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

магістр
(назва освітнього ступеня)

на тему: Вебсистема централізованого зберігання файлів із підтримкою семантичного пошуку

Виконав: студент VI курсу групи СПм-61
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

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

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

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

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

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль
2025

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

Факультет Комп'ютерно інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра Програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М. Р
(підпис) (прізвище та ініціали)
«__» _____ 2025 р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістра
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Солтису Максиму Васильовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Вебсистема централізованого зберігання файлів із підтримкою семантичного пошуку

Керівник роботи доктор фіз.-мат. наук професор кафедри ПІ Михалик Дмитро Михайлович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 2024 року № _____

2. Термін подання студентом завершеної роботи _____

3. Вихідні дані до роботи Предметна область, технічне завдання, вимоги та специфікація, програмне рішення

4. Зміст роботи (перелік питань, які потрібно розробити): _____

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів): _____
Ілюстративні зображення, інформативні зображення для доповнення тексту, таблиці, знімки екрану з проробленою роботою.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 14.10.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.			*Виконано*

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

Керівник роботи _____ Петрик М. Р.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота магістра на тему «Вебсистема централізованого зберігання файлів із підтримкою семантичного пошуку» виконана Солтисом Максимом Васильовичем, студентом Тернопільського національного технічного університету імені Івана Пулюя, факультету комп'ютерно-інформаційних систем і програмної інженерії, кафедри програмної інженерії, групи СПм-61.

Відомості про обсяг: сторінок – 97, рисунків – 28, таблиць – 2, частин – 4, додатків – 2, бібліографічних посилань – 34.

Метою роботи є створення ефективної та безпечної вебсистеми, що забезпечує централізоване зберігання файлів, зручний доступ до них і можливість семантичного пошуку за змістом документів. У першому розділі виконано аналіз предметної області та визначено вимоги до системи. Обґрунтовано вибір основних технологій, зокрема Node.js і NestJS для серверної частини, PostgreSQL для зберігання даних, Docker для контейнеризації, а також ElasticSearch і OpenAI embeddings для реалізації семантичного пошуку.

Розроблена система підтримує реєстрацію й аутентифікацію користувачів, управління файлами та каталогами, попередній перегляд і завантаження файлів, створення тимчасових посилань для доступу, а також інтелектуальний пошук, що враховує зміст файлів, а не лише їх назви. Значну увагу приділено питанням безпеки, зокрема зберіганню паролів у хешованому вигляді та захисту від несанкціонованого доступу.

Об'єктом дослідження є система централізованого зберігання та доступу до файлів. Предметом дослідження є методи й інструменти розробки вебсистем із підтримкою семантичного пошуку з використанням Node.js, NestJS, PostgreSQL, ElasticSearch та OpenAI embeddings.

Ключові слова: централізоване зберігання файлів, семантичний пошук, ElasticSearch, OpenAI embeddings, Node.js, NestJS, PostgreSQL, Docker.

ABSTRACT

The master’s qualification work on the topic "Development of a System for Centralized File Storage and Access with Semantic Search Support" was written by Maksym Vasylovych Soltys, a student of Ivan Puluj Ternopil National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, group SPm-61.

Information about the scope: pages – 97, figures – 28, tables – 2, sections – 4, appendices – 2, bibliogr. – 34.

The aim of the thesis is to develop an efficient and secure web-based system that provides centralized file storage, convenient access to data, and semantic search based on document content. The first chapter presents an analysis of the subject area and defines system requirements. The choice of core technologies is justified, including Node.js and NestJS for the server-side implementation, PostgreSQL for data storage, Docker for containerization, as well as ElasticSearch and OpenAI embeddings for semantic search functionality.

The developed system supports user registration and authentication, file and folder management, file preview and download, generation of temporary access links, and intelligent search that considers the content of files rather than only their names. Special attention is paid to security aspects, including password hashing and protection against unauthorized access.

The object of the research is a centralized file storage and access system. The subject of the research is the methods and tools for developing web-based systems with semantic search support using Node.js, NestJS, PostgreSQL, ElasticSearch, and OpenAI embeddings.

Keywords: centralized file storage, semantic search, ElasticSearch, OpenAI embeddings, Node.js, NestJS, PostgreSQL, Docker.

ЗМІСТ

ВСТУП

Стрімке зростання обсягів цифрової інформації та активне впровадження інформаційних технологій у різні сфери діяльності призводять до потреби у якісних рішеннях для зберігання та ефективного оброблення даних. У сучасному середовищі, де доступність інформації та швидкість її пошуку є критично важливими, централізовані системи управління файлами виступають основою для побудови корпоративних сервісів, освітніх платформ, хмарних сховищ і внутрішніх інформаційних порталів.

Актуальність теми зумовлена тим, що традиційні сховища, створені для зберігання та доступу до файлів, вже не задовольняють сучасні вимоги користувачів. Вони забезпечують лише базові механізми пошуку за ключовими словами або іменами документів і не враховують семантичного змісту інформації. У таких умовах виникає потреба у розробці систем, здатних не лише надійно зберігати дані, але й розуміти їхній зміст та забезпечувати інтелектуальний доступ до них.

Магістерська робота є логічним розвитком і суттєвим рефакторингом раніше виконаного бакалаврського проєкту, тема якого стосувалася створення серверної системи для централізованого доступу до файлів. У попередній роботі реалізовано базові функціональні можливості: механізми авторизації, зберігання файлів та керування правами доступу, побудовані на основі технологій Node.js, NestJS, PostgreSQL та Docker. Однак зі зростанням вимог до взаємодії з інформацією та необхідністю підвищення зручності використання система потребувала розвитку та розширення.

У межах магістерського дослідження виконано глибокий рефакторинг існуючого рішення та значне розширення його функціональності. На основі початкового серверного ядра створено повноцінну вебсистему з інтерфейсом, реалізованим за допомогою Next.js, React та бібліотеки Ant Design, що забезпечує інтерактивну та інтуїтивну взаємодію користувача з файловим середовищем. Однією з ключових інновацій є впровадження семантичного пошуку, для чого

інтегровано ElasticSearch як пошуковий рушій та застосовано векторні представлення тексту (embeddings), отримані через моделі штучного інтелекту.

Таким чином, метою даної кваліфікаційної роботи є створення вебсистеми, яка об’єднує функції централізованого зберігання файлів і підтримує пошук інформації за смисловою подібністю, що підвищує швидкість доступу до необхідних документів. Реалізація такої системи вимагає розробки серверної і клієнтської частин, побудови механізмів семантичної індексації, організації безпечного доступу до даних та забезпечення масштабованості рішення.

Об’єктом дослідження є процеси проєктування, вдосконалення і розгортання веборієнтованих систем для зберігання та пошуку файлів. Предметом дослідження є методологія та технології, що забезпечують створення систем із підтримкою семантичного аналізу вмісту, включаючи NestJS, Node.js, PostgreSQL, ElasticSearch, embeddings, Docker, Next.js та React

Отже, робота спрямована на розробку оновленої версії системи, яка не лише зберігає напрацювання попереднього проєкту, але й значно вдосконалює його за рахунок архітектурних оптимізацій, створення вебінтерфейсу та впровадження семантичного пошуку, що є важливим етапом еволюції систем керування інформацією. У подальших розділах розглянуто дослідження предметної області, проектування, імплементацію та оцінювання результатів побудованого рішення.

РОЗДІЛ 1: ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА МЕТОДОЛОГІЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

У даному розділі здійснюється дослідження предметної області систем централізованого зберігання та доступу до файлів, а також аналізуються підходи до розробки сучасних вебсистем у цій сфері. Розглядаються основні потреби користувачів, що працюють з цифровими файлами, визначаються функціональні та нефункціональні вимоги до програмного забезпечення, а також аналізуються існуючі рішення та підходи до організації файлових сховищ. Окрему увагу приділено методології розробки програмного забезпечення та обґрунтуванню вибору технологій, які формують основу для подальшого проєктування та реалізації вебсистеми з підтримкою розширених можливостей пошуку та управління файлами.

1.1 Дослідження та характеристика предметної області

У сучасних умовах цифровізації обробка, зберігання та швидкий доступ до інформації стають ключовими факторами ефективності діяльності організацій. Оскільки обсяги даних постійно зростають, виникає необхідність у побудові надійних інструментів, що забезпечують структуроване зберігання, доступність та захищеність цифрових ресурсів. Саме тому вебсистеми для централізованого управління файлами набувають широкого поширення у бізнесі, освіті, наукових установах, державному секторі та сфері охорони здоров’я.

Традиційні рішення забезпечують зберігання та базову категоризацію документів, дозволяючи користувачам організовувати інформацію за допомогою папок, міток та прав доступу. Такі системи скорочують час пошуку, усувають дублювання інформації та спрощують командну роботу, забезпечуючи спільний доступ до файлів і можливість їх редагування в межах встановлених дозволів. Головними вимогами до таких платформ є надійність, доступність, масштабованість та безпека даних [1].

Важливим аспектом централізованих сховищ є захист інформації. Оскільки кіберзагрози постійно зростають, критично важливо гарантувати конфіденційність і цілісність даних. Це досягається через застосування механізмів аутентифікації, авторизації, шифрування та логування дій користувачів, а також резервного копіювання [1]. Використання контейнеризації (наприклад, Docker) сприяє масштабуваності та гнучкому розширенню функціональності системи без необхідності значних витрат на інфраструктуру.

Однак сучасні потреби виходять за межі простого зберігання даних. Зі збільшенням обсягів інформації традиційні методи пошуку на основі ключових слів стають менш ефективними: користувач не завжди знає точні назви файлів або формулювання, а запити можуть формуватися природною мовою. Тому актуальним напрямом розвитку є семантичний пошук — можливість знаходити інформацію за її смисловою подібністю, а не за прямим текстовим збігом.

Семантичний пошук базується на методах штучного інтелекту, зокрема на векторному представленні тексту (embeddings) [17], що дозволяє перетворювати зміст документів у числові представлення. Такі вектори зберігають контекст і значення, тому система може знаходити релевантні документи навіть тоді, коли у тексті та запиті використовуються різні формулювання. Впровадження семантичного пошуку суттєво підвищує ефективність доступу до великих інформаційних сховищ, особливо в умовах різнорідних документів і неструктурованого тексту.

Для реалізації таких підходів широко використовуються пошукові платформи, наприклад ElasticSearch [16], що поєднують класичні методи аналізу тексту (BM25, інвертовані індекси) із підтримкою векторних баз даних та kNN-пошуку. Це дозволяє створювати гібридні моделі пошуку, де система оцінює документи як за ключовими словами, так і за семантичною близькістю. Використання embeddings, згенерованих моделями OpenAI [17] та іншими AI-рушіями, відкриває можливість реалізації інтелектуального доступу до даних і створює основу для систем підтримки прийняття рішень.

Отже, дана предметна область дослідження охоплює постановку задач зберігання інформації, організації доступу до неї, безпеки, а також інтелектуального пошуку, що дозволяє не лише акумулювати дані, але й витягувати з них знання. Це формує підґрунтя для розробки сучасних вебсистем, здатних адаптуватися до зростання обсягів інформації та потреб користувачів, і обґрунтовує актуальність створення системи, яка поєднує централізоване зберігання з пошуком за змістом документів, що і є фокусом даної роботи.

1.2 Аналіз та опис сценаріїв взаємодії користувачів із системою

Актори системи:

На рисунку 1 представлений основний актор системи.

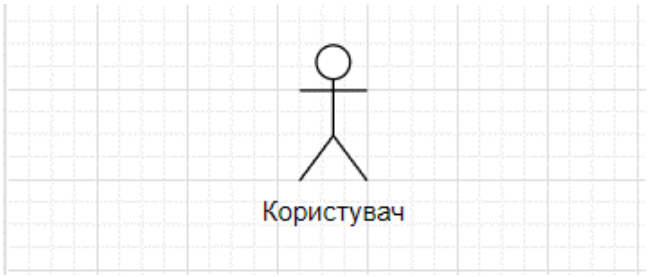

Рисунок 1.1 – Основний актор системи

Опис актора системи представлений в таблиці 1.1.

Табл. 1.1 – Визначення акторів

Актор	Опис
Користувач	Користувач має можливість створювати обліковий запис і проходити автентифікацію в системі. Після входу до системи користувач може здійснювати завантаження файлів до сховища та отримувати їх зі сховища, переглядати вміст файлової структури, отримувати детальну інформацію про файли й каталоги, змінювати їхні властивості, а також виконувати операції видалення елементів файлової системи.

Опис сценаріїв взаємодії користувачів із системою 1.2.

Табл. 1.2 – Сценарії взаємодії

Актор	Варіант використання	Короткий опис
Користувач	Реєстрація	Користувач має можливість **створити особистий обліковий запис** у вебсистемі, зазначивши електронну адресу, ім’я, власний пароль та назву сховища, яке буде пов’язане з його профілем. Після успішного заповнення форми реєстрації створюється відповідний запис у системі, ініціалізує персональне файлове середовище та налаштовує параметри доступу.
Користувач	Аутентифікація	Користувач проходить процес підтвердження власності облікового запису через e-mail, що активує його файлове середовище та забезпечує коректну ідентифікацію в системі.
Користувач	Управління папками	Користувач може створювати, перейменовувати та видаляти папки як у корені власного сховища, так і в межах вкладеної структури, організовуючи файловий простір відповідно до своїх потреб.
Користувач	Управління файлами	Користувач може завантажувати файли до будь-якого каталогу сховища, редагувати їхні властивості, видаляти або завантажувати на локальний пристрій. Система забезпечує базову інформаційну структуру та безпечну роботу з файлами.

Продовження таблиці 1.2

Користувач	Навігація по файловій системі	Користувач має змогу переглядати дерево каталогів і переходити між ними, швидко знаходячи потрібні елементи через структуроване представлення сховища.
Користувач	Пошук у файловій системі	Система підтримує різні види пошуку: традиційний – за назвами файлів та каталогів, і семантичний пошук — за змістовою схожістю документа, що забезпечує точніший доступ до інформації.
Користувач	Додавання нотаток	Додавати нотатки для папок та файлів для забезпечення інформативності.
Користувач	Генерування посилань для доступу до файлів	Система надає можливість створювати унікальні посилання для забезпечення тимчасового доступу третіх осіб до завантаження вибраних файлів.
Користувач	Перегляд файлів та зображень	Користувач може відкривати зображення, документи (наприклад PDF) та інші підтримувані формати безпосередньо у вебінтерфейсі, що дозволяє переглядати вміст без попереднього завантаження.

Моделювання варіантів використання системи:

Рисунок 1.2 – Діаграма прецедентів вебсистеми

1.3 Постановка завдання

На основі проведеного аналізу предметної області [1] було сформовано перелік функціональних і нефункціональних вимог, які визначають основні характеристики та обмеження розроблюваної системи:

Функціональні вимоги:

1. Аутентифікація та реєстрація користувачів:

1.1.Система повинна надавати можливість створення нового облікового запису, включно з підтвердженням його належності користувачу.

1.2. Система повинна забезпечувати автентифікацію користувачів шляхом введення електронної адреси та пароля.

2. Управління файлами та каталогами:

2.1. Система має надавати користувачам можливість завантажувати до власного сховища файли різних типів, зокрема документи, зображення та відеоматеріали.

2.2. Користувач має можливість створювати, перейменовувати та видаляти каталоги і вкладені каталоги для структуризації даних.

2.3. Має бути реалізована підтримка drag-and-drop для зручного переміщення файлів та папок.

3. Доступ сторонніх користувачів до файлів:

3.1. Система повинна надавати функціональність створення унікальних посилань для передачі доступу до окремих файлів неавторизованим отримувачам.

4. Пошук і фільтрація файлів:

4.1. Підтримка класичного пошуку за назвою, атрибутами або метаданими файлів.

4.2. Має бути реалізовано семантичний пошук із використанням векторних представлень (embeddings), що дозволяє знаходити документи за змістовою схожістю, навіть при відсутності прямого збігу у формулюваннях.

4.3. Система повинна дозволяти фільтрацію файлів за різними параметрами (формат, дата, розмір, розташування в каталозі тощо).

5. Безпека даних:

5.1. Дані користувача (в тому числі й паролі) повинні зберігатися у захищеному, зашифрованому вигляді.

5.2. Необхідно забезпечити резервне копіювання інформації для запобігання втрати даних у випадку збоїв.

6. Продуктивність та масштабованість:

6.1. Система повинна коректно функціонувати при одночасній роботі значної кількості користувачів.

6.2. Необхідно забезпечити швидкий доступ до файлів та результатів пошуку, включно зі зростанням обсягів даних.

Нефункціональні вимоги:

1. Продуктивність:

1.1. Система повинна забезпечувати швидке реагування інтерфейсу та серверної частини на дії користувача.

1.2. Архітектура має підтримувати оброблення великої кількості одночасних запитів, включаючи виконання семантичного пошуку без істотного падіння продуктивності.

2. Надійність:

2.1. Система повинна гарантувати високу доступність сервісів і мінімізувати час простою.

2.2. Компоненти платформи мають бути відмовостійкими, з можливістю автоматичного відновлення або реплікації даних після збоїв.

3. Портативність та доступність:

3.1. Вебінтерфейс повинен коректно функціонувати в основних браузерах (Google Chrome, Edge, Opera, Firefox тощо).

3.2. Система має підтримувати розгортання на різних серверах або у хмарних середовищах (Docker, Kubernetes, cloud-hosting) без втрати функціональності чи даних.

4. Модульність та підтримуваність:

4.1. Архітектура повинна бути модульною та дозволяти додавати нові можливості (наприклад, нові типи пошуку чи витягання контенту) без значних змін у решті системи.

4.2. Код та API повинні бути документованими і читабельними для спрощення супроводу, тестування та розширення.

5. Зручність використання:

5.1. Користувацький інтерфейс має бути інтуїтивним, забезпечуючи легкий доступ до функцій зберігання, організації та пошуку файлів, включно із семантичним пошуком.

1.4 Технології розробки системи

JavaScript [2] виступає базовою мовою для серверної та браузерної частин системи. Її популярність пояснюється універсальністю, широкою екосистемою та здатністю працювати як у браузері, так і в серверному середовищі.

Важливою характеристикою JavaScript є її асинхронна модель виконання, що реалізована через механізм Event Loop. Він обробляє події та callback-функції у циклі, дозволяючи виконувати операції вводу/виводу, мережеві запити й оновлення інтерфейсу без блокування основного потоку. Це дозволяє системі обробляти численні користувацькі дії – пошук, перегляд файлів, взаємодію із семантичними запитами – без затримок.

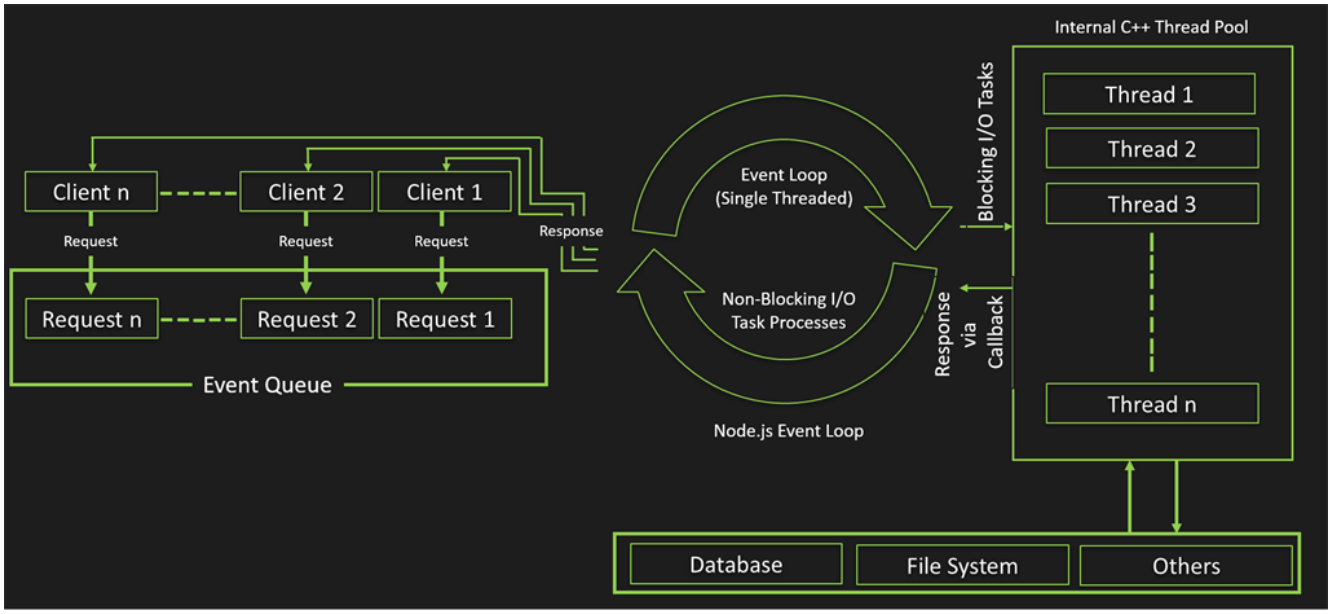

Рисунок 1.3 – Діаграма роботи механізму «Event Loop»

Event Loop також використовується в Node.js [4], де забезпечує паралельну обробку HTTP-запитів та фонового виконання задач. Така особливість є критично важливою для нашої системи, оскільки дозволяє одночасно обробляти завантаження файлів, індексацію тексту й виконання пошукових запитів.

JavaScript [2] розвивається згідно зі стандартами ECMAScript, що забезпечує постійне вдосконалення мови та сумісність із сучасними підходами веброзробки.

TypeScript [3] – надбудова над JavaScript, що додала те, чого вебіндустрія довго потребувала – статичну типізацію.

Його переваги:

- помилки виявляються на етапі компіляції, а не у продакшені;
- код стає документованим самим собою;
- знижуються ризики рефакторингу великих модулів;
- структура відносин між сутностями системи стає прогнозованою.

Модель взаємодії клієнтської частини, бекенд-модулів, сервісів семантичного пошуку та доступу до ElasticSearch [16] реалізована у TypeScript [3], що дозволяє:

- визначати контракти API;
- задавати структури embeddings;

- спрощувати фільтрацію та валідацію даних.

TypeScript [3] є важливим елементом підтримуваності і масштабованості нової версії системи.

Node.js [4] – середовище виконання JavaScript коду за межами браузера, створене на основі V8 (двигуна Google Chrome). Особливістю Node.js [4] є:

- той самий Event Loop, який дозволяє виконувати неблокуючі операції;
- висока продуктивність при великій кількості одночасних підключень;
- низькі витрати пам’яті.

Node.js [4] увійшов у проєкт як основа серверної частини, бо:

- підтримує REST-комунікацію [5];
- дозволяє інтегруватися з ElasticSearch [16] через офіційний SDK;
- ефективно обробляє асинхронні задачі – індексацію документів, формування embeddings і обробку пошукових запитів.

Його роль у системі – серверний центр управління, що координує взаємодію між користувачем, базою даних, сховищем файлів і сервісом пошуку.

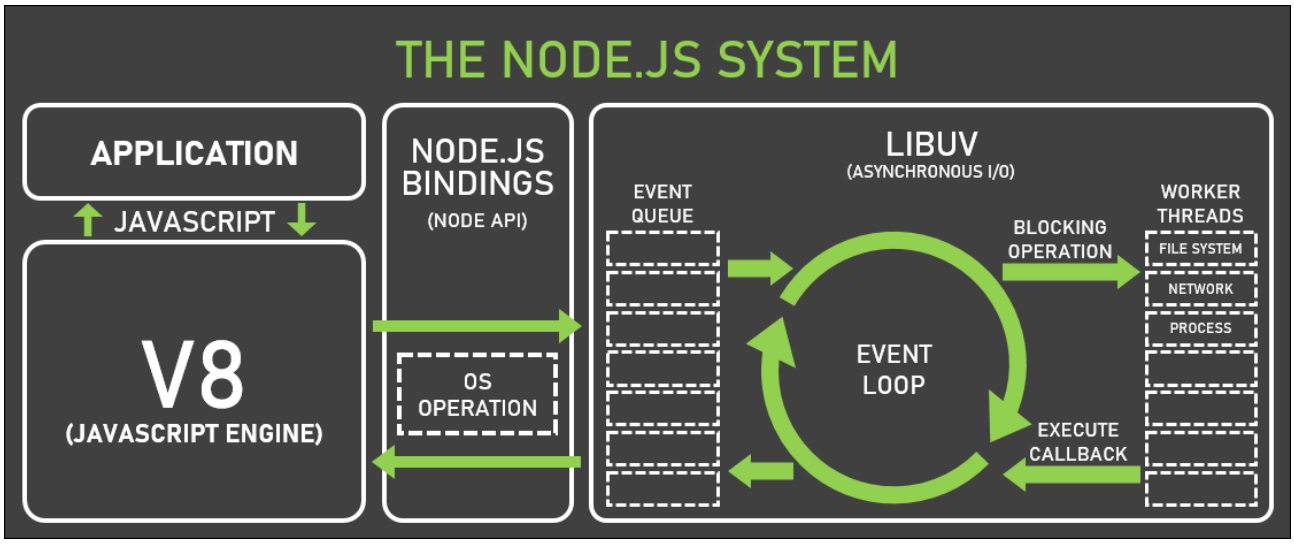

Рисунок 1.4 – Система платформи Node.js [4]

REST (Representational State Transfer) [5] обрано як архітектурний стиль взаємодії між клієнтською і серверною частинами.

Причини:

- простота інтеграції з UI та зовнішніми системами;
- використання стандартних HTTP-методів (GET, POST, PUT, DELETE);
- легкість масштабування.

REST API [5] дає змогу інтегрувати:

- NestJS-сервер [6],
- індексаційний сервіс ElasticSearch [16],
- embeddings-сервіс OpenAI [17],
- клієнтський інтерфейс Next.js [11].

Це створює розподілену архітектуру, у якій можна додавати нові модулі – наприклад, OCR-розпізнавання або аналіз зображень без зміни ядра системи.

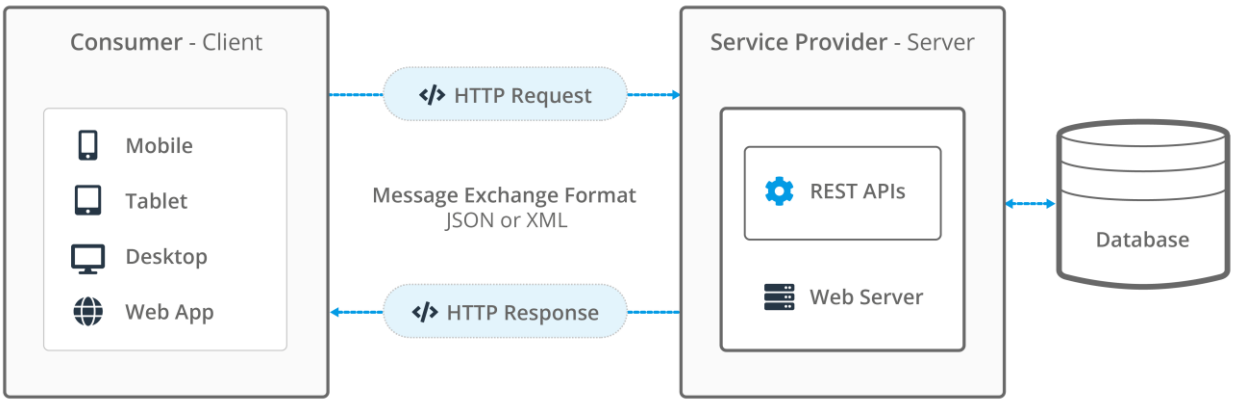

Рисунок 1.5 – Архітектура REST API [5]

NestJS – серверний фреймворк, побудований на патерні інверсії управління. Його переваги:

- чітке розділення модулів (auth, файли, пошук, indexing service);
- легке тестування;
- високий рівень структурованості.

Особливо корисним у цій роботі NestJS [6] став тому, що:

- дозволив інкапсулювати семантичний пошук у вигляді сервісу,

- створив окремі модулі для роботи з embedding-моделями, зберігання векторів у ElasticSearch [16] і текстової індексації.

Таким чином NestJS [6] формує архітектурний каркас, на якому розгортається логіка системи.

PostgreSQL [8] – об’єктно-реляційна СУБД з відкритим кодом.

Вона забезпечує:

- транзакційність;
- цілісність даних;
- підтримку складних типів;
- масштабованість.

У проєкті PostgreSQL виконує роль:

- сховища користувачів, прав доступу, метаданих файлів і записів логування;
- зберігає зв’язки з документами, індексованими у ElasticSearch.

Для роботи з PostgreSQL [8] застосовано ORM-бібліотеку TypeORM [9], що дозволяє:

- працювати з таблицями через класи моделей;
- використовувати міграції;
- описувати бізнес-логіку на рівні сутностей.

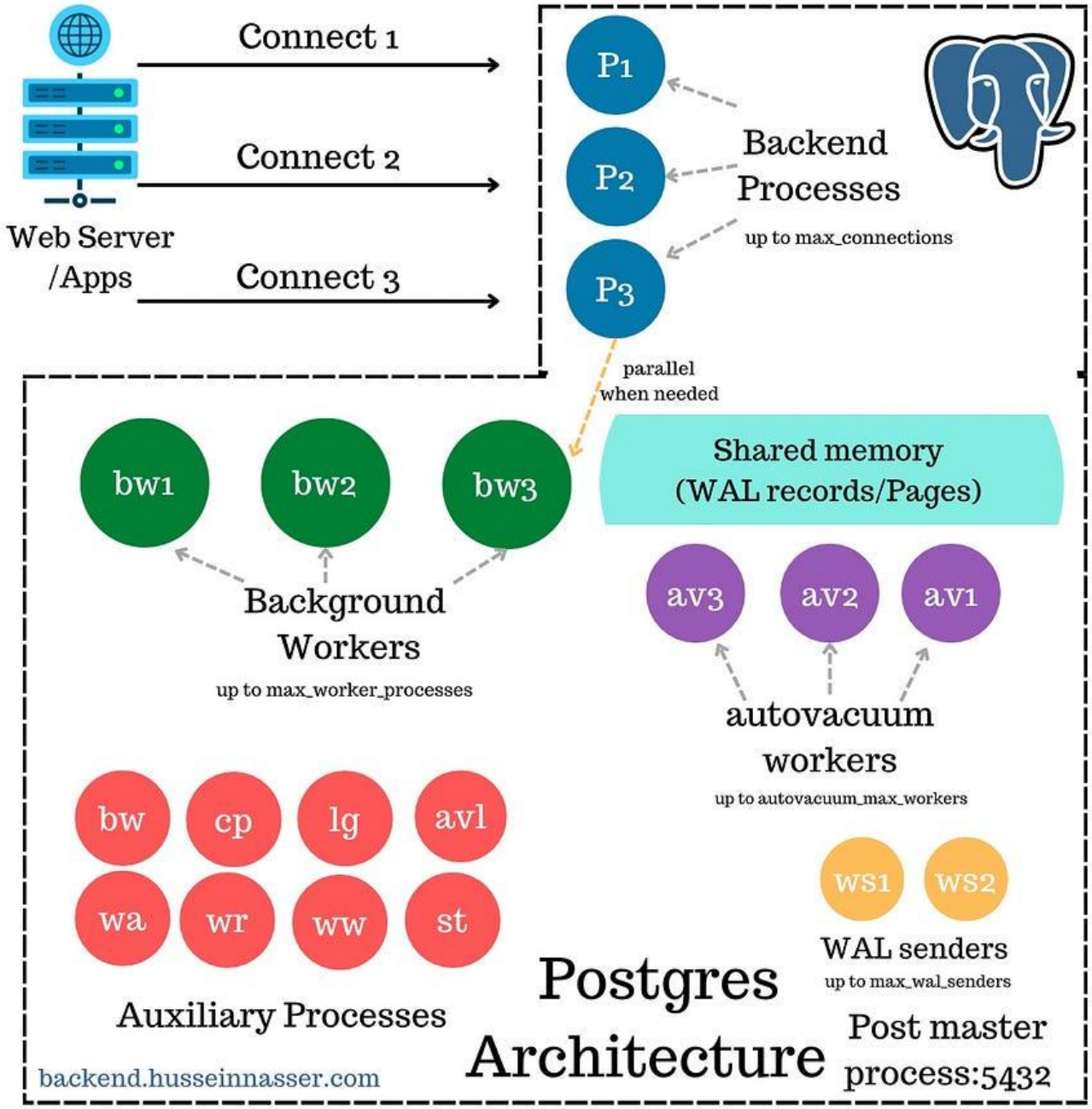

Рисунок 1.6 – Архітектура процесів PostgreSQL [8]

Для реалізації взаємодії з PostgreSQL [8] було обрано ORM-бібліотеку TypeORM [9]. ORM підхід дозволяє працювати з даними як із об’єктами, а не через SQL-інструкції, що:

- спрощує розробку;
- зменшує кількість помилок у запитах;
- підвищує підтримуваність кодової бази.

TypeORM [9] забезпечує:

- мапінг сутностей на таблиці – кожен клас відповідає структурі таблиці.
- механізм міграцій, що дозволяє версійно змінювати структуру бази даних без втрати даних.
- підтримку зв’язків (One-to-Many, Many-to-Many), що важливо для моделювання користувачів, файлів та структури каталогів.
- транзакційну роботу, необхідну для операцій, що змінюють стан системи (наприклад, переміщення файлів чи редагування метаданих).

TypeORM [9] застосовано також для зберігання записів про результати індексації документів, метаданих файлів і прив’язки між даними у PostgreSQL та записами в ElasticSearch [16]. Це дає змогу підтримувати єдине джерело правди у системі.

ElasticSearch [16] – це розподілений пошуковий рушій, створений для:

- аналізу великих обсягів текстів;
- швидкого знаходження інформації;
- побудови складних систем пошуку.

Його ключові можливості для цієї системи:

- інвертовані індекси для класичного пошуку;
- алгоритм BM25 для текстового ранжування;
- підтримка векторних полів і k-nearest-neighbors (kNN) пошуку;
- API для комбінованих пошукових стратегій.

ElasticSearch [16] дозволяє виконувати гібридний пошук, де результат ранжується як:

- за збігом ключових слів,
- за векторною близькістю embeddings.

Це робить його вдалим інструментом для семантичних систем.

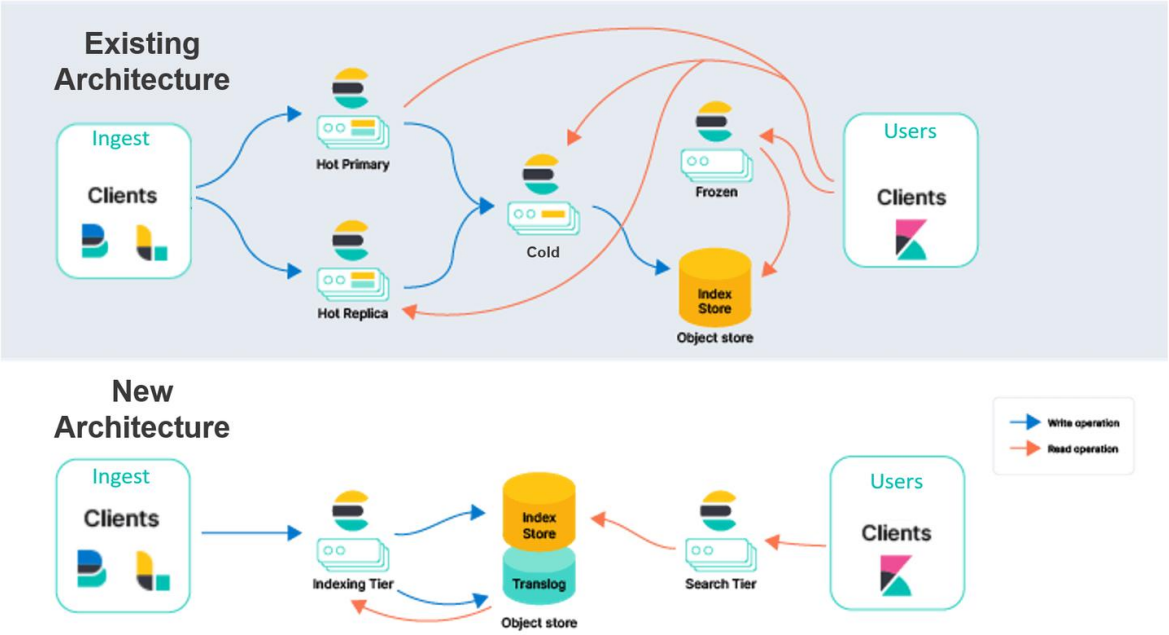

Рисунок 1.7 – Пошук за допомогою ElasticSearch [16]

Щоб система могла шукати файли за змістовою подібністю, текст потрібно перетворити на числа. OpenAI embeddings [17] саме це й роблять – генерують векторне представлення значення тексту.

Навіщо це потрібно?

- Якщо в документі написано “The car accelerated to 120 km/h”, а користувач шукає “vehicle movement”, класичний пошук не знайде збігів.
- Embeddings дозволяють системі зрозуміти, що ці слова семантично близькі.

У проєкті embeddings використовуються для:

- індексації змісту файлів у ElasticSearch;
- оцінки схожості документів;
- побудови гібридного пошуку (BM25 + kNN);
- реалізації “інтелектуального доступу” до даних.

Embeddings формують новий якісний рівень системи – від простого сховища файлів до системи знань, яка розуміє зміст.

Docker [10] – ключовий інструмент контейнеризації.

Його використання дозволяє:

- ізолювати модулі (PostgreSQL, ElasticSearch, Node.js-API);
- запускати систему в однаковому середовищі незалежно від комп'ютера розробника;
- масштабувати сервіси окремо.

Docker особливо важливий для розгортання ElasticSearch, який потребує стабільного середовища та доступу до конфігурацій, що легко реалізується через контейнери.

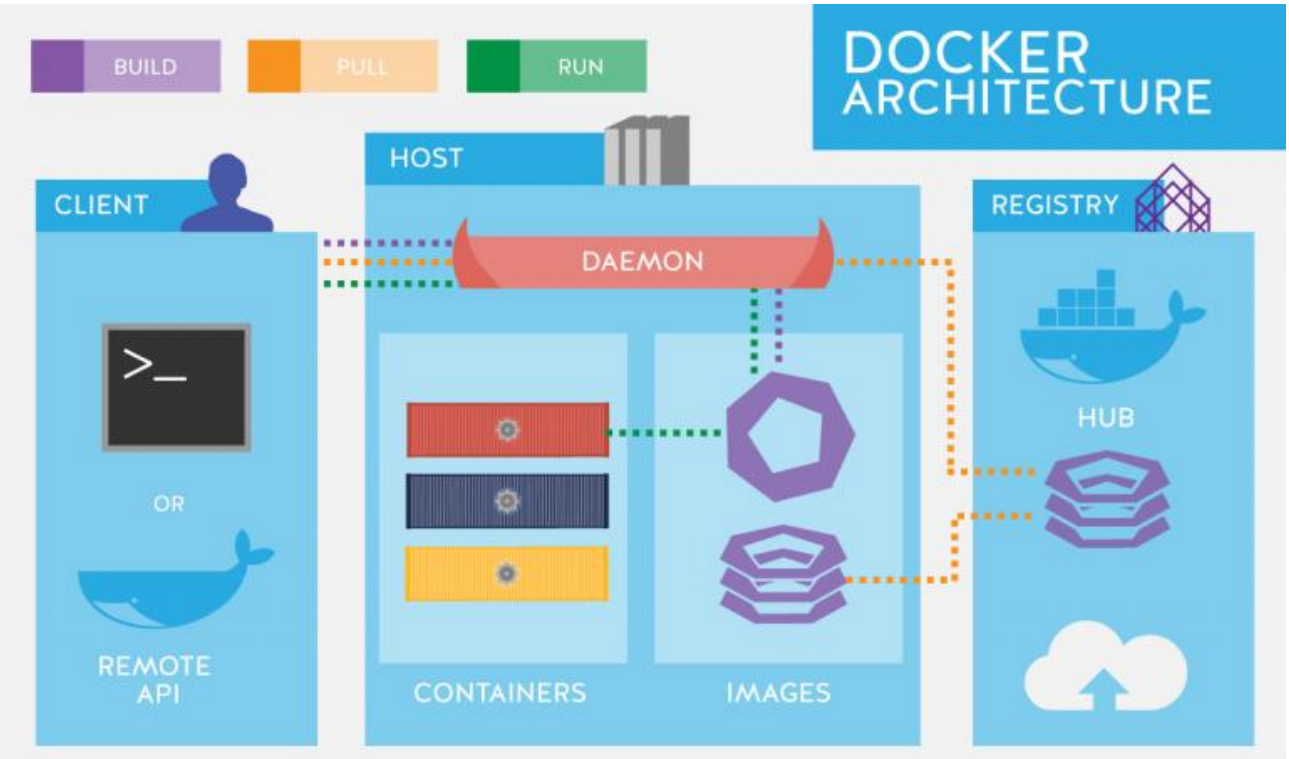

Рисунок 1.8 – Архітектура Docker [10]

Оскільки система реалізує семантичний пошук, перш ніж будувати embeddings — потрібно видобути текстовий зміст документів. Для цього використано набір вузькоспеціалізованих бібліотек.

pdf-parse — легковагова бібліотека, що дозволяє видобувати текст із PDF-документів.

Її переваги:

- робота без встановлення зовнішніх двигунів (на відміну від poppler);
- підтримка багато-сторінкових документів;

- отримання тексту разом із базовими метаданими (розмір, кількість сторінок тощо).

pdf-parse використано для:

- аналізу документів перед embeddings-індексацією;
- попереднього перегляду PDF у інтерфейсі;
- генерації текстового представлення для пошуку.

Для файлів Microsoft Word (.docx) застосовано бібліотеку Mammoth, яка:

- перетворює Word-документи у структурований текст;
- корректно витягує форматування;
- працює швидше та точніше, ніж підхід DOCX → PDF → OCR.

Вона інтегрована як окремий “reader” у сервіс індексації, що дозволяє автоматично отримувати зміст документів Word для семантичної обробки.

Tesseract — відкрита OCR-система (optical character recognition), яка здатна розпізнавати текст на фотографіях, сканах і PDF-файлах без текстового шару.

У системі Tesseract використовується для:

- індексації тексту у сканованих PDF;
- витягу зображень, що містять документи (скани договорів, актів тощо);
- емантичного пошуку за текстами, що існують лише у графічному вигляді.

Процес виглядає так:

Зображення → OCR → текст → embeddings → індекс у ElasticSearch.

Після витягування тексту з файлів за допомогою pdf-parse, Mammoth або Tesseract – він:

1. нормалізується (очищення, токенізація);
2. перетворюється на embeddings моделлю OpenAI;
3. додається в ElasticSearch як векторне та текстове поле;
4. бере участь у гібридному BM25+vector пошуку.

Таким чином, бібліотеки аналізу файлів виступають обов’язковим попереднім шаром RAG-архітектури, без якого семантичний пошук був би неможливим.

Клієнтська частина системи реалізована на Next.js [11], який поєднує SSR і SPA-модель.

React відповідає за:

- компоненти інтерфейсу;
- реактивну взаємодію;
- стан файлової системи.

Ant Design забезпечує:

- якісну UI-бібліотеку;
- відображення файлових дерев, індексів пошуку та модальних форм;
- візуалізацію результатів семантичного пошуку.

Результат — інтерфейс, де користувач швидко:

- навігує каталогами,
- знаходить файли за змістом,
- переглядає документи без завантаження.

Для тестування було обрано Jest — надійну та високопродуктивну бібліотеку, що підвищує якість коду та знижує кількість помилок у процесі розробки.

Ключові Переваги Jest [12]:

- Універсальність**:** Jest [12] підтримує широкий спектр типів тестування, включно з модульними, інтеграційними та кінцевими (end-to-end), дозволяючи всебічно перевіряти функціональність застосунку.
- Висока Швидкість**:** Бібліотека забезпечує ефективне виконання тестів завдяки паралельному запуску та кешування результатів.
- Зручність Відладки**:** Jest пропонує ефективні засоби відладки та аналізу тестів, що значно прискорює процес виявлення та виправлення несправностей.

Використання Jest гарантує стабільність і надійність застосунку, мінімізуючи експлуатаційні ризики [12].

Під час конструювання системи було використано широкий спектр технологій, що працюють у взаємозв’язку та формують цілісну архітектуру системи. На рівні виконання бізнес-логіки використовується стек JavaScript/TypeScript [2, 3] із Node.js [4] та NestJS [4], що забезпечує модульність,

асинхронність та високу продуктивність серверної частини. PostgreSQL у комбінації з TypeORM виступає надійним сховищем структурованих даних, тоді як Docker формує ізольоване та стабільне середовище розгортання. На клієнтському рівні Next.js, React та Ant Design забезпечують динамічний інтерфейс для роботи з файлами та пошуком, пропонуючи користувачеві швидку й зручну взаємодію з інформаційним простором системи.

Окрему роль відіграють технології аналізу вмісту та семантичного пошуку. ElasticSearch реалізує традиційні та гібридні моделі пошуку, тоді як embeddings OpenAI дозволяють системі інтерпретувати зміст документів й виконувати пошук за смисловою подібністю. Інструменти витягання тексту, такі як pdf-parse, Mammoth та Tesseract, створюють міст між файлами та їхнім семантичним представленням, що дозволяє вводити неструктуровані дані у модель пошуку. У сукупності ці технології трансформують просте файлове сховище у вебсистему з інтелектуальними властивостями, здатну не лише зберігати інформацію, а й опрацьовувати її залежно від змісту, забезпечуючи ефективний доступ до знань.

РОЗДІЛ 2: АРХІТЕКТУРНЕ ПРОЄКТУВАННЯ ТА СТРУКТУРНА ОРГАНІЗАЦІЯ СИСТЕМИ

У цьому розділі розглядається архітектурне проєктування вебсистеми централізованого зберігання та доступу до файлів, а також структурна організація її основних компонентів. Описується загальна архітектура системи, взаємодія між клієнтською та серверною частинами, а також принципи побудови внутрішньої структури програмного забезпечення. Особлива увага приділяється проєктуванню сутностей бази даних, визначенню зв’язків між ними та формуванню логічної моделі даних, що забезпечує цілісність, масштабованість і зручність подальшого розвитку системи.

2.1 Моделювання та характеристика сутностей системи

Під час проектування вебсистеми централізованого керування файлами одним із визначальних етапів стало проєктування моделі даних. Саме схема бази даних оприділяє метод організації, структурування і взаємодії інформації між окремими компонентами системи. Від її побудови залежить можливість ефективного зберігання документів різних форматів, підтримка масштабування, контроль прав доступу та забезпечення цілісності даних. У даному розділі розглядається логіка моделювання сутностей, аргументація обраної структури даних і способи формування зв’язків між елементами системи. Такий підхід дозволяє сформувати стійку основу для роботи з файлами та впровадження механізмів пошуку.

Схематичне відображення сутностей реалізовано у вигляді ORM-моделей, створених завдяки бібліотеці TypeORM [9]. Для прикладу, модель UserEntity, що відповідає таблиці users, відображає основні характеристики користувача системи. До її складу входять унікальний ідентифікатор, електронна пошта, хешований пароль, повне ім’я користувача та зв’язок один до одного із сутністю DriveEntity, що репрезентує персональне файлове сховище власника. Така структура дозволяє

пов’язати обліковий запис із його файловим простором і забезпечити коректний розподіл доступів при взаємодії із системою.

Лістинг 2.1 – Модель сутності користувача (UserEntity), реалізована з використанням TypeORM [9]

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

  @Column()
  email: string;

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

  @Column()
  fullName: string;

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

DriveEntity відповідає структурі таблиці *drives* у базі даних, використовується з метою відображення персонального файлового простору користувача. Ця сутність містить унікальний ідентифікатор у форматі UUID, а також назву сховища, яке асоціюється з конкретним профілем користувача. Через зв’язок із моделлю UserEntity встановлюється належність диска певному користувачу, що дозволяє забезпечити логічне розмежування даних між різними обліковими записами. Окрім цього, DriveEntity пов’язана з сутністю FolderEntity, яка виконує роль кореневого каталогу й визначає ієрархічну структуру всієї файлової системи, у межах якої розміщуються підпапки та документи.

Лістинг 2.2 – Модель сутності сховища (DriveEntity), реалізована з використанням TypeORM [9]

```
@Entity()
export class DriveEntity {
  @PrimaryGeneratedColumn('uuid')
```

```
  id: string;

  @Column()
  name: string;

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

  @OneToOne(() => FolderEntity)
  @JoinColumn()
  rootFolder: FolderEntity;

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

FolderEntity відображає таблицю folders, описує каталог, який є структурним елементом файлової системи. До складу цієї сутності входять унікальний ідентифікатор, ім’я папки, початкове найменування, часові позначки створення, оновлення та можливого видалення, а також властивість, що визначає, чи виступає даний каталог кореневим у межах певного диска. Внутрішня ієрархія папок реалізована через рекурсивний зв’язок із батьківським елементом, що дозволяє відтворювати дерево каталогів будь-якої глибини. Кожна папка може містити підпапки та файли, що формує логічну структуру користувацького сховища. Додатково сутність пов’язана з власником, що встановлює належність каталогу конкретному користувачу, а також із моделлю FolderNoteEntity, яка забезпечує можливість додавання текстових приміток до папок як елементу метаінформації.

Лістинг 2.3 – Модель сутності папки (FolderEntity), реалізована з використанням TypeORM [9]

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

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

```
  @Column()
  originalFolderName: string;

  @Column({ nullable: false, type: 'text' })
  folderPath: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @DeleteDateColumn()
  deletedAt?: Date;

  @Column()
  isDriveRoot: boolean = false;
  @Column({ default: false })
  isDriveTrashFolder: boolean = false;
  @ManyToOne(() => FolderEntity, (folder) => folder.children, {
    nullable: true,
  })
  @JoinColumn()
  parent?: FolderEntity;

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

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

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

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

  @BeforeUpdate()
  updateFolderPath() {
    if (!this.parent) {
      return;
    }
    this.folderPath = path.join(
      this.parent.folderPath,
      this.originalFolderName,
    );
  }
}
```

FileEntity відображає таблицю files, описує файловий ресурс, збережений у системі. Дана сутність включає унікальний ідентифікатор, ім’я, вихідне найменування, розмір, MIME-тип, а також часові позначки створення, оновлення і можливого видалення. Така структура забезпечує можливість зберігати основні характеристики документа та контролювати його життєвий цикл. Файл пов’язується з певним каталогом за допомогою сутності FolderEntity, яка визначає його фізичне розміщення у файловій системі. Крім того, він асоційований із користувачем, який здійснив завантаження, що дозволяє ідентифікувати власника або автора ресурсу. До файлів також можуть додаватися примітки, що представлені моделлю нотаток, завдяки чому система підтримує зберігання додаткових метаданих, пов’язаних із визначеним документом.

Лістинг 2.4 – Частина моделі сутності файла (FileEntity), реалізована з використанням TypeORM [9]

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

  @Column()
  fileName: string;

  @Column()
  originalFileName: string;

  @Column({ nullable: false, type: 'text' })
  filePath: string;

  @Column()
  size: number;

  @Column()
  mimeType: string;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @DeleteDateColumn()
```

```
    deletedAt?: Date;

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

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

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

    @BeforeUpdate()
    updateFilePath() {
      if (!this.folder) {
        return;
      }
      this.filePath = path.join(this.folder.folderPath,
this.originalFileName);
    }
  }
```

FileDownloadLinkEntity відображає таблицю file_download_links, використовується для моделювання механізму надання доступу до файлів через тимчасові посилання. Дана сутність містить унікальний ідентифікатор, токен, за яким здійснюється завантаження документу, а також часові позначки створення, оновлення та закінчення строку дії. Така структура дозволяє керувати життєвим циклом посилання і забезпечувати обмежений у часі доступ до файлу без необхідності автентифікації користувача.

Лістинг 2.5 – Модель сутності посилання для завантаження (FileDownloadLinkEntity), реалізована з використанням TypeORM [9]

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

  @Column()
  token: string;

  @CreateDateColumn()
```

```
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;

  @Column()
  expiresAt: Date;

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

Note – вбудована сутність застосовується як узагальнений елемент для моделей FileNoteEntity та FolderNoteEntity, описуючи основні властивості текстових приміток. До її складу входять заголовок, зміст, а також часові позначки створення і оновлення. Використання такої спільної структури дозволяє уникати дублювання коду і забезпечує повторне застосування логіки формування нотаток у різних частинах системи, незалежно від того, чи додається примітка до каталогу, чи до конкретного файлу.

Лістинг 2.6 – Модель вбудованої сутності Note, реалізована з використанням TypeORM [9]

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

  @Column()
  content: string;

  @CreateDateColumn()
  createdAt: Date;

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

FileNoteEntity відповідає таблиці file_notes, використовується для зберігання приміток, пов’язаних із конкретним файлом. Дана сутність містить унікальний ідентифікатор та вбудовану структуру Note, яка описує заголовок, зміст та часові

позначки створення й оновлення запису. Через асоціацію з FileEntity визначається файл, якому належить дана нотатка, що дозволяє доповнювати документи контекстною інформацією або коментарями.

Лістинг 2.7 – Модель сутності нотатки файла (FileNoteEntity), реалізована з використанням TypeORM [9]

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

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

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

FolderNoteEntity відображає таблицю folder_notes, використовується для зберігання приміток, пов’язаних із каталогами файлової системи. Дана сутність містить унікальний ідентифікатор та вбудовану структуру Note, що включає заголовок, текстове наповнення та часові характеристики створення й оновлення запису. Через зв’язок із FolderEntity визначається папка, до якої належить певна примітка, що дозволяє доповнювати каталоги додатковою інформацією та підтримувати можливість створення кількох коментарів для одного розділу.

Лістинг 2.8 – Модель сутності нотатки каталогу (FolderNoteEntity), реалізована з використанням TypeORM [9]

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

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

  @ManyToOne(() => FolderEntity, (folder) => folder.notes, {
```

```
    nullable: false,
  })
  folder: FolderEntity;
}
```

На основі аналізу вимог та логіки функціонування системи було сформовано набір програмних класів, що відображають структурні та поведінкові особливості її компонентів. Ці класи та їхні взаємозв’язки представлені у вигляді діаграми класів, яка слугує концептуальною моделлю внутрішньої архітектури програмного забезпечення. Діаграма надає візуальне відображення основних сутностей, їхніх атрибутів і залежностей між ними, демонструючи, як дані переходять між компонентами та які ролі виконують окремі елементи у системі. Кожен клас у цій моделі має власну відповідальність і містить властивості та методи, необхідні для реалізації конкретного функціоналу — починаючи від обробки користувацьких даних і управління файлами, закінчуючи підтримкою ієрархії папок та механізмів доступу. Такий підхід дозволяє забезпечити структурованість, розширюваність і зрозумілість програмного коду, а також створює підґрунтя для подальшої реалізації, тестування та масштабування системи.

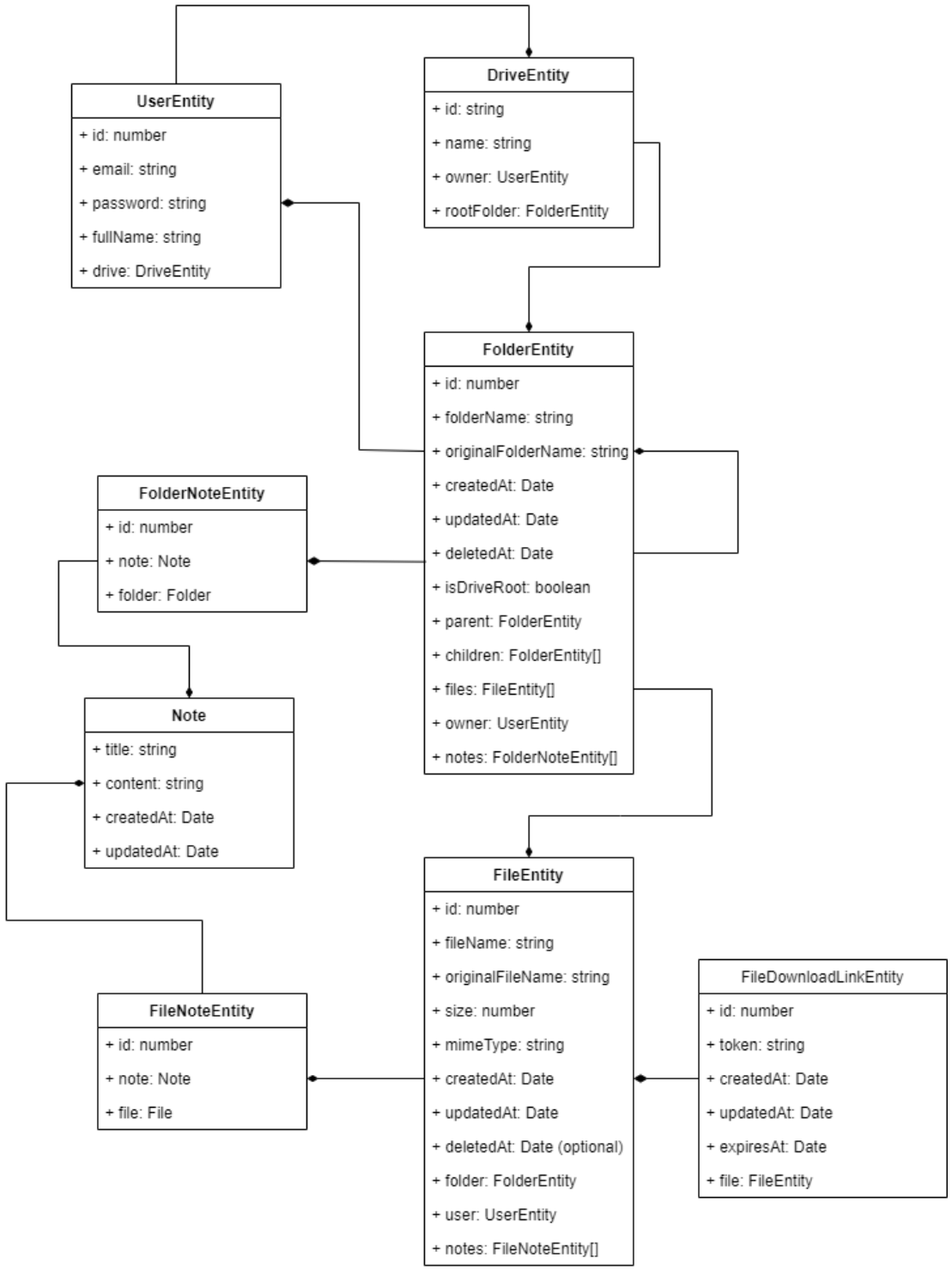

Рисунок 2.1 – UML діаграма зпроектованої системи

Головні сутності системи формують логічне ядро її архітектури, визначаючи структуру зберігання даних та правила взаємодії між ними. На рівні моделі даних виділено чотири базові класи. UserEntity містить облікову інформацію та визначає права доступу, формуючи точку входу користувача в систему. DriveEntity виступає персональним файловим простором, що належить

конкретному користувачу, і забезпечує відокремленість даних між акаунтами. Власне організація файлового дерева реалізована через FolderEntity, яке описує ієрархію каталогів і дозволяє будувати вкладені структури різної складності. FileEntity забезпечує зберігання метаданих файлів, шляхів доступу та логічних прив’язок до папок.

Між класами існують композиційні зв’язки, що підкреслюють залежність частин системи одна від одної. Наприклад, сховище не може існувати без його власника, папка не функціонує поза межами диска, а файл завжди є елементом певної папки. Така модель забезпечує цілісність та консистентність даних, а також дозволяє легко виконувати каскадні операції – наприклад, видалення каталогу разом з усіма вкладеними файлами.

Додатково сутності інтегруються із пошуковим індексом та механізмами семантичної інтерпретації. FileEntity містить зв’язки з даними індексації, включно з текстовим представленням і embeddings, що виступають базою для семантичного пошуку в ElasticSearch. Це дозволяє не лише зберігати файли, а й забезпечувати змістовний доступ до їхнього вмісту. У свою чергу FolderEntity використовується для групування результатів пошуку та побудови навігації у клієнтському інтерфейсі, що реалізовано у Next.js.

Завдяки такій структурі система підтримує не лише фізичну організацію сховища, але й семантичне розуміння його складових. Модель даних виступає фундаментом, на якому реалізовано авторизацію, управління файлами, індексацію та алгоритми пошуку. Це підкреслює її ключове значення – правильна побудова сутностей визначає гнучкість системи, простоту її масштабування та можливість розширення новими функціями без суттєвого втручання в ядро.

2.2 Визначення зв’язків між сутностями системи

Модель даних розробленої вебсистеми базується на низці сутностей, що відображають логічну структуру сховища файлів та їхню взаємодію. Сутності виконують роль абстракцій реальних об’єктів — користувача, дискового простору,

папки та файлу — і формують основу для побудови функціональності системи. Для повноцінного управління інформацією важливо не лише визначити ці об’єкти, але й описати зв’язки між ними, оскільки саме вони забезпечують узгодженість даних, описують залежності й дозволяють реалізовувати каскадну поведінку у процесі модифікації інформації.

У реляційних базах даних зв’язки визначають спосіб співіснування сутностей і вказують, яким чином записи однієї таблиці співвідносяться із записами іншої. Для системи централізованого зберігання файлів найбільш характерним є зв’язок «один до багатьох», де один користувач може мати кілька дискових сховищ, декілька папок та великий набір файлів. Така модель дозволяє описувати ієрархічну структуру файлового простору, де користувач є власником ресурсів, а система може легко відстежувати належність даних конкретному обліковому запису. Подібний тип зв’язку застосовується також між диском і папками чи папкою та файлами — один диск може містити безліч каталогів, а кожний каталог може включати необмежену кількість файлів.

Усередині системи також використовується композиційний зв’язок між сутностями, що вказує на залежність існування одного об’єкта від іншого. Наприклад, диск не може існувати без власника-користувача, а папка або файл не можуть бути створені поза межами сховища. Це забезпечує узгодженість даних і дозволяє реалізовувати каскадні операції — видалення користувача веде до видалення його дисків, а видалення папки — до очищення всіх вкладених файлів. Така модель спрощує керування ресурсами і виключає появу «осиротілих» записів.

Система семантичного пошуку, інтегрована у вебплатформу, доповнює традиційну модель даних власними сутностями. З FileEntity пов’язано представлення документа у пошуковому індексі, включно з витягнутим текстом та embeddings — числовими векторами, що описують зміст файлу. Хоча їх фізичне зберігання здійснюється в ElasticSearch, у реляційній базі фіксується логічний зв’язок між файлом і його індексом, що забезпечує консистентність та дозволяє повторно створювати семантичний опис при зміні документа.

Таким чином, модель сутностей не лише визначає структуру сховища, а й формує системні правила взаємодії між компонентами. Завдяки чітким зв’язкам база даних підтримує цілісність інформації, спрощує навігацію, забезпечує коректне виконання операцій над файлами та дозволяє розширювати систему новою функціональністю — зокрема, механізмами семантичного пошуку — без порушення узгодженості її ядра.

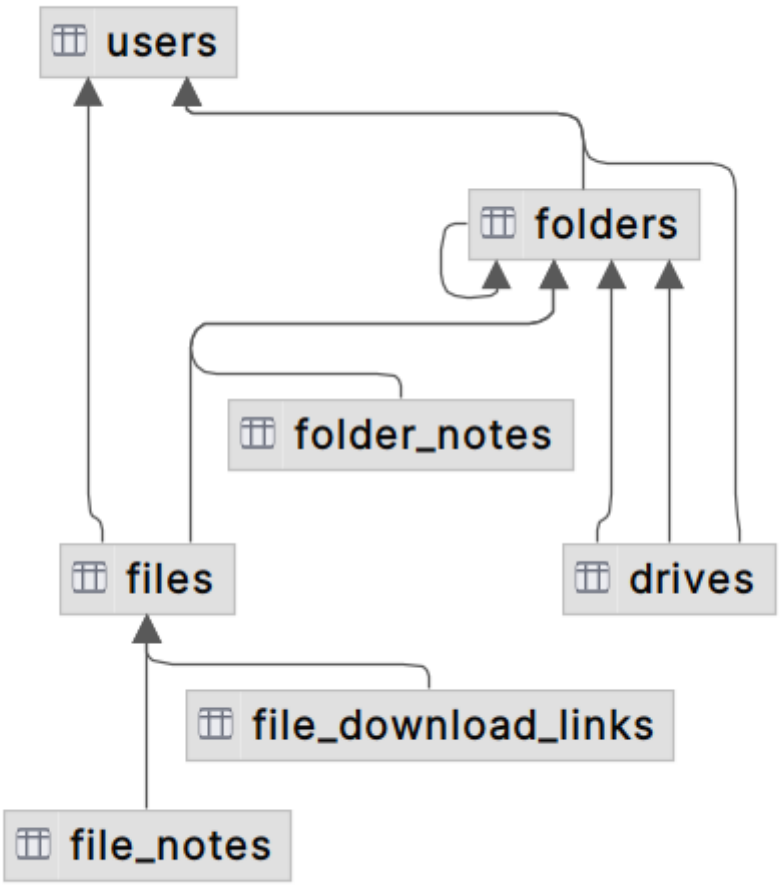

Рисунок 2.2 – Концептуальна модель бази даних системи

Користувач і його персональне сховище (диск) формують одну з базових пар взаємопов’язаних сутностей системи. Між ними встановлено зв’язок типу «один до одного», що означає: кожен користувач володіє лише одним дисковим простором, і жоден диск не може декільком обліковим записам. Така модель забезпечує

однозначну належність даних, спрощує контроль доступу та дозволяє ізолювати файлові ресурси між різними користувачами, гарантувавши логічне розмежування та захищеність їхнього вмісту.

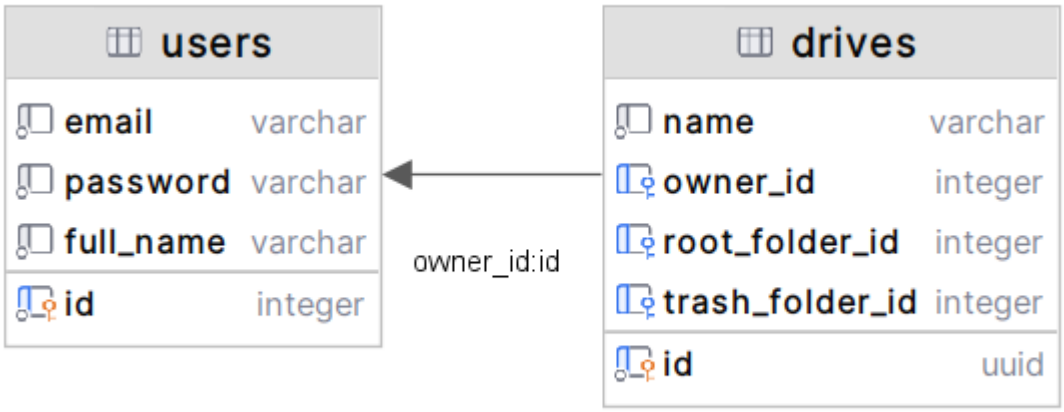

Рисунок 2.3 – Зв’язок між користувачами та дисками (users - drives)

Взаємозв’язок між дисковим простором та каталогами має розширену структуру, що включає два ключові типи папок: кореневий каталог та каталог-кошик. Кожен диск має одну кореневу папку, яка формує вихідну точку файлової ієрархії та містить у собі всі інші каталоги й файли. Окрім цього, диск містить окрему папку-кошик, яка призначена для тимчасового зберігання видалених елементів і забезпечує механізм відновлення або остаточного очищення даних. Оба посилання реалізовані через відношення «один до одного», що гарантує унікальність кореневого каталогу та кошика для кожного сховища.

Усі інші каталоги системи будуються за принципом деревоподібної структури: кожна папка (крім кореневої та кошика) містить посилання на свій батьківський каталог. Завдяки цьому формується вкладена ієрархія з можливістю навігації, групування та каскадних операцій — видалення каталогу автоматично охоплює його вміст, а переміщення у кошик зберігає логічну структуру даних. Така модель дозволяє одночасно мати впорядковане середовище зберігання та механізм безпечного видалення, без втрати інформаційної цілісності та ієрархії.

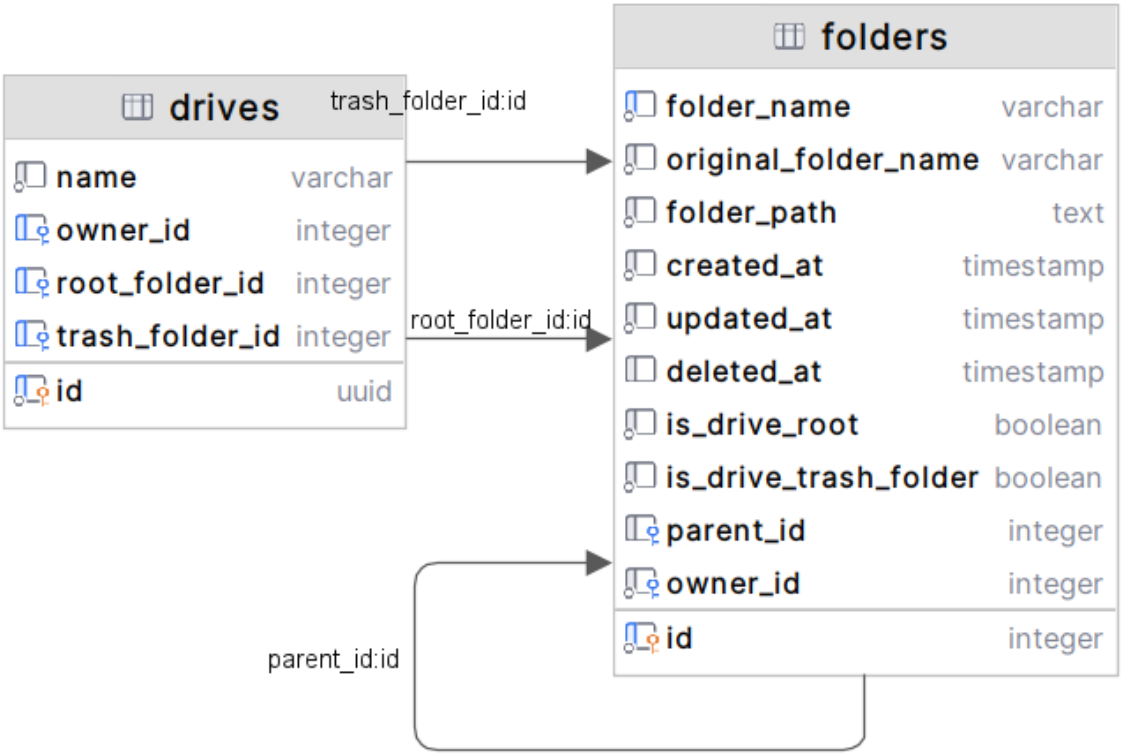

Рисунок 2.4 – Зв’язок між каталогами та дисками (folders - drives)

Зв’язок між каталогами та файлами реалізовано за принципом «один до багатьох», де кожна папка може містити множину файлів, тоді як окремий файл належить лише одному каталогу. Така модель формує логічну структуру зберігання, у якій файли групуються за категоріями або змістовими ознаками, що забезпечує впорядкованість даних та спрощує навігацію в системі. Прив’язка файлу до конкретного каталогу також відіграє важливу роль у механізмах доступу, спадкових прав та каскадних операцій, наприклад при видаленні папки або переміщенні її в кошик.

У контексті семантичного пошуку даний зв’язок дозволяє не лише фізично впорядковувати файли, але й використовувати їхнє розташування як додаткову контекстну інформацію під час індексації. Таким чином, файл не просто існує у сховищі — він набуває значення в межах своєї ієрархії, що покращує можливість пошуку, фільтрації та відображення результатів у користувацькому інтерфейсі.

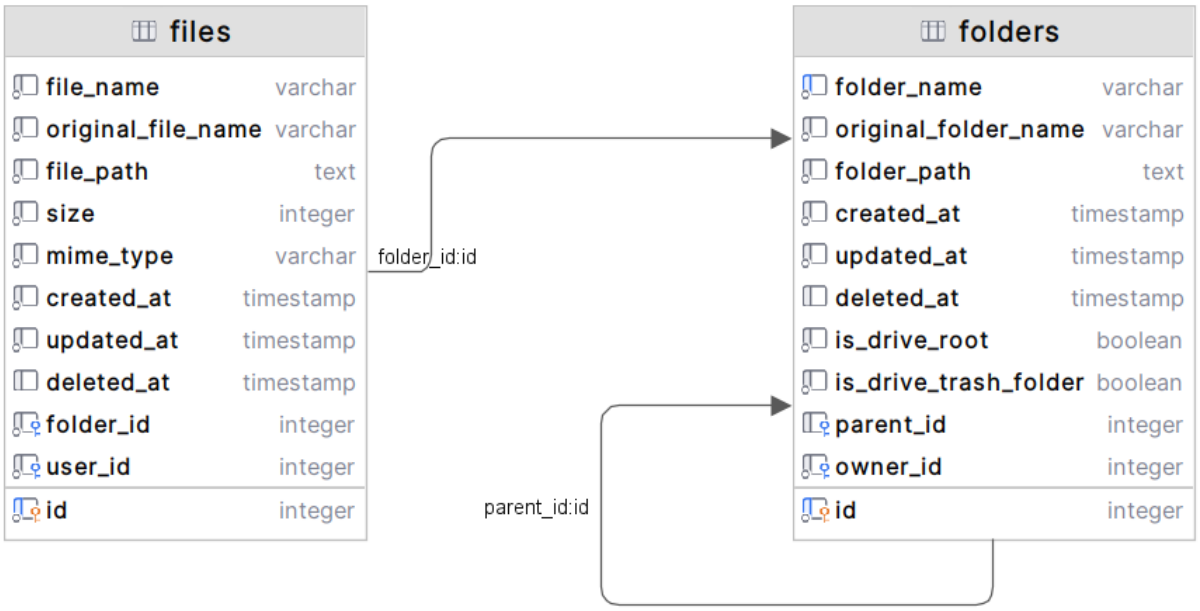

Рисунок 2.5 – Зв’язок між каталогами та файлами (folders - files)

Зв’язки між файлом та допоміжними сутностями — нотатками і посиланнями на завантаження — реалізовані за моделлю «один до багатьох». Це означає, що один файл може містити довільну кількість коментарів, пояснень чи службових записів, представлених у таблиці file_notes. Такий механізм дозволяє користувачам фіксувати додаткову інформацію, уточнення або нагадування, що підвищує інформативність та зручність роботи з документами. Кожна нотатка водночас має жорстку прив’язку лише до конкретного файлу, що забезпечує однозначність належності та виключає інформаційні конфлікти.

Аналогічна логіка застосовується для створення посилань на скачування: файл може мати кілька лінків, наприклад для одноразового доступу, публічного поширення або внутрішнього обміну, і ці лінки зберігаються у таблиці file_download_links. Кожне посилання існує лише в межах одного файлу, що дозволяє відслідковувати їх використання, обмежувати термін дії чи контролювати права доступу. Така модель взаємодії розширює можливості системи, створюючи інфраструктуру для співпраці та контролю над поширенням цифрових документів.

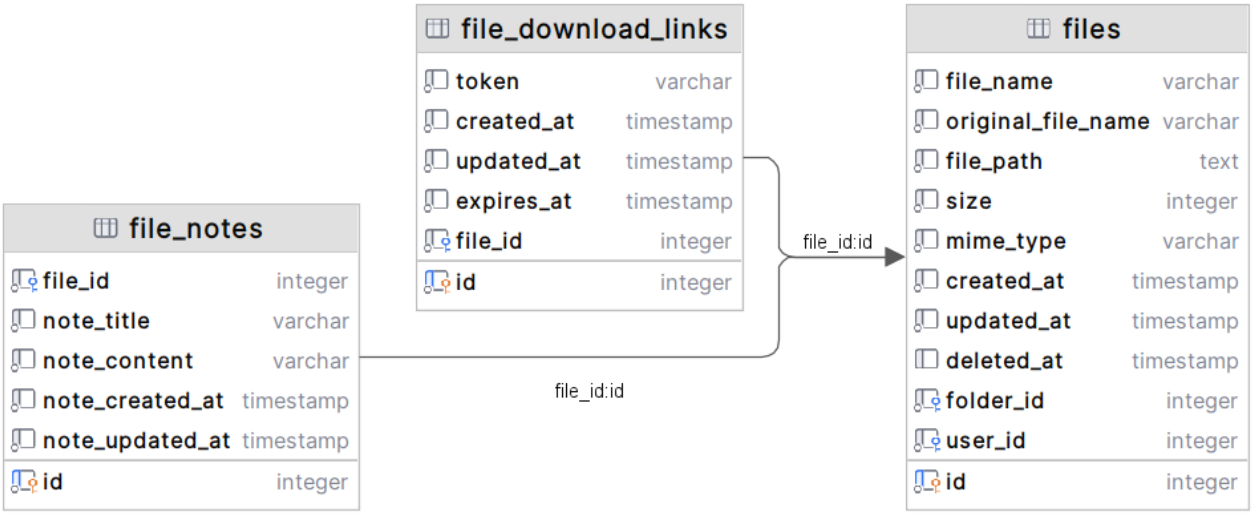

Рисунок 2.6 – Зв’язок між файлами та нотатками для файлів (files - file_notes) і файлами та посиланнями для завантаження файлів (file - file_download_links)

Взаємодія між папками та нотатками для них реалізована за принципом «один до багатьох». Окрема папка може містити множину коментарів або описових записів, що дає можливість користувачам фіксувати додаткову інформацію про її вміст чи призначення. Нотатка завжди належить лише одній папці, що забезпечує чітку структурованість і дозволяє уникнути неоднозначностей у визначенні контексту її використання. Такий зв’язок підвищує інформативність системи — папки можуть супроводжуватися службовими позначками, нагадуваннями або поясненнями, що спрощує навігацію в дереві каталогів та покращує управління ресурсами, особливо у великих файлових структурах.

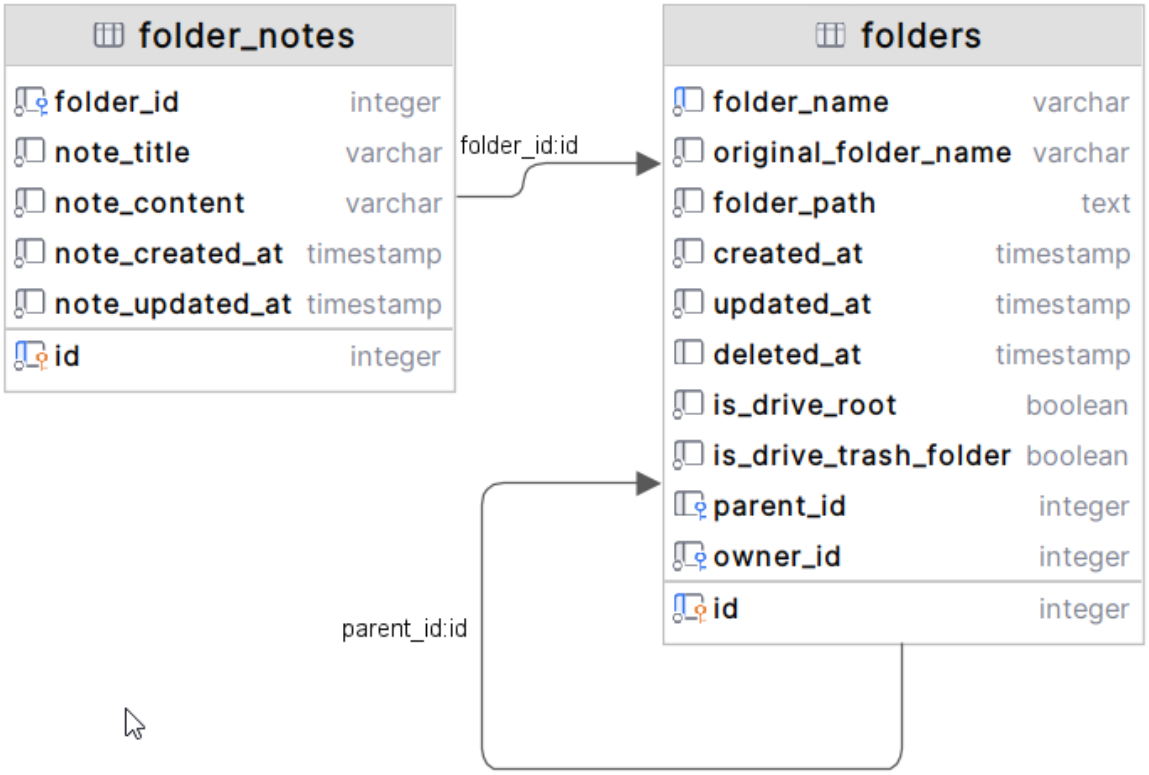

Рисунок 2.7 – Зв’язок між папками та нотатками для папок (folders - folder_notes)

Побудова схеми бази даних є ключовим етапом розробки системи, оскільки саме на цьому рівні визначається структура зберігання інформації, правила її узгодженості та механізми взаємодії між об’єктами. ER-діаграма, наведена у даному розділі, слугує візуальним відображенням концептуальної моделі даних і описує основні сутності системи — користувача, диск, папки, файли та пов’язані з ними допоміжні об’єкти, такі як нотатки й посилання на завантаження.

Діаграма демонструє не лише перелік сутностей, а й характер їх взаємозв’язків, що відображають логіку функціонування файлової платформи. Наприклад, зв’язок один до одного між користувачем і диском окреслює межі персонального сховища, а зв’язки один до багатьох між каталогами, файлами, нотатками та посиланнями дозволяють будувати ієрархію та розширювати інформаційний контекст кожного елемента. Таким чином, ER-модель виступає основою для реалізації бізнес-логіки системи, забезпечує можливість каскадних

операцій над даними, полегшує проєктування API та гарантує цілісність інформації незалежно від механізмів її обробки.

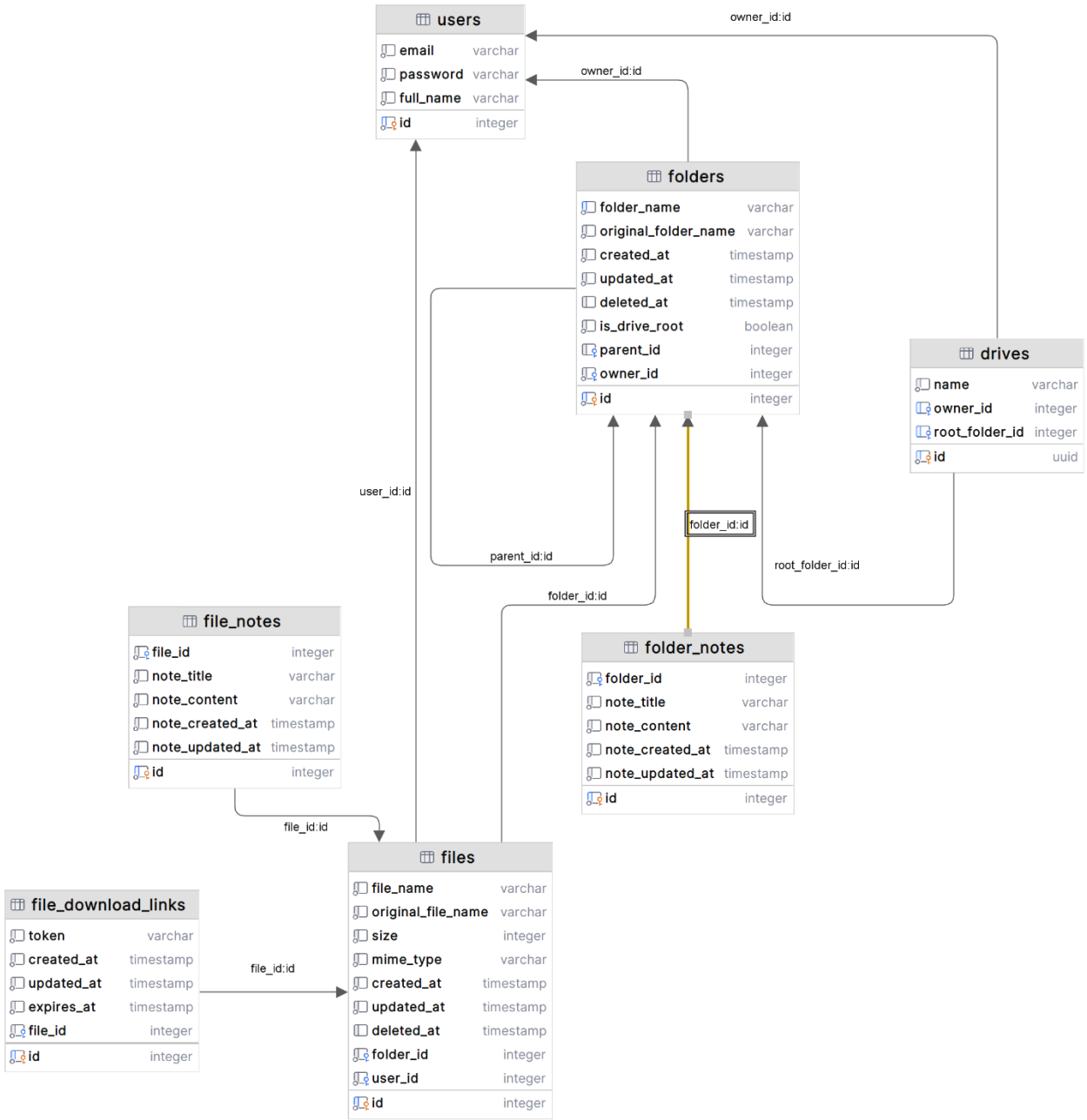

Рисунок 2.8 – ER-діаграма (Entity–Relationship) структури бази даних системи [14]

Entity-Relationship діаграма демонструє структуровану та узгоджену модель даних, яка забезпечує основу для реалізації всіх функціональних можливостей системи. Завдяки чітко визначеним сутностям і зв’язкам між ними база даних підтримує впорядковане зберігання документів і папок, гарантує належність інформації конкретним користувачам та дозволяє виконувати типові операції

керування файлами без ризику втрати цілісності. Така схема не лише оптимізує доступ до даних і їх навігацію, але й створює передумови для розширюваності системи — модель легко доповнювати новими типами сутностей, індексаційними механізмами чи сервісами семантичного пошуку без необхідності радикальної перебудови її логіки. Таким чином, ER-модель виступає фундаментом, який забезпечує стабільність, масштабованість і розвиток платформи у процесі її подальшої еволюції.

РОЗДІЛ 3: РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБСИСТЕМИ

У цьому розділі описано практичну реалізацію вебсистеми централізованого зберігання файлів із підтримкою семантичного пошуку, а також проведення її тестування та оцінку якості реалізації. Розглядаються ключові аспекти реалізації серверної та клієнтської частин системи, особливості впровадження основного функціоналу роботи з файлами, механізмів пошуку та взаємодії користувача з вебінтерфейсом. Окрему увагу приділено процесу тестування, що дозволяє перевірити коректність роботи системи, її стабільність, продуктивність і відповідність визначеним вимогам.

3.1 Програмна реалізація функціональних компонентів системи

Конструювання функціоналу системи централізованого зберігання файлів ґрунтується на використанні сучасного технологічного стеку, що забезпечує надійність та масштабованість рішення. Серверна частина побудована на платформі Node.js у поєднанні з NestJS, що дозволяє організувати модульну архітектуру та чітке розмежування бізнес-логіки. Робота з реляційною базою даних реалізована за допомогою TypeORM, який надає можливість керувати сутностями, зв’язками та міграціями на високому рівні абстракції. Браузерний інтерфейс створений з використанням Next.js і React, що забезпечує швидке рендерування, інтерактивність та зручну навігацію для користувача. Завдяки цьому система є легко підтримуваною, розширюваною та готовою до інтеграції додаткового функціоналу, такого як семантичний пошук на основі ElasticSearch та OpenAI Embeddings.

Одним із ключових етапів взаємодії користувача з системою є процес реєстрації. На цьому етапі здійснюється створення облікового запису користувача, збереження його основних даних, а також ініціалізація персонального файлового простору, який використовується як базове сховище для подальшого розміщення файлів і папок. Реєстрація включає генерацію унікального ідентифікатора,

хешування пароля та створення початкової структури директорій — зокрема кореневої папки користувача та папки-кошика для відновлення видалених даних. Такий підхід забезпечує логічну ізольованість файлових просторів різних користувачів та формує цілісну ієрархію, яка слугує основою для подальшої роботи з файлами. У цьому розділі наведено фрагменти коду, що ілюструють реалізацію реєстрації та демонструють принципи обробки вхідних даних, створення сутностей та взаємодії із базою даних.

Лістинг 3.1 – Метод контроллера для реєстрації користувача (клас AuthController)

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

Лістинг 3.2 – Метод реєстрації користувача у системі (клас AuthService)

```
    async registerUser(registerDto: RegisterUserDto):
Promise<IJwtTokens> {
        const { email, password, fullName, driveName } =
registerDto;
        const queryRunner =
this.entityManager.connection.createQueryRunner();
        await queryRunner.startTransaction();
        const { manager } = queryRunner;

        try {
          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,
          },
          { manager },
        );

        const drive = await
this.userService.createUserDrive(user, driveName, {
          manager,
        });

        const isIndexCreated = await
this.elasticService.createDriveIndex(
          drive.id,
        );
        if (!isIndexCreated) {
          throw new
InternalServerErrorException('FAILED_TO_CREATE_INDEX');
        }

        await queryRunner.commitTransaction();

        return this.generateTokens({
          id: user.id,
          email: user.email,
        });
      } catch (error) {
        await queryRunner.rollbackTransaction();
        console.error(error);
        throw error;
      } finally {
        await queryRunner.release();
      }
    }
```

Лістинг 3.3 – Метод створення сховища користувача (клас UserService)

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

```
        await driveRepository.save(drive);

        drive.rootFolder = await
this.folderService.createRootFolder(drive, {
          manager,
        });
        drive.trashFolder = await
this.trashFolderService.createTrashFolder(drive, {
          manager,
        });

        await driveRepository.save(drive);

        return drive;
      }
```

Лістинг 3.4 – Метод створення індексу для сховища користувача в ElasticSearch (клас ElasticService)

```
    async createDriveIndex(driveId: string): Promise<boolean> {
        const indexName = this.getIndexName(driveId);

        try {
          const exists = await
this.elasticsearchService.indices.exists({
            index: indexName,
          });

          if (exists) {
            this.logger.log(`Index ${indexName} already exists`);
            return false;
          }

          const response = await
this.elasticsearchService.indices.create({
            index: indexName,
            mappings: {
              properties: {
                id: { type: 'integer' },
                originalName: {
                  type: 'text',
                  fields: {
                    keyword: { type: 'keyword' },
                  },
                },
                fileName: { type: 'keyword' },
                content: {
```

```
                  type: 'text',
                  analyzer: 'standard',
                },
                embedding: {
                  type: 'dense_vector',
                  dims: parseInt(
                    this.configService.get('ELASTICSEARCH_EMBEDDING
_DIMENSIONS'),
                  ),
                  index: true,
                  similarity: 'cosine',
                },
                userId: { type: 'integer' },
              },
            },
            settings: {
              number_of_shards: 1,
              number_of_replicas: 1,
            },
          } as IndicesCreateRequest & { mappings:
MappingTypeMapping });

          if (response.index && response.acknowledged) {
            this.logger.log(`Index ${indexName} created
successfully`);
            return true;
          }

          this.logger.warn(`Failed to create index ${indexName}`);
          return false;
        } catch (error) {
          this.logger.warn(
            `Failed to create index ${indexName}:
${error.message}`,
            error.stack,
          );
          return false;
        }
      }
```

Процес аутентифікації користувача є одним із базових механізмів системи, оскільки саме він забезпечує контроль доступу до персонального сховища та захист конфіденційних даних. Під час входу користувач проходить перевірку облікових даних, після чого система формує пару токенів — access token та refresh token. Ці токени містять зашифровану інформацію про користувача й виступають

інструментом підтвердження автентичності без необхідності повторного введення пароля при кожному запиті.

Access token використовується для короткотривалого доступу до захищених ресурсів і супроводжує більшість запитів до серверної частини. Refresh token має більший строк життя та дозволяє отримати новий access token у разі його завершення, забезпечуючи безперервність роботи користувача в системі. Такий підхід мінімізує ризики компрометації даних й підвищує рівень безпеки, тому що облікові дані ніколи не передаються повторно після первинного входу. У подальших фрагментах наведено приклади серверної реалізації процесу аутентифікації та механізму генерації токенів.

Лістинг 3.5 – Метод контроллера для аутентифікації користувача у системі (клас AuthController)

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

Лістинг 3.6 – Метод аутентифікації користувача у системі (клас AuthService)

```
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,
    });
```

```
  }
```

Лістинг 3.7 – Метод пошуку користувача за емейлом (клас UserService)

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

Процес завантаження файлів є одним із центральних елементів роботи системи, оскільки саме він забезпечує можливість формування персонального сховища користувача та організації його даних. Під час завантаження користувач обирає цільовий каталог і визначає один або декілька файлів, які необхідно додати до системи. Такий підхід дозволяє гнучко структурувати інформацію та підтримує індивідуальну логіку організації даних для кожного користувача.

Після отримання запиту серверна частина виконує збереження файлів у відповідну папку, створення записів у базі даних та формування метаданих, що описують кожен файл. Збережені матеріали стають доступними для подальших операцій — перегляду, редагування, переміщення та видалення. Реалізований механізм гарантує надійне зберігання даних, а також забезпечує цілісність файлової структури, включно з прив’язкою до конкретного каталогу. Нижче подано фрагменти коду, які демонструють основні етапи реалізації процесу завантаження файлів.

Лістинг 3.8 – Метод контроллера для завантаження файлів у сховище (клас FileController)

```
@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);
      }
```

Лістинг 3.9 – Метод завантаження файлів у сховище (клас FileService)

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

        const { manager } = queryRunner;
        try {
          const folder = await manager.findOne(FolderEntity, {
            where: {
              id: folderId,
              owner: { id: userId },
            },
            relations: ['owner', 'owner.drive'],
          });

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

          await forEach(files, async (file) => {
            const ext = path.extname(file.originalname);

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

            const fileEntity = manager.create(FileEntity, {
              fileName: uniqueFileName,
              originalFileName: file.originalname,
              filePath: path.join(folder.folderPath,
file.originalname),
              size: file.size,
              mimeType: file.mimetype,
              user: { id: userId },
              folder,
            });

            await manager.save(fileEntity);

            const filePath = await
this.fileSystem.getFilePath(fileEntity);
```

```
            await fs.promises.writeFile(
              filePath,
              file.buffer as unknown as Uint8Array,
            );

            const isFileExist = await
this.fileSystem.isExistInFileSystem(filePath);

            if (!isFileExist) {
              throw new InternalServerErrorException(
                'FILE_NOT_SAVED_IN_FILE_SYSTEM',
              );
            }

            const extractor =
FileTextExtractorFactory.getExtractor(
              fileEntity.originalFileName,
            );

            const content = await extractor.extractText(filePath);

            if (!content) {
              throw new
InternalServerErrorException('FAILED_TO_READ_FILE_CONTENT');
            }

            const chunks = await this.chunkText(content);

            const areChunksAdded = await map(
              chunks,
              async (chunk): Promise<boolean> => {
                const embedding = await
this.elasticService.getEmbedding(chunk);

                if (!embedding) {
                  throw new
InternalServerErrorException('FAILED_TO_GET_EMBEDDING');
                }

                return this.elasticService.addDocumentToIndex(
                  folder.owner.drive.id,
                  fileEntity,
                  { content: chunk, embedding },
                );
              },
            );

            return areChunksAdded.every((isAdded) => isAdded);
          });
```

```
            await queryRunner.commitTransaction();

            return {
              message: 'FILES_UPLOADED_SUCCESSFULLY',
              date: new Date(),
            };
          } catch (error) {
            await queryRunner.rollbackTransaction();
            throw error;
          } finally {
            await queryRunner.release();
          }
        }
```

Лістинг 3.10 – Метод додавання фрагменту видобутого тексту із файлу до індекса сховища (клас ElasticService)

```
      async addDocumentToIndex(
          driveId: string,
          file: FileEntity,
          { content, embedding }: { content: string; embedding:
number[] },
        ): Promise<boolean> {
          const indexName = this.getIndexName(driveId);

          try {
            const document = {
              id: file.id,
              originalName: file.originalFileName,
              fileName: file.fileName,
              content,
              embedding,
              userId: file.user.id,
            };

            const response = await this.elasticsearchService.index({
              index: indexName,
              document,
            });

            this.logger.debug('addDocumentToIndex response',
response);

            // Successful index when result is 'created' or 'updated'
            if (['created', 'updated'].includes(response.result)) {
              return true;
            }
```

```
            this.logger.warn(
              `Failed to add document to index ${indexName}:
${JSON.stringify(
                 response,
              )}`,
            );
            return false;
          } catch (error) {
            this.logger.warn(
              `Failed to add document to index ${indexName}:
${error.message}`,
              error.stack,
            );
            return false;
          }
        }
```

Функціонал формування архіву каталогу є важливою частиною системи, оскільки забезпечує користувачам можливість швидко отримати повну копію вибраної структури даних. Під час виконання цієї операції користувач обирає потрібний каталог, після чого система проводить рекурсивне обходження всієї його структури, включно з вкладеними підкаталогами та файлами, і формує єдиний архівний файл. Такий підхід дозволяє зберегти логіку структурування користувацьких даних та гарантує, що після розпакування структура каталогу буде повністю відтворена.

Лістинг 3.11 – Метод контроллера для завантаження архіву каталогу (клас FolderController)

```
@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,
    });
```

```
    }
```

Лістинг 3.12 – Метод завантаження архіву каталогу (клас FolderService)

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

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

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

          const { size } = 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;
        }
      }
```

Механізм створення унікальних посилань для отримання файлів забезпечує зручний і безпечний спосіб передавання даних третім особам без необхідності надання доступу до всього сховища. Під час генерації посилання користувач обирає файл, який хоче надати для завантаження, а також зазначає період дії, після завершення якого посилання автоматично втратить актуальність. На основі цих параметрів система створює унікальний ідентифікатор, формує URL і заносить відповідний запис до бази даних разом з датою завершення дії.

Такий функціонал дозволяє користувачам контролювати поширення своїх файлів, забезпечуючи обмежений у часі доступ і мінімізуючи ризики

несанкціонованого завантаження даних. Тимчасові посилання є зручним інструментом для одноразового обміну матеріалами, надсилання великих файлів або передавання документів користувачам, які не мають облікового запису в системі. У подальших фрагментах коду наведено реалізацію алгоритму генерації посилань, їх валідації та перевірки строку дії під час завантаження файлу.

Лістинг 3.12 – Метод контроллера для отримання посилання для завантаження файлу (клас FileController)

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

Лістинг 3.13 – Метод отримання посилання для завантаження файлу (клас FileService)

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

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

    const token = uuid();

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

    const fileDownloadLinkObject = await
this.fileDownloadLinkRepository.save(
      this.fileDownloadLinkRepository.create({
        file,
```

```
        token,
        expiresAt,
      }),
    );

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

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

Одним із важливих компонентів системи є модуль пошуку, який забезпечує ефективне знаходження файлів у сховищі користувача. Для різних сценаріїв використання система підтримує три типи пошуку: прямий, семантичний та гібридний. Кожен з них реалізує власний підхід до обробки запитів, що дозволяє задовольнити як базові, так і більш складні потреби пошуку інформації у файлах.

Прямий пошук реалізований як класичний механізм пошуку за метаданими файлів у базі даних PostgreSQL. У цьому режимі система виконує запит до таблиці файлів, порівнюючи пошукову фразу з полем original_file_name. Таким чином, користувач отримує швидкі та точні результати у випадках, коли відома назва файла або її частина. Завдяки використанню можливостей PostgreSQL, включаючи оператори часткового співпадіння та сортування, прямий пошук працює швидко й ефективно навіть при великій кількості файлів у сховищі.

Семантичний пошук ґрунтується на аналізі змісту файлів, а не лише їхніх назв. Для цього використовується індексація текстового вмісту в ElasticSearch і формування векторних embedding’ів за допомогою моделей OpenAI. Під час обробки запиту текст перетворюється на вектор, після чого ElasticSearch виконує kNN-пошук, знаходячи документи зі схожим змістом. Цей підхід дає змогу знаходити файли, понятійно близькі до запиту, навіть якщо користувач формулює пошук іншими словами. Семантичний пошук є особливо корисним для великих текстових файлів, PDF-документів, DOCX-файлів та зображень, де текст витягується за допомогою OCR.

Гібридний пошук поєднує результати прямого та семантичного пошуку, щоб сформувати максимально релевантну відповідь. Після виконання окремих запитів система використовує метод RRF (Reciprocal Rank Fusion), що дозволяє комбінувати результати двох підходів у єдиний впорядкований список. Такий метод ранжування враховує сильні сторони обох пошуків: точність прямого пошуку за назвою та глибину семантичного аналізу. У результаті користувач отримує збалансовані результати, де у верхніх позиціях знаходяться як файли з відповідною назвою, так і документи з найбільш релевантним змістом.

Лістинг 3.14 – Метод контроллера для пошуку файлів (клас SearchController)

```
@Controller('search')
@UseGuards(AccessTokenGuard)
@ApiTags('search')
@ApiBearerAuth()
export class SearchController {
  constructor(private readonly searchService: SearchService) {}

  @Get()
  @ApiResponse({
    status: 200,
    description: 'Search results with pagination',
    type: SearchPaginationResponseDto,
  })
  async search(
    @Query() searchDto: SearchDtoReq,
    @CurrentUser() user: IUser,
  ): Promise<SearchPaginationResponseDto> {
    return await this.searchService.search(searchDto, user);
  }
}
```

Лістинг 3.15 – Методи пошуку файлів за назвами файлів та їх контентом (клас SearchService)

```
async search(
    { searchType, ...rest }: SearchDtoReq,
    user: IUser,
  ): Promise<SearchPaginationResponseDto> {
    const drive = await
this.entityManager.findOneOrFail(DriveEntity, {
      where: {
        owner: {
```

```
            id: user.id,
          },
        },
      });

      switch (searchType) {
        case 'direct':
          return this.directSearch({ ...rest }, user);
        case 'semantic':
          return this.semanticSearch(drive.id, rest.query);
        case 'hybrid':
          return this.hybridSearch(drive.id, rest.query, {
...rest }, user);
        default:
          break;
      }

      throw new BadRequestException('INVALID_SEARCH_TYPE');
    }

    async directSearch(
      {
        query,
        type,
        orderBy,
        orderDirection,
        page = 1,
        pageSize = 20,
      }: Omit<SearchDtoReq, 'searchType'>,
      user: IUser,
    ): Promise<SearchPaginationResponseDto> {
      const orderByColumn = OrderBy[orderBy];
      const direction = OrderDirection[orderDirection];

      const limit = pageSize;
      const offset = (page - 1) * pageSize;

      const searchQueries = {
        file: `
            SELECT id,
                   'file' AS type,
                   original_file_name AS original_name,
                   size,
                   file_path AS path,
                   created_at,
                   updated_at
            FROM files
            WHERE original_file_name ILIKE $1 AND user_id = $2
          `,
        folder: `
```

```
            SELECT id,
                   'folder' AS type,
                   original_folder_name AS original_name,
                   0 AS size,
                   folder_path AS path,
                   created_at,
                   updated_at
            FROM folders
            WHERE original_folder_name ILIKE $1 AND owner_id = $2
          `,
        };

        const countQueries = {
          file: `SELECT COUNT(*) as count FROM files WHERE
original_file_name ILIKE $1 AND user_id = $2`,
          folder: `SELECT COUNT(*) as count FROM folders WHERE
original_folder_name ILIKE $1 AND owner_id = $2`,
          all: `
            SELECT COUNT(*) as count FROM (
              (SELECT id FROM files WHERE original_file_name ILIKE
$1 AND user_id = $2)
              UNION ALL
              (SELECT id FROM folders WHERE original_folder_name
ILIKE $1 AND owner_id = $2)
            ) AS combined
          `,
        };

        const countSql = countQueries[type];
        const countResult = await
this.entityManager.query(countSql, [
          `%${query}%`,
          user.id,
        ]);
        const total = parseInt(countResult[0]?.count || '0', 10);

        let sql: string;
        switch (type) {
          case 'file':
          case 'folder':
            sql = `${searchQueries[type]} ORDER BY ${orderByColumn}
${direction} LIMIT $3 OFFSET $4`;
            break;
          case 'all':
            sql = `
              (${searchQueries.file})
              UNION
              (${searchQueries.folder})
              ORDER BY ${orderByColumn} ${direction}
              LIMIT $3 OFFSET $4
```

```
            `;
            break;
          default:
            throw new BadRequestException('INVALID_SEARCH_TYPE');
        }

        const results = await this.entityManager.query(sql, [
          `%${query}%`,
          user.id,
          limit,
          offset,
        ]);

        const data: SearchDtoRes[] = results.map((row: any) => ({
          id: row.id,
          originalName: row.original_name,
          size: row.size,
          path: row.path,
          createdAt: row.created_at,
          updatedAt: row.updated_at,
          type: row.type,
        }));

        const totalPages = Math.ceil(total / pageSize);
        const currentPageItems = data.length;

        return {
          data,
          meta: {
            page,
            pageSize,
            total,
            totalPages,
            currentPageItems,
          },
        };
      }

      async semanticSearch(driveId: string, query: string):
Promise<any> {
        const results = await
this.elasticService.searchDocuments(driveId, query);

        if (!results.length) {
          return {
            data: [],
            meta: {
              page: 1,
              pageSize: 20,
              total: 0,
```

```
            totalPages: 0,
            currentPageItems: 0,
          },
        };
      }

      // Preserve order of documents as returned by ElasticSearch
using file IDs
      // and build map of best content snippet per file (first
hit has highest score)
      const orderedIds = Array.from(new Set(results.map((item:
any) => item.id)));

      const bestContentByFileId = new Map<number, string>();
      results.forEach((item: any) => {
        if (!bestContentByFileId.has(item.id) && item.content) {
          bestContentByFileId.set(item.id, item.content);
        }
      });

      const files = await this.entityManager.find(FileEntity, {
        where: {
          id: In(orderedIds),
        },
        relations: ['folder'],
      });

      const fileById = new Map<number, FileEntity>();
      files.forEach((file) => {
        fileById.set(file.id, file);
      });

      // Rebuild files array strictly following the order from
ElasticSearch
      const orderedFiles: FileEntity[] = [];
      orderedIds.forEach((id) => {
        const file = fileById.get(id);
        if (file) {
          orderedFiles.push(file);
        }
      });

      const data: SearchDtoRes[] = orderedFiles.map((file) => {
        const content = bestContentByFileId.get(file.id);

        return {
          id: file.id,
          originalName: file.originalFileName,
          size: file.size,
          path: file.filePath,
```

```
            createdAt: file.createdAt,
            updatedAt: file.updatedAt,
            type: 'file',
            snippet: content ? this.buildSnippet(content, query) :
undefined,
          };
        });

        return {
          data,
          meta: {
            page: 1,
            pageSize: 20,
            total: files.length,
            totalPages: Math.ceil(files.length / 20),
            currentPageItems: files.length,
          },
        };
      }

      async hybridSearch(
        driveId: string,
        query: string,
        { page = 1, pageSize = 20 }: Omit<SearchDtoReq,
'searchType' | 'query'>,
        user: IUser,
      ): Promise<SearchPaginationResponseDto> {
        const results = await
this.elasticService.hybridSearchDocuments(
          driveId,
          query,
          {
            limit: pageSize * 2, // Get more results for better
pagination
          },
        );

        if (!results.length) {
          return {
            data: [],
            meta: {
              page,
              pageSize,
              total: 0,
              totalPages: 0,
              currentPageItems: 0,
            },
          };
        }
```

```
        // Preserve order of documents as returned by ElasticSearch
using file IDs
        // and build map of best content snippet per file (first
hit has highest score)
        const orderedIds = Array.from(new Set(results.map((item:
any) => item.id)));

        const bestContentByFileId = new Map<number, string>();
        results.forEach((item: any) => {
          if (!bestContentByFileId.has(item.id) && item.content) {
            bestContentByFileId.set(item.id, item.content);
          }
        });

        // Apply pagination
        const startIndex = (page - 1) * pageSize;
        const endIndex = startIndex + pageSize;
        const paginatedIds = orderedIds.slice(startIndex,
endIndex);

        const files = await this.entityManager.find(FileEntity, {
          where: {
            id: In(paginatedIds),
          },
          relations: ['folder'],
        });

        const fileById = new Map<number, FileEntity>();
        files.forEach((file) => {
          fileById.set(file.id, file);
        });

        // Rebuild files array strictly following the order from
ElasticSearch
        const orderedFiles: FileEntity[] = [];
        paginatedIds.forEach((id) => {
          const file = fileById.get(id);
          if (file) {
            orderedFiles.push(file);
          }
        });

        const data: SearchDtoRes[] = orderedFiles.map((file) => {
          const content = bestContentByFileId.get(file.id);

          return {
            id: file.id,
            originalName: file.originalFileName,
            size: file.size,
            path: file.filePath,
```

```
            createdAt: file.createdAt,
            updatedAt: file.updatedAt,
            type: 'file',
            snippet: content ? this.buildSnippet(content, query) :
undefined,
          };
        });

        const total = orderedIds.length;
        const totalPages = Math.ceil(total / pageSize);
        const currentPageItems = data.length;

        return {
          data,
          meta: {
            page,
            pageSize,
            total,
            totalPages,
            currentPageItems,
          },
        };
      }
```

3.2 Візуалізація функціоналу розробленої системи

У цьому підрозділі представлено візуальні елементи вебсистеми, що демонструють ключові етапи взаємодії користувача з інтерфейсом. Скриншоти дають можливість наочно оцінити реалізовані функції, логіку побудови інтерфейсу та зручність користування системою. У межах цього розділу подано описи основних сторінок, включаючи реєстрацію, авторизацію та роботу з файловим сховищем, що дозволяє зрозуміти особливості поведінки системи та її функціональні можливості з позиції кінцевого користувача.

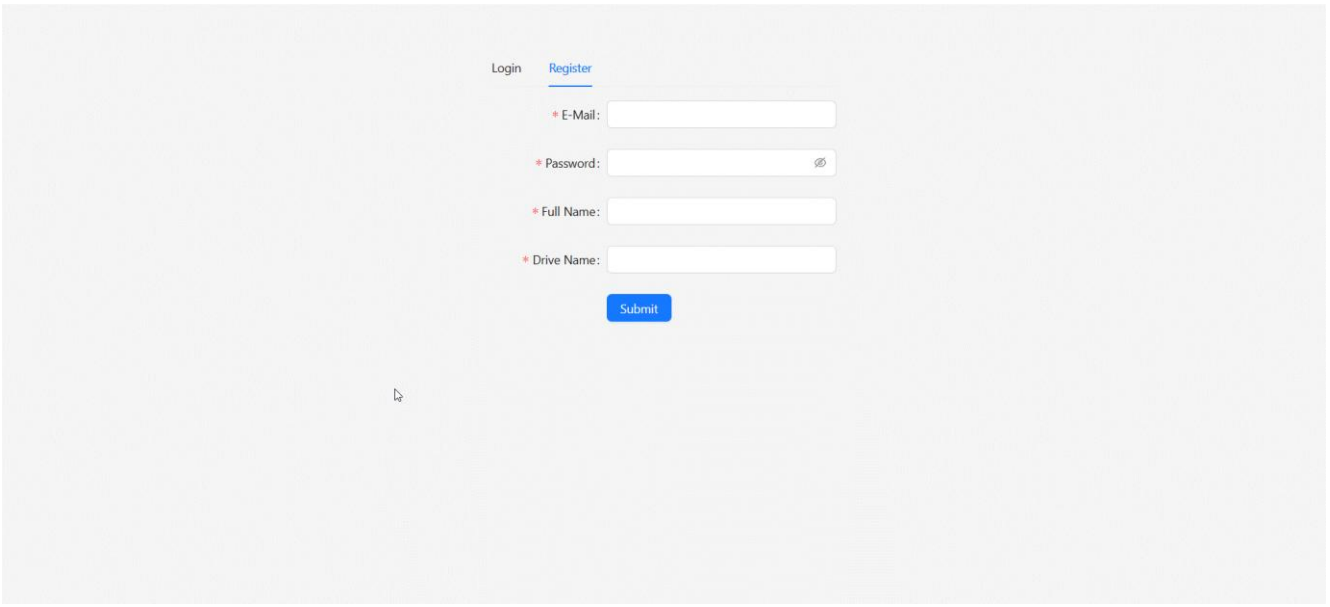

Рисунок 3.1 – Інтерфейс реєстрації нового користувача вебсистеми

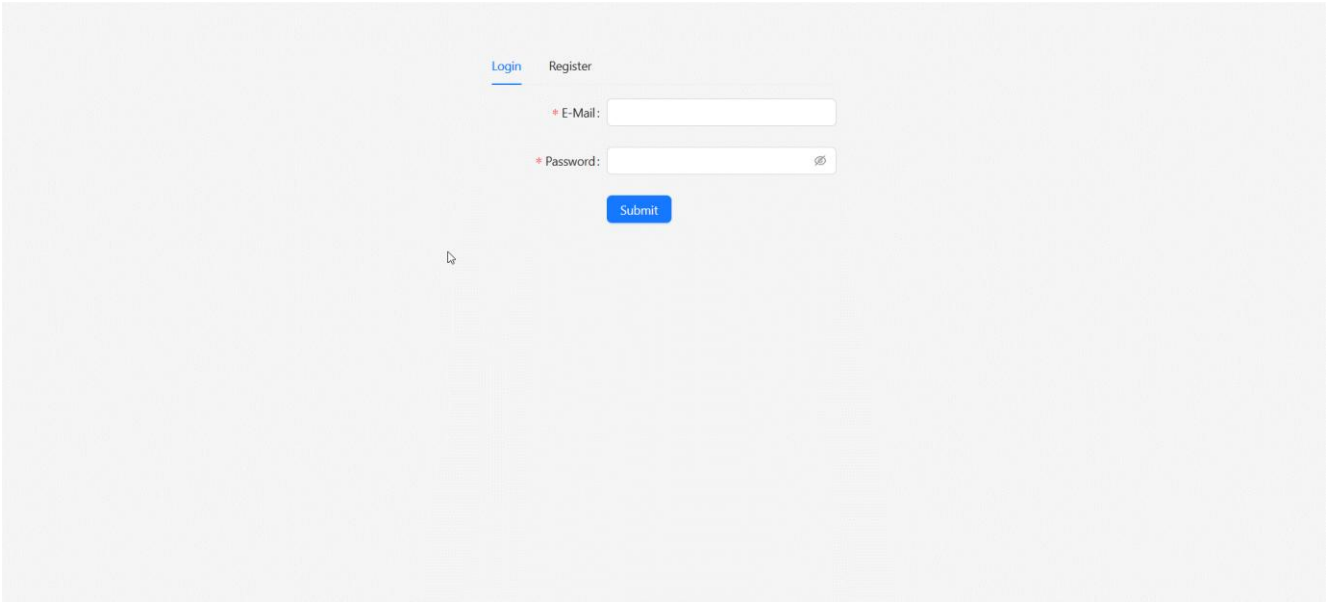

Рисунок 3.2 – Інтерфейс авторизації користувача у вебсистемі

На рисунках представлено інтерфейси реєстрації та авторизації, які забезпечують початкову взаємодію користувача з вебсистемою централізованого зберігання файлів. Обидві сторінки виконані у мінімалістичному стилі та містять лише необхідні елементи управління, що сприяє зручності та зрозумілості процесу. Сторінка реєстрації передбачає введення електронної пошти, пароля, повного імені та назви особистого диска, який автоматично створюється для нового користувача. Сторінка авторизації містить поля для введення електронної пошти та пароля, що

дозволяє вже зареєстрованим користувачам отримати доступ до свого файлового простору. Обидва інтерфейси відповідають сучасним вимогам до юзабіліті, забезпечуючи чітку структуру, легкість навігації та інтуїтивність використання.

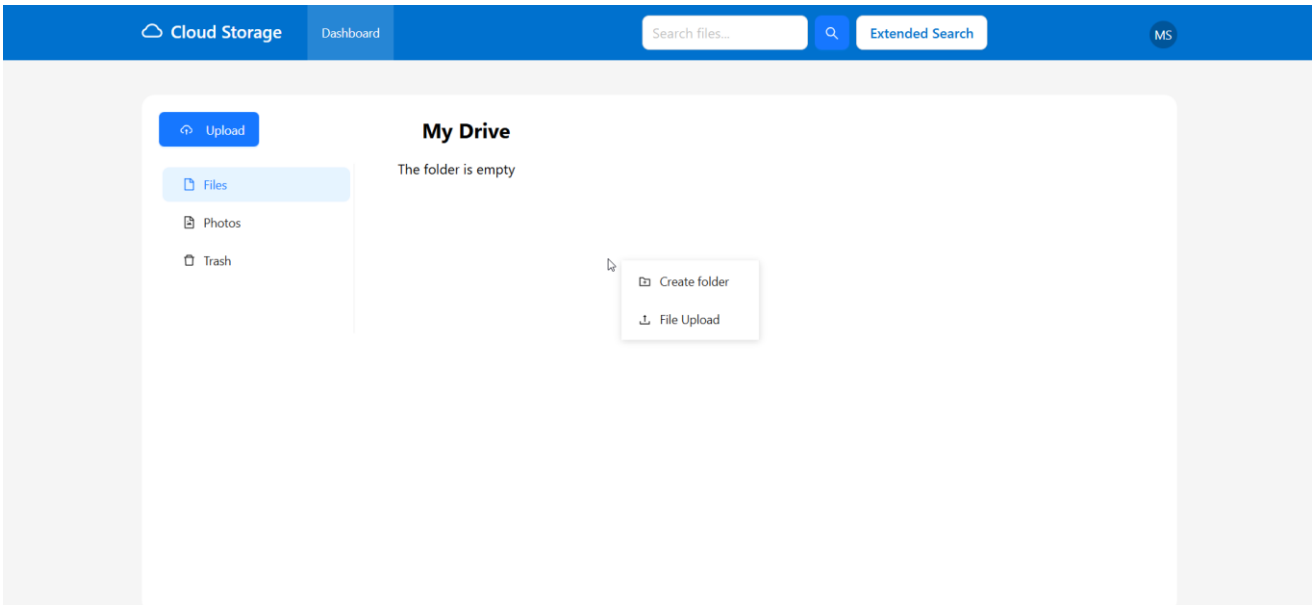

Рисунок 3.3 – Інтерфейс головної директорії користувача у вебсистемі

На рисунку представлено головний інтерфейс вебсистеми після входу користувача. На робочій панелі відображається коренева директорія сховища («My Drive»), у якій наразі немає файлів або папок. Ліва бічна панель містить основні розділи навігації: загальний файловий простір, папку з фотографіями та корзину. У верхній частині інтерфейсу розміщено поле пошуку та кнопка розширеного пошуку, що забезпечує швидкий доступ до функцій фільтрації та семантичного пошуку. Користувачу також доступні елементи керування, такі як створення папки та завантаження файлів безпосередньо з екрану, що робить взаємодію із системою інтуїтивною та зручною.

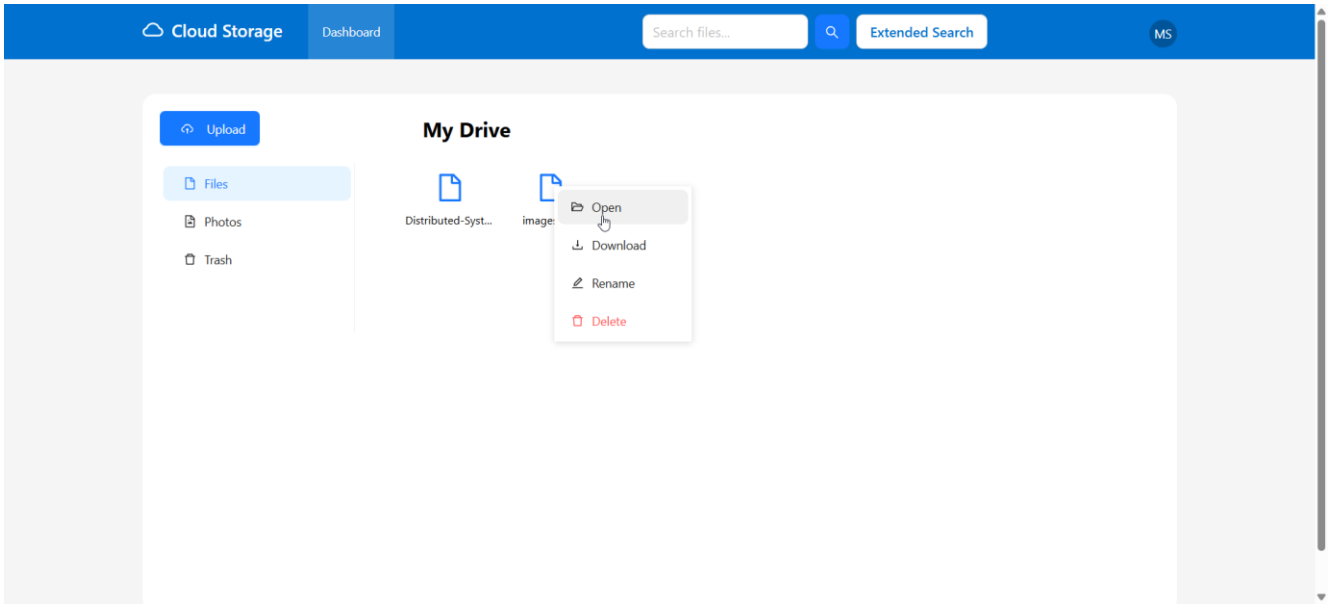

Рисунок 3.4 – Контекстне меню керування файлами у вебсистемі

На рисунку продемонстровано взаємодію користувача з файлом у вебсистемі через контекстне меню. Після виклику контекстного меню правою кнопкою миші для вибраного файлу відкривається список доступних операцій, серед яких відкриття файлу, завантаження на локальний пристрій, перейменування та видалення. Такий підхід забезпечує оперативний доступ до ключових функцій керування файлами, підвищує зручність роботи з інтерфейсом та дозволяє ефективно організовувати вміст власного сховища. Контекстне меню є важливим елементом користувацького досвіду, оскільки надає інтуїтивний спосіб виконання найпоширеніших дій без додаткової навігації чи переходів.

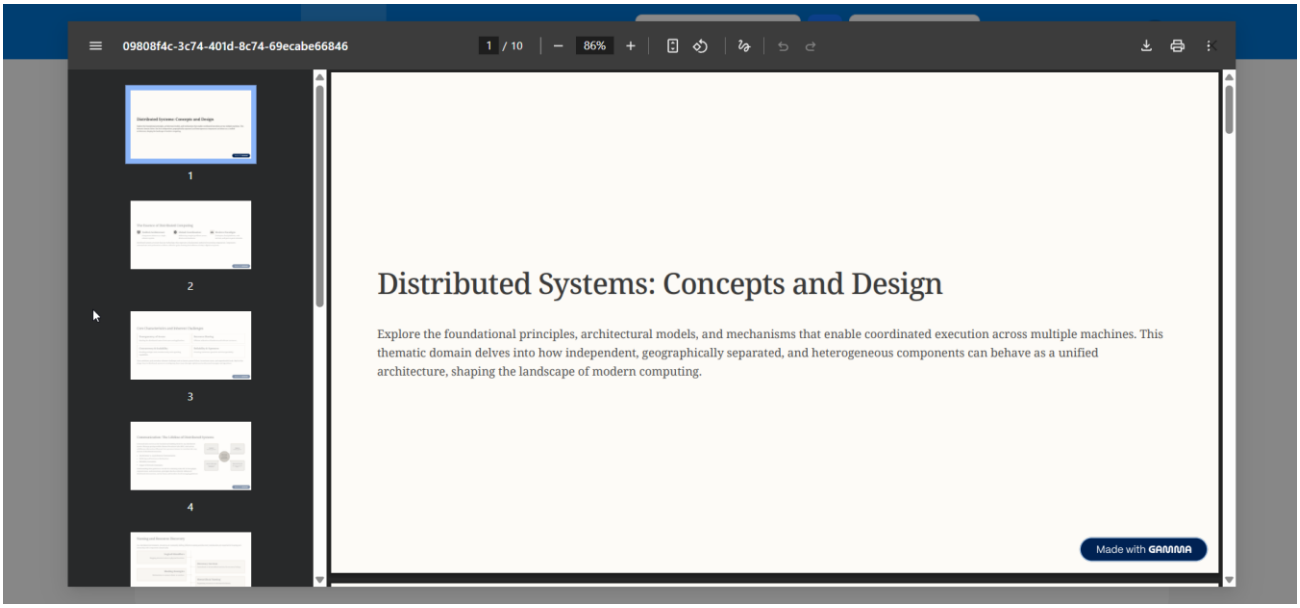

Рисунок 3.5 – Інтерфейс попереднього перегляду PDF-документів у вебсистемі

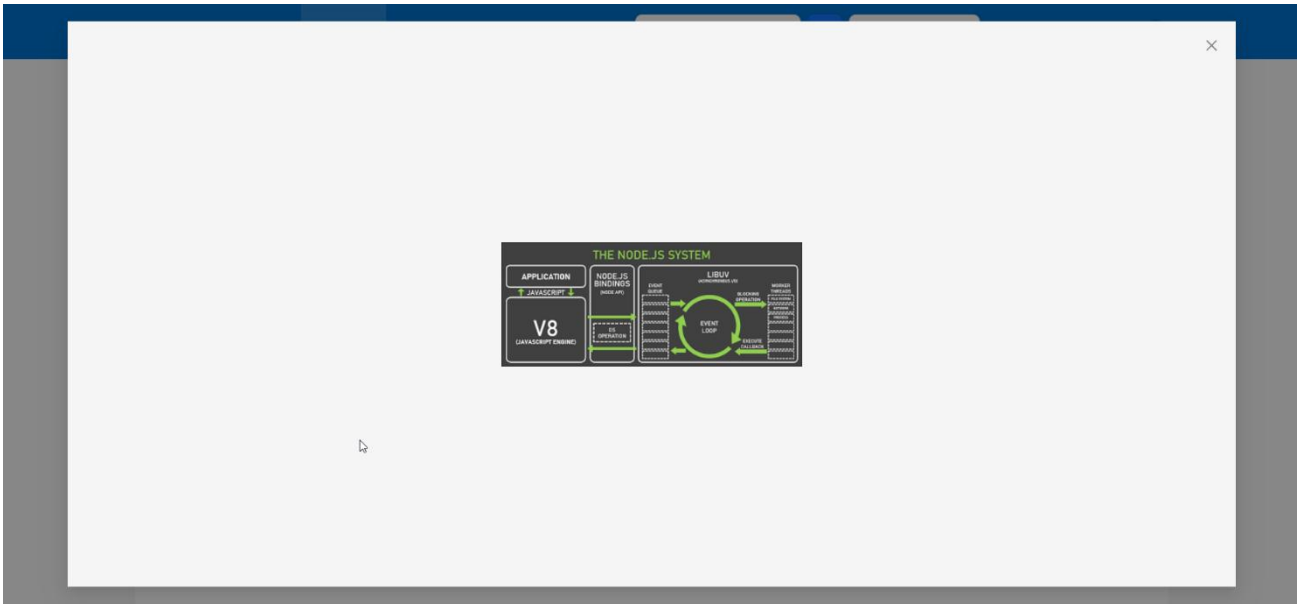

Рисунок 3.6 – Інтерфейс попереднього перегляду зображень у вебсистемі

Функціонал попереднього перегляду файлів забезпечує зручну взаємодію користувача з документами та мультимедійними матеріалами без необхідності їхнього завантаження. На першому екрані представлено інтерфейс перегляду PDF-документів, який дозволяє гортати сторінки, змінювати масштаб та швидко орієнтуватися у структурі файлу через панель мініатюр. На другому зображенні показано перегляд зображень, який відкривається у модальному вікні та дозволяє масштабувати файл, переглядати його у високій якості та закривати перегляд

одним кліком. Такий підхід суттєво підвищує зручність використання системи, роблячи її зручною для роботи з навчальними матеріалами, технічною документацією та графічними файлами.

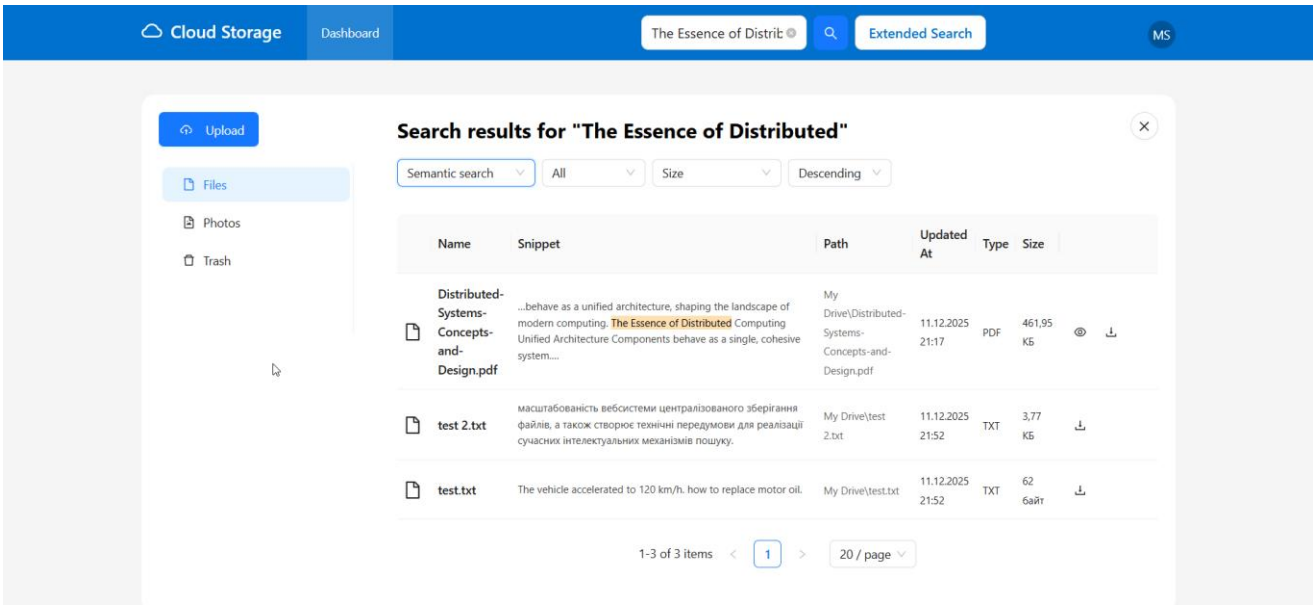

Рисунок 3.7 – Відображення результатів семантичного пошуку у вебсистемі

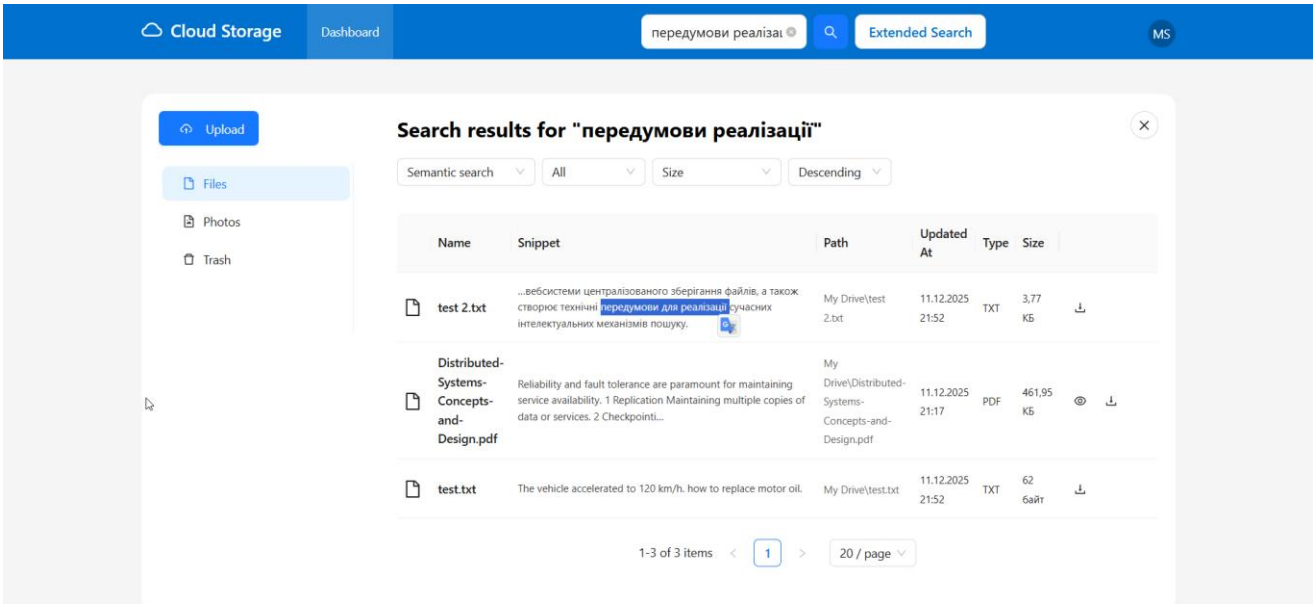

Рисунок 3.8 – Відображення результатів семантичного пошуку у вебсистемі

На наведених рисунках продемонстровано роботу механізму семантичного пошуку у вебсистемі. Після введення пошукового запиту система за допомогою попередньо згенерованих векторних представлень порівнює зміст файлів і формує

результати, упорядковані за релевантністю. На екрані відображається таблиця, що містить ім’я файлу, фрагмент знайденого контексту (*snippet*), шлях до файлу, дату оновлення, тип та розмір. Семантичний пошук дає змогу знаходити документи навіть у випадках, коли запит не містить точного збігу з текстом у файлах, але передає схожий зміст або тематику. Це дозволяє суттєво підвищити якість навігації по великому сховищу файлів, забезпечуючи інтелектуальне виявлення матеріалів за змістовими ознаками.

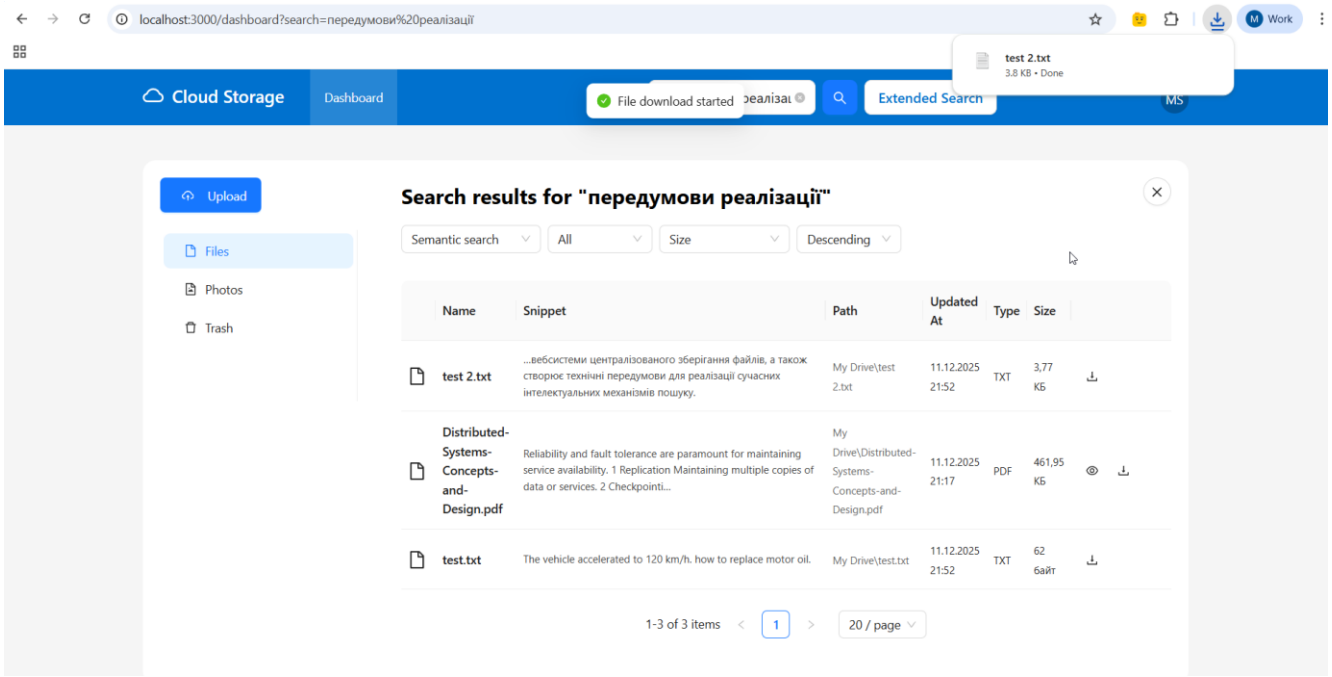

Рисунок 3.9 – Процес завантаження файлу з інтерфейсу пошуку у вебсистемі

На рисунку продемонстровано процес завантаження файлу з інтерфейсу вебсистеми. Після натиснення відповідної кнопки в таблиці результатів користувач отримує стандартне сповіщення браузера про початок завантаження, що свідчить про успішне опрацювання запиту сервером. Інтерфейс зберігає повний контекст пошуку: у верхній частині екрана відображено активний запит, тип пошуку та фільтри, а нижче — таблицю знайдених документів, де користувач може переглядати, відкривати або завантажувати файли. Додаткове повідомлення у правому верхньому куті браузера підтверджує, що файл було успішно отримано

локально. Такий підхід забезпечує зручність роботи та прозорість усіх операцій, пов’язаних з керуванням файлами.

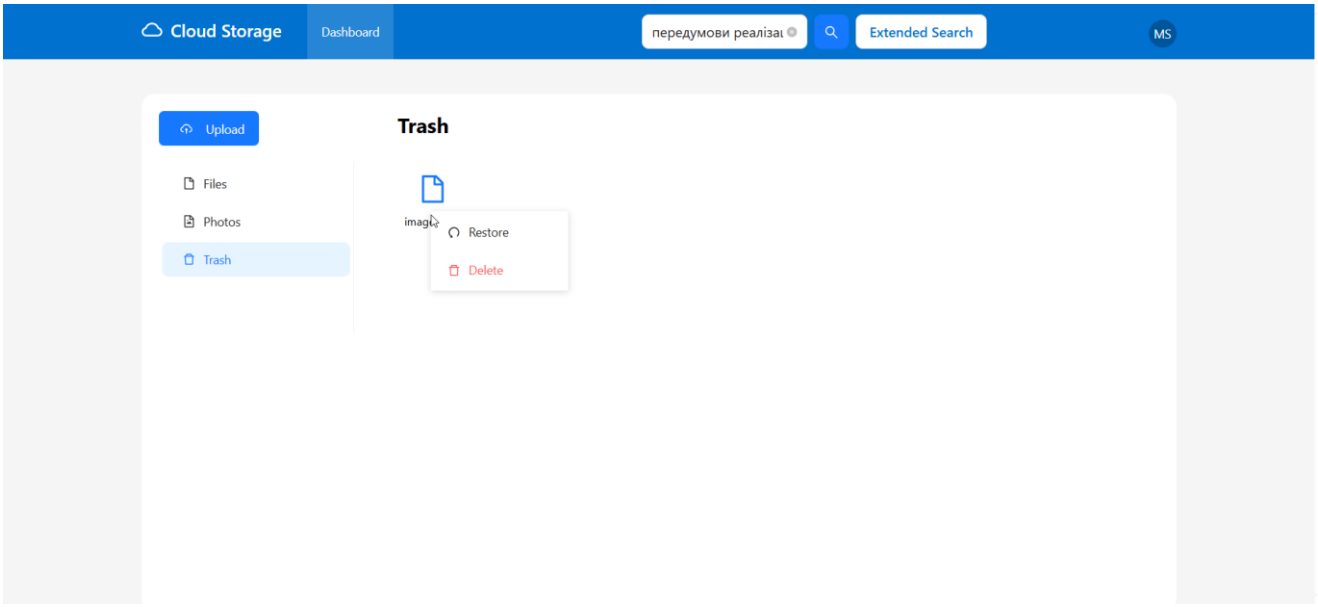

Рисунок 3.10 – Інтерфейс корзини із можливістю відновлення або остаточного видалення файлу

На цьому скріншоті продемонстровано інтерфейс розділу «Trash», у якому відображаються файли, попередньо видалені користувачем. Система не видаляє їх остаточно одразу, що дає можливість відновити помилково видалені дані. Для кожного файлу доступне контекстне меню з двома діями: Restore — для повернення елемента у вихідну директорію, та Delete — для остаточного видалення без можливості відновлення. Такий підхід підвищує безпечність роботи користувача з файлами та мінімізує ризики випадкової втрати важливої інформації.

3.3 Тестування функціональних компонентів і аналіз якості програмної системи

Тестування є невід’ємною складовою життєвого циклу розробки програмного забезпечення та має визначальне значення для забезпечення його якості, стабільності та відповідності встановленим вимогам [15]. На цьому етапі

здійснюється комплексна перевірка правильності реалізації функціональних можливостей системи, виявлення логічних помилок, дефектів програмної реалізації та потенційних вразливостей, а також аналіз стійкості роботи програмного продукту в різних умовах експлуатації.

Для вебсистеми централізованого зберігання файлів із підтримкою семантичного пошуку процес тестування є особливо важливим, оскільки будь-які збої або помилки можуть призвести до втрати даних, порушення доступу до інформації або зниження рівня безпеки. Саме тому тестування спрямоване не лише на перевірку коректності роботи окремих функціональних модулів, але й на забезпечення цілісності, доступності та захищеності даних користувачів перед введенням системи в промислову експлуатацію.

Основною ціллю тестування є вчасне виявлення дефектів і їх усунення ще на етапі розробки, що дозволяє мінімізувати ризики некоректної роботи системи в експлуатаційному середовищі. Проведення тестування також дає змогу переконатися, що система задовільняє вказані функціональні та нефункціональні вимоги, працює стабільно під навантаженням і забезпечує коректну обробку запитів користувачів. Таким чином, тестування сприяє підвищенню загальної якості програмного продукту та зниженню можливих експлуатаційних і репутаційних ризиків.

Процес тестування складається з декількох взаємопов’язаних етапів. На початковому етапі виконується дослідження вимог до системи, що дозволяє визначити ключові сценарії використання та критерії успішності тестування. На основі цього аналізу розробляються сценарії тестування, які забезпечують перевірку базових функціональних компонентів системи, а також перевіряють граничні та нетипові ситуації. Далі здійснюється безпосереднє виконання тестів, у ході якого фіксуються всі виявлені відхилення від прогнозованої поведінки системи. Отримані результати підлягають подальшому аналізу з метою виявлення першопричин виявлених помилок, оцінювання їхнього впливу на роботу системи та визначення оптимальних способів усунення виявлених недоліків. За

необхідності проводиться регресійне тестування для перевірки стабільності вже реалізованого функціоналу після внесення змін.

У межах даної роботи основна увага була приділена модульному (unit) тестуванню, яке спрямоване на перевірку окремих ізольованих компонентів вебсистеми. Unit тести дають змогу переконатися в правильності роботи контролерів, сервісів та допоміжних модулів, а також забезпечують раннє виявлення логічних помилок у коді. Для тестування серверної частини системи було використано автоматизовані тести, що охоплюють основні бізнес-логіки, зокрема управління користувачами, файлами, каталогами, пошуком та кошиком.

Результати тестування підтверджують коректність реалізації основного функціоналу системи та її готовність до подальшого використання і розширення. Проведення комплексного unit тестування забезпечило підвищення надійності програмного коду, спростило процес супроводу системи та створило передумови для її масштабування в майбутньому. На рисунку нижче наведено приклад виконання автоматизованих модульних тестів, що демонструє успішне проходження всіх тестових сценаріїв у системі.

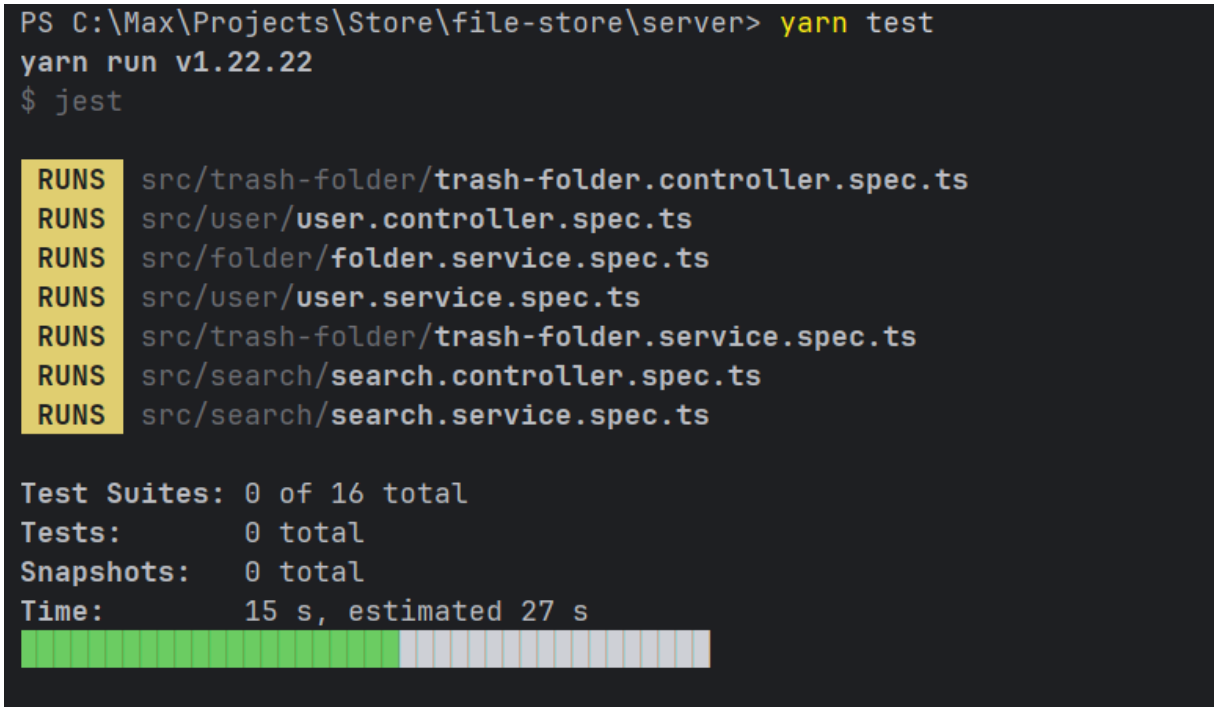

Рисунок 3.11 – Проведення модульного тестування для ідентифікації дефектів у кодовій базі

```
PS C:\Max\Projects\Store\file-store\server> yarn test
yarn run v1.22.22
$ jest
 PASS  src/search/search.service.spec.ts (21.916 s)
 PASS  src/search/search.controller.spec.ts (23.968 s)
 PASS  src/folder/folder.service.spec.ts (24.444 s)
 PASS  src/user/user.service.spec.ts (24.575 s)
 PASS  src/trash-folder/trash-folder.service.spec.ts (24.687 s)
 PASS  src/user/user.controller.spec.ts (26.016 s)
 PASS  src/trash-folder/trash-folder.controller.spec.ts (26.18 s)
 PASS  src/utils/file-system.service.spec.ts
 PASS  src/file/file.service.spec.ts
 PASS  src/search/elastic.service.spec.ts
 PASS  src/auth/auth.service.spec.ts
 PASS  src/folder/folder.controller.spec.ts (6.317 s)
 PASS  src/note/note.controller.spec.ts
 PASS  src/auth/auth.controller.spec.ts
 PASS  src/file/file.controller.spec.ts (5.253 s)
 PASS  src/note/note.service.spec.ts

Test Suites: 16 passed, 16 total
Tests:       118 passed, 118 total
Snapshots:   0 total
Time:        30.614 s
Ran all test suites.
Done in 31.42s.
```

Рисунок 3.12 – Результати модульного тестування

Автоматизоване тестування виконувалося із використанням фреймворку Jest. Загальний час виконання всіх тестів склав приблизно 30–31 секунду, що свідчить про прийнятну продуктивність тестового середовища та ефективну організацію тестових сценаріїв. Такий час виконання дозволяє регулярно запускати тести під час розробки без суттєвого впливу на швидкість роботи розробника.

Рівень покриття коду unit-тестами охоплює основні бізнес-логіки системи, включаючи обробку запитів, роботу з базою даних, валідацію вхідних даних та сценарії помилок. Це забезпечує високий ступінь впевненості у коректності реалізації функціоналу та знижує ймовірність появи критичних помилок під час експлуатації системи. Досягнуті показники тестування підтверджують достатній рівень якості програмного коду та готовність системи до практичного використання й подальшого розширення.

РОЗДІЛ 4: ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

У даному розділі розглядаються питання охорони праці та безпеки в надзвичайних ситуаціях, які є невід'ємною складовою професійної діяльності фахівців у сфері інформаційних технологій. Особлива увага приділяється аналізу умов праці користувачів комп'ютерних систем, потенційних факторів ризику для здоров'я, а також вимогам чинного законодавства України щодо забезпечення безпечних і нешкідливих умов праці. Розділ спрямований на обґрунтування необхідності дотримання норм охорони праці, ергономічних вимог та правил безпеки з метою запобігання професійним захворюванням і зменшення негативного впливу шкідливих факторів на організм людини.

4.1 Охорона праці.

Під час виконання магістерської кваліфікаційної роботи на тему «Вебсистема централізованого зберігання файлів із підтримкою семантичного пошуку» особлива увага приділялася питанням охорони праці. Процес розробки програмної системи передбачав тривалу роботу з комп'ютерною технікою, використання екранних пристроїв, периферійного обладнання та програмних засобів, що зумовлює необхідність дотримання вимог чинного законодавства України у сфері безпеки та охорони праці.

Відповідно до Закону України «Про охорону праці», охорона праці визначається як система правових, соціально-економічних, організаційно-технічних, санітарно-гігієнічних і лікувально-профілактичних заходів та засобів, спрямованих на збереження життя, здоров'я і працездатності людини у процесі трудової діяльності [22]. Дія цього закону поширюється також на осіб, діяльність яких пов'язана з інтелектуальною працею та використанням інформаційних технологій, зокрема програмістів і розробників вебсистем.

Організація робочого процесу під час розробки вебсистеми здійснювалася з урахуванням вимог ДНАОП 0.00-4.15-98 «Положення про розробку інструкцій з охорони праці», яке регламентує порядок створення та дотримання інструкцій з безпеки праці для працівників відповідних професій [23]. Також враховувалися положення НПАОП 0.00-4.12-05, що визначає порядок проведення інструктажів, навчання та перевірки знань з питань охорони праці [24].

Робоче місце розробника вебсистеми було обладнане персональним комп'ютером і периферійними пристроями та організоване відповідно до вимог Державних санітарних норм і правил НПАОП 0.00-7.11-12 «Загальні вимоги стосовно забезпечення роботодавцями охорони праці працівників» [27]. Монітор розміщувався на відстані 50–70 см від очей користувача, при цьому верхня межа екрана знаходилася на рівні або трохи нижче лінії зору, що сприяло зменшенню навантаження на органи зору.

Клавіатура та маніпулятор типу «миша» розташовувалися таким чином, щоб забезпечити зручне положення рук і мінімізувати напруження м'язів верхніх кінцівок. Для роботи використовувалося регульоване крісло зі спинкою, яке дозволяло підтримувати правильну поставу та зменшувати статичне навантаження на опорно-руховий апарат. Дотримання цих вимог сприяло профілактиці професійних захворювань, пов'язаних із тривалою роботою за комп'ютером.

Особлива увага приділялася умовам освітлення робочого місця. Освітлення відповідало вимогам ДБН В.2.5-28:2018 «Природне і штучне освітлення» [28]. Приміщення мало достатній рівень природного освітлення, а у разі його недостатності використовувалося штучне освітлення з рівномірним розподілом світлового потоку. Це дозволяло уникнути появи відблисків на екрані монітора та зменшити втому очей під час тривалої роботи.

Під час роботи з комп'ютерною технікою дотримувалися вимоги НПАОП 0.00-7.15-18 «Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями» [25]. Зокрема, регламентувалася тривалість безперервної роботи за комп'ютером, передбачалося чергування інтелектуально напружених

завдань із перервами для відпочинку, а також виконання вправ для зменшення навантаження на зір і м’язи.

Електробезпека під час виконання роботи забезпечувалася відповідно до вимог ДНАОП 0.00-1.21-98 «Правила безпечної експлуатації електроустановок споживачів» [26]. Усі електричні пристрої були справними, мали захисне заземлення та підключалися до електромережі з використанням сертифікованих кабелів і мережевих фільтрів. Перед початком роботи здійснювалася перевірка стану електрообладнання, а у разі виявлення несправностей його експлуатація припинялася.

Окремо враховувалися психофізіологічні фактори, пов’язані з характером розумової праці. Розробка вебсистеми централізованого зберігання файлів із підтримкою семантичного пошуку потребує високої концентрації уваги, аналізу великих обсягів інформації та прийняття складних інженерних рішень. З метою зниження емоційного напруження та перевтоми застосовувалися принципи раціональної організації праці, планування робочого часу та регулярні перерви.

Таким чином, у процесі виконання магістерської кваліфікаційної роботи було забезпечено дотримання основних вимог охорони праці відповідно до чинного законодавства України. Реалізація організаційних, санітарно-гігієнічних і технічних заходів дозволила створити безпечні та комфортні умови праці, що сприяло ефективній розробці вебсистеми та збереженню здоров’я виконавця.

4.2 Фактори ризику і можливі порушення здоров’я користувачів комп’ютерної мережі.

У сучасному суспільстві комп’ютерні мережі та інформаційні технології стали невід’ємною частиною професійної діяльності людини. Значна кількість працівників різних галузей здійснює свою діяльність із використанням персональних комп’ютерів, мережевих ресурсів і програмних засобів. Такий характер праці зумовлює тривалий вплив на організм людини комплексу фізичних, психофізіологічних та ергономічних факторів, які за відсутності належних умов

можуть негативно впливати на стан здоров’я користувачів комп’ютерної мережі [29].

Праця користувачів комп’ютерної мережі належить до категорії розумової діяльності з незначним фізичним навантаженням, однак характеризується підвищеним навантаженням на зоровий аналізатор, нервову систему та опорно-руховий апарат. Тривале перебування у статичному положенні, постійна концентрація уваги, обробка великих обсягів інформації та робота з екранними пристроями створюють передумови для виникнення функціональних порушень і професійних захворювань [30].

Одним із основних факторів ризику є інтенсивне зорове навантаження. Під час роботи з персональним комп’ютером користувач постійно фокусує зір на екрані монітора, що може призводити до швидкої втоми очей, сухості слизової оболонки, почервоніння та зниження гостроти зору. Негативний вплив посилюється за умови неправильного розташування дисплея, недостатньої або надмірної яскравості екрана, низької контрастності зображення та наявності відблисків [31]. За систематичного впливу цих факторів можливий розвиток астенопії та інших порушень зорової функції.

Важливу роль у формуванні зорового навантаження відіграє освітлення робочого місця. Недостатній рівень освітленості або нерівномірний розподіл світлового потоку спричиняють підвищене напруження органів зору, що негативно позначається на працездатності та загальному самопочутті користувача. Відповідно до нормативних вимог, освітлення повинно забезпечувати оптимальні умови для зорової роботи, не створювати різких контрастів між екраном та навколишніми поверхнями і не викликати появу відблисків [32]. Порушення цих вимог може призводити до зорової втоми, головного болю та зниження концентрації уваги.

Ще одним суттєвим фактором ризику є статичне навантаження на опорно-руховий апарат. Тривале перебування у сидячому положенні без зміни пози призводить до перенапруження м’язів спини, шиї та плечового поясу. Неправильна організація робочого місця, невідповідна висота робочого столу або крісла,

відсутність підтримки поперекового відділу хребта можуть спричиняти розвиток порушень постави, болю в спині та захворювань опорно-рухового апарату [31, 33]. Особливо небезпечним є поєднання статичного навантаження з недостатньою руховою активністю протягом робочого дня.

Не менш важливим фактором ризику є психоемоційне навантаження. Робота користувача комп'ютерної мережі часто пов'язана з високим рівнем відповідальності, необхідністю швидкого прийняття рішень, обробкою великих обсягів інформації та дотриманням жорстких часових обмежень. За відсутності раціонального режиму праці це може призводити до перевтоми нервової системи, підвищеного рівня стресу, емоційного виснаження та зниження працездатності [34]. Тривалий психоемоційний стрес негативно впливає на загальний стан здоров'я та може спричиняти порушення сну й концентрації уваги.

Окрему групу факторів ризику становлять несприятливі параметри мікроклімату робочого приміщення. Недостатня вентиляція, підвищена або знижена температура повітря, низька вологість негативно впливають на фізіологічний стан користувача, спричиняючи швидку втому, зниження працездатності та дискомфорт. Забезпечення оптимальних параметрів мікроклімату є важливою умовою підтримання нормального функціонування організму людини під час тривалої роботи з комп'ютерною технікою [35].

Вплив на здоров'я користувачів має також організація робочого простору з точки зору ергономіки та естетики. Захаращене робоче місце, невдалий підбір кольорової гами та низькі коефіцієнти відбиття поверхонь можуть негативно впливати на зорове сприйняття та психологічний стан людини. Рекомендовані коефіцієнти відбиття для стелі, стін і підлоги дозволяють створити сприятливе світлове середовище, зменшити навантаження на органи зору та підвищити комфортність праці [36].

Таким чином, користувачі комп'ютерної мережі зазнають впливу комплексу факторів ризику, серед яких провідне місце займають зорове та психоемоційне навантаження, статичний характер праці, несприятливі умови освітлення та мікроклімату. За відсутності належної організації робочого місця ці фактори

можуть призводити до порушень зору, захворювань опорно-рухового апарату, розладів нервової системи та загального зниження працездатності. Дотримання ергономічних вимог, раціональний режим праці та забезпечення комфортних умов роботи є необхідними передумовами збереження здоров’я користувачів комп’ютерної мережі [29–36].

ВИСНОВКИ

У результаті виконання магістерської кваліфікаційної роботи було розроблено вебсистему централізованого зберігання файлів із підтримкою семантичного пошуку, яка поєднує класичні механізми управління файловими ресурсами з сучасними підходами до інтелектуального пошуку інформації. Створена система забезпечує повний цикл роботи з файлами, починаючи з їх завантаження у сховище та організації у каталоги і закінчуючи пошуком, попереднім переглядом, завантаженням і наданням контрольованого доступу іншим користувачам. Використання сучасних вебтехнологій дозволило реалізувати гнучке, масштабоване та надійне програмне рішення, орієнтоване на роботу з великими обсягами даних.

На початковому етапі дослідження було проведено ґрунтовний аналіз предметної області, у ході якого визначено основні проблеми традиційних систем зберігання файлів, зокрема складність пошуку інформації у великих сховищах та обмежені можливості роботи зі змістом документів. На основі цього аналізу сформульовано функціональні та нефункціональні вимоги до системи, що охоплюють питання зручності використання, безпеки, продуктивності та масштабованості. Отримані результати стали основою для проєктування архітектури системи та вибору технологічного стеку.

У процесі розробки було спроєктовано та реалізовано серверну частину системи з використанням Node.js і фреймворка NestJS, що дозволило побудувати модульну та добре структуровану архітектуру. Для зберігання метаданих файлів, користувачів і структури каталогів було використано реляційну базу даних PostgreSQL, яка забезпечує надійність, цілісність даних та ефективну роботу з великими наборами записів. Контейнеризація за допомогою Docker спростила процес розгортання системи та створила передумови для її подальшого масштабування.

Значна увага в роботі була приділена реалізації функціоналу роботи з файлами. Система підтримує завантаження файлів у сховище з можливістю вибору

цільового каталогу, збереження ієрархічної структури папок та обробку різних типів файлів. Реалізовано механізми попереднього перегляду документів і зображень без необхідності їх завантаження на локальний пристрій, що підвищує зручність роботи користувача. Окремо реалізовано функції завантаження файлів зі сховища, роботи з кошиком та відновлення або остаточного видалення даних, що мінімізує ризик випадкової втрати важливої інформації.

Ключовою особливістю розробленої системи є впровадження семантичного пошуку, який суттєво розширює можливості взаємодії користувача з файловим сховищем. Для реалізації цього механізму використано ElasticSearch у поєднанні з векторними представленнями тексту, згенерованими за допомогою OpenAI embeddings. Такий підхід дозволяє здійснювати пошук файлів на основі їхнього змісту, а не лише за назвами або атрибутами. У роботі також реалізовано гібридний підхід до пошуку, що поєднує прямий пошук за іменами файлів у базі даних із семантичним аналізом контенту, забезпечуючи більш релевантні та інформативні результати.

Важливим аспектом реалізації системи є забезпечення безпеки даних користувачів. У системі впроваджено механізми реєстрації та автентифікації з використанням токенів доступу, зберігання паролів у хешованому вигляді та контроль доступу до файлових ресурсів. Реалізація тимчасових посилань для завантаження файлів дозволяє надавати доступ третім особам із чіткими обмеженнями у часі, що підвищує рівень захисту інформації.

Для перевірки коректності реалізації та оцінки якості програмного продукту було проведено комплексне тестування, зокрема модульне тестування серверної частини. Unit-тести охопили основні бізнес-логіки системи, включаючи роботу з файлами, каталогами, пошуком і механізмами доступу. Результати тестування підтвердили стабільність роботи системи, відповідність заявленим вимогам і готовність розробленого рішення до практичного використання.

Загалом, розроблена вебсистема демонструє ефективне поєднання традиційних підходів до управління файлами з сучасними технологіями семантичного пошуку та штучного інтелекту. Отримані результати підтверджують

доцільність використання векторних представлень тексту для пошуку інформації у файлових сховищах. Набутий під час виконання роботи досвід є цінним для подальшої професійної діяльності у сфері веброзробки, проєктування інформаційних систем і застосування технологій для обробки та аналізу даних.

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. NestJS 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. Whatt 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. Elastic Docs [Електронний ресурс] – Режим доступу до ресурсу: https://www.elastic.co/docs.

17. OpenAI embeddings [Електронний ресурс] – Режим доступу до ресурсу: https://platform.openai.com/docs/guides/embeddings.

18. Automated Parallelization of Software for Identifying Parameters of Intraparticle Diffusion and Adsorption in Heterogeneous Nanoporous Media : дис. канд. техн. наук / ., 2023.

19. Mykhalyk D. Automated processing and analysis of medical texts / Mykhalyk D., 2023.

20. Методичні вказівки до виконання кваліфікаційної роботи магістра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі: Михалик Д.М., Цуприк Г.Б., Бревус В.М., Мудрик І.Я. – Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 44 с. URL: https://elartu.tntu.edu.ua/handle/lib/50316

21. Документація UML [Електронний ресурс] – Режим доступу: http://www.uml.org/.

22. Закон України «Про охорону праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/2694-12.

23. ДНАОП 0.00-4.15-98 «Про затвердження Положення про розробку інструкцій з охорони праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0226-98.

24. НПАОП 0.00-4.12-05 «Типове положення про порядок проведення навчання і перевірки знань з питань охорони праці». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0231-05.

25. НПАОП 0.00-7.15-18 «Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0508-18.

26. ДНАОП 0.00-1.21-98 «Про затвердження Правил безпечної експлуатації електроустановок споживачів» [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0093-98.

27. НПАОП 0.00-7.11-12 «Загальні вимоги стосовно забезпечення роботодавцями охорони праці працівників» [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0226-12.

28. ДБН В.2.5-28:2018 «Природне і штучне освітлення». [Електронний ресурс] URL: https://zakon.rada.gov.ua/laws/show/z0000-18.

29. Ергономічні вимоги для організації робочого місця. Портал споживача. URL: https://www.gpp.in.ua/robota/ergonomichni-vimogi-dlyaorganizatsiji-robochogo-mistsya (дата звернення: 05.06.2023).

30. ДСТУ 8604:2015. Дизайн і ергономіка. Робоче місце для виконання робіт у положенні сидячи. Загальні ергономічні вимоги. На заміну ГОСТ 12.2.032-78 ; чинний від 2017-07-01. Вид. офіц. Київ, 2015. 9 с. URL: http://online.budstandart.com/ua/catalog/doc-page.html?id_doc=71028 (дата звернення: 05.06.2025).

31. Бедрій І.Я., Нечай В.Я. Безпека життєдіяльності. Навчальний посібник. – Львів: Манголія 2006, 2007. 499 с.

32. Основи охорони праці. / Під ред. Ткачука К.Н., Халімовського Н.О. – К.: Основа, 2006. 448 с.

33. Стручок В.С. Методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання «БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ» / Тернопіль: ФОП Паляниця В. А., –156 с. Отримано з https://elartu.tntu.edu.ua/handle/lib/39196.

34. Стручок В.С. Навчальний посібник «ТЕХНОЕКОЛОГІЯ ТА ЦИВІЛЬНА БЕЗПЕКА. ЧАСТИНА «ЦИВІЛЬНА БЕЗПЕКА»» / Тернопіль: ФОП Паляниця В. А., – 156 с. Отримано з http://elartu.tntu.edu.ua/handle/lib/39424.

ДОДАТКИ

ДОДАТОК А

Апробація результатів

УДК 004.41

Солтис М.В. – ст. гр. СП-61

Тернопільський національний технічний університет імені Івана Пулюя

ВЕБСИСТЕМА ЦЕНТРАЛІЗОВАНОГО ЗБЕРІГАННЯ ФАЙЛІВ ІЗ ПІДТРИМКОЮ СЕМАНТИЧНОГО ПОШУКУ

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

Soltys M.

Ternopil Ivan Puluj National Technical University

WEB SYSTEM FOR CENTRALIZED FILE STORAGE WITH SEMANTIC SEARCH SUPPORT

Supervisor: PhD in Technical Sciences, Professor Mykhalyk D.

Ключові слова: семантичний пошук, централізоване зберігання даних, вебсистема, індексація файлів, векторні ембедінги, ElasticSearch.

Keywords: semantic search, centralized data storage, web system, file indexing, vector embeddings, ElasticSearch.

Моя науково-дослідна робота присвячена розробці вебсистеми для централізованого зберігання файлів із підтримкою семантичного пошуку. Основною метою є створення інтелектуальної платформи, яка не лише зберігає дані, а й забезпечує їх змістовне пошукове опрацювання за допомогою сучасних алгоритмів векторного представлення тексту. Така система значно підвищує ефективність взаємодії користувачів із великими обсягами інформації, дозволяючи шукати файли не лише за назвою, а й за змістом.

На початковому етапі проводиться аналіз потреб користувачів та вимог до структури даних, визначаються ключові сценарії взаємодії із системою. На основі цього формується архітектура, яка включає сервіс обробки файлів, модуль індексації контенту та клієнтську частину з інтуїтивно зрозумілим інтерфейсом. Для клієнтської частини використовується фреймворк Next.js, тоді як серверна логіка реалізована на NestJS з використанням TypeScript та PostgreSQL.

Особлива увага приділяється реалізації семантичного пошуку. Для цього застосовується ElasticSearch, який зберігає індекси файлів і підтримує як повнотекстовий, так і векторний пошук. Векторизація вмісту виконується за допомогою OpenAI Embeddings, що забезпечує розуміння змістового контексту документів. Такий підхід дає можливість знаходити релевантні файли навіть тоді, коли користувач не знає точну назву документа або вводить запит у довільній формі.

Під час розробки системи значна увага приділяється безпеці даних, включаючи авторизацію на основі JWT-токенів, контроль доступу до файлів та обмеження прав для зовнішніх посилань на завантаження. Також реалізуються інструменти обробки файлів різних форматів (PDF, DOCX, зображення), що дозволяє індексувати текст із більшості поширених типів даних.

Завершальним етапом є тестування сервісу, яке включає перевірку коректності індексації, точності семантичного пошуку, швидкодії, масштабованості та стійкості системи до великих навантажень. Тестування дозволяє оптимізувати роботу алгоритмів та гарантує стабільність системи в реальних умовах.

Результати проведеної роботи свідчать про те, що поєднання централізованого сховища з інформаційним пошуком на основі векторних представлень значно підвищує ефективність роботи з файлами та відкриває широкі можливості для подальших досліджень у сфері інтелектуальних інформаційних систем.

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 Б

Структура проєкту

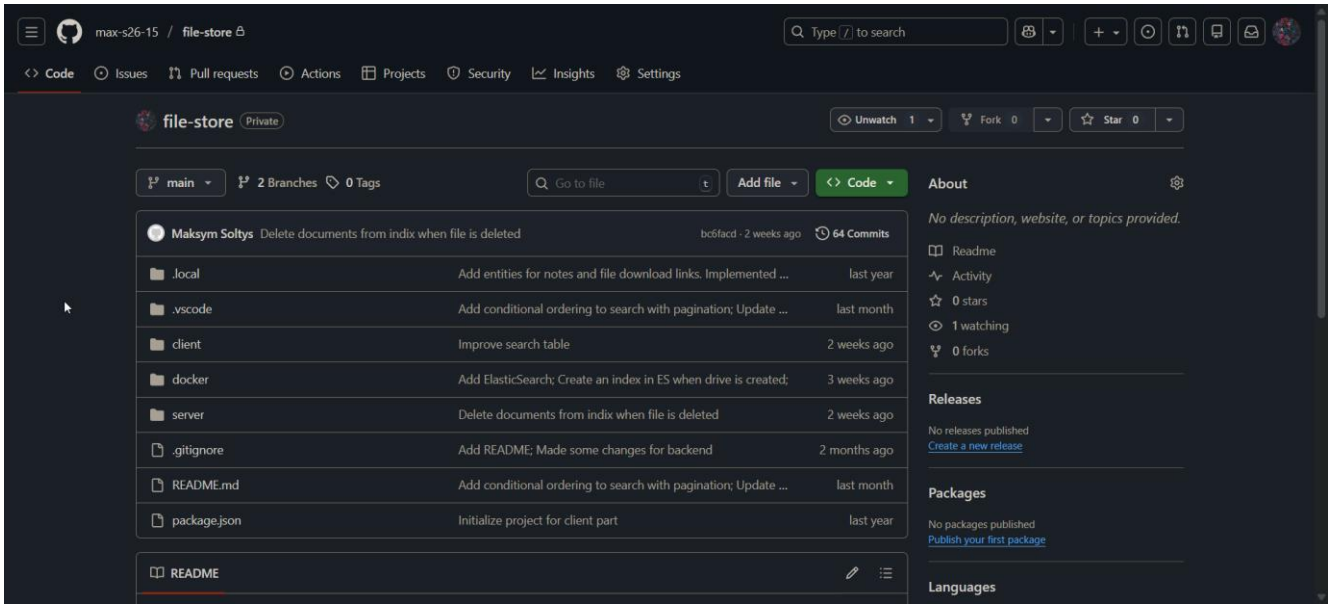

Рисунок Б.1 – Структура проекту на GitHub

Посилання на проєкт: https://github.com/max-s26-15/file-store