Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка системи для централізованого зберігання та доступу до файлів

Виконав(ла):
студент(ка) 4 курсу групи СП-42
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

(підпис) Солтис М. В. (прізвище та ініціали)

Керівник (підпис) Михалик Д. М. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю. А. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2024

АНОТАЦІЯ

Кваліфікаційна робота бакалавра на тему «Розробка системи для централізованого зберігання та доступу до файлів» виконана Солтисом Максимом Васильовичем, студентом Тернопільського національного технічного університету імені Івана Пулюя, Факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42.

Відомості про обсяг: сторінок – 79, рисунків – 40, таблиць – 2, частин – 4, додатків – 3, посилань – 22.

Метою даної роботи є створення ефективної та безпечної системи, яка забезпечить зручне зберігання та доступ до файлів для користувачів. У першому розділі проведено аналіз предметної області та визначено вимоги до системи для зберігання файлів. Розроблено діаграми варіантів використання, обґрунтовано вибір ключових технологій, включаючи платформу Node.js та фреймворк NestJS для серверної частини, PostgreSQL як базу даних та Docker для контейнеризації.

Система для централізованого зберігання та доступу до файлів призначена для забезпечення безпечного і зручного способу зберігання та обміну файлами між користувачами. Вона підтримує функції аутентифікації користувачів, управління файлами та каталогами та створення унікальних посилань для завантаження.

Особливу увагу приділено безпеці системи, включаючи зберігання хешованих паролів та захист від несанкціонованого доступу. У результаті було створено систему, яка відповідає сучасним вимогам до програмних продуктів у сфері зберігання та управління файлами.

Об'єктом дослідження є система для централізованого зберігання та доступу до файлів. Предметом дослідження є інструменти та методи розробки веб-додатків для зберігання файлів із використанням Node.js, NestJS, PostgreSQL та Docker.

Ключові слова: централізоване зберігання файлів, Node.js, NestJS, PostgreSQL, Docker, проектування, система управління файлами, аутентифікація користувачів.

ABSTRACT

Bachelor's qualification work on the topic "Development of a System for Centralized File Storage and Access" was written by Soltys Maksym Vasylovych, a student of Ivan Puluj Ternopil National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, group SP-42.

Information about the scope: pages – 79, figures – 40, tables – 2, sections – 4, appendices – 3, references – 22.

The purpose of this work is to develop an efficient and secure system for centralized file storage and access. The work includes an analysis of the subject area, defining system requirements, developing use-case diagrams, and justifying the choice of the development environment and key technologies, such as Node.js and the NestJS framework for the server-side, PostgreSQL as the database, and Docker for containerization.

The system for centralized file storage and access is designed to provide a secure and convenient way for users to store and share files. It supports user registration and authentication, file and folder management, and the creation of unique download links for files. The system also allows for the creation of archives for directories, providing a simple and efficient way to organize and store large volumes of data.

Special attention was paid to the security of the system, including the storage of hashed passwords and protection against unauthorized access. As a result, a system was created that meets modern requirements for software products in the field of file storage and management.

The object of research is a modern system for centralized file storage and access. The subject of research is the tools and methods of developing web applications for file storage using Node.js, NestJS, PostgreSQL, and Docker.

Keywords: centralized file storage, Node.js, NestJS, PostgreSQL, Docker, design, file management system, user authentication.

ЗМІСТ

ВСТУП

У сучасному світі, де обсяги цифрової інформації постійно зростають, зберігання та управління цими даними стає однією з ключових задач для багатьох організацій. Інформаційні технології проникають у всі сфери діяльності, забезпечуючи ефективність та швидкість обробки інформації. З цієї причини потреба у надійних системах для централізованого зберігання та доступу до файлів є надзвичайно актуальною.

Актуальність теми – сучасне суспільство вимагає наявності ефективних та надійних рішень для зберігання та управління інформацією. В умовах постійного збільшення обсягу цифрових даних виникає необхідність у створенні централізованих систем для їхнього зберігання та доступу. Такі системи можуть значно полегшити роботу організацій та забезпечити більш ефективний доступ до інформації. Вони дозволяють зменшити витрати на інфраструктуру, підвищити безпеку даних та забезпечити їхню доступність у будь-який час і з будь-якого місця.

Централізовані системи зберігання даних стають важливою частиною інфраструктури сучасних компаній, організацій та установ. Вони забезпечують єдине місце для зберігання великих обсягів інформації, спрощуючи процеси її пошуку, доступу та управління. Завдяки цьому, підприємства можуть підвищити свою продуктивність, скоротити час на пошук необхідної інформації та знизити ризик втрати даних. В умовах постійного збільшення обсягів інформації та необхідності її безпечного зберігання, створення таких систем є надзвичайно важливим завданням.

Мета роботи – розробка системи для централізованого зберігання та доступу до файлів, яка забезпечуватиме надійність, масштабованість та зручність використання. Для досягнення цієї мети необхідно створити програмний продукт, який дозволятиме користувачам зберігати, організовувати та отримувати доступ до своїх файлів у зручний спосіб. Така система повинна мати інтуїтивно зрозумілий

інтерфейс, забезпечувати високий рівень безпеки даних та бути здатною до масштабування відповідно до потреб користувачів.

Завдання полягає у створенні програмного продукту, який дозволятиме користувачам зберігати, організовувати та отримувати доступ до своїх файлів у зручний спосіб. Для цього необхідно розробити інтерфейс користувача, серверну частину для управління даними та механізми забезпечення безпеки збереженої інформації. Інтерфейс користувача повинен бути простим та зручним у використанні, забезпечуючи швидкий доступ до необхідних файлів та можливість їх організації за різними критеріями. Серверна частина повинна забезпечувати надійне зберігання даних, їхню швидку обробку та захист від несанкціонованого доступу.

Об’єктом дослідження є процеси проектування та реалізації централізованої системи зберігання файлів. Це включає вивчення сучасних методів та технологій, які використовуються для створення таких систем, аналіз їхньої ефективності та безпеки, а також розробку власних рішень для забезпечення надійного зберігання та доступу до файлів. Особлива увага приділяється питанням безпеки даних, оскільки в сучасних умовах інформаційні загрози стають все більш актуальними.

Предметом дослідження є методи та технології, які використовуються для створення ефективної та безпечної системи зберігання та доступу до файлів. Для розробки системи використовуються сучасні технології, такі як TypeScript, Node.js, NestJS, PostgreSQL, Next.js та Docker. Ці інструменти дозволяють реалізувати надійну серверну частину, зручний інтерфейс користувача та забезпечити високий рівень безпеки даних. Використання цих технологій дозволяє створити систему, яка відповідає сучасним вимогам до продуктивності, надійності та безпеки.

На додаток до цього, розвиток хмарних технологій відкриває нові можливості для централізованого зберігання даних. Використання хмарних сервісів дозволяє зменшити витрати на інфраструктуру, забезпечити високу доступність та надійність зберігання даних, а також підвищити гнучкість системи. У даній роботі також розглядаються питання інтеграції з хмарними сервісами, їхні переваги та можливі недоліки.

Отже, створення системи для централізованого зберігання та доступу до файлів є важливим і актуальним завданням у сучасних умовах. Така система дозволить забезпечити ефективне управління даними, їхню надійність та безпеку, а також забезпечить зручність використання для кінцевих користувачів. У даній роботі буде розглянуто всі етапи розробки такої системи, починаючи від аналізу вимог та проектування, і закінчуючи реалізацією та тестуванням готового продукту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОПИС ЕТАПІВ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Аналіз предметної області

У сучасному світі обробка та зберігання інформації є критично важливими аспектами для будь-якої організації. Збільшення обсягів цифрових даних вимагає наявності надійних та ефективних рішень для їхнього зберігання та управління. Централізовані системи зберігання файлів дозволяють забезпечити зручний доступ до інформації, її безпеку та масштабованість, що робить їх незамінними інструментами у багатьох галузях, включаючи бізнес, науку, медицину та освіту.

Централізовані системи зберігання даних дозволяють користувачам зберігати, організовувати та отримувати доступ до своїх файлів у зручний спосіб, скорочуючи час на пошук інформації та підвищуючи ефективність роботи. Основними вимогами до таких систем є надійність, масштабованість, безпека та зручність використання [1].

Однією з ключових переваг централізованих систем зберігання даних є їхня здатність забезпечувати високий рівень безпеки. У сучасному світі, де кіберзагрози стають все більш поширеними, важливо забезпечити захист даних від несанкціонованого доступу, втрати чи пошкодження. Це досягається завдяки використанню сучасних методів аутентифікації та авторизації, шифрування даних, а також регулярного створення резервних копій [1].

Ще однією важливою перевагою є масштабованість таких систем. В умовах постійного зростання обсягів даних необхідно забезпечити можливість легкого розширення системи без значних затрат на інфраструктуру. Використання технологій контейнеризації, таких як Docker, дозволяє досягти високого рівня гнучкості та масштабованості системи [1].

Централізовані системи зберігання файлів дозволяють користувачам завантажувати та зберігати різноманітні типи файлів, включаючи документи, зображення, відео та інші медіафайли. Ці системи забезпечують структуроване

зберігання даних, дозволяючи користувачам створювати каталоги та підкаталоги для легкого управління своїми файлами. Це сприяє кращій організації

та швидшому доступу до необхідної інформації, що є особливо важливим для великих організацій з великим обсягом даних.

Централізовані системи також забезпечують функції спільного використання файлів, що дозволяє кільком користувачам одночасно отримувати доступ до тих самих файлів, редагувати їх та вносити зміни в режимі реального часу. Це особливо корисно для командної роботи, де учасники можуть спільно працювати над проектами, обмінюватися інформацією та забезпечувати актуальність даних.

Однією з ключових характеристик таких систем є можливість встановлення прав доступу для користувачів. Адміністратори системи можуть визначати, хто з користувачів матиме доступ до певних файлів або папок, а також встановлювати рівень доступу – наприклад, тільки для читання або з можливістю редагування. Це забезпечує додатковий рівень безпеки та дозволяє контролювати доступ до конфіденційної інформації.

Централізовані системи зберігання файлів також підтримують інтеграцію з іншими сервісами та додатками. Це може включати інтеграцію з офісними пакетами для редагування документів, системами управління проектами для організації роботи команди або сервісами для резервного копіювання даних. Така інтеграція дозволяє створити єдину екосистему для управління інформацією, що значно підвищує ефективність роботи користувачів.

Важливим аспектом є забезпечення надійності та доступності даних. Централізовані системи зберігання файлів використовують методи реплікації даних та створення резервних копій для захисту інформації від втрат у випадку технічних збоїв або аварій. Це дозволяє забезпечити безперебійну роботу системи та доступність даних у будь-який час.

Загалом, централізовані системи зберігання файлів є важливими інструментами для ефективного управління інформацією. Вони забезпечують зручний доступ до даних, їхню безпеку, масштабованість та інтеграцію з іншими сервісами, що дозволяє організаціям ефективно працювати з великими обсягами

інформації та забезпечувати її захист. Впровадження таких систем є актуальним завданням для багатьох компаній, що прагнуть підвищити ефективність своєї роботи та забезпечити безпеку даних.

1.2 Визначення варіантів використання системи користувачами

Визначення акторів системи:

На рисунку 1 представлені основні актори системи.

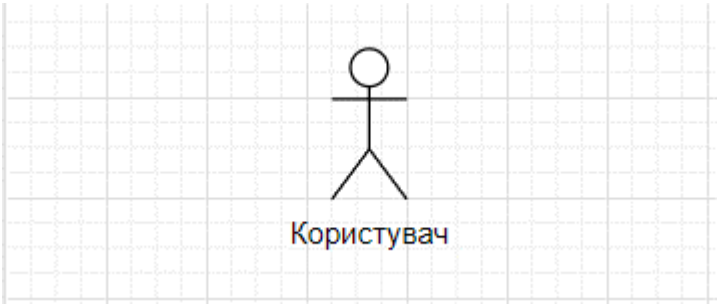

Рисунок 1.1 – Основні актори системи

Короткий опис акторів представлений в таблиці 1.1.

Табл. 1.1 – Визначення акторів

Актор	Короткий опис
Користувач	Має можливість реєструватися та проходити аутентифікацію у системі. Також має можливість завантажувати файли у сховище та зі сховища, переглядати елементи файлової системи, переглядати детальну інформацію про елементи файлової системи, редагувати дані елементів файлової системи, видаляти елементи файлової системи.

Визначення варіантів використання представлені таблиці 1.2.

Табл. 1.2 – Варіанти використання

Актор	Варіант використання	Опис
Користувач	Реєстрація	Можливість зареєструватися у системі, вказавши електронну пошту, ім’я, назву свого сховища та пароль.
Користувач	Аутентифікація	Можливість пройти процес встановлення належності користувачеві інформації в системі.
Користувач	Управління каталогами	Можливість створювати, редагувати та видаляти каталоги всередині кореневого каталога сховища та всередині інших каталогів.
Користувач	Управління файлами	Можливість завантажувати файли у будь-який каталог власного сховища, редагувати інформацію про них, видаляти та завантажувати на комп’ютер.
Користувач	Навігація та пошук у файловій системі	Можливість пошуку та фільтрації елементів файлової системи.
Користувач	Додавання нотаток	Можливість додавати нотатки для файлів та каталогів для інформативності.
Користувач	Надання доступу до файлів через посилання	Можливість генерувати унікальні посилання для надання доступу для завантаження файлів третім особам.

Розробка варіантів використання системи

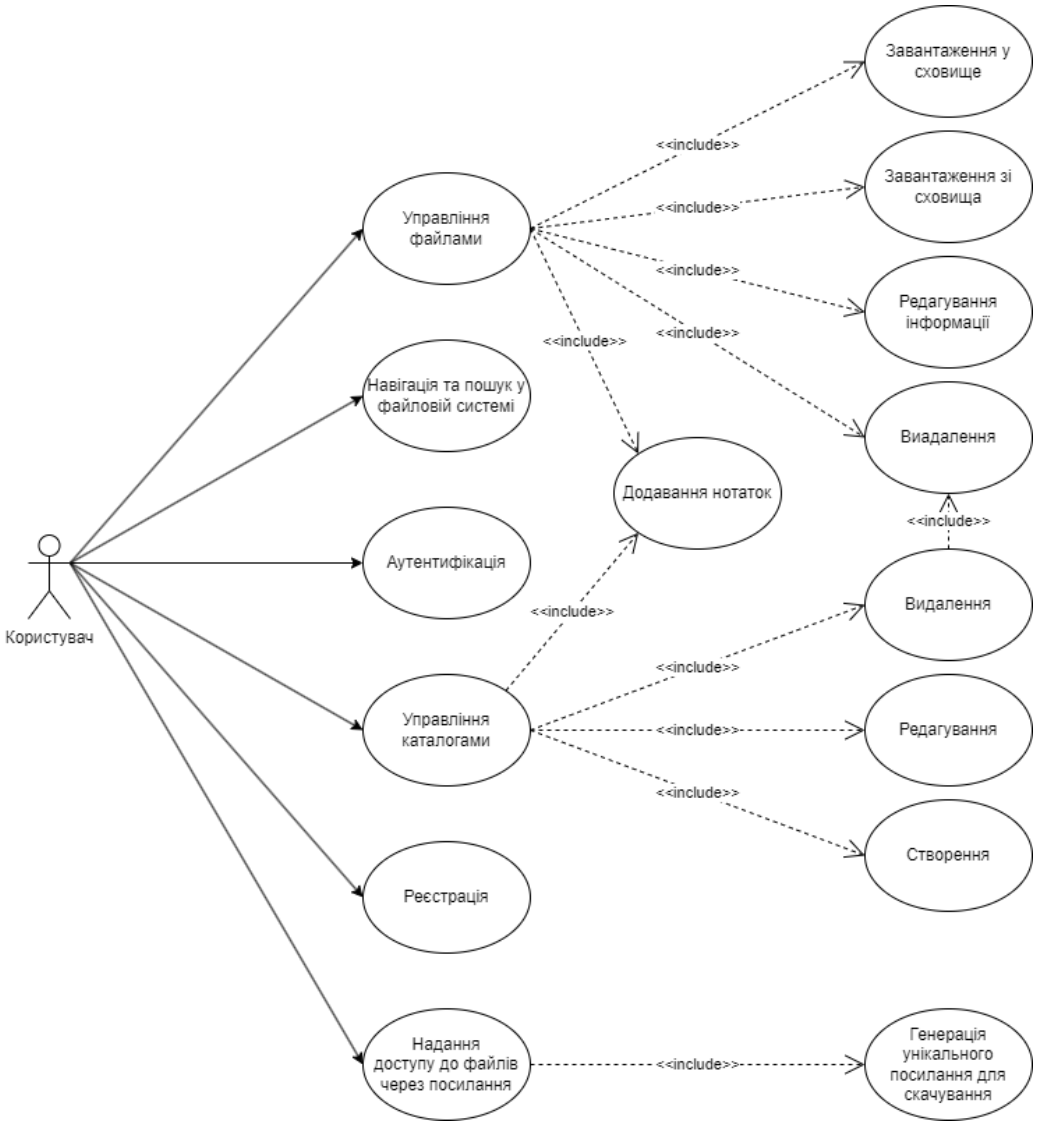

Рисунок 1.2 – Діаграма прецедентів системи

1.3 Постановка завдання

Після детально аналізу предметної області було виділено наступні функціональні та нефункціональні вимоги:

Функціональні вимоги:

1. Реєстрація та аутентифікація користувачів:

1.1. Система повинна підтримувати реєстрацію нових користувачів.

1.2. Користувачі повинні мати можливість входу в систему за допомогою електронної пошти та пароля.

2. Управління файлами:

2.1. Користувачі повинні мати можливість завантажувати файли різних форматів (документи, зображення, відео тощо).

2.2. Система повинна дозволяти створювати, перейменовувати, видаляти каталоги та підкаталоги для організації файлів.

2.3. Підтримка перетягування файлів та папок для зручного управління.

3. Доступ та спільне використання файлів:

3.1. Можливість створення посилань для спільного доступу до файлів без необхідності реєстрації.

4. Пошук та фільтрація файлів:

4.1. Система повинна мати функцію пошуку файлів за назвою, типом та іншими атрибутами.

4.2. Підтримка фільтрації файлів за різними критеріями (дата завантаження, розмір, тип файлу).

5. Забезпечення безпеки даних:

5.1. Всі чутливі дані користувача дані повинні зберігатися в зашифрованому вигляді.

5.2. Регулярне створення резервних копій даних для захисту від втрат.

6. Масштабованість та продуктивність:

6.1. Система повинна підтримувати одночасну роботу великої кількості користувачів без зниження продуктивності.

6.2. Забезпечення швидкого доступу до файлів навіть при великій кількості даних.

Нефункціональні вимоги:

1. Продуктивність:

1.1. Система повинна забезпечувати швидкий відгук на дії користувачів.

1.2. Система повинна бути здатна обробляти великі обсяги запитів одночасно без зниження продуктивності.

2. Надійність:

2.1. Система повинна забезпечувати високу доступність, мінімізуючи час простою.

2.2. Всі компоненти системи повинні бути відмовостійкими, з підтримкою автоматичного відновлення після збоїв.

3. Портативність та доступність:

3.1. Система повинна бути сумісною з різними браузерами, зокрема із такими як Google Chrome, Opera, Edge та Firefox.

3.2. Можливість легкого перенесення системи на інші сервери або хмарні платформи без втрати даних та функціональності.

4. Модульність та підтримуваність:

4.1. Кодова база повинна бути модульною, дозволяючи легко додавати нові функції або змінювати існуючі без значного впливу на інші частини системи.

4.2. Система повинна мати добре документований код та API для забезпечення легкості підтримки та розширення.

5. Зручність використання:

5.1. Інтерфейс користувача повинен бути інтуїтивно зрозумілим та простим у використанні, забезпечуючи позитивний користувацький досвід.

1.4 Технології розробки системи

Для реалізації системи централізованого зберігання та доступу до файлів було обрано низку сучасних технологій, які забезпечують високу продуктивність, надійність та масштабованість. Розглянемо кожну з них детальніше.

JavaScript є однією з найпопулярніших мов програмування у світі, широко використовуваною для розробки веб-додатків. Вона була створена в середині 1990-х років і з тих пір значно еволюціонувала, ставши невід'ємною частиною сучасної веб-розробки. JavaScript дозволяє додавати інтерактивність та динамічність до веб-сторінок, що робить користувацький досвід більш зручним та захоплюючим.

Однією з ключових особливостей JavaScript є її можливість виконуватись на стороні клієнта, що дозволяє значно знизити навантаження на сервер та підвищити швидкість відгуку додатків. Крім того, JavaScript підтримує асинхронні операції завдяки механізму “Even Loop”, що дозволяє виконувати кілька задач одночасно, не блокуючи користувацький інтерфейс. Це робить мову ідеальною для створення високопродуктивних веб-додатків.

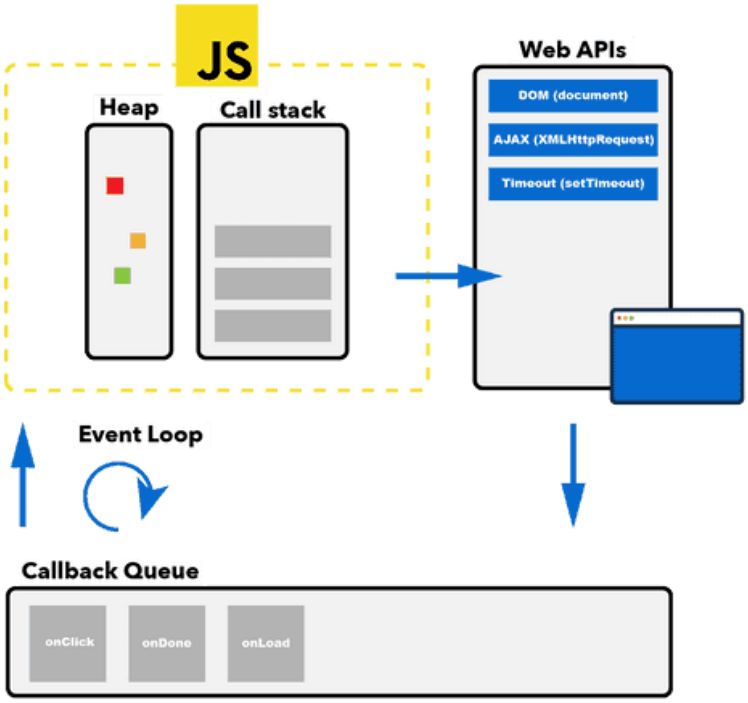

Рисунок 1.3 – Діаграма роботи механізму «Event Loop»

JavaScript також має велику і активну спільноту розробників, яка постійно створює та підтримує численні бібліотеки та фреймворки. Серед найвідоміших з них можна виділити React, Angular та Vue.js, які значно спрощують розробку

складних клієнтських інтерфейсів. Однак JavaScript не обмежується лише клієнтською стороною. З появою Node.js ця мова стала використовуватись і на сервері. Node.js дозволяє виконувати JavaScript-код на сервері, що робить можливим створення повноцінних серверних додатків. Популярні фреймворки для серверної розробки, такі як Express.js, NestJS та Koa, забезпечують потужні інструменти для розробки серверних частин додатків [2].

Node.js відзначається своєю подієвою архітектурою та асинхронною природою, що дозволяє створювати високонавантажені серверні додатки, здатні обробляти велику кількість одночасних запитів. Це робить Node.js ідеальним для реальних застосувань, таких як чати, стрімінгові сервіси та інші додатки, які потребують швидкого реагування.

JavaScript постійно оновлюється і вдосконалюється, з новими стандартами ECMAScript, що впроваджують нові можливості та поліпшення. Це забезпечує стабільний розвиток мови та її відповідність сучасним вимогам веб-розробки. Завдяки своїй гнучкості, багатофункціональності та активній спільноті, JavaScript залишається однією з найбільш перспективних мов програмування для розробки як клієнтських, так і серверних додатків.

TypeScript був обраний як основна мова програмування для розробки системи завдяки своїм перевагам у порівнянні зі звичайним JavaScript. TypeScript — це надбудова над JavaScript, яка додає статичну типізацію та інші потужні інструменти для розробників. Це дозволяє знаходити помилки на етапі компіляції, а не під час виконання коду, що значно знижує ризик виникнення помилок в продуктивному середовищі.

Одна з головних переваг TypeScript — це статична типізація. Завдяки їй розробники можуть визначати типи змінних, параметрів функцій і повертаємих значень, що дозволяє виявляти помилки на ранніх етапах розробки. Це значно покращує якість коду та полегшує його підтримку. Крім того, TypeScript підтримує сучасні можливості JavaScript, включаючи останні версії ECMAScript, що дозволяє використовувати всі новітні функції мови.

Вибір TypeScript для розробки цієї системи був зумовлений його здатністю забезпечувати високу якість коду, покращувати продуктивність розробки та знижувати ризик виникнення помилок. Це дозволяє створювати надійні та масштабовані додатки, які легко підтримувати та розширювати [3].

Node.js був обраний завдяки своїй здатності забезпечувати високопродуктивну та масштабовану серверну інфраструктуру. Це платформа для виконання коду JavaScript на сервері, яка забезпечує подієво-орієнтовану, неблокуючу архітектуру. Завдяки цьому Node.js може ефективно обробляти великі обсяги запитів та швидко реагувати на них. Це дозволяє створювати високонавантажені серверні додатки, що забезпечують швидкий доступ до даних та їх обробку.

Однією з ключових переваг Node.js є його швидкість. Завдяки використанню двигуна V8 від Google, який компілює JavaScript в машинний код, Node.js забезпечує надзвичайно високу швидкість виконання програм. Крім того, Node.js має багату екосистему модулів і пакетів, доступних через npm (Node Package Manager), що дозволяє легко інтегрувати різні бібліотеки та інструменти в проект [4].

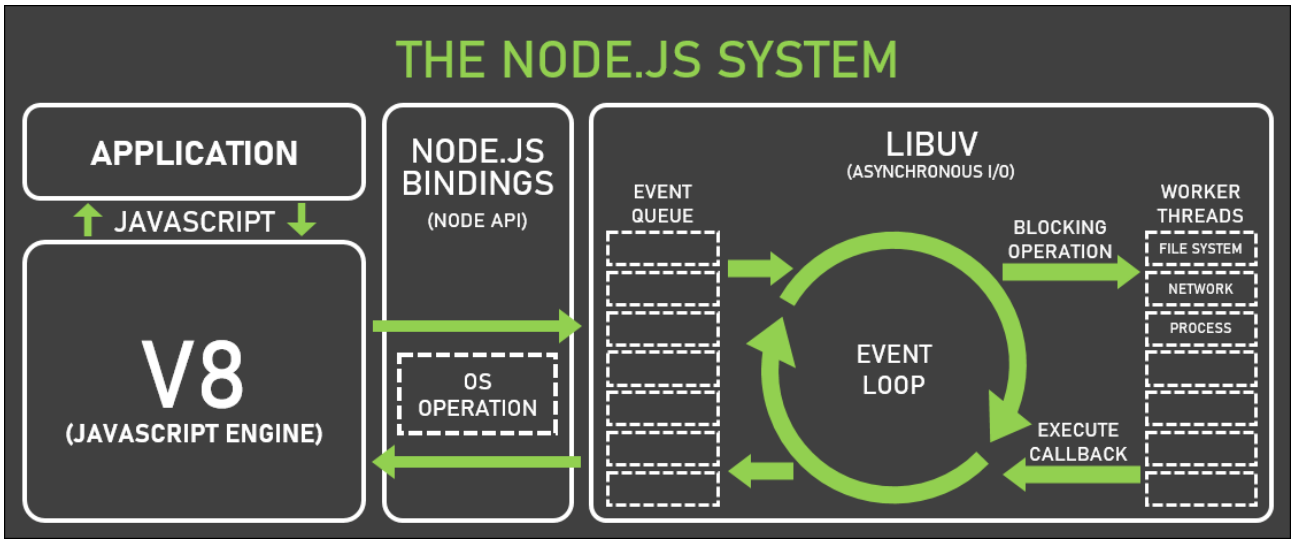

Рисунок 1.4 – Система платформи Node.js

Для взаємодії між клієнтом і сервером була обрана архітектура REST API (Representational State Transfer). REST API дозволяє створювати інтуїтивно зрозумілі та легко масштабовані сервіси, які використовують стандартні HTTP методи (GET, POST, PUT, DELETE) для здійснення операцій над ресурсами. Це забезпечує високу сумісність та гнучкість системи, дозволяючи клієнтам отримувати доступ до ресурсів та маніпулювати ними у зручний спосіб.

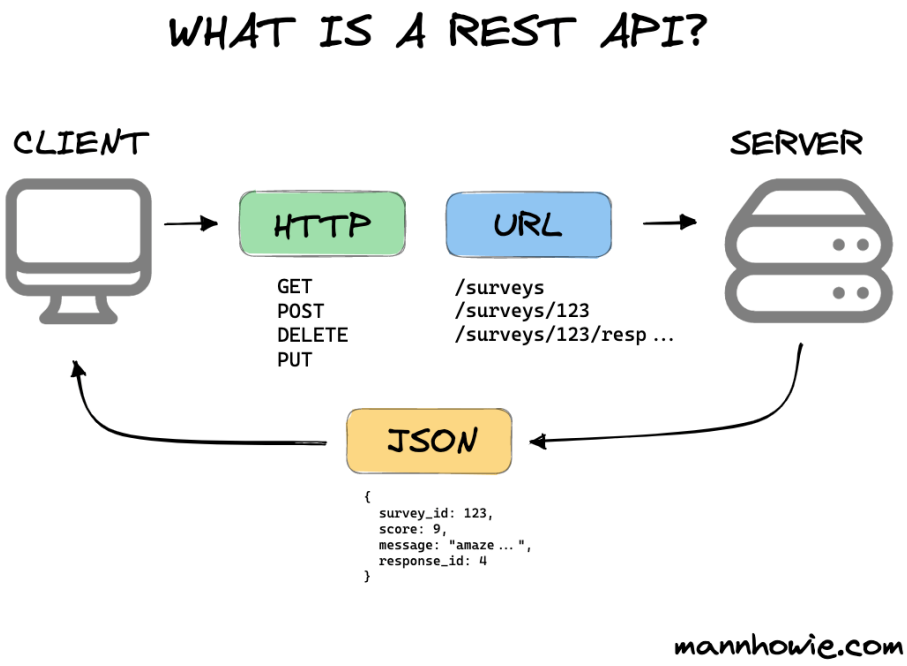

Рисунок 1.5 – Архітектура REST API

REST API забезпечує можливість створення розподілених систем, де клієнт і сервер можуть бути розташовані на різних машинах або навіть у різних мережах. Це дозволяє створювати масштабовані рішення, які можуть обробляти великі обсяги запитів і даних. Крім того, REST API спрощує інтеграцію з іншими системами та сервісами, забезпечуючи взаємодію через стандартні протоколи та формати даних (наприклад, JSON) [5].

NestJS був обраний через свою модульну архітектуру та використання TypeScript для покращення якості коду. Це сучасний фреймворк для створення серверних додатків на базі Node.js, який забезпечує розширені можливості для

статичної типізації. Завдяки цьому NestJS сприяє зменшенню кількості помилок та підвищенню надійності додатків.

NestJS побудований на принципах об'єктно-орієнтованого програмування та інверсії керування, що дозволяє створювати легко масштабовані та підтримувані додатки. Завдяки модульній архітектурі, можна легко розділити функціональність додатка на окремі модулі, що спрощує їх розробку та тестування. NestJS також підтримує використання різних бібліотек і інструментів, що дозволяє інтегрувати необхідну функціональність без зайвих зусиль [6].

Для роботи з архівами файлів було обрано бібліотеку Archiver. Archiver дозволяє створювати, змінювати та розпаковувати архіви різних форматів, таких як ZIP та TAR, що є важливим для зберігання та доступу до великої кількості файлів у системі. Використання архівів дозволяє зменшити обсяг даних, що зберігаються, та спрощує управління ними.

Archiver надає зручний API для роботи з архівами, що дозволяє легко інтегрувати функціональність архівації в додаток. Завдяки цьому можна створювати резервні копії даних, забезпечувати їх збереження та відновлення у разі необхідності. Крім того, Archiver підтримує стиснення даних, що дозволяє зменшити обсяг зберіганих файлів та підвищити ефективність використання дискового простору [7].

PostgreSQL була обрана як система управління базами даних завдяки своїй надійності, розширеним можливостям та високій продуктивності. PostgreSQL є потужною об'єктно-реляційною системою з відкритим вихідним кодом, яка активно розвивається протягом багатьох років. Вона забезпечує зберігання, організацію та управління великими обсягами даних, що є критичним для даної системи.

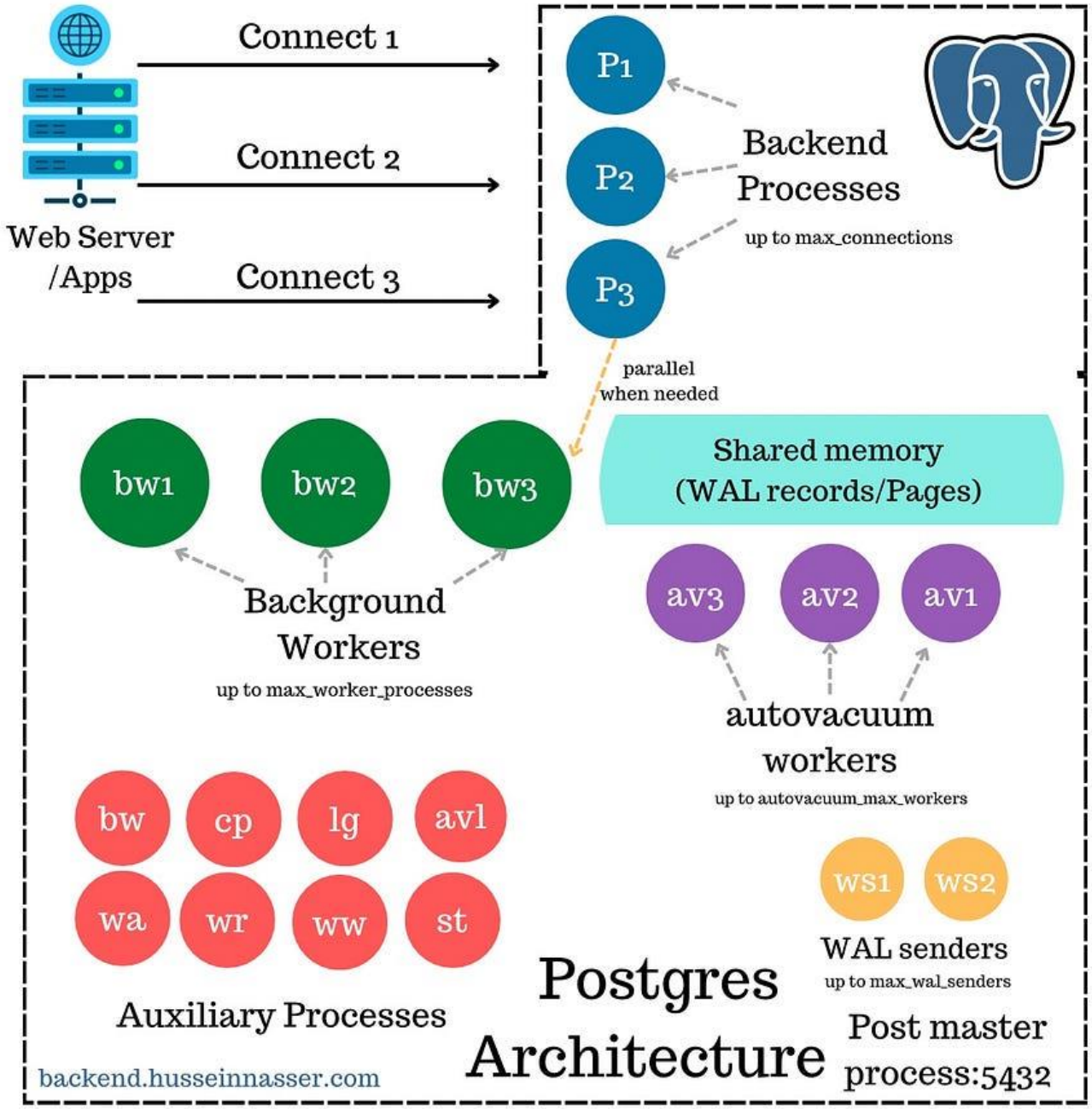

Рисунок 1.6 – Архітектура процесів PostgreSQL

PostgreSQL підтримує різноманітні типи даних, включаючи складні структури, такі як JSON та XML, що дозволяє зберігати та обробляти дані у різних форматах. Крім того, PostgreSQL надає розширені можливості для забезпечення цілісності даних, такі як транзакції, зовнішні ключі, перевірки обмежень та інші механізми. Це дозволяє забезпечити високу надійність та консистентність даних у системі [8].

TypeOrm використовується для взаємодії з базою даних PostgreSQL. Це ORM (Object-Relational Mapping) бібліотека для TypeScript та JavaScript, яка дозволяє використовувати об'єктно-орієнтований підхід для роботи з базою даних. TypeOrm забезпечує автоматичне відображення об'єктів на таблиці бази даних, що спрощує розробку та підтримку коду.

Завдяки TypeOrm можна легко виконувати CRUD (Create, Read, Update, Delete) операції над даними, використовувати транзакції, налаштовувати зв'язки між таблицями та виконувати складні запити. TypeOrm також підтримує використання міграцій, що дозволяє легко управляти змінами у схемі бази даних та забезпечувати їх відповідність під час розробки та розгортання додатка [9].

Docker був використаний для контейнеризації додатка. Контейнеризація дозволяє створювати ізольовані середовища для виконання додатків, що забезпечує їх портативність, масштабованість та полегшує процес розгортання. Docker дозволяє упаковувати додаток разом з усіма його залежностями у вигляді контейнерів, які можуть бути легко перенесені на інші машини або середовища.

Використання Docker забезпечує консистентність середовища розробки, тестування та розгортання, що дозволяє уникнути проблем, пов'язаних з різницею у налаштуваннях систем. Крім того, Docker дозволяє ефективно використовувати ресурси системи, забезпечуючи ізоляцію додатків та їх залежностей. Це дозволяє легко масштабувати додаток та забезпечувати його безперебійну роботу навіть при великих навантаженнях [10].

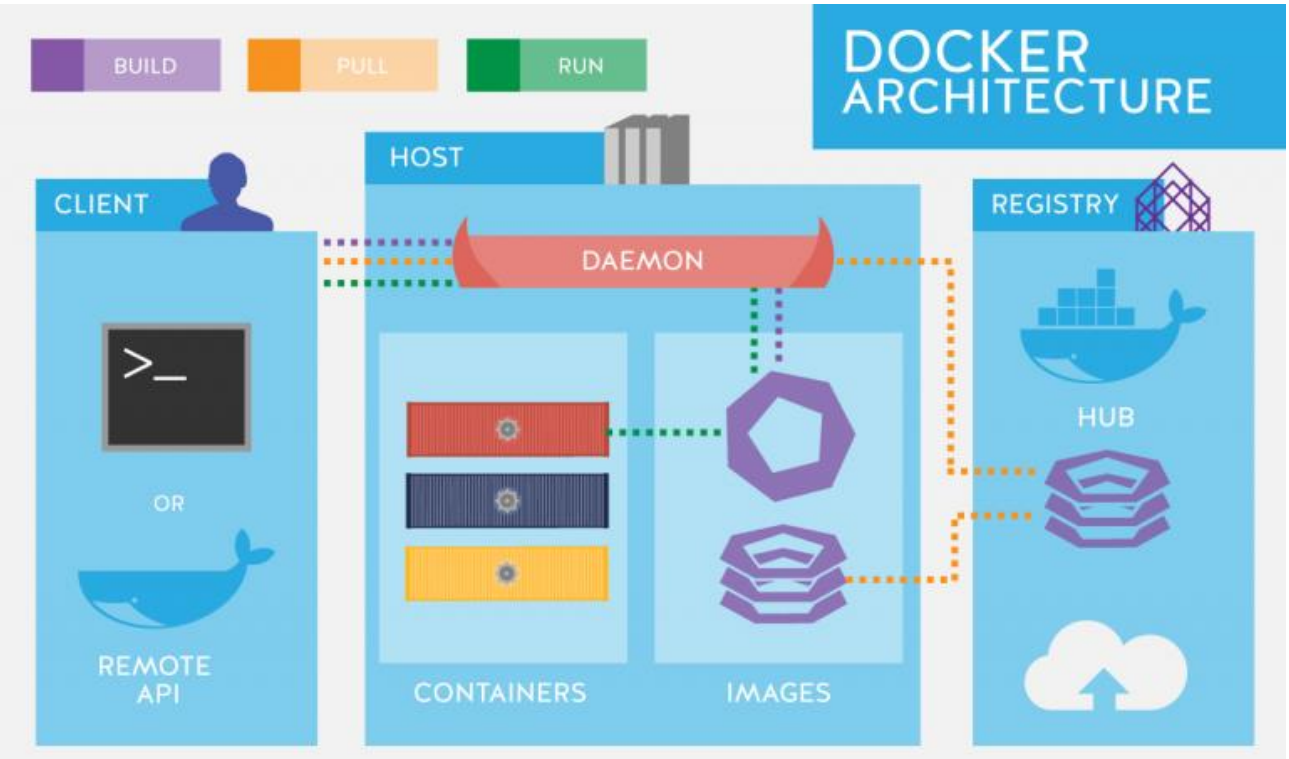

Рисунок 1.7 – Архітектура Docker

Для розробки клієнтської частини додатка був обраний фреймворк Next.js. Next.js забезпечує серверний рендеринг, що покращує продуктивність додатка та оптимізує його для пошукових систем. Серверний рендеринг дозволяє швидше завантажувати сторінки та покращує взаємодію користувача з додатком.

Next.js також забезпечує підтримку статичних сайтів, що дозволяє створювати швидкі та легкі веб-додатки. Він надає зручні інструменти для створення маршрутизації, обробки даних та інтеграції з різними API. Завдяки цьому можна створювати високопродуктивні та інтерактивні користувацькі інтерфейси [11].

В якості інструмента для тестування була обрана технологія Jest. Це потужна бібліотека для написання та виконання тестів, яка дозволяє забезпечити високу якість коду та зменшити кількість помилок під час розробки. Jest підтримує різні типи тестів, включаючи модульні, інтеграційні та кінцеві тести, що дозволяє перевіряти функціональність додатка на різних рівнях.

Jest забезпечує швидке виконання тестів завдяки паралельному запуску та кешуванню результатів. Крім того, він надає зручні інструменти для відладки та

аналізу тестів, що дозволяє швидко знаходити та виправляти помилки. Використання Jest дозволяє забезпечити надійність та стабільність додатка, зменшуючи ризики виникнення проблем під час його експлуатації [12].

Обрані технології були ретельно підібрані з урахуванням вимог проекту та їх можливостей забезпечити надійну, масштабовану та ефективну систему для централізованого зберігання та доступу до файлів. Кожна з них грає важливу роль у створенні цілісного рішення, що відповідає сучасним стандартам розробки програмного забезпечення.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ТА СТРУКТУРИ СИСТЕМИ

2.1 Проектування та опис сутностей системи

При проектуванні системи для централізованого зберігання та доступу до файлів одним з ключових етапів є побудова схеми бази даних. Схема бази даних визначає, як дані будуть організовані, зберігатися та взаємодіяти між собою. Важливо забезпечити ефективність, масштабованість та безпеку зберігання даних, враховуючи специфіку зберігання файлів різних форматів, об'ємів та вимог до доступу. У цьому розділі буде розглянуто основні принципи проектування схеми бази даних, обґрунтування вибору структури даних та взаємозв'язок між різними елементами системи, що дозволить створити надійну та ефективну платформу для управління файлами.

Нижче представлені рисунки відповідних сутностей, реалізованих за допомогою TypeORM [9].

Сутність UserEntity, яка відповідає таблиці users у базі даних, представляє користувача системи. Вона включає такі атрибути: id (унікальний ідентифікатор користувача), email (електронна адреса), password (хешований пароль для безпеки даних користувачів) та fullName (повне ім'я користувача) та зв'язок і з сутністю DriveEntity [9].

```
@Entity()  Show usages
export class UserEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  email: string;

  @Column( options: { select: false })
  password: string;

  @Column()
  fullName: string;

  @OneToOne( typeFunctionOrTarget: () => DriveEntity,  inverseSide: (drive : DriveEntity ) => drive.owner)
  drive?: DriveEntity;
}
```

Рисунок 2.1 – Сутність UserEntity, реалізована за допомогою TypeORM [9]

Сутність DriveEntity, що відповідає таблиці drives у базі даних, служить для представлення користувацького диска. Вона включає атрибути id (унікальний ідентифікатор диска у форматі UUID) та name (назва диска). Зв'язок із сутністю UserEntity визначає власника диска. Крім того, зв'язок із сутністю FolderEntity вказує на кореневу папку, яка організовує всю структуру файлів і підпапок на диску [9].

```
@Entity()  Show usages
export class DriveEntity {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  id: string;

  @Column()
  name: string;

  @OneToOne( typeFunctionOrTarget: () => UserEntity,  inverseSide: (user : UserEntity ) => user.drive)
  @JoinColumn()
  owner: UserEntity;

  @OneToOne( typeFunctionOrTarget: () => FolderEntity)
  @JoinColumn()
  rootFolder: FolderEntity;
}
```

Рисунок 2.2 – Сутність DriveEntity, реалізована за допомогою TypeORM [9]

Сутність FolderEntity, яка відповідає таблиці folders у базі даних, представляє каталог в системі. Вона включає такі атрибути: id (унікальний ідентифікатор папки), folderName (ім'я папки), originalFolderName (оригінальне ім'я папки), createdAt (дата створення), updatedAt (дата оновлення), deletedAt (дата видалення, якщо папка видалена), та isDriveRoot (позначка, чи є папка кореневою для диска).

Рекурсивний зв'язок parent визначає батьківський каталог, що дозволяє створювати ієрархію каталогів. Каталог може містити інші підкаталоги (зворотний зв'язок через children), а також файли (files). Зв'язок з користувачем (owner) вказує, кому належить папка. Також папка може мати нотатки (notes), оскільки реалізовано зв’язок із сутністю FolderNoteEntity [9].

```
@Entity()  Show usages
export class FolderEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column( options: { unique: true })
  folderName: string;

  @Column()
  originalFolderName: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @DeleteDateColumn()
  deletedAt?: Date;

  @Column()
  isDriveRoot: boolean;

  @ManyToOne( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.children,  options: {
    nullable: true,
  })
  @JoinColumn()
  parent?: FolderEntity;
```

Рисунок 2.3

```
  @ManyToOne( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.children,  options: {
    nullable: true,
  })
  @JoinColumn()
  parent?: FolderEntity;

  @OneToMany( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.parent)
  children: FolderEntity[];

  @OneToMany( typeFunctionOrTarget: () => FileEntity,  inverseSide: (file : FileEntity ) => file.folder)
  files: FileEntity[];

  @ManyToOne( typeFunctionOrTarget: () => UserEntity,  options: { nullable: false })
  @JoinColumn()
  owner: UserEntity;

  @OneToMany( typeFunctionOrTarget: () => FolderNoteEntity,  inverseSide: (folderNote : FolderNoteEntity ) => folderNote.folder)
  notes: FolderNoteEntity[];
}
```

Рисунок 2.4

Сутність FolderEntity, реалізована за допомогою TypeORM (рис. 2.3 – 2.3) [9].

Сутність FileEntity, яка відповідає таблиці files у базі даних, представляє файл у системі. Вона містить такі атрибути: id (унікальний ідентифікатор файлу), fileName (ім'я файлу), originalFileName (оригінальне ім'я файлу), size (розмір файлу), mimeType (тип файлу), createdAt (дата створення), updatedAt (дата оновлення) та deletedAt (дата видалення, якщо файл видалений).

Файл асоціюється з папкою через сутність FolderEntity, що вказує на його розташування. Він також пов'язаний з користувачем через сутність UserEntity, яка вказує на того, хто завантажив файл. Додатково, файл може мати кілька нотаток (notes), що дозволяє зберігати додаткову інформацію, пов'язану з файлом [9].

```
@Entity()  Show usages
export class FileEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  fileName: string;

  @Column()
  originalFileName: string;

  @Column()
  size: number;

  @Column()
  mimeType: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @DeleteDateColumn()
  deletedAt?: Date;

  @ManyToOne( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.files,  options: {
    nullable: false,
  })
```

Рисунок 2.5

```
  @ManyToOne( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.files,  options: {
    nullable: false,
  })
  folder: FolderEntity;

  @ManyToOne( typeFunctionOrTarget: () => UserEntity,  options: {
    nullable: false,
  })
  user: UserEntity;

  @OneToMany( typeFunctionOrTarget: () => FileNoteEntity,  inverseSide: (fileNote : FileNoteEntity ) => fileNote.file)
  notes: FileNoteEntity[];
}
```

Рисунок 2.6

Сутність FileEntity, реалізована за допомогою TypeORM (рис. 2.5 – 2.6) [9]

Сутність FileDownloadLinkEntity відповідає таблиці file_download_links у базі даних, представляє посилання на завантаження файлу. Вона включає такі атрибути: id (унікальний ідентифікатор посилання), token (унікальний токен посилання), createdAt (дата створення посилання), updatedAt (дата останнього оновлення посилання) та expiresAt (дата закінчення терміну дії посилання) [9].

```
@Entity()  Show usages
export class FileDownloadLinkEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  token: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @Column()
  expiresAt: Date;

  @ManyToOne( typeFunctionOrTarget: () => FileEntity,  options: {
    nullable: false,
  })
  file: FileEntity;
}
```

Рисунок 2.7 – Сутність FileDownloadLinkEntity, реалізована за допомогою TypeORM [9]

Сутність Note, яка використовується як вбудована сутність для FileNoteEntity та FolderNoteEntity, представляє загальні атрибути нотатки. Вона включає такі атрибути: title (заголовок нотатки), content (вміст нотатки), createdAt (дата створення нотатки) та updatedAt (дата останнього оновлення нотатки).

Ця вбудована сутність дозволяє уникнути дублювання коду та забезпечує повторне використання логіки для нотаток, що використовуються у різних контекстах, таких як файли та папки [9].

```
export class Note {  Show usages
  @Column()
  title: string;

  @Column()
  content: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}
```

Рисунок 2.8 – Вбудована сутність Note, реалізована за допомогою TypeORM [9]

Сутність FileNoteEntity, яка відповідає таблиці file_notes у базі даних, представляє нотатку для файлу. Вона включає такі атрибути: id (унікальний ідентифікатор нотатки) та note (вбудована сутність Note, що містить заголовок, вміст, дату створення та оновлення нотатки).

Нотатка пов'язана з файлом через сутність FileEntity, вказуючи на файл, до якого вона належить [9].

```
@Entity()  Show usages
export class FileNoteEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column( type: () => Note)
  note: Note;

  @ManyToOne( typeFunctionOrTarget: () => FileEntity,  inverseSide: (file : FileEntity ) => file.notes,  options: {
    nullable: false,
  })
  file: FileEntity;
}
```

Рисунок 2.9 – Сутність FileNoteEntity, реалізована за допомогою TypeORM [9]

Сутність FolderNoteEntity відповідає таблиці folder_notes у базі даних та представляє нотатку для папки. Вона включає такі атрибути: id (унікальний ідентифікатор нотатки) та note (вбудована сутність Note, що містить заголовок, вміст, дату створення та оновлення нотатки).

Нотатка має зв'язок із сутністю FolderEntity, що вказує на папку, до якої належить нотатка. Це дозволяє кожній папці мати кілька нотаток, що забезпечує зберігання додаткової інформації, пов'язаної з папкою [9].

```
@Entity()  Show usages
export class FolderNoteEntity {
  @PrimaryGeneratedColumn()
  id: number;

  @Column( type: () => Note)
  note: Note;

  @ManyToOne( typeFunctionOrTarget: () => FolderEntity,  inverseSide: (folder : FolderEntity ) => folder.notes,  options: {
    nullable: false,
  })
  folder: FolderEntity;
}
```

Рисунок 2.10 – Сутність FolderNoteEntity, реалізована за допомогою TypeORM [9]

На основі проведеного аналізу вимог системи та її компонентів було визначено класи додатку, які представлені на діаграмі класів [19]. Діаграма наочно демонструє сутності системи, їхні атрибути та взаємозв’язки. Кожен із цих класів

має свої специфічні атрибути та методи, що забезпечують реалізацію функціональності системи.

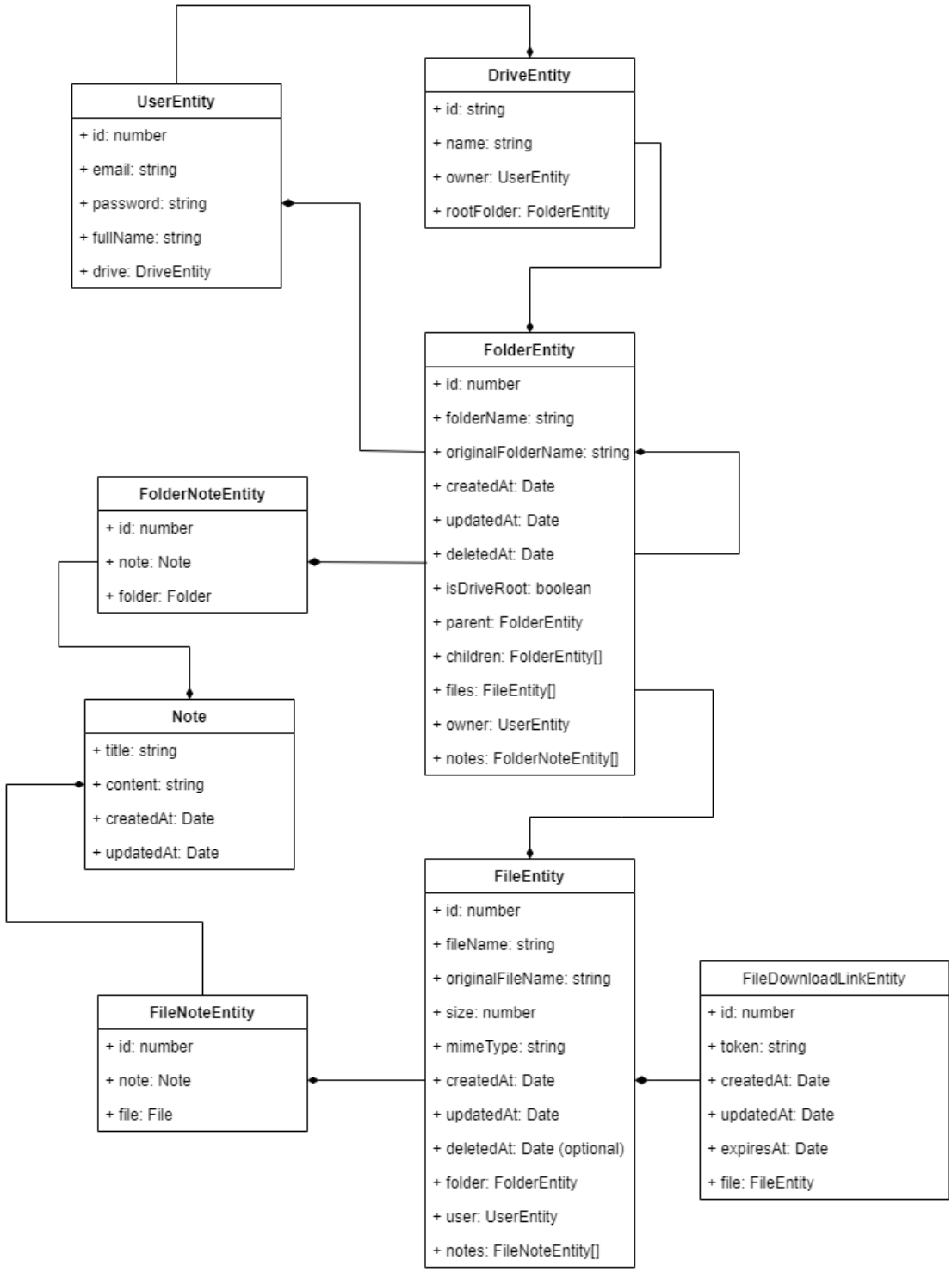

Рисунок 2.11 – UML діаграма класів системи

Основні класи системи включають UserEntity, DriveEntity, FolderEntity та FileEntity. Ці класи формують фундамент додатку, де UserEntity відповідає за

зберігання даних користувачів, DriveEntity описує диски користувачів, FolderEntity організовує структуру папок, а FileEntity управляє файлами. Взаємозв’язки між класами вказують на композиційні відносини, що забезпечують цілісність системи. Наприклад, зв’язок між UserEntity та DriveEntity підкреслює, що диск не може існувати без користувача, тоді як зв’язок між FolderEntity та FileEntity підкреслює залежність файлів від папок.

2.2 Визначення зв’язків між сутностями системи

Описані сутності представляють основні елементи нашої системи для централізованого зберігання та доступу до файлів. Кожна сутність має свої атрибути та зв'язки з іншими сутностями, що забезпечує ефективне управління даними. Для повного розуміння структури бази даних і взаємодії між різними елементами, важливо розглянути зв'язки між цими сутностями. Нижче наведено детальний опис взаємозв'язків між таблицями, які забезпечують цілісність даних та зручність доступу до інформації в системі.

У реляційних базах даних зв’язки між таблицями грають ключову роль у забезпеченні цілісності даних та їх структурованого зберігання. Зв’язки дозволяють встановити взаємозалежність між різними сутностями, що, в свою чергу, допомагає ефективно організовувати та управляти даними. Основними типами зв’язків між таблицями в реляційних базах даних є: один до одного, один до багатьох та багато до багатьох [8].

Зв’язок один до одного (One-to-One) означає, що кожному рядку в одній таблиці відповідає один і тільки один рядок в іншій таблиці. Такий тип зв’язку використовується, коли необхідно розділити дані на кілька таблиць для оптимізації або з міркувань безпеки. В цьому випадку ключ, який є посиланням, є унікальним, тобто для кожного рядка в одній таблиці існує унікальний рядок в іншій таблиці, і жоден з рядків не може мати більше одного відповідного рядка в іншій таблиці [13].

Зв’язок один до багатьох (One-to-Many) є найпоширенішим типом зв’язку в реляційних базах даних. Він означає, що кожному рядку в одній таблиці може відповідати багато рядків в іншій таблиці, але кожен рядок у другій таблиці пов'язаний лише з одним рядком у першій таблиці. В цьому випадку ключ, який є посиланням, є неунікальним, тобто для кожного рядка в першій таблиці може існувати кілька відповідних рядків у другій таблиці [13].

Зв’язок багато до багатьох (Many-to-Many) використовується, коли рядки в одній таблиці можуть бути пов'язані з багатьма рядками в іншій таблиці і навпаки. Для реалізації такого типу зв’язку зазвичай використовується проміжна таблиця, яка містить зовнішні ключі до обох зв’язаних таблиць. В цьому випадку ключі, які є посиланнями, є неунікальними в проміжній таблиці, що дозволяє встановлювати кілька зв’язків між рядками обох таблиць [13].

Основними сутностями в базі даних є користувачі (users), файли (files), папки (folders), нотатки до файлів (file_notes), нотатки до каталогів (folder_notes), посилання на завантаження файлів (file_download_links) та диски (drives). Кожна з цих сутностей має певні атрибути та взаємозв'язки, які дозволяють зберігати та управляти інформацією ефективно.

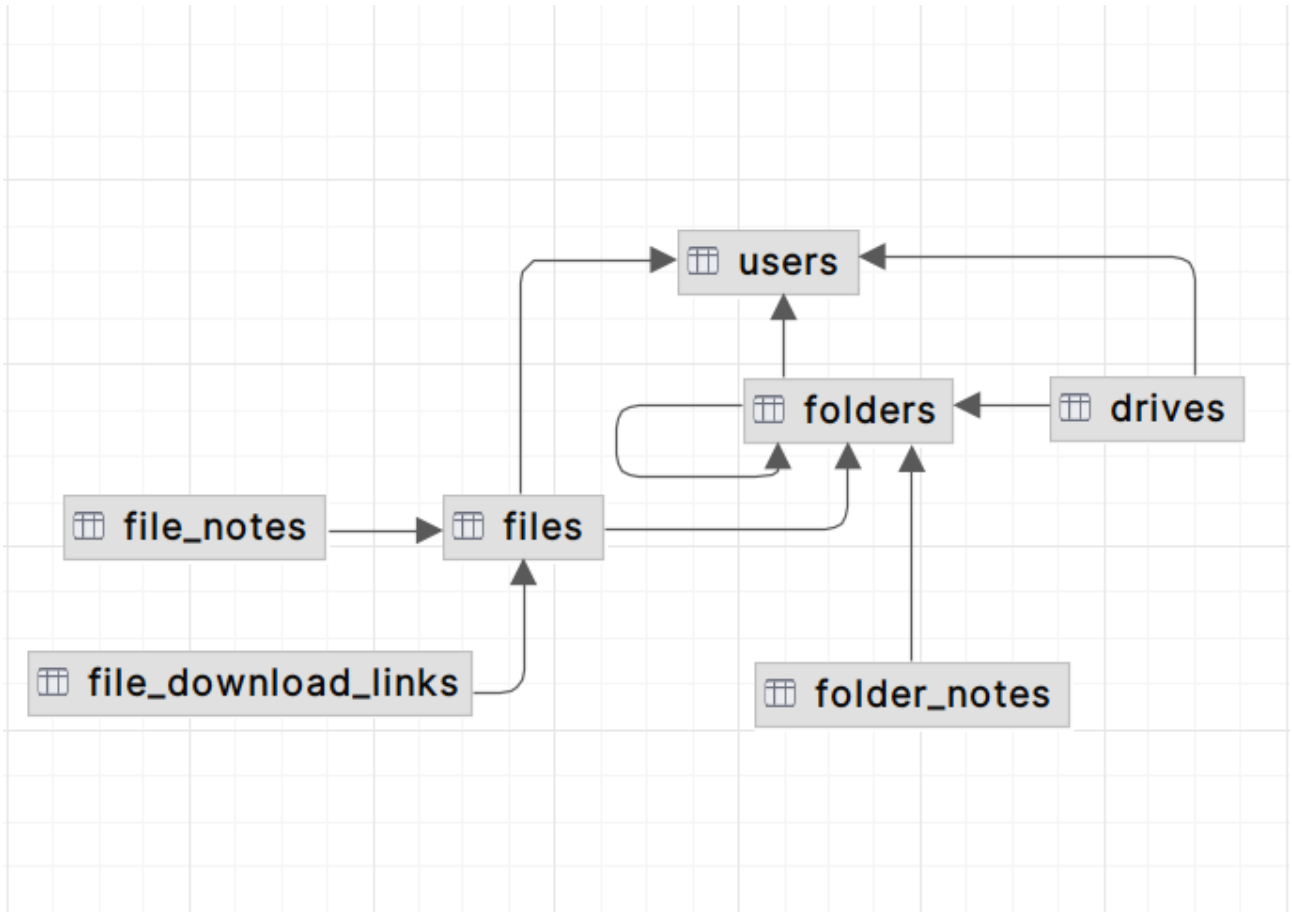

Рисунок 2.12 – Концептуальна схема бази даних

Користувач і диск є одними із ключових сутностей системи. У даній системі існує зв'язок один до одного між користувачами (users) та дисками (drives). Це означає, що кожен користувач може мати лише один диск, і кожен диск належить лише одному користувачу. Така структура дозволяє чітко визначити власника диска та забезпечити унікальність зберігання даних для кожного користувача.

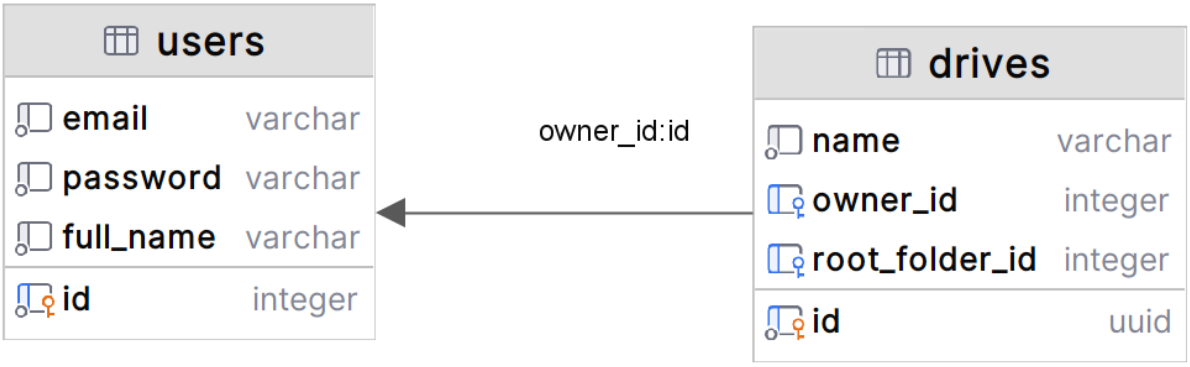

Рисунок 2.13 – Зв’язок між користувачами та дисками (users - drives)

Зв'язок між каталогами (folders) та дисками (drives) – один до одного. Це означає, що кожен диск має одну кореневу папку, яка відповідає за організацію всіх файлів і підкаталогів на цьому диску. Така структура забезпечує чітку ієрархію зберігання даних. Також кожен каталог, окрім кореневого каталога диска, має посилання на батьківський каталог.

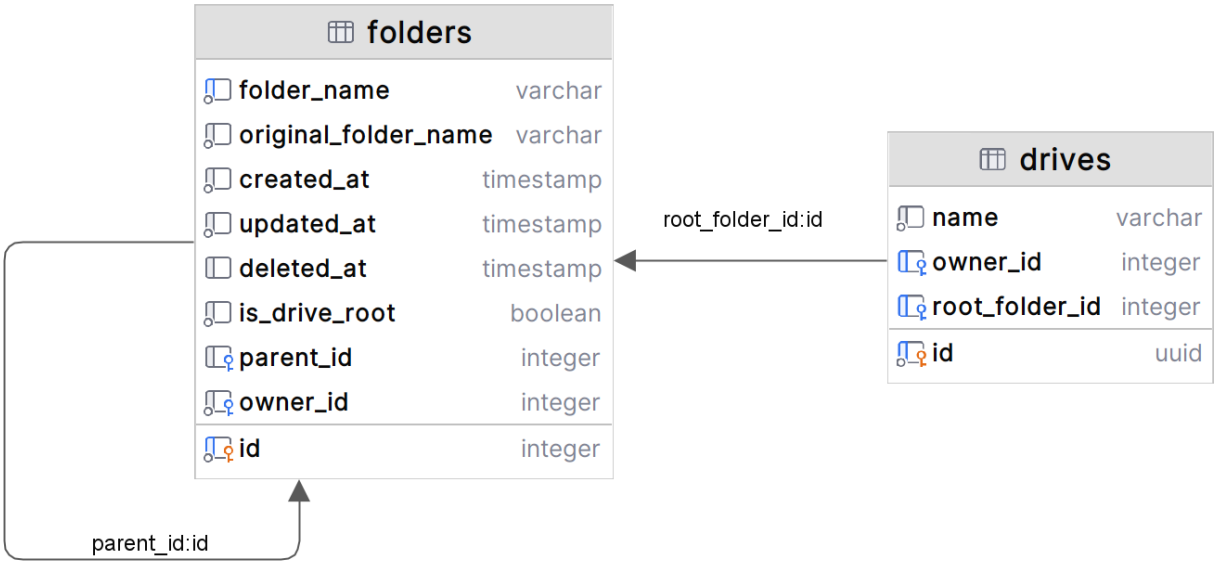

Рисунок 2.14 – Зв’язок між каталогами та дисками (folders - drives)

Зв'язок між каталогами (folders) та файлами (files) у нашій системі є зв'язком один до багатьох. Це означає, що кожен каталог може містити кілька файлів, але кожен файл може бути розміщений лише в одній папці. Така структура дозволяє організувати файли в логічні групи, забезпечуючи зручність у їхньому пошуку та управлінні.

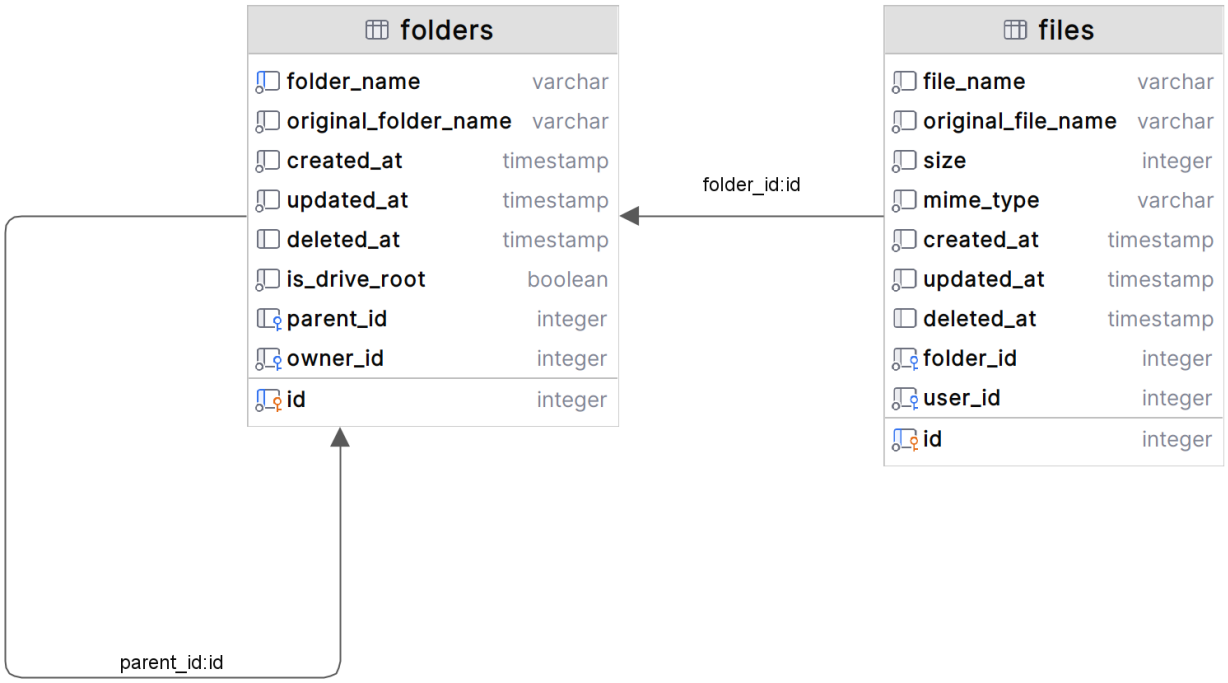

Рисунок 2.15 – Зв’язок між каталогами та файлами (folders - files)

Зв'язок між файлами (files) та нотатками для файлів (file_notes) є зв'язком один до багатьох, як і зв'язок між файлами (files) та посиланнями для завантаження файлів (file_download_links). Дані зв’язки означають, що кожен файл може мати кілька нотаток та кілька посилань для завантаження, але кожна нотатка і кожне посилання можуть бути пов'язані лише з одним файлом.

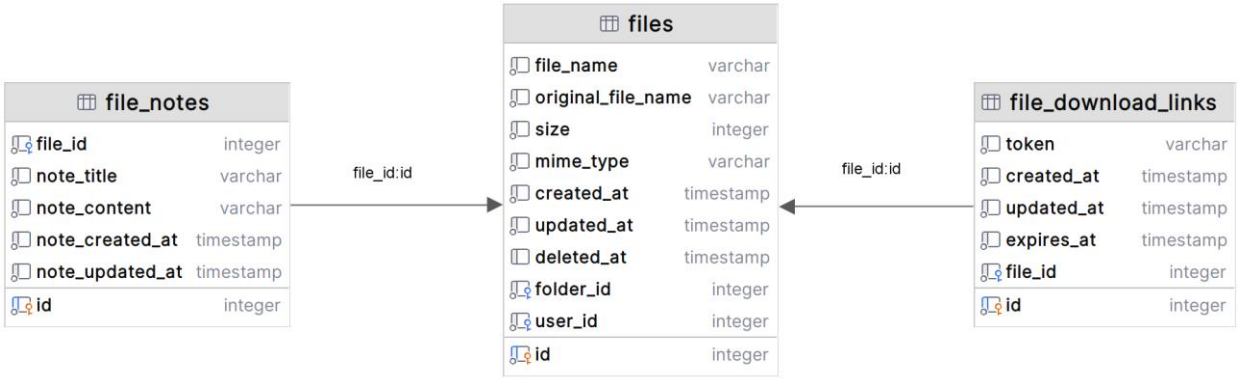

Рисунок 2.16 – Зв’язок між файлами та нотатками для файлів (files - file_notes) і файлами та посиланнями для завантаження файлів (file - file_download_links)

Зв'язок між папками (folders) та нотатками для папок (folder_notes) є зв'язком один до багатьох. Даний зв’язок означає, що кожна папка може мати кілька нотаток, але кожна нотатка може бути пов'язана лише з однією папкою. Така структура дозволяє зберігати додаткову інформацію, пов'язану з кожною папкою, забезпечуючи зручність в управлінні даними.

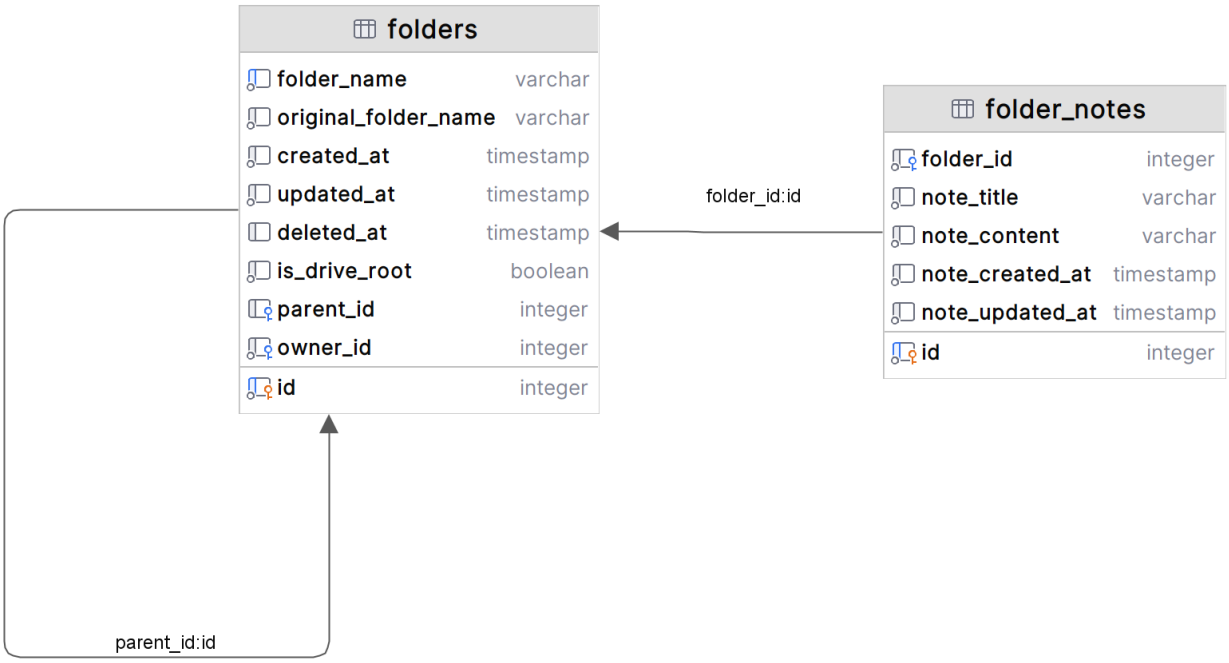

Рисунок 2.17 – Зв’язок між папками та нотатками для папок (folders - folder_notes)

Побудова схеми бази даних є важливим етапом у розробці системи централізованого зберігання та доступу до файлів. ER-діаграма, представлена у цьому розділі, відображає взаємозв'язки між основними сутностями системи, такими як користувачі, файли, папки, нотатки до файлів, нотатки до папок, посилання на завантаження файлів та диски. Ця діаграма демонструє, як дані організовані та як вони взаємодіють між собою.

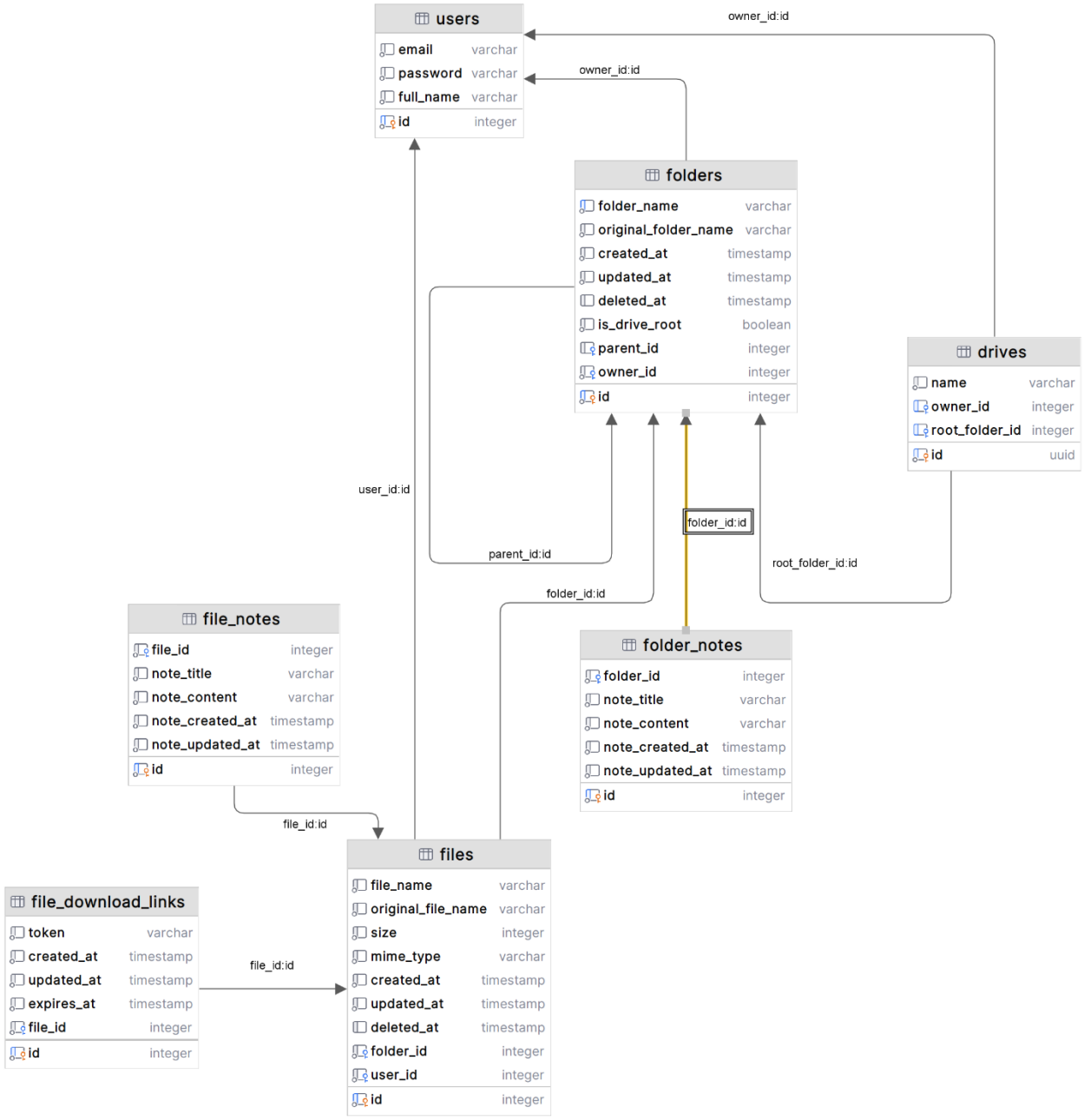

Рисунок 2.18 – Entity-Relationship Diagram [14]

ER-діаграма відображає надійну структуру зберігання даних, яка дозволяє системі ефективно керувати файлами та каталогами для кожного користувача. Така модель бази даних сприяє організованому зберіганню даних, полегшує доступ до них і забезпечує можливість подальшого розширення функціональності системи.

3 КОНСТРУЮВАННЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Реалізація функціоналу системи

Реалізація функціоналу системи для централізованого зберігання та доступу до файлів базується на використанні сучасних технологій, таких як Node.js, NestJS, TypeORM та Archiver [4, 6, 7, 9]. Використання Node.js дозволило створити високопродуктивну серверну частину, що забезпечує швидку обробку запитів користувачів. NestJS, завдяки своїй модульній архітектурі, сприяє структурованому та організованому підходу до розробки, що полегшує підтримку і масштабування системи. TypeORM забезпечує зручне управління базою даних, дозволяючи працювати з PostgreSQL [8] на високому рівні абстракції. За допомогою бібліотеки Archiver реалізовано функціонал створення архівів, що забезпечує зручну організацію та зберігання великих обсягів даних. У цьому розділі буде детально описано процес розробки ключових функцій системи, включаючи реєстрацію та аутентифікацію користувачів, управління файлами та папками, а також створення унікальних посилань для завантаження файлів.

Реєстрація є важливим процесом у використанні системи, оскільки під час цього етапу створюється обліковий запис користувача та ініціалізується диск для зберігання файлів. Цей процес забезпечує створення унікального ідентифікатора для кожного користувача, зберігання його облікових даних та створення простору для організації та зберігання файлів. Така структура дозволяє ефективно управляти даними користувача і забезпечує безпечний та організований доступ до файлів. Нижче наведені скріни коду, що демонструють реалізацію цього процесу.

```
@Post( path: 'register')  no usages
@ApiResponse( options: {
  status: 201,
  description: 'User registered successfully',
  type: AuthResDto,
})
async register(@Body() registerDto: RegisterUserDto): Promise<AuthResDto> {
  return this.authService.registerUser(registerDto);
}
```

Рисунок 3.1

```
async registerUser(registerDto: RegisterUserDto): Promise<IJwtTokens> {  Show usages
  const { email : string , password : string , fullName : string , driveName : string  } = registerDto;

  const isUsernameTaken : boolean  = await this.userService.isEmailTaken(email);

  if (isUsernameTaken) throw new ConflictException( objectOrError: 'USERNAME_ALREADY_TAKEN');

  const hashedPassword : string  = await this.hashUserPassword(password);

  const user : UserEntity  = await this.userService.createUser( {email, password, fullName}: {
    email,
    password: hashedPassword,
    fullName,
  });

  await this.userService.createUserDrive(user, driveName);

  return this.generateTokens( payload: {
    id: user.id,
    email: user.email,
  });
}
```

Рисунок 3.2

```
async createUserDrive(  Show usages
  user: UserEntity,
  driveName: string,
): Promise<DriveEntity> {
  const drive : DriveEntity  = this.driveRepository.create({
    name: driveName,
    owner: user,
  });

  await this.driveRepository.save(drive);

  drive.rootFolder = await this.folderService.createRootFolder(drive);

  await this.driveRepository.save(drive);

  return drive;
}
```

Рисунок 3.3

Лістинги коду функціоналу реєстрації користувача у системі та створення диска для зберігання файлів (рис. 3.1 – 3.3)

Процес входу користувача в систему є ключовим для забезпечення безпеки та надання доступу до особистих даних користувача. Під час цього процесу користувач отримує пару токенів: токен доступу та токен оновлення. Ці токени містять зашифровану інформацію, необхідну для аутентифікації кожного запиту користувача, що дозволяє забезпечити безпечний доступ до системи без необхідності повторної авторизації. Токен доступу використовується для короткочасної аутентифікації, тоді як токен оновлення дозволяє отримати новий токен доступу без повторного входу в систему. Нижче наведені скріни коду, що демонструють реалізацію цього процесу.

```
@Post( path: 'login')  Show usages
@ApiOkResponse( options: {
  description: 'User logged in successfully',
  type: AuthResDto,
})
async login(@Body() loginDto: LoginDto): Promise<AuthResDto> {
  return await this.authService.login(loginDto);
}
```

Рисунок 3.4

```
async login({ email, password }: LoginDto): Promise<IJwtTokens> {  Show usages
  const user : UserEntity  = await this.userService.findUserWithPasswordByEmail(email);

  if (!user) throw new NotFoundException( objectOrError: 'USER_NOT_FOUND');

  const isPasswordValid = await bcrypt.compare(password, user.password);

  if (!isPasswordValid) throw new BadRequestException( objectOrError: 'INVALID_PASSWORD');

  return this.generateTokens( payload: {
    id: user.id,
    email: user.email,
  });
}
```

Рисунок 3.5

```
async findUserWithPasswordByEmail(email: string): Promise<UserEntity> {  Show usages
  return this.userRepository.findOne( options: {
    where: { email },
    select: ['id', 'email', 'password'],
  });
}
```

Рисунок 3.6

Лістинги коду функціоналу входу користувача в систему (рис. 3.4 – 3.6)

Процес завантаження файлів на диск відіграє ключову роль у функціональності системи, дозволяючи користувачам організовувати та зберігати свої дані. Користувач може вказати папку, в яку він бажає завантажити файли, та вибрати один або кілька файлів для завантаження. Це забезпечує гнучкість та

зручність у використанні системи, дозволяючи користувачам структурувати свої дані відповідно до власних потреб. Під час завантаження файлів система автоматично зберігає їх у вказану папку, забезпечуючи їх надійне зберігання та доступність. Нижче наведені скріни коду, що демонструють реалізацію цього процесу.

```
@Post( path: 'upload')  Show usages
@ApiConsumes( mimeTypes: 'multipart/form-data')
@UseInterceptors(FilesInterceptor( fieldName: 'files'))
async uploadFile(
  @UploadedFiles()
  files: Array<MulterFile>,
  @CurrentUser() user: IUser,
  @Body() uploadFileDto: UploadFileDto,
): Promise<IApiResponse> {
  return await this.fileService.saveFiles(files, user, uploadFileDto);
}
```

Рисунок 3.7

```
export class FileService {

  async saveFiles(  Show usages
    files: Array<MulterFile>,
    { id: userId }: IUser,
    { folderId }: UploadFileDto,
  ): Promise<IApiResponse> {
    const queryRunner : QueryRunner  = this.entityManager.connection.createQueryRunner();
    await queryRunner.startTransaction();

    const { manager : EntityManager  } = queryRunner;
    try {
      const folder : FolderEntity  = await manager.findOne(FolderEntity, options: {
        where: {
          id: folderId,
          owner: { id: userId },
        },
      });

      if (!folder) {
        throw new NotFoundException( objectOrError: 'FOLDER_NOT_FOUND_OR_NOT_OWNED_BY_USER');
      }

      await forEach(files, callback: async (file : Express.Multer.File ) : Promise<void>  => {
        const ext : string  = path.extname(file.originalname);

        const uniqueFileName : string  = `${uuid()}${ext}`;

        const fileEntity : FileEntity  = manager.create(FileEntity, plainObject: {
          fileName: uniqueFileName,
```

Рисунок 3.8

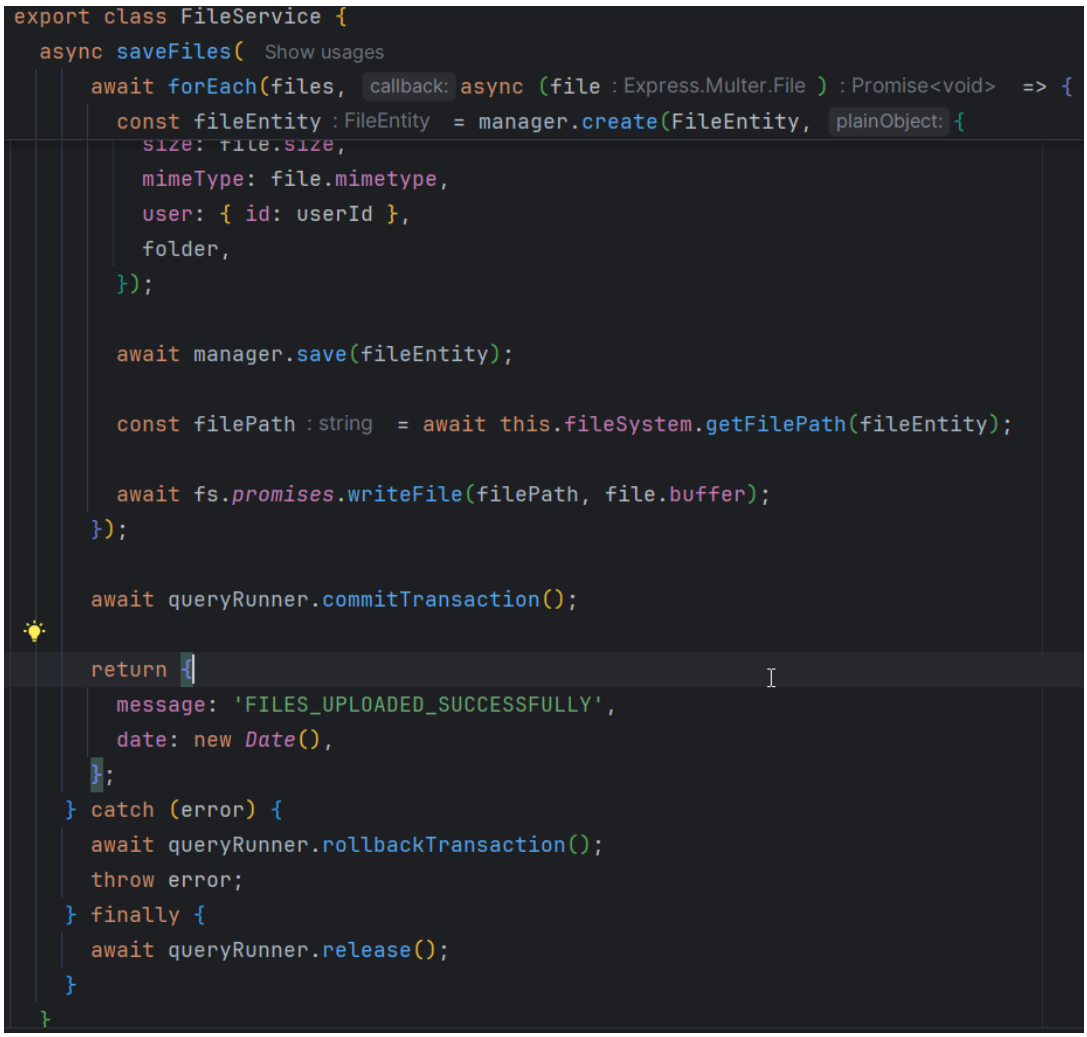

Рисунок 3.9

Лістинги коду функціоналу завантаження файлів на диск (рис. 3.7 – 3.9)

Функціонал завантаження архіву каталога з усіма його файлами та підкаталогами є важливим інструментом для забезпечення зручного доступу до даних. Користувач може вибрати каталог, який він хоче завантажити, і система автоматично створює архів, що включає всі файли та підкаталоги цього каталога. Це дозволяє користувачам легко зберігати резервні копії своїх даних або переносити їх на інші пристрої. Під час створення архіву система зберігає структуру папок, що забезпечує збереження організації даних користувача. Нижче наведені скріни коду, що демонструють реалізацію цього процесу.

```
export class FolderController {
  @Get( path: 'download/:id') Show usages
  async downloadArchive(
    @Param( property: 'id', ParseIntPipe) id: number,
    @CurrentUser() user: IUser,
    // eslint-disable-next-line @typescript-eslint/no-unused-vars
    @Res( options: { passthrough: true }) res: Response,
  ): Promise<StreamableFile> {
    const { stream : ReadStream , contentType : string , fileName : string , size : number  } =
      await this.folderService.downloadArchive(id, user);

    return new StreamableFile(stream, options: {
      type: contentType,
      disposition: `attachment; filename="${fileName}"`,
      length: size,
    });
  }
```

Рисунок 3.10

```
export class FolderService {
  async downloadArchive(id: number, user: IUser): Promise<StreamableFileType> {  Show usages
    try {
      const folder : FolderEntity  = await this.entityManager.findOneBy(FolderEntity, where: {
        id,
        owner: user,
      });

      if (!folder) {
        throw new NotFoundException( objectOrError: 'FOLDER_NOT_FOUND_OR_NOT_OWNED_BY_USER');
      }

      const archivePath : string  = await this.fileSystem.createArchive(folder);

      const { size : number  } = await fs.promises.stat(archivePath);

      return {
        stream: fs.createReadStream(archivePath),
        fileName: path.basename(archivePath),
        contentType: 'application/zip',
        size: size / FILE_CHUNK_SIZE,
      };
    } catch (error) {
      console.log('error', error);
      throw error;
    }
```

Рисунок 3.11

Лістинги коду функціоналу завантаження архіва каталогу на комп’ютер користувача (рис. 3.10 – 3.11)

Функціонал генерації унікального посилання для завантаження файлів надає користувачам можливість легко ділитися своїми файлами з іншими. Користувач вказує час, після якого посилання стане недійсним, а також вибирає файл, для якого потрібно створити посилання. У результаті система генерує унікальне посилання, яке можна передати третім особам для завантаження файлу, а також дату, до якої посилання буде дійсним. Це забезпечує безпеку та контроль доступу до файлів, дозволяючи обмежити час дії посилання. Нижче наведені скріни коду, що демонструють реалізацію цього процесу.

```
@Post( path: 'download-link/:id')  Show usages
createDownloadLink(
  @Param( property: 'id', ParseIntPipe) id: number,
  @Body() dto?: CreateDownloadLinkDto,
): Promise<ResCreateDownloadLinkDto> {
  return this.fileService.createDownloadLink(id, dto?.expirationTime);
}
```

Рисунок 3.12

```
export class FileService {
  async createDownloadLink(  Show usages
    id: number,
    expirationTime?: number,
  ): Promise<ResCreateDownloadLinkDto> {
    const file : FileEntity  = await this.fileRepository.findOne( options: {
      where: { id },
      relations: ['user'],
    });

    if (!file) {
      throw new NotFoundException( objectOrError: 'FILE_NOT_FOUND');
    }

    const token : string  = uuid();

    const expiresAt : Date  = new Date(
      value: Date.now() + (expirationTime ?? FILE_DOWNLOAD_LINK_DEFAULT_EXPIRATION),
    );

    const fileDownloadLinkObject : FileDownloadLinkEntity  = await this.fileDownloadLinkRepository.save(
      this.fileDownloadLinkRepository.create({
        file,
        token,
        expiresAt,
      }),
    );

    const appUrl : string  = this.configService.get<string>('APP_URL');
```

Рисунок 3.13

```
    const fileDownloadLinkObject : FileDownloadLinkEntity  = await this.fileDownloadLinkRepository.save(
      this.fileDownloadLinkRepository.create({
        file,
        token,
        expiresAt,
      }),
    );

    const appUrl : string  = this.configService.get<string>('APP_URL');

    return {
      downloadLink: `${appUrl}/file/download-link/${token}`,
      expirationDate: fileDownloadLinkObject.expiresAt,
    };
  }
```

Рисунок 3.14

Лістинги коду функціоналу генерації унікального посилання для завантаження файлів третіми особами (3.12 – 3.14)

3.2 Тестування системи та оцінка якості реалізації

Тестування є невід'ємною частиною розробки програмного забезпечення [15], яка забезпечує високу якість продукту та його відповідність вимогам. Процес тестування дозволяє виявити та виправити помилки, покращити продуктивність та забезпечити надійність системи перед її впровадженням. У випадку системи для централізованого зберігання та доступу до файлів, тестування є критично важливим, оскільки від цього залежить безпека, цілісність та доступність даних користувачів.

Основною метою тестування є виявлення дефектів у системі та їх усунення до того, як продукт потрапить до кінцевого користувача. Це дозволяє зменшити ризики, пов'язані з неправильним функціонуванням системи, та уникнути потенційних фінансових втрат та репутаційних ризиків. Тестування також допомагає переконатися, що система працює відповідно до визначених специфікацій і виконує свої функції ефективно та безпечно [15].

Тестування включає декілька етапів, кожен з яких має свої особливості та важливість [15]:

- Аналіз вимог: Першим кроком є детальний аналіз вимог до системи, щоб зрозуміти, які функціональні та нефункціональні вимоги необхідно перевірити. Це дозволяє розробити ефективні тестові сценарії та визначити критерії успіху.
- Розробка тестових сценаріїв: На основі аналізу вимог створюються тестові сценарії, які охоплюють всі аспекти роботи системи. Тестові сценарії повинні бути ретельно сплановані та описані, щоб забезпечити повне покриття функціональності системи.
- Виконання тестів: Після розробки тестових сценаріїв починається їх виконання. Це може включати як автоматизоване тестування, так і ручне тестування. Під час виконання тестів важливо фіксувати всі виявлені помилки та відхилення від очікуваної поведінки системи.

- Аналіз результатів тестування: Після виконання тестів результати аналізуються для визначення причин помилок та шляхів їх усунення. Це також допомагає визначити, які частини системи потребують додаткового тестування або оптимізації.
- Регресійне тестування: Регресійне тестування проводиться для перевірки, що зміни в коді не спричинили появу нових помилок у вже перевірених частинах системи.

В контексті розробки системи для централізованого зберігання та доступу до файлів, одним з ключових етапів тестування є Unit тести. Unit тестування спрямоване на перевірку окремих модулів або компонентів системи на правильність їх роботи. Це дозволяє виявити помилки на ранніх стадіях розробки та забезпечити правильну роботу кожного компонента в ізоляції. Unit тести допомагають переконатися, що кожна функція виконує свої завдання відповідно до специфікацій і може взаємодіяти з іншими компонентами системи.

Проведення ретельного тестування є невід'ємною частиною успішної розробки програмного забезпечення. Unit тести є важливим етапом, який забезпечує надійність, безпеку та ефективність системи. Використовуючи ці методи тестування, можна значно зменшити ризики та забезпечити високоякісний продукт для кінцевих користувачів. Нижче наведений рисунок, що демонструє роботу Unit тестів у системі.

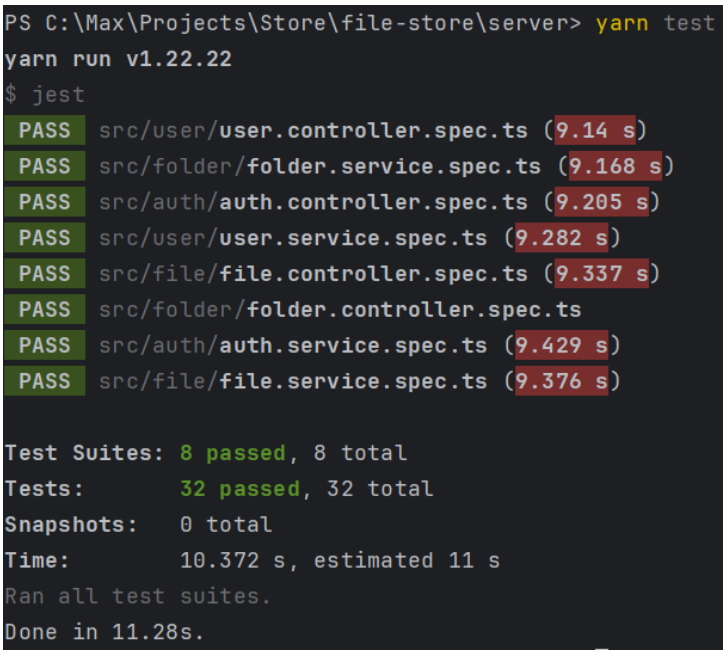

Рисунок 3.15 – Запуск Unit тестів для виявлення помилок у коді

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

4.1 Перша допомога при отруєнні СДОР.

Безпека життєдіяльності охоплює всі заходи, спрямовані на запобігання та мінімізацію ризиків для здоров'я та безпеки людей. Це включає в себе заходи з пожежної безпеки, безпеки на дорозі, безпеки на робочому місці, безпеки в домашньому середовищі, а також заходи з медичної та гігієнічної безпеки. Основна мета безпеки життєдіяльності полягає в тому, щоб забезпечити людей усіма необхідними знаннями, навичками та умовами для того, щоб уникнути травм і захворювань, а також для того, щоб допомогти людям у випадку надзвичайних ситуацій [22].

Охорона праці - це система заходів, спрямованих на забезпечення безпеки та здоров'я працівників під час виконання їх трудових обов'язків. Основна мета охорони праці полягає в уникненні травматизму, професійних захворювань та забезпеченні безпечних умов праці для всіх працівників [22].

Система охорони праці включає в себе ряд заходів, таких як проведення оцінки ризиків, розробка та впровадження безпечних технологій та методів роботи, навчання працівників правилам безпеки, постійний контроль за дотриманням норм безпеки, а також надання необхідного захисту та засобів індивідуального захисту [22].

Охорона праці є важливою складовою частиною будь-якого підприємства або організації, оскільки вона сприяє покращенню якості життя працівників та зменшенню втрат виробництва, пов'язаних з травмами та захворюваннями [22].

Сильнодіючі отруйні речовини (СДОР) – це хімічні речовини, які призначаються для застосування в народногосподарських цілях і володіють токсичністю, здатною викликати масові ураження людей, тварин і рослин. Серед них найбільш часто зустрічаються хлор, аміак, сірководень, синильна кислота, сірчистий ангідрид, бромистий водень [21].

При потраплянні отруйних речовин на шкіру необхідно терміново змити уражену ділянку тіла водою з милом. Досить часто ефективним засобом для виведення отрути є промивання шлунка, за умови, що токсична речовина потрапила в організм через стравохід (отруєння харчові, медикаментозні, грибами, алкоголем тощо) [20]. Для цього потерпілому потрібно дати випити відразу кілька склянок води з питною содою (1 чайна ложка на 1 склянку води) чи зі слабким розчином марганцевокислого калію (блідо-рожевий колір) і, натиснувши на корінь язика, викликати блювання. Промивання шлунка проводять 3 – 4 рази [21]. Після цього дають 5 – 8 таблеток активованого вугілля, яке має хороші адсорбційні властивості. Для очищення кишечника використовують сольове (20г гіркої англійської солі на 0,5 склянки води) або будь-яке інше проносне (наприклад, гуталакс). Потім потрібно дати випити потерпілому міцний чай чи каву. Давати молоко не рекомендується, оскільки, в більшості випадків, воно прискорює потрапляння токсичних речовин у кишечник і перешкоджає виведенню їх із організму [20]. При втраті свідомості потерпілого необхідно покласти без подушки, краще на живіт, голову повернути в сторону, щоб уникнути попадання блювотних мас у дихальні шляхи, давати нюхати ватку, змочену нашатирним спиртом [21]. Слід зазначити, що коли потерпілий перебуває в непритомному стані, то йому категорично заборонено робити промивання шлунка, оскільки вода може потрапити в дихальні шляхи та спричинити смерть. Цю процедуру може проводити лише лікар з використанням спеціальних засобів. При отруєнні медикаментозними препаратами чи алкоголем до прибуття лікаря не можна залишати хворого одного, оскільки в нього може розвинутись збудження. Якщо отруєння виникло внаслідок потрапляння в шлунок кислоти чи лугу (оцтова кислота, нашатирний спирт, кальцинована сода тощо), то до прибуття швидкої допомоги необхідно негайно видалити слину та слиз із рота потерпілого. Загорнувши чайну ложку в шматок марлі, хустинку чи серветку протирають ротову порожнину [21]. Якщо виникли ознаки задухи, проводять штучне дихання (краще способом “рот до носа”, оскільки слизова оболонка рота обпечена). Промивати шлунок самостійно в будь-якому випадку категорично заборонено, оскільки це може посилити блювання, призвести

до попадання кислоти чи лугу в дихальні шляхи. Можна лише дати потерпілому випити 2 – 3 склянки (не більше!) води, щоб розбавити кислоту чи луг і зменшити тим самим їх припікальну дію. Не можна пробувати “нейтралізувати” агресивні рідини, даючи слабкий луг при отруєнні кислотою, чи слабку кислоту при отруєнні лугом, оскільки при цьому утворюється велика кількість вуглекислого газу, що призводить до розтягування шлунка, посилення болю та кровотечі [20].

4.2 Заходи захисту обладнання від статичної електрики

З дитинства кожен з нас стикався з проявами статичної електрики – коли гладив кішку або знімав вовняний светр, а потім торкався до металевого корпусу електроприладів. Даний ефект супроводжується тихим потріскуванням, а на кінчиках пальців відчувається дуже дискомфортне поколювання. Сама по собі статична електрика не є небезпечною, проте якщо природа її виникнення походить від побутової електромережі, є причини для хвилювання.

Небезпека від статичної електрики виходить у тому випадку, якщо вона спровокована попереднім впливом напруги на корпус якого-небудь приладу, який потім було відключено від мережі. У цьому випадку міг накопичитися великий заряд, здатний серйозно вразити людину струмом. Наслідком цього можуть стати опіки або тимчасове оніміння кінцівок, які контактували з пристроєм [19]. Часом у побуті подібна небезпека також може виходити від розеток через те, що при їх нормальній експлуатації електроди вилки труться об пластик та отримують певний статичний заряд, здатний багаторазово посилитися за рахунок навколишнього електромагнітного поля.

Крім того, статична електрика може бути небезпечно не сама по собі, а як фактор, який провокує загоряння. Наприклад, маленька іскра, яка народжується від тертя декількох різнорідних матеріалів, може перекинутися на сусідні об'єкти. Пожежі, причиною яких послужила електростатика, нерідко відбуваються на

складах зерна та різних отрутохімікатів, а також палива та різноманітних спиртовмісних рідин. Усі речовини, які мають гарну горючість, а також дрібнодисперсні порошки потенційно можуть спалахнути від, здавалося б, абсолютно безпечної статичної електрики.

Серед негативних ефектів даного явища, які іноді зустрічаються, слід особливо згадати електромагнітні перешкоди. Якщо статика виявляє себе біля приладів та пристроїв, що мають чутливі до радіохвиль компоненти, вона здатна збити їхні налаштування або навіть вивести з ладу конденсатори. Через статичну електрику можуть виникати спотворення сигналу при прослуховуванні радіо або збої у роботі дистанційних пультів від телевізорів, підсвічування зі світлодіодних стрічок, радіокерованих моделей або іншого подібного обладнання.

Як же захиститися від статичної електрики? Усе різноманіття методів захисту від електростатики зводиться до вибору з двох шляхів: або необхідно створити умови для того, щоб не пов'язані електрони розсіювалися самі по собі, не провокуючи перехід зі стрекотом, або попередити саме виникнення ефекту, не даючи накопичитися заряду. Найпростішим способом позбутися від будь-яких можливих електроударів статикою є звичайне заземлення електроприладів. Передбачається, що корпуси пристроїв, хоча й не перебувають під напругою безпосередньо, можуть поступово накопичувати заряд. Якщо ми забезпечимо стік цього заряду у землю через окремий кабель, то дотики до корпусу перестануть становити загрозу для здоров'я.

У побутових приладах заземлення зазвичай виконується за допомогою третьої жовто-зеленої жили у живильному шнурі. Вона з'єднується з відповідним контактом у розетці та по дроту веде до захисного заземлення та на вулиці. У автомобілях та багатьох пересувних механізмах заземлення ще більш очевидне: до кузова або корпусу кріпиться смужка зі струмопровідного матеріалу або ланцюжок, який при їзді торкається асфальту та забезпечує стік статичного заряду у землю.

Ще один широко відомий спосіб позбутися від зайвих електронів на предметах полягає у тому, щоб збільшити електропровідність діелектричних

матеріалів. Зробивши це, Ви отримаєте можливість відводити зайвий заряд на інші об'єкти, знижуючи сумарний потенціал. Потрібний ефект досягається із застосуванням різних спреїв та аерозолів, які наносяться на предмети. Окрім того, на великі прилади та пристрої можна наклеювати спеціальні плівки, які збирають заряд на себе. Принцип дії й у тих, і у інших однаковий: просто у другому випадку плівка наклеюється відразу, а у першому вона стає результатом висихання складу на поверхні об'єкту.

Схожий ефект дає й звичайне зволоження повітря: якщо у будинку висока вологість, предмети меблів та інші поверхні набувають дуже тонкої плівки-нальоту, яка забезпечує підвищену електричну провідність. Ще краще іонізувати повітря у приміщенні: іонізатор відразу генерує необхідну кількість позитивно та негативно заряджених частинок та викидає їх потоком за допомогою вентилятора. Завдяки гарному поширенню, кожен іон швидко «знаходить своє місце», притягаючись до мікрочасток протилежної полярності та нейтралізуючи заряд [20].

У промисловості, де будь-яка іскра може представляти серйозну небезпеку, застосовують інші підходи. Наприклад, розробляють нові принципи здійснення виробничого процесу, які повністю виключають або мінімізують ймовірність накопичення заряду на поверхні верстатів та агрегатів, готують мікроклімат відповідним чином, використовують антистатичні речовини при обробці обладнання та спецодягу персоналу [20]. За рахунок того, що світильники та допоміжні засоби виробництва знаходяться поза зоною можливого торкання людиною, знижується ймовірність контакту між різнозарядженими тілами та виникнення іскри. На високонебезпечних виробництвах співробітники проходять через так звану клітку Фарадея – це великий бокс, стінки якого сформовані з металевої сітки з маленькими клітинками. Конструкція переймає на себе будь-який розряд та відводить його у землю окремими кабелем [20].

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи бакалавра було розроблено систему для централізованого зберігання та доступу до файлів з використанням Node.js, NestJS, PostgreSQL та Docker. Ця система забезпечує користувачам зручний і безпечний спосіб управління файлами, надаючи можливості реєстрації та аутентифікації, управління файлами та папками, генерації унікальних посилань для завантаження файлів та створення архівів для каталогів.

Під час роботи було проведено детальний аналіз предметної області, визначено вимоги до системи та обґрунтовано вибір технологій. Особлива увага приділялася безпеці системи, включаючи зберігання хешованих паролів та захист від несанкціонованого доступу. Було розроблено детальну архітектуру системи, яка включала проектування бази даних та моделювання взаємодії між компонентами системи.

Проектування та реалізація системи відбувалася з використанням сучасних методів та технологій, що забезпечило високу продуктивність і надійність системи. Використання Node.js та NestJS для серверної частини дозволило створити гнучку та масштабовану архітектуру, яка забезпечує швидку обробку запитів та ефективне управління ресурсами. PostgreSQL було обрано як надійну і потужну базу даних, що дозволяє ефективно зберігати і обробляти великі обсяги даних.

Особлива увага приділялася тестуванню системи, зокрема Unit тестам. Це дозволило виявити та усунути можливі помилки на ранніх етапах розробки, забезпечивши високу якість кінцевого продукту. Проведене тестування підтвердило, що система відповідає визначеним вимогам та працює стабільно і надійно в різних умовах експлуатації.

Система для централізованого зберігання та доступу до файлів призначена для забезпечення безпечного і зручного способу зберігання та обміну файлами між користувачами. Вона підтримує функції реєстрації та аутентифікації користувачів, управління файлами та папками, а також створення унікальних посилань для

завантаження файлів. Система також дозволяє створювати архіви для каталогів, що забезпечує простий і ефективний спосіб організації та зберігання великих обсягів даних.

В процесі розробки було детально описано функціональні та нефункціональні вимоги до системи. Функціональні вимоги включали можливість реєстрації та аутентифікації користувачів, створення, редагування та видалення файлів і папок, а також генерацію посилань для завантаження. Нефункціональні вимоги включали забезпечення безпеки даних, високу продуктивність, масштабованість та зручність у користуванні. Було обґрунтовано вибір стеку технологій, зокрема використання Node.js для реалізації серверної частини, NestJS як серверного фреймворка для структурування коду, PostgreSQL для зберігання даних та Docker для контейнеризації додатку.

Розробка системи включала декілька етапів, таких як проектування бази даних, створення архітектури системи, розробка серверної та клієнтської частин, а також інтеграція різних компонентів. Було створено RESTful API для забезпечення взаємодії між клієнтською та серверною частинами, що дозволило реалізувати всі необхідні функції системи. Особлива увага приділялася забезпеченню безпеки даних користувачів, включаючи шифрування паролів та захист від атак.

Проведене тестування включало як ручні тести, так і автоматизовані тести, що дозволило виявити та усунути можливі помилки і недоліки системи. Ретельне тестування забезпечило стабільність роботи системи і відповідність її функціональних можливостей заявленим вимогам. Усі компоненти системи були протестовані на відповідність вимогам щодо продуктивності, безпеки та зручності використання.

Розроблена система дозволяє користувачам зручно та безпечно зберігати свої файли, організовувати їх у папки та надавати доступ до них іншим користувачам через унікальні посилання. Вона відповідає сучасним вимогам до програмних продуктів у сфері зберігання та управління файлами, забезпечуючи високу надійність і безпеку збережених даних.

Отримані знання та досвід під час розробки даного проекту є цінними для подальшої професійної діяльності у сфері розробки веб-додатків та використання сучасних технологій і методологій розробки. Завдяки виконаній роботі було здобуто практичні навички у використанні Node.js, NestJS, PostgreSQL та Docker, що є важливими інструментами у сучасному світі програмної інженерії.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Centralized Data Storage System: How your Business can benefit [Електронний ресурс]. – 2022. – Режим доступу до ресурсу: https://www.ilink-digital.com/insights/blog/centralized-data-storage-system-how-your-business-can-benefit/.
2. JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
3. TypeScript is JavaScript with syntax for types [Електронний ресурс] – Режим доступу до ресурсу: https://www.typescriptlang.org.
4. About Node.js [Електронний ресурс] – Режим доступу до ресурсу: https://nodejs.org/en/about.
5. REST API як спосіб спілкування компонент веб-додатків [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://foxminded.ua/shcho-take-rest-api/.
6. Документація [Електронний ресурс] – Режим доступу до ресурсу: https://docs.nestjs.com/.
7. Archiver API [Електронний ресурс] – Режим доступу до ресурсу: https://www.archiverjs.com/docs/archiver.
8. PostgreSQL 16.3 Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/current/.
9. Документація TypeORM [Електронний ресурс] – Режим доступу до ресурсу: https://typeorm.io.
10. What is Docker? [Електронний ресурс] – Режим доступу до ресурсу: https://aws.amazon.com/docker/?nc1=h_ls.
11. Waht is Next.js? [Електронний ресурс] – Режим доступу до ресурсу: https://nextjs.org/docs#what-is-nextjs.
12. Документація Jest [Електронний ресурс] – Режим доступу до ресурсу: https://jestjs.io/docs/getting-started.

13. Relationships in SQL [Електронний ресурс] – Режим доступу до ресурсу: https://blog.devart.com/types-of-relationships-in-sql-server-database.html#:~:text=There%20are%20five%20types%20of,%2C%20and%20self%2Dreferencing%20relationships..
14. What is an Entity Relationship Diagram (ERD)? [Електронний ресурс] – Режим доступу до ресурсу: https://www.lucidchart.com/pages/er-diagrams.
15. Software testing [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Software_testing.
16. Automated Parallelization of Software for Identifying Parameters of Intraparticle Diffusion and Adsorption in Heterogeneous Nanoporous Media : дис. канд. техн. наук / ., 2023.
17. Mykhalyk D. Automated processing and analysis of medical texts / Mykhalyk D., 2023.
18. Mudryk I. Hybrid artificial intelligence systems for complex neural network analysis of abnormal neurological movements with multiple cognitive signal nodes / Mudryk I., 2020.
19. Документація UML [Електронний ресурс] – Режим доступу: http://www.uml.org/.
20. Постанова КМ України від 26.06.2013 «Про затвердження Порядку здійснення навчання населення діям у надзвичайних ситуаціях». ДСТУ 5058:2008 «Навчання населення діям у надзвичайних ситуаціях. Основні положення».
21. Стеблюк М. І. Порядок і правила надання першої допомоги при ураженні небезпечними речовинами / М. І. Стеблюк // цивільна оборона / М. І. Стеблюк., 2013.
22. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ. // Одеський державний аграрний університет. – 2017. – С. 437.

ДОДАТКИ

Додаток А Лістинг коду системи

Лістинг коду 1 – файл main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ConfigService } from '@nestjs/config';
import { BadRequestException, Logger, ValidationPipe } from
'@nestjs/common';
import { DocumentBuilder, SwaggerModule } from '@nestjs/swagger';
import { FILE_UPLOAD_DIR, TMP_DIR } from './common/constants';
import { FileSystemService } from './utils';

const logger = new Logger();

async function bootstrap(): Promise<void> {
  const app = await NestFactory.create(AppModule);

  const fileSystem = app.get(FileSystemService);

  await fileSystem.ensureFolderExists(FILE_UPLOAD_DIR);
  await fileSystem.ensureFolderExists(TMP_DIR);

  app.useGlobalPipes(
    new ValidationPipe({
      transform: true,
      transformOptions: { enableImplicitConversion: true },
    }),
  );

  app.enableCors({
    origin: '*',
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    preflightContinue: false,
    optionsSuccessStatus: 204,
  });

  app.useGlobalPipes(
    new ValidationPipe({
      transform: true,
```

```
      transformOptions: { enableImplicitConversion: true },
      whitelist: true,
      forbidNonWhitelisted: true,
      exceptionFactory: (errors) => new BadRequestException(errors),
    }),
  );

  const config = new DocumentBuilder()
    .setTitle('File Store')
    .setVersion('1.0')
    .addBearerAuth()
    .build();

  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('swagger', app, document, {
    swaggerOptions: {
      persistAuthorization: true,
    },
  });

  const configService = app.get(ConfigService);

  const port = configService.get('PORT');
  const host = configService.get('HOST');

  await app.listen(port, host);

  logger.log('URL', await app.getUrl());
}

bootstrap();
```

Лістинг коду 2 – файл app.module.ts

```
import { Module } from '@nestjs/common';
import { ConfigModule } from '@nestjs/config';
import { UserModule } from './user/user.module';
import { FileModule } from './file/file.module';
import { DatabaseModule } from './database/database.module';
```

```
import { AuthModule } from './auth/auth.module';
import { FolderModule } from './folder/folder.module';
import { ScheduleModule } from '@nestjs/schedule';
import { NoteModule } from './note/note.module';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      envFilePath: '../.env',
    }),
    ScheduleModule.forRoot(),
    DatabaseModule,
    UserModule,
    FileModule,
    AuthModule,
    FolderModule,
    NoteModule,
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

Лістинг коду 3 – файл auth.controller.ts

```
import { Body, Controller, Post, UseGuards } from '@nestjs/common';
import { AuthService } from './auth.service';
import { ConfigService } from '@nestjs/config';
import { LoginDto, RefreshDto, RegisterUserDto } from '../common/dto';
import { IJwtTokens } from '../common/types';
import { RefreshTokenGuard } from '../common/guards';
import { ApiOkResponse, ApiResponse, ApiTags } from '@nestjs/swagger';
import { AuthResDto } from '../common/dto';

@Controller('auth')
@ApiTags('auth')
export class AuthController {
```

```
  private readonly accessTokenAge: number;
  private readonly refreshTokenAge: number;

  constructor(
    private readonly authService: AuthService,
    private configService: ConfigService,
  ) {
    this.accessTokenAge =
      this.configService.get<number>('ACCESS_TOKEN_AGE_MINUTES') * 60 *
1000;

    this.refreshTokenAge =
      this.configService.get<number>('REFRESH_TOKEN_AGE_DAYS') *
      24 *
      60 *
      60 *
      1000;
  }

  @Post('login')
  @ApiOkResponse({
    description: 'User logged in successfully',
    type: AuthResDto,
  })
  async login(@Body() loginDto: LoginDto): Promise<AuthResDto> {
    return await this.authService.login(loginDto);
  }

  @Post('register')
  @ApiResponse({
    status: 201,
    description: 'User registered successfully',
    type: AuthResDto,
  })
  async register(@Body() registerDto: RegisterUserDto): Promise<AuthResDto>
{
    return this.authService.registerUser(registerDto);
  }
```

```
  @UseGuards(RefreshTokenGuard)
  @Post('refresh')
  async refresh(@Body() refreshDto: RefreshDto): Promise<IJwtTokens> {
    return await this.authService.refresh(refreshDto.refreshToken);
  }
}
```

Лістинг коду 4 – файл auth.service.ts

```
import {
  BadRequestException,
  ConflictException,
  Injectable,
  InternalServerErrorException,
  NotFoundException,
} from '@nestjs/common';
import { UserService } from '../user/user.service';
import { LoginDto, RegisterUserDto } from '../common/dto';
import { IJwtTokens, IUserPayload } from '../common/types';
import * as bcrypt from 'bcrypt';
import { JwtService } from '@nestjs/jwt';
import { ConfigService } from '@nestjs/config';
import { map } from 'p-iteration';

@Injectable()
export class AuthService {
  private readonly accessTokenAge: string;
  private readonly refreshTokenAge: string;

  constructor(
    private userService: UserService,
    private jwtService: JwtService,
    private configService: ConfigService,
  ) {
    this.accessTokenAge =
      this.configService.get<number>('ACCESS_TOKEN_AGE_MINUTES') + 'm';

    this.refreshTokenAge =
      this.configService.get<number>('REFRESH_TOKEN_AGE_DAYS') + 'd';
```

```
  }

  async login({ email, password }: LoginDto): Promise<IJwtTokens> {
    const user = await this.userService.findUserWithPasswordByEmail(email);

    if (!user) throw new NotFoundException('USER_NOT_FOUND');

    const isPasswordValid = await bcrypt.compare(password, user.password);

    if (!isPasswordValid) throw new
BadRequestException('INVALID_PASSWORD');

    return this.generateTokens({
      id: user.id,
      email: user.email,
    });
  }

  async registerUser(registerDto: RegisterUserDto): Promise<IJwtTokens> {
    const { email, password, fullName, driveName } = registerDto;

    const isUsernameTaken = await this.userService.isEmailTaken(email);

    if (isUsernameTaken) throw new
ConflictException('USERNAME_ALREADY_TAKEN');

    const hashedPassword = await this.hashUserPassword(password);

    const user = await this.userService.createUser({
      email,
      password: hashedPassword,
      fullName,
    });

    await this.userService.createUserDrive(user, driveName);

    return this.generateTokens({
      id: user.id,
      email: user.email,
```

```
  });
}

async refresh(refreshToken: string): Promise<IJwtTokens> {
  const payload = (await this.jwtService.verifyAsync(refreshToken, {
    secret: this.configService.get<string>('JWT_REFRESH_SECRET'),
  })) as IUserPayload;

  if (!payload.id || !payload.email) {
    throw new InternalServerErrorException('INVALID_PAYLOAD');
  }

  const user = await this.userService.findByEmail(payload.email);

  if (!user) throw new NotFoundException('USER_NOT_FOUND');

  return this.generateTokens({
    id: user.id,
    email: user.email,
  });
}

async generateTokens(payload: IUserPayload): Promise<IJwtTokens> {
  if (!payload.id || !payload.email) {
    throw new InternalServerErrorException('INVALID_PAYLOAD');
  }

  let accessToken: string, refreshToken: string;

  const signOptions = [
    {
      expiresIn: this.accessTokenAge,
      secret: this.configService.get<string>('JWT_SECRET'),
      payload,
    },
    {
      expiresIn: this.refreshTokenAge,
      secret: this.configService.get<string>('JWT_REFRESH_SECRET'),
      payload,
```

```
      },
    ];

    try {
      const tokens = await map(
        signOptions,
        async ({ payload, expiresIn, secret }) => {
          return await this.jwtService.signAsync(payload, {
            expiresIn: expiresIn,
            secret: secret,
          });
        },
      );

      accessToken = tokens[0];
      refreshToken = tokens[1];
    } catch (error) {
      throw new InternalServerErrorException('TOKEN_GENERATION_FAILED');
    }

    return { accessToken, refreshToken };
  }

  private async hashUserPassword(password: string): Promise<string> {
    return await bcrypt.hash(password, 10);
  }
}
```

Лістинг коду 5 – файл folder.controller.ts

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  ParseIntPipe,
  Post,
  Put,
```

```
  Res,
  StreamableFile,
  UseGuards,
} from '@nestjs/common';
import { FolderService } from './folder.service';
import { AccessTokenGuard } from '../common/guards';
import { ApiBearerAuth, ApiTags } from '@nestjs/swagger';
import { CurrentUser } from '../common/decorators/current-user';
import { IApiResponse, IUser } from '../common/types';
import { CreateFolderDto } from './dto/create-folder.dto';
import { RenameFolderDto } from './dto/rename-folder.dto';
import { MoveFolderDto } from './dto/move-folder.dto';
import { DeleteFolderDto } from './dto/delete-folder.dto';
import { Response } from 'express';

@Controller('folder')
@UseGuards(AccessTokenGuard)
@ApiTags('folder')
@ApiBearerAuth()
export class FolderController {
  constructor(private readonly folderService: FolderService) {}

  @Get('download/:id')
  async downloadArchive(
    @Param('id', ParseIntPipe) id: number,
    @CurrentUser() user: IUser,
    @Res({ passthrough: true }) res: Response,
  ): Promise<StreamableFile> {
    const { stream, contentType, fileName, size } =
      await this.folderService.downloadArchive(id, user);

    return new StreamableFile(stream, {
      type: contentType,
      disposition: `attachment; filename="${fileName}"`,
      length: size,
    });
  }

  @Post('create')
```

```
async createFolder(
  @CurrentUser() user: IUser,
  @Body() createFolderDto: CreateFolderDto,
): Promise<IApiResponse> {
  await this.folderService.createFolder(user, createFolderDto);

  return {
    message: 'FOLDER_CREATED_SUCCESSFULLY',
    date: new Date(),
  };
}

@Put('rename/:id')
async renameFolder(
  @Param('id', ParseIntPipe) id: number,
  @CurrentUser() user: IUser,
  @Body() renameFolderDto: RenameFolderDto,
): Promise<IApiResponse> {
  await this.folderService.renameFolder(id, user, renameFolderDto);

  return {
    message: 'FOLDER_RENAMED_SUCCESSFULLY',
    date: new Date(),
  };
}

@Put('move')
async moveFolder(
  // @Param('id', ParseIntPipe) id: number,
  @CurrentUser() user: IUser,
  @Body() moveFolderDto: MoveFolderDto,
): Promise<IApiResponse> {
  await this.folderService.moveFolders(user, moveFolderDto);

  return {
    message: 'FOLDER_MOVED_SUCCESSFULLY',
    date: new Date(),
  };
}
```

```
  @Delete('delete')
  async deleteFolder(
    @CurrentUser() user: IUser,
    @Body() deleteFolderDto: DeleteFolderDto,
  ): Promise<IApiResponse> {
    return this.folderService.deleteFolders(user, deleteFolderDto);
  }
}
```

Лістинг коду 6 – файл file.controller.ts

```
import {
  Body,
  Controller,
  Delete,
  Get,
  Param,
  ParseIntPipe,
  Post,
  Put,
  Res,
  StreamableFile,
  UploadedFiles,
  UseGuards,
  UseInterceptors,
} from '@nestjs/common';
import { ApiBearerAuth, ApiConsumes, ApiTags } from '@nestjs/swagger';
import { FileService } from './file.service';
import { FilesInterceptor } from '@nestjs/platform-express';
import { IApiResponse, IUser, MulterFile } from '../common/types';
import { AccessTokenGuard } from '../common/guards';
import { CurrentUser } from '../common/decorators/current-user';
import { FileEntity } from '../database/entities';
import { RenameFileDto } from './dto/rename-file.dto';
import { MoveFileDto } from './dto/move-file.dto';
import { DeleteFileDto } from './dto/delete-file.dto';
import { UploadFileDto } from './dto/upload-file.dto';
import { Response } from 'express';
```

```
import {
  CreateDownloadLinkDto,
  ResCreateDownloadLinkDto,
} from './dto/create-download-link.dto';
import { Public } from '../common/decorators';

@Controller('file')
@UseGuards(AccessTokenGuard)
@ApiTags('file')
@ApiBearerAuth()
export class FileController {
  constructor(private readonly fileService: FileService) {}

  @Get()
  async findAll(): Promise<FileEntity[]> {
    return this.fileService.findAll();
  }

  @Get('download/:id')
  async downloadFile(
    @Param('id', ParseIntPipe) id: number,
    @CurrentUser() user: IUser,
    @Res({ passthrough: true }) res: Response,
  ): Promise<StreamableFile> {
    const { stream, contentType, fileName, size } =
      await this.fileService.getFileStream(id, user);

    return new StreamableFile(stream, {
      type: contentType,
      disposition: `attachment; filename="${fileName}"`,
      length: size,
    });
  }

  @Get('download-link/:token')
  @Public()
  async downloadByToken(
    @Param('token') token: string,
  ): Promise<StreamableFile> {
```

```
  const { file } = await this.fileService.getFileInfoByToken(token);

  const { stream, contentType, fileName, size } =
    await this.fileService.getFileStream(file.id, file.user);

  return new StreamableFile(stream, {
    type: contentType,
    disposition: `attachment; filename="${fileName}"`,
    length: size,
  });
}

@Post('download-link/:id')
createDownloadLink(
  @Param('id', ParseIntPipe) id: number,
  @Body() dto?: CreateDownloadLinkDto,
): Promise<ResCreateDownloadLinkDto> {
  return this.fileService.createDownloadLink(id, dto?.expirationTime);
}

@Post('upload')
@ApiConsumes('multipart/form-data')
@UseInterceptors(FilesInterceptor('files'))
async uploadFile(
  @UploadedFiles()
  files: Array<MulterFile>,
  @CurrentUser() user: IUser,
  @Body() uploadFileDto: UploadFileDto,
): Promise<IApiResponse> {
  return await this.fileService.saveFiles(files, user, uploadFileDto);
}

@Put('rename/:id')
async renameFile(
  @Param('id', ParseIntPipe) id: number,
  @CurrentUser() user: IUser,
  @Body() renameFileDto: RenameFileDto,
): Promise<IApiResponse> {
  return await this.fileService.renameFile(id, user, renameFileDto);
```

```
  }

  @Put('move')
  async moveFile(
    // @Param('id', ParseIntPipe) id: number,
    @CurrentUser() user: IUser,
    @Body() moveFileDto: MoveFileDto,
  ): Promise<IApiResponse> {
    return await this.fileService.moveFiles(user, moveFileDto);
  }

  @Delete('delete')
  async deleteFiles(
    @CurrentUser() user: IUser,
    @Body() deleteFileDto: DeleteFileDto,
  ): Promise<IApiResponse> {
    return await this.fileService.deleteFiles(user, deleteFileDto);
  }
}
```

Додаток Б Публікація у науковій конференції

VII Міжнародна студентська науково - технічна конференція
"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

УДК 004.41
Солтис М.В. – ст. гр. СП-42
Тернопільський національний технічний університет імені Івана Пулюя

РОЗРОБКА СИСТЕМИ ДЛЯ ЦЕНТРАЛІЗОВАНОГО ЗБЕРІГАННЯ ТА ДОСТУПУ ДО ФАЙЛІВ

Науковий керівник: д. ф.-м. н., професор Михалик Д. М.

Soltys M.
Ternopil Ivan Puluj National Technical University

DEVELOPMENT OF A SYSTEM FOR CENTRALIZED STORAGE AND ACCESS TO FILES

Supervisor: PhD in Technical Sciences, Professor Mykhalyk D.

Ключові слова: централізоване зберігання, доступ до файлів, система управління файлами, безпека даних, мережеві протоколи.

Keywords: centralized storage, file access, file management system, data security, network protocols.

Моя наукова робота зосереджена на розробці системи для централізованого зберігання та доступу до файлів, використовуючи сучасні технології зберігання даних. Ця система призначена для покращення ефективності управління даними та доступу до них в корпоративних середовищах.

Початково я досліджую специфічні потреби користувачів та технічні вимоги, щоб визначити ключові характеристики, які повинна мати система. Заснований на цьому аналізі, я вибираю відповідні технології та методи зберігання даних.

Розробка архітектури системи включає вибір між розподіленими та централізованими підходами, а також визначення протоколів доступу до файлів. Я обираю централізований підхід для забезпечення кращого контролю та безпеки даних.

Основні аспекти реалізації системи включають налаштування серверів зберігання, впровадження заходів шифрування даних та розробку інтерфейсів для доступу користувачів. Забезпечення безпеки є критично важливим, тому впроваджується багаторівнева система захисту.

Перед впровадженням системи проводиться ретельне тестування для перевірки її надійності, продуктивності та безпеки. Тестування допомагає виявити потенційні проблеми та оптимізувати процеси управління файлами.

Завершуючи, система централізованого зберігання та доступу до файлів демонструє значне поліпшення у ефективності управління даними та забезпечення їх безпеки. Вона відкриває можливості для подальших досліджень та розвитку в області інформаційних технологій.

Література

1. Effective Strategies for Managing Network-Attached Storage Systems [Електронний ресурс] – Режим доступу до ресурсу: https://www.linkedin.com/advice/3/what-best-practices-managing-network-storage-qcpze.
2. Data Security in Cloud Computing [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/learn/what-is-cloud-data-security.

Додаток В Диск з роботою