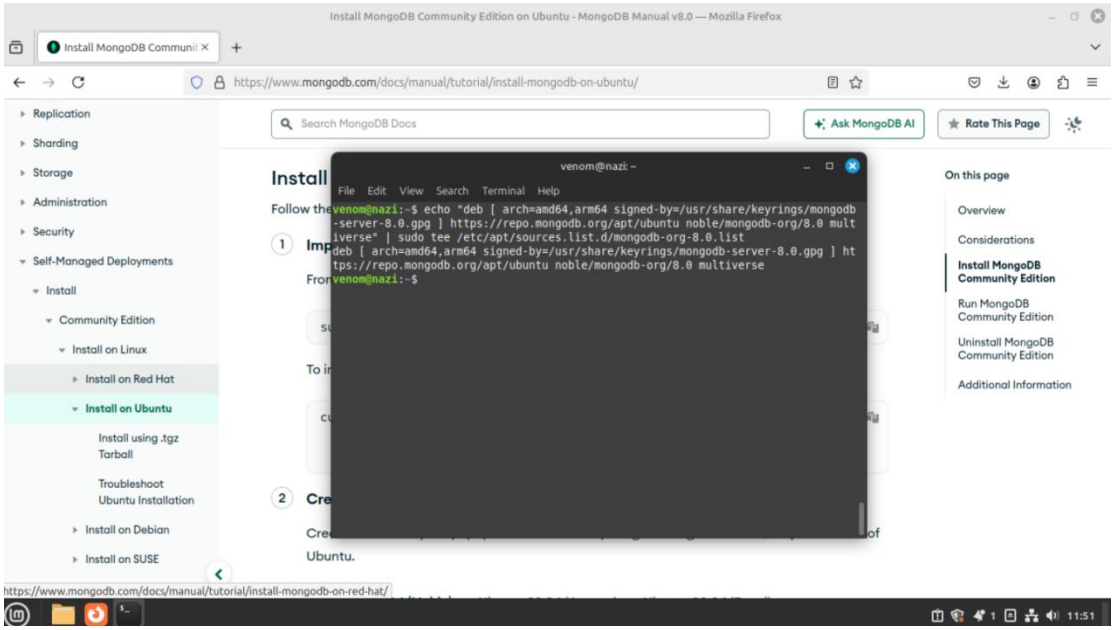

Fig.5.4 MongoDB list file creation

After all that's done it's possible now to install MongoDB Community server using the command: sudo apt-get install mongodb-org. The next figure shows the output of the installation of MongoDB community server.

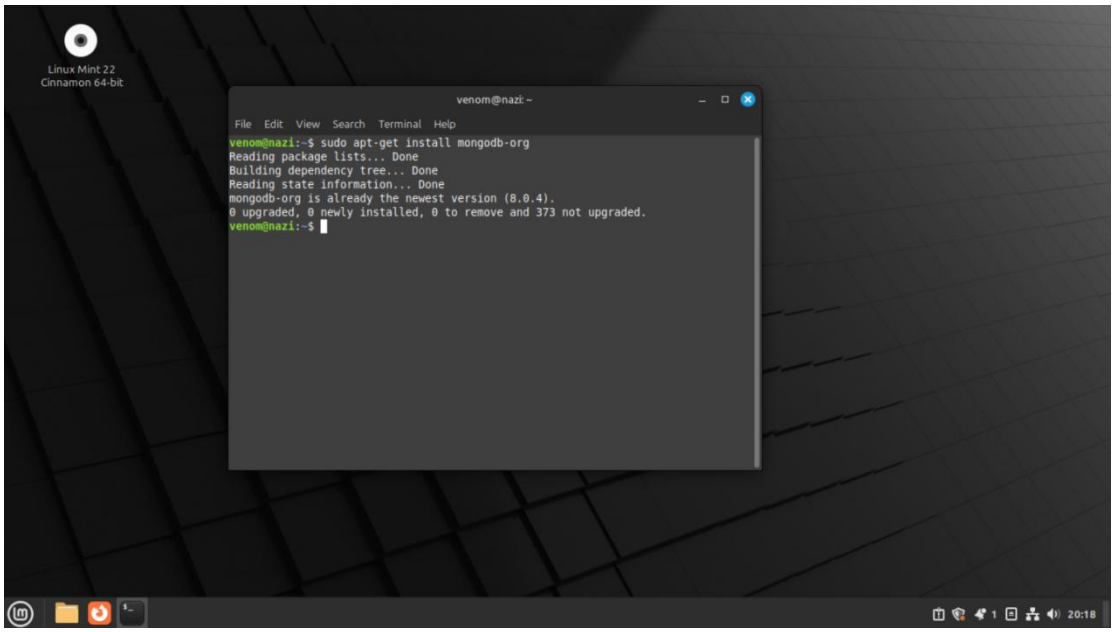

Fig.5.5 MongoDB Community server installation

After installing MongoDB successfully it's possible to run MongoDB server as a system service. The next figure shows the process of starting MongoDB server and showing the status of MongoDB server.

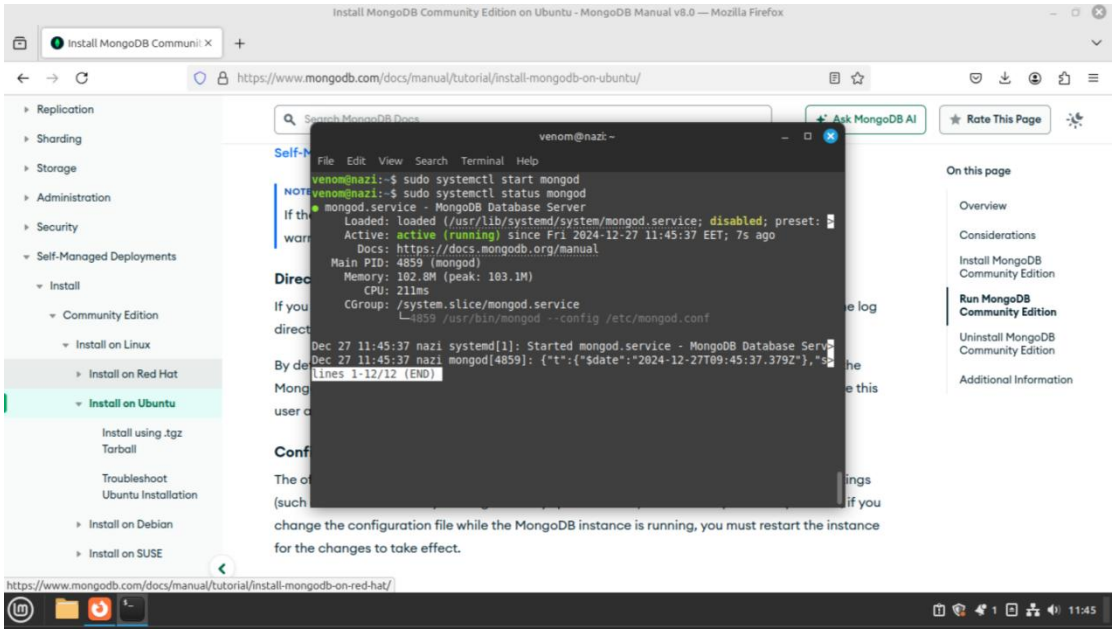

Fig.5.6 MongoDB server start and show status

By finishing the installation of Node.js and MongoDB, we can then start the development of the cloud storage application.

1.24 Model-View-Controller (MVC)

The Model-View-Controller (MVC) is a design pattern that separate the application into three main logical components Model, View, and Controller. Each component is built to handle specific development aspects of the system to isolate the business logic (The main functionalities of the system) and the application logic (The common tasks that are not related to the main functionality of the system).

Functionalities of the MVC components:

- **Model**: refers to the data-related logic that the end user works with, in our application it's the file management, authentication, and authorization.

- **View**: used for all the user interface logic of the application. Based on the resource the user requested the application must render the requested interface if the user is authorized perform this action.
- **Controller**: It enables interconnection between the Model component and View component, by defining the operations the Model must do and interact with the View component to render to render the result to the end user.

By adhering the the MVC architecture principles, we created three main folders each corresponds to one of the main MVC components. These folders are: **models**, **views**, **controllers**.

1.25 Model Component

Model component handles the resource-related operations of the system. The main resources of our application are: **Users** and **Files**. So in the models folder we created two files each one serves its own purpose: **fileModel.js** and **userModel.js**.

In the Model component we will implement both the User Model and the File Model.

5.4.1 File Model

File Model is implemented in fileModel.js file. In this file we created the mongoose schema which defines the structure of the file documents that will be stored in the files collection in our MongoDB database. And in this file we created and exported the model of the file schema so it’s usable in multiple different files of the project.

```
const mongoose = require('mongoose');

const fileSchema = new mongoose.Schema({
  name: {
    type: String,
    required: [true, 'File must have a name'],
  },
  originalName: {
    type: String,
    required: [true, 'File must have an originalName'],
  },
  user: {
    type: mongoose.Schema.ObjectId,
    ref: 'User',
    required: [true, 'File must have User!'],
  },
});

const File = mongoose.model('File', fileSchema);
module.exports = File;
```

Fig.5.7 File Schema and Model implementation

File schema consists of three fields:

- **name**: string represents the name of the file a user did upload, and it has a required validator which triggers an error if the input did not contain the required field. The error message is the second element in the array used to declare the required validator in Mongoose schema.
- **originalName**: string represents the original name of the file when the user sent it. This field is used to be assigned to the filename attachment header when sending the file to the user.
- **user**: is an Object Id (Mongoose Schema Data Type) that refers to the id of the user who uploaded the file. This field is utilized to ensure only the user who created that file have the permission to retrieve it.

5.4.2 User Model

User model is implemented in userModel.js file. In this file we created the mongoose schema which defines the structure of the user documents that will be stored in the users collection in our MongoDB database. The user schema is extensive making it impractical to present it in a single figure. Therefore, it's implementation will be outlined in multiple steps for clarity.

```
const userSchema = mongoose.Schema({
  name: {
      type: String,
      required: [true, 'User must have a name!'],
    },
    email: {
      type: String,
      required: [true, 'Please provide your email!'],
      unique: [true, 'This email is already in use'],
      lowerCase: true,
      validate: [validator.isEmail, 'Please provide a valid email!'],
    },
    photo: {
      type: String,
      default: 'default.jpg',
    },
    role: {
      type: String,
      enum: ['user', 'admin'],
      default: 'user',
    },
  // ...etc,
});
```

Fig.5.8 User Schema (part one)

In the first part of the user schema we defined:

- **name**: string representing the username of the user. This field have a **required** validator set to true, and if it's not provided in a user create document query it will trigger an error with the message shown in the figure ('User must have a name!').
- **email**: string representing the user's email address, This field have **unique** schema option and two validators **required** and custom validator

'**validator.isEmail**' (a function that checks if the input string is a valid email). And an error will be triggered if the input string didn't match all the validators.

- **photo**: is a string represents the path of the user's profile picture. This field have default schema option set to the value of 'default.jpg'. which is the default picture for all the users in our application.
- **role**: represents the user permissions and access level to the system. This filed have a **default** schema option value of 'user' and an **enum** validator with two values of ('user' and 'admin') which are the only roles in our application. This field is used to prevent normal users to perform higher-privilege operations such as deleting a user document, or retrieving all the database documents.

```
const userSchema = mongoose.Schema({
  // ...etc,
  password: {
    type: String,
    required: [true, 'Please provide a password'],
    minlength: [8, `password shouldn't be less than 8 characters`],
    maxlength: [32, `password shouldn't be more than 32 characters`],
    select: false,
  },
  passwordConfirm: {
    type: String,
    required: [true, 'Please confirm your password'],
    validate: {
      validator: function (el) {
        return el === this.password;
      },
      message: 'Please Confirm your password',
    },
  },
  passwordChangedAt: {
    type: Date,
    select: false,
  },
});
```

Fig.5.9 User Schema (part two)

- **password**: string represents the user key phrase that is used to sign in to the user's account (retrieving a JWT token). This field have three validators, **required**, **minlength** (ensures that the password isn't smaller than a fixed integer), and **maxlength** (ensures that the password isn't longer that a fixed integer). In addition to the validators this field have a **select** schema option set to false which prevents the password from being queried along the user document without selecting it specifically.
- **passwordConfirm**: string that ensures the user typed the password correctly. This field have a custom validator that checks if this field input is equal to the password field, and if it's not equal it will trigger an error. This field is not persisted in the database and only used to validate the user input.
- **passwordChangedAt**: is a date data type that represents the last time the user changed his password. And according to that time we will reject any JWT token issued to the user before that time period. This field has select schema option set to false which prevents this field from being queried along the user document without specifically selecting it.

Once the user Schema is declared, we define the methods and the query middlewares that we will use to separate the business logic from the application logic such as hashing the passwords or checking if the password has changed after JWT token issue date. The user model contains multiple methods and middleware functions such as:

Password hashing middleware function

Is a pre query middleware that will be run before sending save or create MongoDB query to the database. And this function checks if the password is not modified it will move to the next middleware. And in case of the password was modified this middleware function will hash the password with bcrypt and then moves to the next middleware. The next figure shows the code of the password hashing middleware function.

```
userSchema.pre('save', async function (next) {
  // Only run this function if password was actually modified
  if (!this.isModified('password')) return next();

  // Hash the password with the salt of 12
  this.password = await bcrypt.hash(this.password, 12);

  // Delete the passwordConfirm field
  this.passwordConfirm = undefined;
  next();
});
```

Fig.5.10 Password hashing middleware function

The salt is the second argument bcrypt takes to hash the password, which is a random string that is added to the plain text password so after hashing them both the hash algorithm's output is no longer predictable.

Password change date middleware function

Is a pre query middleware function that run before sending save or create MongoDB query to the database. This function checks if the password field has been modified, in case if it was modified the document object (accessed in the middleware function via the ***this*** keyword) we add the passwordChangedAt field and gives it the value of **Date.now() - 1000**. And Date.now() returns the number of milliseconds elapsed since the epoch (midnight of January, 1, 1970). The next figure shows the code of the password change date middleware function.

```
userSchema.pre('save', async function (next) {
  if (!this.isModified('password' || this.isNew)) return next();
  this.passwordChangedAt = Date.now() - 1000;
  next();
});
```

Fig.5.11 Password change date middleware function

Note: we subtracted 1000 of Date.now() output so that the password appears as it changed from a second ago. This approach prevents the issue when a JWT token is sent right back after changing the password since JWT timestamps are in seconds and not in milliseconds, if the passwordChangedAt is set to **Date.now()** it might have an earlier or the same issue date (**iat** - issued at) of the JWT token so when we verify the token it will be rejected because of the issue date is the same or very close to the passwordChangedAt date.

Password checking method

Mongoose schema method that uses bcrypt to handle the password validation. This method takes two arguments:

- **candidatePassword**: which is the password sent by the user in order to get access to his resources.
- **userPassword**: which is the encrypted user's password that is stored in our MongoDB database.

This method returns the result of the bcrypt compare which compares and returns a boolean if the first argument (candidatePassword) is a the plain text of the second argument (userPassword), and by doing this we can know if the user entered the correct password. The next figure shows the implementation of the password checking method.

```
userSchema.methods.checkPassword = async function (
  candidatePassword,
  userPassword
) {
  return await bcrypt.compare(candidatePassword, userPassword);
};
```

Fig.5.12 Password checking method

Password change timestamp validation method

Mongoose schema method that we utilize to check if a JWT token is issued before a user changed his password. As the field passwordChangedAt field is not persisted in the database unless the user has changed the password, so we check if the passwordChangedAt first, if it doesn't exist that means the user have never changed his password and the method returns false. And if the passwordChangedAt field is present we check if the JWT token timestamp is less than the user's document password change time stamp, if the JWT token timestamp is less than the password change date this method returns true indicating that the JWT token is not valid anymore. The next figure shows the implementation of the password change timestamp validation method.

```
userSchema.methods.changedPasswordAfter = function (JWTTimestamp) {
  if (this.passwordChangedAt) {
    const changedTimestamp = this.passwordChangedAt.getTime() / 1000;
    return JWTTimestamp < changedTimestamp;
  }
  return false;
};

const User = mongoose.model('User', userSchema);

module.exports = User;
```

Fig.5.13 Password change timestamp validation method

Once the user schema and the schema methods and middleware functions are implemented, we create the model and export it so it's usable in other files of the project.

Note: the user's document field passwordChangedAt is a UNIX based date data type, so the time is measured in milliseconds. And as JWT token timestamp is measured in seconds, we divide the **passwordChangedAt** value by 1000 so it's converted to seconds.

1.26 Controller Component

Controller component handles the communication between the Model and the View by processing the user's input, interacting with the model, and send a response or renders a page.

There is multiple controllers each created to handle a specific task, and these tasks includes authenticating a user, uploading/downloading a file, rendering a page, …etc.

5.5.1 Authentication Controller

Authentication controller handles the creation and signing of JSON Web Tokens (JWT), performing JWT validation, and sending the generated token via cookies. In addition we created the login, sign-up, and protect middlewares in the authentication controller file named **authController.js**.

JWT sign function

JWT token sign function (**signToken**) takes a single argument where we pass the id of the user and that argument will be assigned to the JWT payload. We used jsonwebtoken module to sign the token, jsonwebtoken module has a method called sign that takes the payload as the first argument, the **secret** as the second argument and we used JWT_SECRET environment variable as the value for the **secret**, and the third argument is an options object for the JWT where we defined JWT_EXPIRES_IN environment variable as the value for the expiration date of the JWT token. The next figure shows the implementation of the **signToken** function.

```
const signToken = (id) =>
  jwt.sign({ id }, process.env.JWT_SECRET, {
    expiresIn: process.env.JWT_EXPIRES_IN,
  });
```

Fig.5.14 JWT sign function implementation

Create and Send token function

This function (createSendToken) handles the creation and sending the JWT token to the user via cookies. This function accepts three arguments: user object, status code, and the response object. First this function signs the token using the id property of the user object, then create the cookie options where we define the expiration date and the secure flag, we defined the value of the expiration date to 90 days after the current time by adding the value of the current date in milliseconds and the value of the environment variable JWT_COOKIE_EXPIRES_IN converted to milliseconds. Once the cookie options are defined, we assign the cookie to the response object and give it a name of jwt and the value of the token signed earlier, and the third argument we assign the cookie options to the response object, then the function sends the response in a JSON format with the status, token, and the user object after removing the private information from the user document. The next figure shows the implementation of **createSendToken** function.

```
const createSendToken = (user, statusCode, res) => {
  const token = signToken(user.id);
  const cookieOptions = {
    expires: new Date(
      Date.now() + process.env.JWT_COOKIE_EXPIRES_IN * 1000 * 60 * 60 * 24
    ),
    httpOnly: true,
  };

  res.cookie('jwt', token, cookieOptions);
  user = {
    _id: user._id,
    name: user.name,
    email: user.email,
    role: user.role,
  };
  res.status(statusCode).json({
    status: 'success',
    token,
    data: {
      user,
    },
  });
};
```

Fig.5.15 createSendToken function implementation

JWT verification function

JWT verification function (**verify**) handles the verification of a JWT token validity. This function accepts a single argument which is the token, and this function utilize the verify method of the jsonwebtoken module and returns the token payload if the token was valid and throws an error in case of an invalid token. The next figure shows the implementation of the **verify** function.

```
const verify = async (token) =>
  new Promise((res, rej) => {
    jwt.verify(token, process.env.JWT_SECRET, (err, val) => {
      if (val) res(val);
      if (err) rej(err);
    });
  });
```

Fig.5.16 JWT verify function

Sign-up middleware function

Sign-up middleware (**signUp**) is an express middleware function that handles the sign-up operation for the users.

signUp middleware takes the name, email, password, and passwordConfirm from request body, using these inputs this function utilizes the User model to create a new user document, and on successful creation of the user document this middleware will send JWT token back to the client. The next figure shows the implementation of the **signUp** middleware.

```
exports.signUp = catchAsync(async (req, res, next) => {
  const { name, email, password, passwordConfirm } = req.body;
  const user = await User.create({ name, email, password, passwordConfirm });
  createSendToken(user, 200, res);
});
```

Fig.5.17 signUp middleware implementation

Note: catchAsync is a wrapper function that is utilized to catch any Promise rejection.

Login middleware function

Login middleware (**login**) is an express middleware function that handles the login operation of users.

The login middleware takes the email and password form request body and ensures that the client sent both. Then searches for a user document with the same email and utilize the checkPassword we defined earlier to check if the password sent in the request body is the same password stored in the user document and send a JWT token back to the client on valid input, and on invalid input the middleware sends an error back to the client The next figure shows the implementation of the login middleware.

```
exports.login = catchAsync(async (req, res, next) => {
  const { email, password } = req.body;
  if (!email || !password)
    return next(new AppError('Please provide email and password', 400));

  const user = await User.findOne({ email }).select('+password');

  if (!user) return next(new AppError('Incorrect email or password', 401));
  if (!(await user.checkPassword(password, user.password)))
    return next(new AppError('Incorrect email or password', 401));

  createSendToken(user, 200, res);
});
```

Fig.5.18 login middleware function

Protect middleware function

Protect middleware (**protect**) is an express middleware function that's utilized to protect specific api endpoints. This middleware checks if the token in the request cookie object is valid by using the verify function we created earlier, then verify it and use the id of the token payload to search for a user with the same id, then using the changedPasswordAfter this middleware checks if the JWT token is issued before the password has changed; and if any statement threw an error protect middleware will send back an error and reject the access

to the resource. The next figure shows the implementation of the protect middleware.

```
exports.protect = catchAsync(async (req, res, next) => {
  let token;
  if (req.cookies.jwt) token = req.cookies.jwt;
  if (!token)
    return next(
      new AppError('You are not logged in! Please log in to get access.', 401)
    );
  const decoded = await verify(token);
  if (!decoded) return next(new AppError('token verification failed!'));
  const currentUser = await User.findById(decoded.id);
  if (!currentUser)
    return next(
      new AppError(
        'The user belonging to this token does no longer exist.',
        403
      )
    );
  if (currentUser.changedPasswordAfter(decoded.iat))
    return next(
      new AppError('User recently changed password! Please log in again.', 401)
    );
  req.user = currentUser;
  res.locals.user = currentUser;
  next();
});
```

Fig.5.19 protect middleware implementation

Access restriction middleware

Access restriction (restrictTo) is a function that returns a middleware which restrict an action to a specific group of users based on their roles. This function accepts an array and check's if the user object role exists in that array and move to the next middleware if the user have permission; if the user's role doesn't exist in the roles array an error message will be sent back to the client. The next figure shows the implementation of the restrictTo middleware.

```
exports.restrictTo = (...roles) => {
  return (req, res, next) => {
    if (!roles.includes(req.user.role))
      return next(
        new AppError('You dont have permission to perform this aciton!', 403)
      );
    next();
  };
};
```

Fig.5.20 restrictTo function implementation

Logged in verification middleware

This middleware (isLoggedIn) is used to add the user object to the response locals object so it's accessible by pug template when rendering the overview page. The token verification in this middleware is exactly the same as the protect middleware. The next figure shows the implementation of isLoggedIn middleware.

```
exports.isLoggedIn = catchAsync(async (req, res, next) => {
  if (req.cookies.jwt) {
    try {
      const decoded = await verify(req.cookies.jwt);
      const currentUser = await User.findById(decoded.id);

      if (!currentUser) return next();

      if (currentUser.changedPasswordAfter(decoded.iat)) return next();

      res.locals.user = currentUser;
      req.user = currentUser;
    } catch (err) {
      return next();
    }
  }
  next();
});
```

Fig.5.21 isLoggedIn middleware implementation

5.5.2 File Controller

File controller handles the all the file-related operations such as file upload, file download, and the creation of file documents in the database. File controller includes the implementation of Multer storage which is a module which handles the multipart/form-data, enabling us to store the files sent by the clients.

File Upload

File upload operation is implemented in two steps:

- Read the file in the multipart/form-data and store it in the storage folder located in the root of the project folder.
- Create the file document with the data provided in the multipart/form-data in the database.

First we define the Multer storage configuration where we defined the destination of the file and the name of the file. Using these configurations we created the upload middleware and exported it so it's usable in other files of the project (mainly used for the route handlers).

The next figure shows the implementation of file uploading mechanism.

```
const multerStorage = multer.diskStorage({
  destination: (req, file, cb) => {
    cb(null, 'storage');
  },
  filename: (req, file, cb) => {
    cb(null, `user-${req.user._id}-${Date.now()}-${file.originalname}`);
  },
});

const upload = multer({
  storage: multerStorage,
});

exports.uploadFile = upload.single('file');
```

Fig.5.22 File upload mechanism

Once the file is uploaded successfully we create the file document in the database with the file's metadata and the id of the user who uploaded it. The next figure shows the implementation of the file document creation middleware (createFile).

```
exports.createFile = catchAsync(async (req, res, next) => {
  const file = req.file || null;
  if (!file) {
    return next(
      new AppError('no file is found in request please try again later!', 400)
    );
  }
  await File.create({
    name: req.file.filename,
    originalName: req.file.originalname,
    user: req.user._id,
  });
  res.status(200).json({
    status: 'success',
    message: 'file uploaded successfully',
  });
});
```

Fig.5.23 File document creation middleware

File download middleware

File download (**getFile**) is an express middleware function that takes the file id from request and searches the database for a file with the same id and with the same user's id to ensure only the real owner of the file can download it. Once file document is retrieved from the database this middleware sends it back to the client requested it if he was authorized to access it. The next figure shows the implementation of getFile middleware function.

```
exports.getFile = catchAsync(async (req, res, next) => {
  const file = await File.findOne({
    _id: req.params.fileId,
    user: req.user._id,
  });

  if (!file) {
    return next(new AppError('No File found with that id!', 404));
  }

  const filePath = path.join(__dirname, `../storage/${file.name}`);

  res.setHeader('Content-Type', 'application/octet-stream');
  res.setHeader('Content-Disposition', `attachment; filename="${file.name}"`);
  res.download(filePath);
});
```

Fig.5.24 File download middleware implementation

5.5.3 User Controller

User controller component is implemented in a file with the name of userControllers.js. This component handles the operations on the user documents stored in the database and this controller is mainly used to perform admin related tasks such as retrieving all the user document from the database.

Retrieving all the users documents

All the users documents are retrieved by a single middleware (getAllUsers). This middleware sends a MongoDB find query and not specifying any filter criteria so it queries all the user documents and sends the query result back to the client in JSON format. This middleware then used in the user routes file and only restricted to users with role of ‘admin’ who can proceed to perform this action.

The next figure shows the implementation of getAllUsers middleware.

```
exports.getAllUsers = catchAsync(async (req, res, next) => {
  const users = await User.find();
  res.status(200).json({
    status: 'success',
    data: {
      users,
    },
  });
});
```

Fig.5.25 getAllUsers middleware implementation

5.5.4 View Controller

View controller is implemented in a file with the name viewController.js, and it handles the rendering of the pages for the client. There is three main pages in our application:

- Overview page: the main page where the user can upload and download his files, and this page is rendered using **getOverview** middleware.
- Login page: the page where the client sends an email with a password to have access to an existed account, and this page is rendered using **getLogin** middleware.
- Sign-up page: the page where a client sends his credentials to create a user account in our application, and this page is rendered using **getSignUp** middleware.

The next figure shows the middleware that renders the pages based on their corresponding pug template (will be showed in this chapter later).

```
exports.getOverview = catchAsync(async (req, res, next) => {
  if (!req.user) return res.redirect('/login');
  const files = await File.find({ user: req.user._id });
  res.status(200).render('overview', {
    files,
  });
});

exports.getLogin = (req, res, next) => {
  res.status(200).render('login');
};

exports.getSignUp = (req, res, next) => {
  res.status(200).render('signUp');
};
```

Fig.5.26 View controller implementation

1.27 View Component

View component is rendered using pug template for easier and faster development. All the pug files are written in a 'views' folder which is a separate folder located in the root folder of the project.

5.6.1 Overview page

Overview page presents the client his uploaded files allowing him to download or upload his files. Since all the pages of the application is illustrated earlier in chapter 2, in this chapter we will only show the implementation of the page itself. The next figure shows the implementation of the overview page.

```
extends base

block content
  .files-head
    form#upload(action="")
      label.fileBtn(for="file") Upload file
      input(type="file" id="file" name="file")
      input(type="submit")

  .files
    if files
      each file in files
        .file-box
          .file-holder
            i.fa-regular.fa-file
            .filename #{file.originalName}
          a.download(href=`/api/files/${file._id}`) Download
```

Fig.5.27 Overview page implementation

5.6.2 **Login page**

Login page provides the client a form containing two inputs email and password so the clients can send his credentials to the server and obtain access to his resources. The next figure shows the implementation of the login page.

```
extends base
block content
  .form-box
    h3 Login
    form#login
      input(type="email" id="email" placeholder="Email")
      input(type="password" id="password" placeholder="Password")
      input(input type="submit" value="Send")
    p.text You don't have an account? please
      | !{' '}
      a(href="/sign-up") Create One
```

Fig.5.28 Login page implementation

Login page form is used to send the information to /api/users/login endpoint that we will implement later in this chapter.

5.6.3 Sign Up page

Sign Up page provides the client a form containing four inputs name, email, password, and password confirm so the client can send his information and create an account in our application. The next figure shows the implementation of the Sign Up page.

```
extends base
block content
  .form-box
    h3 Sign Up
    form#sign-up
      input(type="text" id="name" placeholder="name")
      input(type="email" id="email" placeholder="Email")
      input(type="password" id="password" placeholder="Password")
      input(type="password" id="password-confirm" placeholder="Password Confirm")
      input(input type="submit" value="Send")
    p.text Already have an account? please
      | !{' '}
      a(href="/login") Log in
```

Fig.5.29 Sign Up page implementation

Sign Up page form is used to send the information to /api/users/sign-up endpoint that we will implement later in this chapter.

1.28 Application Programming Interface (API)

This section shows the implementation of the API which consists of three route handlers:

- userRoutes.js: which handles the requests sent to the user-related API endpoints.
- fileRoutes.js: which handles the requests sent to the file-related API endpoints.
- viewRoutes.js: which handles the rendering of the pages (overview, login, sign up).

All the route handlers are mounted in the main application file '**app.js**' which is located in the root of the project folder and exported so it's usable in the server.js file where we initiate the server.

Note: all the code of the project can be retrieved from the GitHub repository https://github.com/emad-saud/Simple-Cloud-Storage

5.7.1 Users Routes

User-related API endpoints are implemented in userRoutes.js and it handles the requests sent to '/api/users' route. There are three sub-routes for the User API:

- '/': handles GET requests and utilizes the getAllUsers middleware exported from **User Controller** to retrieve all the user documents from the database.
- '**/sign-up**': handles POST requests and utilizes the signUp middleware exported from the authentication controller to create a user document and obtain a JWT token for that account.
- '**/login**': Handles POST requests and utilizes the login middleware exported from the authentication controller to login or obtain a JWT token to have access for a specific account.

Once the routes are created, the router object is exported so it's usable in other files of the project. The next figure shows the implementation of the User Routes.

```
const router = express.Router();

router
  .route('/')
  .get(
    authController.protect,
    authController.restrictTo('admin'),
    userController.getAllUsers
  );

router.route('/sign-up').post(authController.signUp);
router.route('/login').post(authController.login);

module.exports = router;
```

Fig.5.30 User Routes implementation

5.7.2 **Files Routes**

Files API endpoints are implemented in fileRoutes.js and it handles the requests sent to '/api/files' routes. There are two sub-routes for the Files API:

- '**/upload**': a protected route (requires valid token) which handles the uploading of files. This route utilizes the uploadFile middleware exported from file controller to store the file uploaded into the 'storage' folder; and also utilizes createFile middleware exported from file controller to create the file document in the database.
- '**/:fileId**': a protected route which handles the downloading of files by utilizing the getFile middleware exported from file controller.

Once the routes are created, the router object is exported so it's usable in other files of the project. The next figure shows the implementation of the Files Routes.

```
const router = express.Router();

router.post(
  '/upload',
  authController.protect,
  fileController.uploadFile,
  fileController.createFile
);

router.route('/:fileId').get(authController.protect, fileController.getFile);

module.exports = router;
```

Fig.5.31 Files Routes implementation

5.7.3 Views Routes

Views routes are implemented in viewRoutes.js, and based on the route the user navigates to, it renders an appropriate page. There is three routes for the views and they are mounted on the root URL of the application '/'. The views routes implemented are:

- '/': handles GET requests to the root URL of the application and utilizes the getOverview middleware to render the overview page.
- '**/login**': handles GET requests to the login route and utilizes the getLogin middleware exported from the Views Controller to render the login page.
- '**/sign-up**': handles GET requests to the sign-up route and utilizes the getSignUp middleware exported from the Views Controller to render the sign up page.

Once the routes are created the router object is exported so it's usable in other files of the application. The next figure shows the implementation of the View Routes.

```
const router = express.Router();

router.get('/', authController.isLoggedIn, viewController.getOverview);
router.get('/login', viewController.getLogin);
router.get('/sign-up', viewController.getSignUp);

module.exports = router;
```

Fig.5.32 View Routes implementation

Once all the route handlers are created and exported, they're mounted in the main application file app.js which we will implement in this chapter.

1.29 Main Application File

The main application file (app.js) is created in the root of the project folder. In this file we configured pug template, embedded cookies and request body to the request object, serve static files, configured express to accept files, sanitized the client input, and mounted the route handlers. Then the main file is exported so it's usable in the server.js file. The next figure shows the imported modules that we used to implement the main application file.

```
const express = require('express');
const path = require('path');
const mongoSanitize = require('express-mongo-sanitize');
const cookieParser = require('cookie-parser');
const xss = require('xss-clean');
```

Fig.5.33 Main file (app.js) modules

The main file (app.js) uses the middleware functions exported from the previous tools to perform multiple operations such as:

- express: is the framework that we used to create the application module that we export from app.js file, and we used the built-in body parser to embed the request body to request object, and used static files middleware to server the static files to the client.
- path: is node module that we used to transform file paths efficiently.
- mongoSanitize: is used to sanitize the user input preventing NoSQL injection.
- cookieParser: is used to embed the cookies to the request object.
- xss: is used to sanitize the user input preventing Cross-Site-Scripting exploits.

The next figure shows the configuration of the application module.

```
1  const app = express();
2
3  app.set('view engine', 'pug');
4  app.set('views', path.join(__dirname, 'views'));
5
6  // Serving static files
7  app.use(express.static(path.join(__dirname, 'public')));
8
9  app.use(express.json());
10
11 app.use(express.urlencoded({ extended: true }));
12
13 app.use(cookieParser());
14
15 app.use(mongoSanitize());
16
17 app.use(xss());
```

Fig.5.34 Application module configuration

Once the application module is configured, the routes handlers are mounted, and the application module is exported.

```
const userRouter = require('./routes/userRoutes');
const fileRouter = require('./routes/fileRoutes');
const viewRouter = require('./routes/viewRoutes');

app.use('/', viewRouter);
app.use('/api/users', userRouter);
app.use('/api/files', fileRouter);

module.exports = app;
```

Fig.5.35 Route handlers mounting

5.8.1 Server Initiation

The server is initiated in server.js file which is located in the root of the project folder, which is used to add the configuration file (config.env) variables to the process object, connect to the database, and initiate the server on port 3000. The next figure shows the implementation of server.js file.

```
const mongoose = require('mongoose');
const dotenv = require('dotenv');
dotenv.config({ path: `${__dirname}/config.env` });

const app = require('./app');

mongoose
  .connect(process.env.DATABASE_LOCAL)
  .then(() => {
    console.log('DB connection successful!');
  })
  .catch((err) => {
    console.log('failed to connect to the database! exiting the program...');
    process.exit(1);
  });

const port = process.env.PORT || 3000;
const server = app.listen(port, () => {
  console.log(`App is running on port ${port}...`);
});
```

Fig.5.36 Server file implementation

The next figure shows the implementation of the configuration file (config.env).

```
DATABASE_LOCAL=mongodb://localhost:27017/cloud
PORT=3000
JWT_SECRET=this-is-my-very-secret-string-ab
JWT_EXPIRES_IN=90d
JWT_COOKIE_EXPIRES_IN=90
```

Fig.5.37 Configuration file (config.env)

Chapter 6: Conclusion

1.30 Conclusion

The main aim of the project was to develop and deploy a secure and user-friendly cloud storage application for effective management of authentication, facilitation of file management, and maintenance of data security. This project is, therefore, aimed at creating a robust solution for cloud-based file storage through a harmonious integration of the features of Node.js, MongoDB, Sophos XG, and Suricata.

Throughout the project, we successfully designed a robust system architecture with clear data modeling. Implemented authentication and file handling back-end functionalities in an efficient and secure manner. Established a secure deployment environment leveraging VMware, Linux Mint, Sophos XG, and Suricata. Created a user-friendly front-end interface for efficient file management.

The only major challenge faced was in configuring the security features to work in tandem with the back-end server. Working around that challenge improved my understanding of network security and practically showed why security needs to be a part of the development life-cycle.

This project demonstrates how thoughtful design and implementation can facilitate to the development of a secure and effective cloud storage application, establishing a basis for advancements and progressing within this domain.

1.31 Future Work

While the project has met its set objectives, there is still significant room for future enhancements to improve functionality, scalability, and user experience. Some of the future features are:

1. Enhanced User Features: Implementing features such as file deletion, renaming of files, and changes in user details, including username, password, and profile picture.
2. Advanced Security Measures: Implementing multi-factor authentication (MFA) to enhance account security and integrating refresh tokens to better manage sessions.

1.32 References

Express.js. (n.d.). *Express Documentation*. Retrieved September 19, 2024, from https://expressjs.com/en/4x/api.html

MongoDB. (n.d.). *MongoDB Documentation*. Retrieved September 24, 2024, from https://www.mongodb.com/docs/

Mongoose (n.d.). *Mongoose Documentation*. Retrieved Septermber 26, 2024, from https://mongoosejs.com/docs/

National Institute of Standards and Technology. (2013). *Cloud computing standards roadmap: version 2* (Special Publication 500-291). U.S. Department of Commerce. Retrieved from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.500-291r2.pdf

Node.js (n.d.). *Node.js Documentation*. Retrieved September 15, 2024, from https://nodejs.org/docs/latest/api/

Rountree, D., & Castrillo, I. (2013). *The basics of cloud computing: Understanding the fundamentals of cloud computing in theory and practice*. Syngress.

OISF. (n.d.) Suricata Documentation. Retrieved October 3, 2024, from https://docs.suricata.io/en/latest/