Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

Допущено до захисту
Завідувач кафедри комп'ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 01 » червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
спеціальність 122 Комп'ютерні науки
освітня програма 122.00.01 Інформатика

Тема: ВЕБ-СЕРВІС ЗБЕРІГАННЯ ТА УПРАВЛІННЯ КОРИСТУВАЦЬКИМИ ФАЙЛАМИ З ПЕРСОНАЛІЗОВАНИМ ДОСТУПОМ

Виконав
студент групи Інб-1-22-4.0д
Савчук Віталій Миколайович

_____
(підпис)
Науковий керівник
кандидат педагогічних наук. доцент
Лілія Олександрівна Варченко-Троценко

_____
(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп'ютерних наук
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 31» жовтня 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

«ВЕБ-СЕРВІС ЗБЕРІГАННЯ ТА УПРАВЛІННЯ КОРИСТУВАЦЬКИМИ ФАЙЛАМИ З ПЕРСОНАЛІЗОВАНИМ ДОСТУПОМ»

Виконавець – студент групи ІНб-1-22-4.0д,
спеціальності 122 Комп'ютерні наук,
освітньої програми 122.00.01 Інформатика
Савчук Віталій Миколайович

1. Вихідні дані: онлайн публікації з тем хмарного зберігання даних та персоналізованого доступу; технічна документація Spring Boot, React.js, PostgreSQL, MinIO та Redis; документація протоколів HTTPS/TLS, BCrypt, RBAC та ownership-based access control; прикладні дані отримані в процесі проєктування, реалізації та тестування веб-сервісу. 2. Основні завдання: провести аналіз предметної області та порівняння існуючих рішень (Google Drive, Dropbox, OneDrive, pCloud, MEGA); спроєктувати архітектуру системи та ownership-based модель доступу; реалізувати серверну частину на Spring Boot та клієнтську на React.js; впровадити механізми безпеки (TLS 1.3, BCrypt, сесійні cookie); провести функціональне, навантажувальне тестування та тестування на безпеку.

3. Пояснювальна записка: обсяг до 61 стор. формату А4 комп'ютерного набору з дотриманням вимог стандарту і методичних рекомендацій кафедри.

4. Графічні матеріали: презентація результатів бакалаврської роботи.

5. Додатки: фрагменти коду backend (Spring Boot) та frontend (React.js); схеми архітектури та моделі доступу; результати тестування системи.

6. Строк подання роботи на кафедру: «_0 1_» ___06___ 2026 р.

Науковий керівник	Виконавець:
к.п.н., доцент	
_____Лілія В.О.	_____Савчук В.М.
31 10 2025 дата	31 10 2025 дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапу дипломної роботи	Строк виконання етапу роботи	Примітка про виконання
1	Затвердження теми кваліфікаційної роботи бакалавра	31.10.2025	виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей і завдань	01.11.2025-11.06.2026	виконано
3	Аналіз процесів зберігання, управління та надання доступу до користувацьких файлів	26.01.2026-15.03.2026	виконано
4	Дослідження архітектури та моделей даних	16.03.2026-31.03.2026	виконано
5	Розробка програмного продукту	01.04.2026-15.04.2026	виконано
6	Тестування програмного продукту	16.04.2026-30.04.2026	виконано
7	Впровадження програмного продукту	01.05.2026-15.05.2026	виконано
8	Підготовка пояснювальної записки, графічних матеріалів, додатків	25.05.2026-12.06.2026	виконано
9	Захист кваліфікаційної роботи	16.06.2026	виконано

«Затверджую»
Науковий керівник
к.п.н., доцент, Лілія В.О.

Індивідуальний план склав
Савчук В.М.

РЕФЕРАТ

Кваліфікаційна робота: 61 с., 15 рис., 4 табл., 31 посилання.

Актуальність: зумовлена стрімким зростанням обсягів персональних даних у цифровому середовищі та посиленням вимог до їх захисту відповідно до Закону України «Про захист персональних даних» і регламенту GDPR. Існуючі комерційні хмарні сервіси (Google Drive, Dropbox, OneDrive та інші) мають суттєві обмеження щодо гнучкості контролю доступу та конфіденційності даних, що зумовлює потребу у створенні власних незалежних рішень для зберігання і управління файлами з персоналізованим доступом.

Об'єкт дослідження: процеси зберігання, управління та надання доступу до користувацьких файлів у веб-орієнтованих системах.

Предмет дослідження: архітектура, моделі даних та механізми персоналізованого доступу у веб-сервісі хмарного зберігання файлів.

Мета роботи: розробити веб-сервіс для зберігання та управління користувацькими файлами з механізмами персоналізованого доступу, який забезпечує безпечне зберігання, гнучке керування правами та зручний інтерфейс для кінцевих користувачів.

Завдання:

1. проведено аналіз предметної області та порівняльний аналіз існуючих рішень;
2. спроєктовано архітектуру системи з ownership-based моделлю доступу;
3. реалізовано серверну частину на Spring Boot з Java та клієнтську частину на React.js;
4. впроваджено механізми безпеки (TLS 1.3, BCrypt);
5. проведено комплексне тестування системи.

Основні результати: розроблено веб-сервіс із технологічним стеком Spring Boot / React.js / PostgreSQL / MinIO / Redis. Реалізовано стримінгову обробку файлів, рекурсивні операції над директоріями, генерацію ZIP-архівів та

ownership-based модель ізоляції даних. Тестування підтвердило 100% проходження функціональних тест-кейсів, стабільну продуктивність під навантаженням до 50 одночасних користувачів та відповідність базовим вимогам безпеки.

Практичне значення: створено готовий до розгортання веб-сервіс, який може застосовуватися в освітніх закладах, невеликих організаціях чи для індивідуальних потреб - без залежності від комерційних хмарних платформ, із повним контролем користувача над власними даними.

Ключові слова: хмарне зберігання, веб-сервіс, персоналізований доступ, Spring Boot, React.js, PostgreSQL, MinIO, JWT, RBAC, ownership-based access control, безпека даних.

ПЕРЕЛІК ВИКОРИСТАНИХ СКОРОЧЕНЬ ТА ПОЗНАЧЕНЬ

2FA – двофакторна автентифікація (two-factor authentication)

ACL – список контролю доступу (access control list)

AES – стандарт шифрування Advanced Encryption Standard

API – інтерфейс прикладного програмування (application programming interface)

CTE – загальні табличні вирази (common table expressions)

GDPR – Загальний регламент захисту даних (General Data Protection Regulation)

HTTPS – захищений протокол передачі гіпертексту (HyperText Transfer Protocol Secure)

JWT – JSON Web Token

RBAC – рольовий контроль доступу (role-based access control)

RLS – безпека на рівні рядків (row-level security)

TLS – безпека транспортного рівня (Transport Layer Security)

ЗМІСТ

ВСТУП

Актуальність теми. У сучасному цифровому світі обсяг даних, що генеруються користувачами, стрімко зростає. Згідно з прогнозами аналітичних агентств, до 2025 року глобальний обсяг даних досягне сотень зетабайтів, значна частина яких належить індивідуальним користувачам у вигляді особистих файлів, фотографій, документів та мультимедійного контенту [30]. Традиційні методи зберігання на локальних носіях (жорсткі диски, флеш-накопичувачі) мають суттєві недоліки: обмежену ємність, ризик втрати даних через апаратні збої, відсутність доступу з різних пристроїв та складність організації спільного доступу.

Хмарні сервіси зберігання файлів (Google Drive, Dropbox, OneDrive, iCloud та інші) частково розв'язують ці проблеми, забезпечуючи віддалене зберігання, синхронізацію між пристроями та базові механізми спільного доступу. Проте більшість комерційних рішень мають обмеження щодо конфіденційності даних, оскільки постачальники сервісів мають технічну можливість доступу до користувацького контенту. Крім того, механізми персоналізованого доступу часто є недостатньо гнучкими: вони обмежуються простими рівнями прав (перегляд, редагування, коментування) і не дозволяють детально налаштувати доступ на рівні окремих файлів чи папок з урахуванням ролей користувачів, тимчасових обмежень чи контекстних умов.

У контексті посилення вимог до захисту персональних даних (зокрема, відповідно до Закону України «Про захист персональних даних»[31] та європейського регламенту GDPR) актуальним стає розробка веб-сервісів, які поєднують зручність хмарного зберігання з розширеними можливостями контролю доступу та підвищеним рівнем безпеки. Створення власного веб-сервісу зберігання та управління користувацькими файлами з персоналізованим доступом дозволяє уникнути залежності від комерційних платформ, забезпечити повний контроль над даними та адаптувати функціональність під конкретні потреби користувачів.

Мета бакалаврської роботи – розробити веб-сервіс для зберігання та управління користувацькими файлами з механізмами персоналізованого доступу, який забезпечує безпечне зберігання, гнучке керування правами доступу та зручний інтерфейс для кінцевих користувачів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз предметної області та існуючих рішень хмарного зберігання файлів;
- виконати порівняльний аналіз функціональних можливостей та механізмів доступу в аналогічних системах;
- сформулювати вимоги до персоналізованого доступу та безпеки даних;
- спроєктувати архітектуру веб-сервісу, структуру бази даних та модель прав доступу;
- реалізувати основні функціональні модулі системи, включаючи завантаження, зберігання, управління файлами та автентифікацію користувачів;
- забезпечити механізми безпеки, зокрема шифрування та захищені протоколи передачі даних;
- провести тестування розробленого сервісу та проаналізувати його ефективність.

Об’єктом дослідження є процеси зберігання, управління та надання доступу до користувацьких файлів у веб-орієнтованих системах.

Предметом дослідження є архітектура, моделі даних та механізми персоналізованого доступу у веб-сервісі хмарного зберігання файлів.

Методи дослідження. У роботі застосовуються методи системного аналізу (для вивчення предметної області та існуючих рішень), порівняльного аналізу (для оцінки аналогів), об’єктно-орієнтованого проєктування (для моделювання системи), програмної реалізації (на основі сучасних веб-технологій) та тестування (функціональне, навантажувальне та на безпеку).

Наукова новизна роботи полягає в удосконаленні архітектури вебсервісів хмарного зберіганя даних через інтеграцію S3-сховища з моделлю доступу на

основі(ownership-based access control),що дозволить забезпечити сувору ізоляцію персональних даних користувачів та оптимізувати обробку великих файлів.

Практична значущість отриманих результатів полягає в створенні готового до використання веб-сервісу, який може застосовуватися в освітніх закладах, невеликих організаціях чи для особистих потреб з метою безпечного зберігання та спільного використання файлів без залежності від комерційних хмарних платформ.

Структура роботи. Бакалаврська робота складається зі вступу, трьох основних розділів, висновків, списку використаних джерел та додатків. У першому розділі проведено аналіз предметної області та постановку завдання. Другий розділ присвячено проєктуванню веб-сервісу. Третій розділ описує реалізацію та тестування системи.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАВДАННЯ

1.1 Аналіз предметної області зберігання та управління файлами в хмарних сервісах

Предметна область зберігання та управління користувацькими файлами в хмарних сервісах є однією з ключових складових сучасних інформаційних технологій, що забезпечує віддалене зберігання, синхронізацію та доступ до даних з різних пристроїв. Хмарні сервіси дозволяють користувачам уникнути обмежень локальних носіїв інформації, таких як обмежена ємність, ризик втрати даних через апаратні збої чи фізичні пошкодження, а також забезпечують зручність спільного доступу та автоматичне резервне копіювання [1].

Згідно з сучасними дослідженнями, концепція персональних даних у хмарних сховищах охоплює широкий спектр файлів, включаючи документи, фотографії, відео та інші мультимедійні об'єкти, що генеруються індивідуальними користувачами. Особлива увага приділяється питанням конфіденційності та контролю доступу, оскільки традиційні комерційні сервіси часто надають постачальникам технічну можливість доступу до даних [2].

Огляд ринку хмарного зберігання свідчить про стрімке зростання обсягів даних: глобальний ринок хмарного зберігання оцінювався в 132,03 млрд дол. США у 2024 році та прогнозується на рівні 161,28 млрд дол. США у 2025 році з подальшим зростанням до 639,40 млрд дол. США до 2032 року за середньорічним темпом зростання 21,7% [3]. Значна частина цього зростання пов'язана з персональними даними, де сервіси на кшталт Sync.com, pCloud та Icedrive виділяються завдяки клієнтському шифруванню та zero-knowledge архітектурі, що забезпечує повний контроль користувача над ключами шифрування [4].

У контексті безпеки та конфіденційності, сучасні тенденції підкреслюють перехід до сервісів з end-to-end шифруванням, де дані захищені як під час передачі, так і на зберіганні. Провідні рішення, такі як Sync.com та pCloud,

пропонують zero-knowledge encryption, що запобігає доступу навіть постачальника до вмісту файлів, на відміну від популярних сервісів на кшталт Google Drive чи Dropbox, де шифрування керується серверною стороною [5].

В Україні використання хмарних сервісів активно розвивається, зокрема в освітніх та корпоративних середовищах, де акцент робиться на адаптивних хмаро орієнтованих системах для зберігання та управління даними. Це дозволяє забезпечити доступність інформації для вчителів та учнів, сприяючи цифровізації освіти [17].

Аналіз функціональних можливостей хмарних сервісів показує, що основними перевагами є масштабованість, автоматична синхронізація між пристроями та механізми спільного доступу. Однак, для персональних даних критичним залишається баланс між зручністю та захистом, де сервіси з клієнтським шифруванням, такі як pCloud та Internxt, демонструють вищий рівень приватності завдяки використанню протоколів на кшталт Twofish та відкритим аудитам безпеки [18].

Тенденції 2025 року вказують на посилення вимог до захисту персональних даних відповідно до регламентів на кшталт GDPR, що стимулює розвиток сервісів з регіональними дата-центрами та посиленим шифруванням. В Україні це актуально з огляду на необхідність дотримання національного законодавства про захист персональних даних та досвід використання хмар для резервного копіювання під час кризових ситуацій [24].

Управління файлами в хмарних сервісах включає функції завантаження, організації в папки, версійного контролю та відновлення видалених файлів. Сучасні рішення, такі як Icedrive, пропонують інтеграцію з локальними пристроями та необмежену версійність, що підвищує надійність зберігання [25].

Предметна область характеризується переходом від простого зберігання до комплексних систем з акцентом на персоналізований доступ, де користувач може детально налаштовувати права для окремих файлів чи груп, поєднуючи зручність з високим рівнем безпеки [29]. Це створює передумови для розробки

спеціалізованих веб-сервісів, адаптованих до потреб індивідуальних користувачів та невеликих груп.

1.2 Огляд існуючих веб-сервісів хмарного зберігання файлів (Google Drive, Dropbox, OneDrive та інші)

Серед існуючих веб-сервісів хмарного зберігання файлів одним з найбільш поширених є Google Drive. Цей сервіс, розроблений компанією Google, надає користувачам 15 ГБ безкоштовного сховища, інтегроване з екосистемою Google Workspace, включаючи інструменти для створення та редагування документів, таблиць і презентацій безпосередньо в браузері. Google Drive підтримує синхронізацію файлів між пристроями, спільний доступ з гнучкими налаштуваннями прав та автоматичне резервне копіювання. Інтерфейс сервісу характеризується простотою та інтуїтивністю, з можливістю швидкого пошуку файлів завдяки інтеграції з пошуковою системою Google [19].

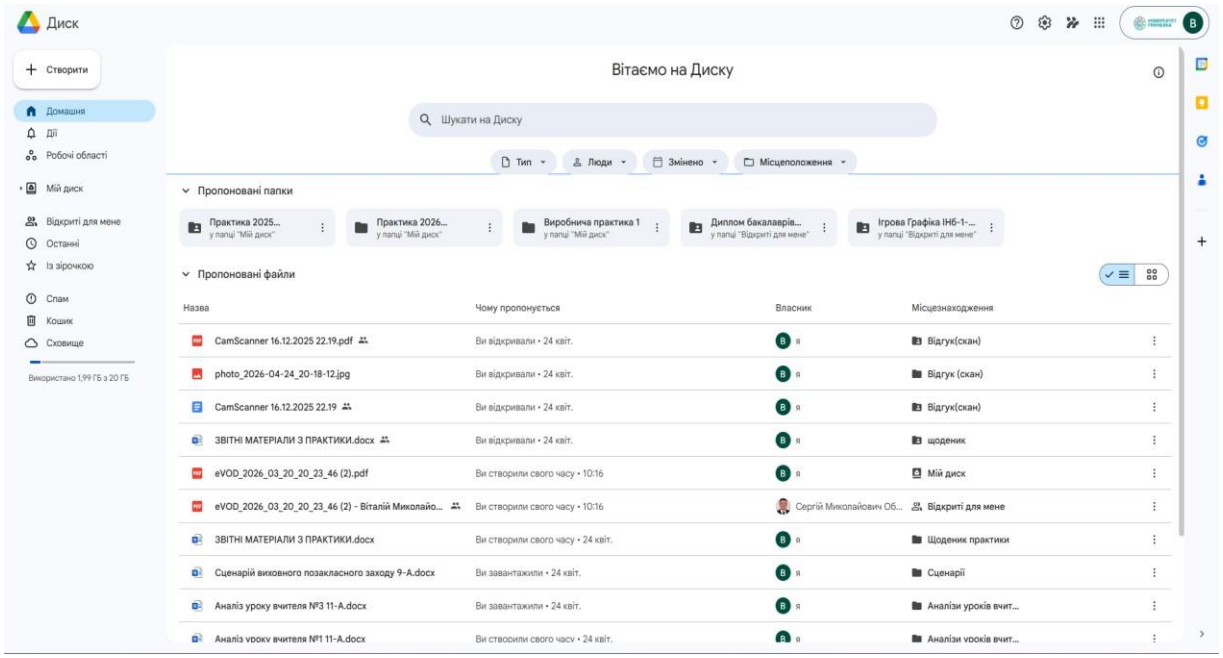

Рисунок 1.1. Головна сторінка веб-інтерфейсу Google Drive

Інший популярний сервіс – Dropbox – пропонує базовий безкоштовний план з 2 ГБ сховища, з акцентом на надійну синхронізацію та спільну роботу над файлами. Dropbox підтримує версійний контроль, відновлення видалених файлів

та інтеграцію з численними сторонніми додатками. Сервіс відомий функцією автомаичного збереження скріншотів та зручним механізмом обміну посиланнями на файли чи папки. Інтерфейс Dropbox є мінімалістичним, з фокусом на швидкий доступ до недавніх файлів та спільних папок.

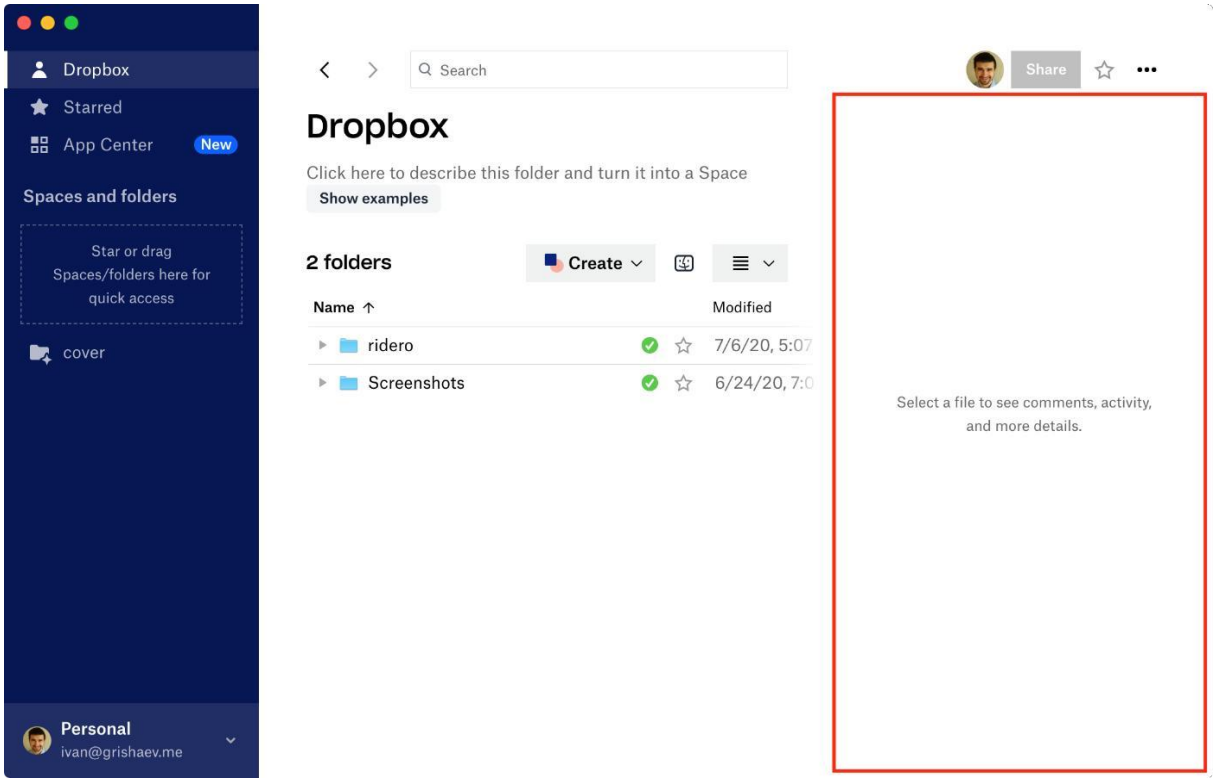

Рисунок 1.2. Головна сторінка веб-інтерфейсу Dropbox

Microsoft OneDrive, інтегрований з екосистемою Microsoft 365, надає 5 ГБ безкоштовного сховища для персональних користувачів. Сервіс тісно пов'язаний з Office Online, дозволяючи редагувати документи Word, Excel та PowerPoint у браузері. OneDrive підтримує автоматичну синхронізацію, спільний доступ та функцію відновлення файлів. У 2025 році інтерфейс сервісу отримав оновлення з покращеною швидкістю завантаження та інтеграцією штучного інтелекту для рекомендацій файлів.

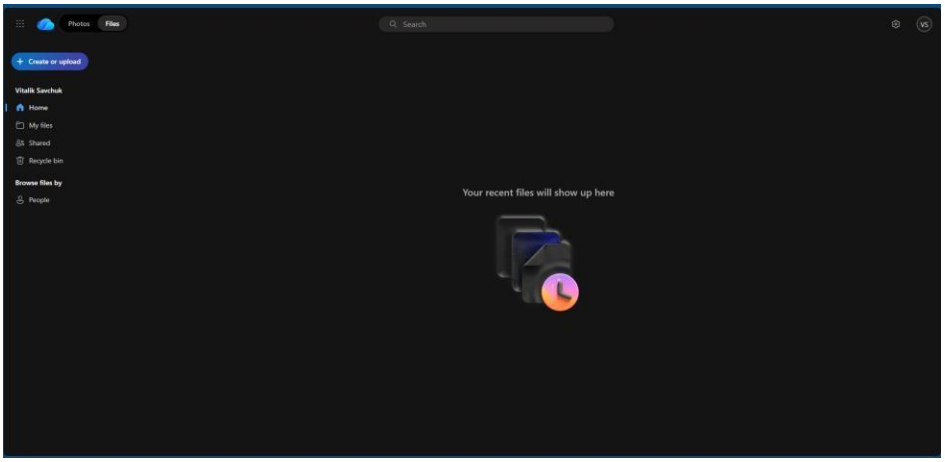

Рисунок 1.3. Головна сторінка веб-інтерфейсу Microsoft OneDrive

Серед альтернативних сервісів з акцентом на конфіденційність виділяється pCloud, який пропонує до 10 ГБ безкоштовного сховища та опцію клієнтського шифрування (Crypto Folder). pCloud підтримує довічну підписку, віртуальний диск для локального доступу без зайняття місця та автоматичне завантаження скріншотів. Сервіс базується в Швейцарії, що забезпечує високий рівень захисту даних відповідно до місцевих законів про конфіденційність [30].

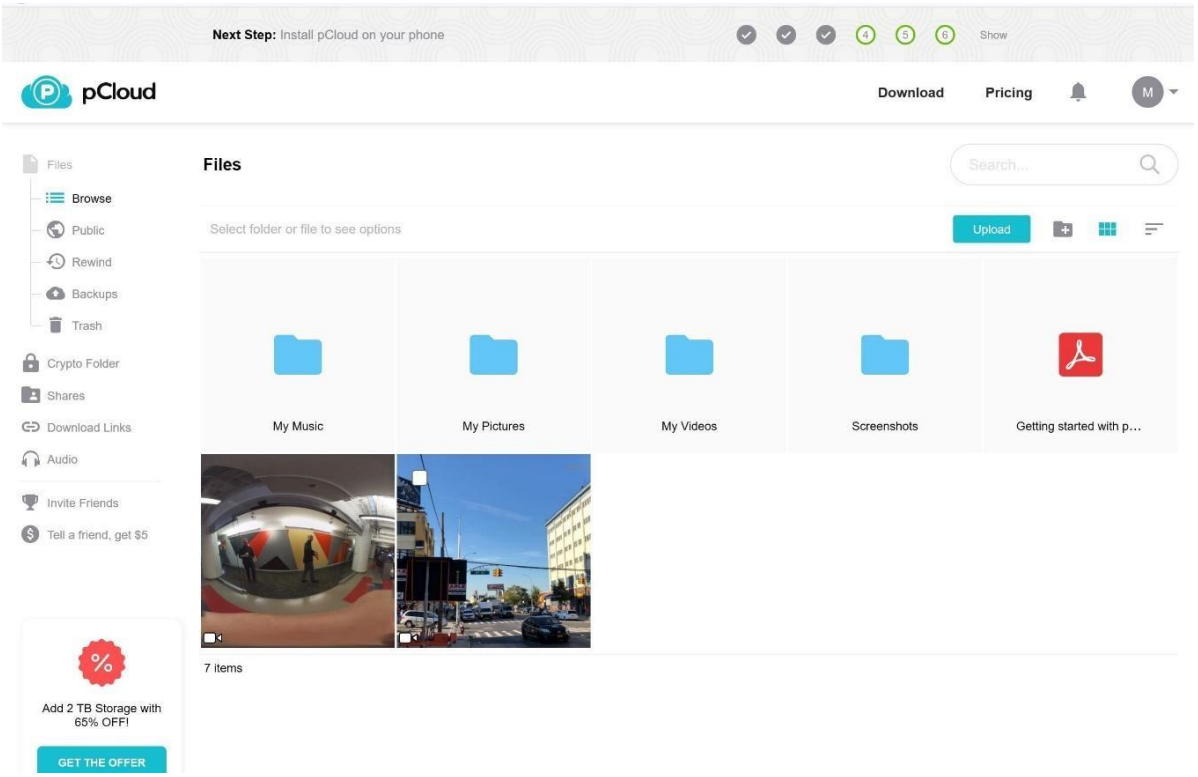

Рисунок 1.4. Головна сторінка веб-інтерфейсу pCloud

MEGA відомий щедрим безкоштовним планом до 20 ГБ та end-to-end шифруванням, де ключі зберігаються лише у користувача. Сервіс пропонує інтегрований чат, відеодзвінки та S3-сумісне об'єктне сховище (MEGA S4). MEGA фокусується на безпеці та швидкій передачі великих файлів, з підтримкою відновлення версій та нульовим знанням про вміст файлів з боку провайдера [40].

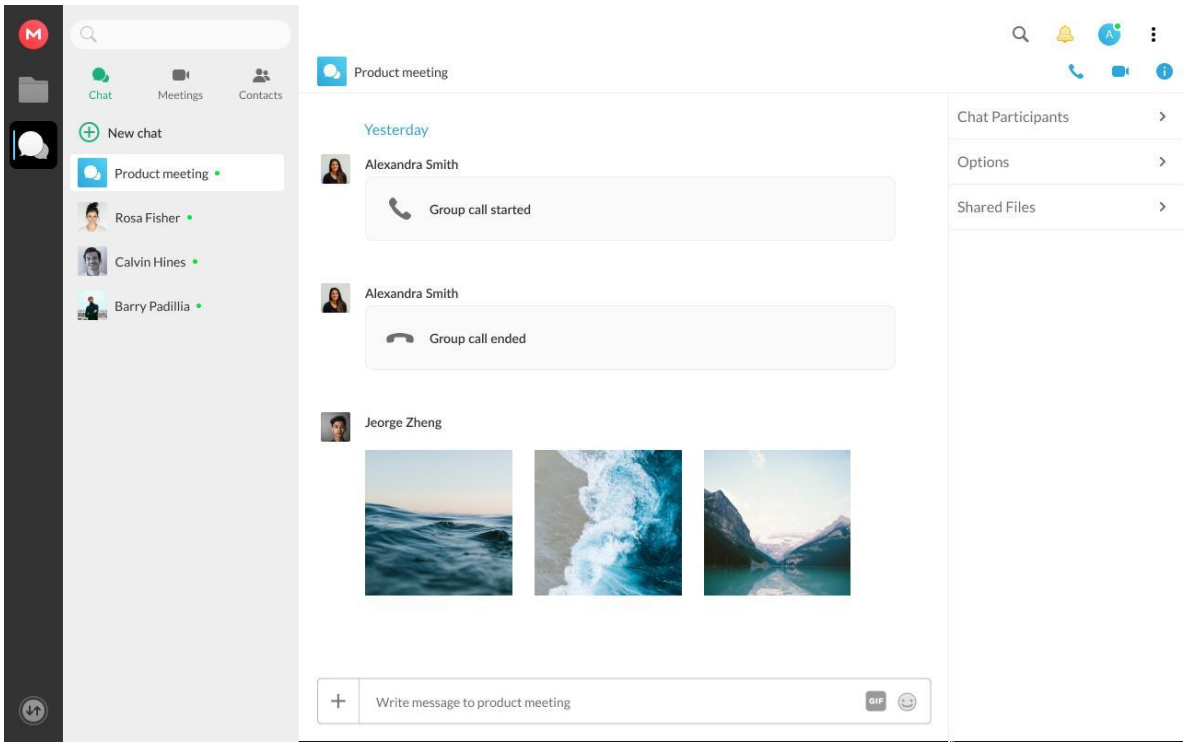

Рисунок 1.5. Головна сторінка веб-інтерфейсу MEGA

Apple iCloud призначений насамперед для користувачів екосистеми Apple, надаючи 5 ГБ безкоштовного сховища з тісною інтеграцією в iOS, macOS та веб-версію. Сервіс забезпечує синхронізацію фото, нотаток, контактів та файлів, з можливістю доступу через iCloud.com. У 2025 році веб-інтерфейс підтримує темний режим, кастомізацію домашньої сторінки та покращену навігацію в Photos [50].

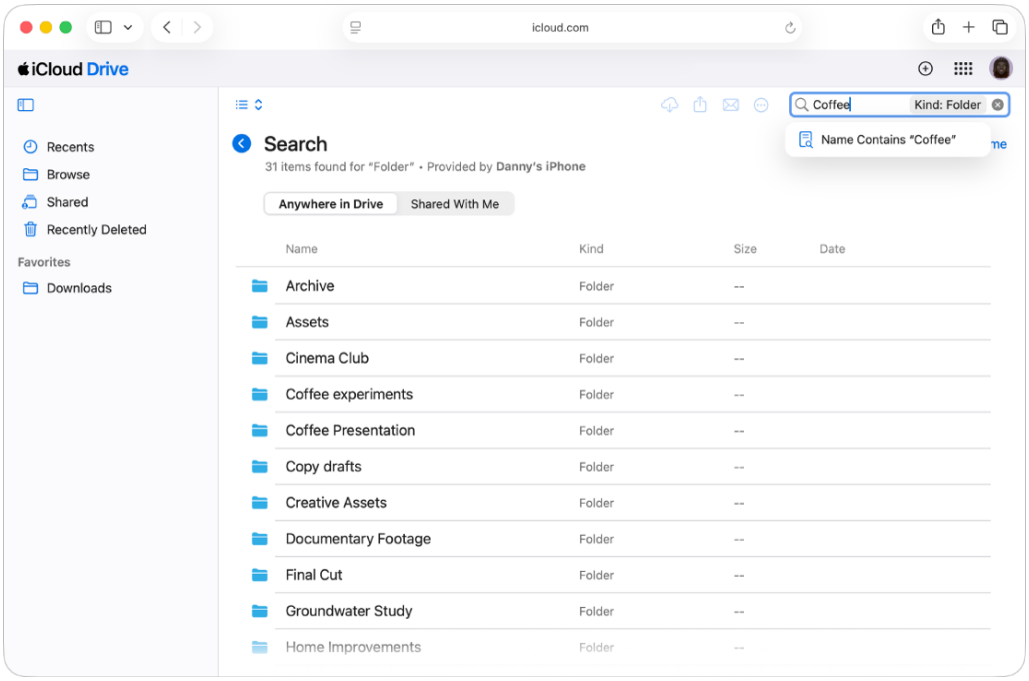

Рисунок 1.6. Головна сторінка веб-інтерфейсу iCloud

Ці сервіси демонструють різноманітність підходів до хмарного зберігання, від інтеграції з офісними інструментами до посиленого акценту на приватності та безпеці даних [3].

1.3 Порівняльний аналіз функціональних можливостей аналогічних систем, вимог до персоналізованого доступу, безпеки даних та постановка завдання на розробку веб-сервісу

Порівняльний аналіз існуючих веб-сервісів хмарного зберігання файлів свідчить про значну різноманітність у функціональних можливостях, механізмах персоналізованого доступу та рівні безпеки даних. Популярні сервіси, такі як Google Drive, Dropbox та Microsoft OneDrive, пропонують розвинені інструменти для спільної роботи, включаючи інтеграцію з офісними додатками та редагування документів у реальному часі, що робить їх зручними для командної взаємодії [6]. Водночас сервіси з акцентом на конфіденційність, наприклад pCloud та MEGA, надають пріоритет end-to-end шифруванню та zero-knowledge архітектурі, де постачальник не має доступу до ключів шифрування, забезпечуючи вищий рівень захисту персональних даних [10].

Щодо персоналізованого доступу, більшість сервісів підтримують базові рівні прав: перегляд, редагування та коментування для окремих файлів чи папок. Google Drive та Dropbox дозволяють встановлювати терміни дії посилань та захищати їх паролем, а OneDrive пропонує додатковий Personal Vault з біометричною автентифікацією для чутливих файлів [7]. Однак гнучкість у налаштуванні ролей та групових прав є обмеженою в комерційних рішеннях, де детальний контроль часто вимагає бізнес-планів. Системи на кшталт MEGA та pCloud пропонують розширені опції спільного доступу з end-to-end шифруванням, але без глибокої інтеграції з продуктивними інструментами [11].

Аналіз вимог до безпеки даних підкреслює необхідність відповідності стандартам, таким як GDPR, що є обов'язковим для обробки даних резидентів ЄС, включаючи випадки, коли сервіси використовуються в Україні [30]. Сервіси з zero-knowledge шифруванням, наприклад Sync.com та Proton Drive, забезпечують повну конфіденційність, оскільки дані шифруються на стороні клієнта [12]. У той же час популярні платформи, як Google Drive та Dropbox, використовують серверне шифрування, що дозволяє постачальнику потенційний доступ до даних, хоча вони відповідають GDPR та іншим нормам [13]. Вимоги до персоналізованого доступу включають рольовий контроль (RBAC), детальні ACL на рівні файлів та папок, а також підтримку тимчасових обмежень і контекстних умов для мінімізації ризиків несанкціонованого доступу [40].

Для наочності порівняння ключових характеристик наведено таблицю 1.1.

На основі проведеного аналізу виявлено, що існуючі сервіси або орієнтовані на зручність та колаборацію з компромісами в конфіденційності, або на максимальну безпеку з обмеженою функціональністю. Відсутність універсального рішення, яке поєднує гнучкий персоналізований доступ на рівні ролей та груп, клієнтське шифрування та адаптацію до вимог національного законодавства України про захист персональних даних, обґрунтовує необхідність розробки власного веб-сервісу.

Таблиця 1.1

Порівняльна характеристика основних веб-сервісів хмарного зберігання файлів

Сервіс	Безкоштовне сховище	End-to-end шифрування	Персоналізований доступ (ролі, паролі на посилання)	Інтеграція з офісними інструментами	Відповідність GDPR
Google Drive	15 ГБ	Ні	Базовий (перегляд, редагування, терміни дії)	Так (Google Workspace)	Так
Dropbox	2 ГБ	Ні	Розширений (паролі, терміни, брендування)	Так (інтеграції)	Так
OneDrive	5 ГБ	Ні (Personal Vault)	Розширений (біометрія, паролі)	Так (Microsoft 365)	Так
pCloud	10 ГБ	Так (опціонально)	Розширений (Crypto Folder)	Ні	Так
MEGA	20 ГБ	Так	Розширений (захищені посилання)	Ні	Так
iCloud	5 ГБ	Так (для більшості)	Базовий (сімейний доступ)	Так (Apple екосистема)	Так

Постановка завдання на розробку веб-сервісу полягає в створенні системи зберігання та управління користувацькими файлами з персоналізованим

доступом, яка забезпечує: клієнтське шифрування даних; рольовий контроль доступу з детальними правами на рівні файлів та папок; відповідність вимогам GDPR та українського законодавства; зручний інтерфейс для завантаження, організації та спільного використання файлів. Розробка орієнтована на індивідуальних користувачів та невеликі групи, з акцентом на незалежність від комерційних постачальників та повний контроль власника над даними.

1.4 Забезпечення безпеки та захисту даних (шифрування, протоколи передачі)

Забезпечення безпеки та захисту даних у веб-сервісах хмарного зберігання файлів є критичним аспектом, оскільки користувацькі дані можуть містити конфіденційну інформацію, що підпадає під вимоги законодавства про захист персональних даних. Основні загрози включають несанкціонований доступ, перехоплення даних під час передачі та компрометацію серверів постачальника послуг. Для мінімізації цих ризиків застосовуються механізми шифрування даних у стані спокою (at rest) та під час передачі (in transit), а також архітектури з різним рівнем контролю над ключами шифрування [8].

Шифрування даних у стані спокою передбачає захист файлів, що зберігаються на серверах. Найпоширенішим стандартом є алгоритм AES-256, який використовується як для серверного (server-side), так і для клієнтського (client-side) шифрування. Серверне шифрування виконується постачальником після отримання даних, що забезпечує базовий захист, але ключі шифрування знаходяться під контролем сервера, дозволяючи потенційний доступ до даних з боку провайдера чи при компрометації акаунта [9]. Навпаки, клієнтське шифрування (end-to-end або zero-knowledge) виконується на пристрої користувача перед завантаженням, де ключі залишаються виключно у користувача, забезпечуючи повну конфіденційність навіть від постачальника послуг [14].

Переваги клієнтського шифрування полягають у вищому рівні приватності: дані передаються та зберігаються в зашифрованому вигляді, а

провайдер не має доступу до plaintext. Це особливо актуально для сервісів на кшталт Proton Drive, Filen чи NordLocker, де застосовуються комбінації AES-256, xChaCha20-Poly1305 та інші алгоритми з zero-knowledge архітектурою [15]. Серверне шифрування, як у Google Cloud чи Microsoft Azure, є зручнішим для масштабування, але менш захищеним у випадку внутрішніх загроз чи регуляторних запитів [18].

Для захисту даних під час передачі обов'язковим є використання протоколу HTTPS з Transport Layer Security (TLS) версії 1.3, який забезпечує конфіденційність, цілісність та автентифікацію з'єднання. TLS 1.3 пропонує покращений handshake з одним round-trip, perfect forward secrecy та видалення вразливих шифрів, що робить його стійкішим до атак на відміну від попередніх версій [31]. Більшість сучасних хмарних сервісів, включаючи Amazon S3 та Google Cloud, підтримують TLS 1.3 для всіх з'єднань, рекомендуючи відмову від старіших протоколів [34].

В Україні захист персональних даних регулюється Законом «Про захист персональних даних», який гармонізується з GDPR, вимагаючи адекватних заходів безпеки, включаючи шифрування [20]. У контексті хмарних сервісів це означає необхідність відповідності принципам мінімізації ризиків та конфіденційності, де клієнтське шифрування дозволяє уникнути залежності від провайдера та забезпечити дотримання вимог щодо зберігання даних [23]. Крім того, для персоналізованого доступу рекомендується поєднання шифрування з рольовим контролем (RBAC) та двофакторною автентифікацією.

Для розроблюваного веб-сервісу доцільним є впровадження клієнтського шифрування з використанням AES-256 та TLS 1.3 для передачі, що забезпечить максимальний захист користувацьких файлів та відповідність сучасним стандартам безпеки [26], [28]. Це дозволить уникнути ризиків, пов'язаних з серверним шифруванням, та надати користувачам повний контроль над їхніми даними.

1.5 Висновки до розділу

У першому розділі бакалаврської роботи проведено комплексний аналіз предметної області зберігання та управління користувацькими файлами в хмарних сервісах. Виявлено, що сучасні хмарні рішення забезпечують зручний доступ до даних з різних пристроїв, автоматичну синхронізацію та базові механізми спільного використання, але мають суттєві відмінності в рівнях конфіденційності та безпеки.

Огляд існуючих веб-сервісів, таких як Google Drive, Dropbox, OneDrive, pCloud, MEGA та iCloud, показав різноманітність підходів: від глибокої інтеграції з продуктивними інструментами до акценту на end-to-end шифруванні та zero-knowledge архітектурі.

Порівняльний аналіз функціональних можливостей, механізмів персоналізованого доступу та безпеки даних виявив, що комерційні сервіси або жертвують конфіденційністю заради зручності колаборації, або пропонують високий захист з обмеженою функціональністю.

На основі проведеного дослідження сформульовано завдання на розробку веб-сервісу, який поєднуватиме зручність управління файлами з розширеними механізмами персоналізованого доступу та підвищеним рівнем захисту даних, орієнтованим на індивідуальних користувачів та невеликі групи. Це створить передумови для подальшого проєктування та реалізації системи в наступних розділах роботи.

РОЗДІЛ 2

ПРОЄКТУВАННЯ ВЕБ-СЕРВІСУ

2.1 Вибір технологічного стеку, обґрунтування архітектури системи та проєктування структури бази даних для зберігання користувацьких файлів та метаданих

Для реалізації веб-сервісу зберігання та управління користувацькими файлами з персоналізованим доступом обрано технологічний стек, який забезпечує високу продуктивність, безпеку, масштабованість та зручність розробки. Frontend-частина розроблена на React.js з використанням функціональних компонентів, хуків та бібліотек для управління станом (наприклад, React Query для запитів до API). Це дозволяє створити динамічний, реактивний інтерфейс з підтримкою drag-and-drop, прогрес-індикаторів та responsive design. Backend реалізовано на Spring Boot 3 з Java 21, що забезпечує потужну екосистему для обробки запитів, інтеграції безпеки (Spring Security) та роботи з зовнішніми сервісами. Для зберігання метаданих обрано реляційну СУБД PostgreSQL, для об'єктного зберігання файлів – MinIO (S3-сумісне сховище), для управління сесіями – Redis. Розгортання системи здійснюється за допомогою Docker Compose з Nginx як зворотним проксі для термінції TLS та маршрутизації.

Обґрунтування вибору стеку базується на вимогах до системи: необхідність ефективної обробки великих файлів (стримінг), суворої ізоляції даних користувачів, простоти розгортання та підтримки. Spring Boot обрано через вбудовану підтримку Spring Security для аутентифікації та авторизації, Spring Data JPA для роботи з базою даних та легку інтеграцію з MinIO Client. Java 21 забезпечує сучасні можливості (virtual threads для покращення concurrency). React.js є оптимальним для клієнтської частини завдяки компонентному підходу та великій екосистемі. MinIO обрано як self-hosted альтернативу Amazon S3, що дозволяє уникнути залежності від зовнішніх провайдерів та забезпечити контроль над даними. PostgreSQL перевершує альтернативи для структурних

метаданих завдяки підтримці транзакцій, індексів та композитних ключів. Redis необхідний для розподіленого зберігання сесій у потенційно масштабованої конфігурації.

Архітектура системи побудована за клієнт-серверною моделлю з елементами мікросервісного підходу. Клієнтська частина (React) взаємодіє з backend через RESTful API (/api/**) за протоколом HTTPS, надсилаючи запити з сесійними cookie. Backend обробляє аутентифікацію, перевірку доступу та бізнес-логіку, взаємодіючи з PostgreSQL для метаданих, MinIO для файлів (через presigned URL або пряме стримінгове завантаження) та Redis для сесій. Nginx виконує термінцію TLS, маршрутизацію (static для frontend, /api для backend) та базовий захист (rate limiting). Така архітектура забезпечує ізоляцію даних (ownership-based access control), ефективну обробку великих файлів без буферизації на application server та горизонтальну масштабованість.

Проєктування структури бази даних орієнтоване на ефективне зберігання метаданих файлів та віртуальних директорій з урахуванням ізоляції за користувачем. Використано єдину таблицю resources для моделювання як файлів, так і директорій (патерн "single table inheritance" або просто enum для типу). Основні поля таблиці:

- id (BIGINT, PRIMARY KEY, автоінкремент);
- path (VARCHAR, нормалізований шлях до батьківської директорії, закінчується на "/" для директорій);
- name (VARCHAR, ім'я файлу чи папки);
- size (BIGINT, розмір у байтах, NULL для директорій);
- type (ENUM: 'FILE', 'DIRECTORY');
- owner_id (BIGINT, FOREIGN KEY на users.id);
- created_at, updated_at (TIMESTAMP).

Унікальність забезпечується композитним унікальним індексом на (owner_id, path, name). Ієрархія директорій моделюється через поле path, з використанням LIKE-запитів для рекурсивного доступу до нащадків (наприклад,

path LIKE '/parent/%'). Додаткова таблиця users містить: id, username, email, password_hash (BCrypt), created_at.

Для оптимізації запитів створено індекси на owner_id, path та (owner_id, path). Міграції схеми керуються Flyway, що забезпечує версійний контроль. Така структура дозволяє ефективно виконувати операції listing, пошук (LIKE %query%), рекурсивне видалення/переміщення та перевірку конфліктів імен, зберігаючи простоту та масштабованість.

Запропонована архітектура та структура бази даних забезпечують надійне, безпечне та ефективне зберігання користувацьких файлів і метаданих, з повною ізоляцією даних та можливістю подальшого розширення (наприклад, додавання версійності чи шарінгу).

2.2 Моделювання системи персоналізованого доступу (ролі, права, токени автентифікації)

Моделювання системи персоналізованого доступу в розроблюваному веб-сервісі базується на принципі ізоляції даних кожного користувача з використанням сесійної автентифікації та прив'язки ресурсів до власника. У системі передбачено лише одну роль – користувач (user), що спрощує архітектуру та виключає необхідність диференціації прав на рівні ролей. Усі зареєстровані користувачі мають однаковий набір функціональних можливостей: реєстрацію, автентифікацію, завантаження, управління та доступ до власних файлів і директорій. Відсутність додаткових ролей, таких як адміністратор, обумовлена орієнтацією сервісу на персональне використання та невеликі групи, де не потрібне централізоване управління.

Автентифікація користувачів реалізується за допомогою серверних HTTP-сесій, підтримуваних Spring Security та Redis для зберігання сесійних даних. Процес автентифікації включає:

- реєстрацію (sign-up) з створенням облікового запису та хешуванням пароля;
- вхід (sign-in) з перевіркою credentials та створенням сесії;

– вихід (sign-out) з інвалідацією сесії.

Після успішної автентифікації сервер створює сесію, ідентифікатор якої зберігається в cookie браузера користувача. Усі подальші запити до захищених ендпоінтів (/api/resource, /api/directory тощо) перевіряються Spring Security на наявність валідної сесії. Це забезпечує stateful аутентифікацію без використання stateless-токенів, таких як JWT, що полегшує управління сесіями в розподіленому середовищі з Redis.

Персоналізований доступ до файлів та директорій моделюється через механізм власності (ownership-based access control). Кожен ресурс (файл або директорія) в базі даних пов'язаний з ідентифікатором власника (owner_id або аналогічне поле в ентіті). Доступ до операцій над ресурсом (перегляд, завантаження, перейменування, видалення) дозволяється виключно користувачу, чия сесія відповідає власнику ресурсу. Метадані ресурсів зберігаються в PostgreSQL з полями, що включають шлях (path), ім'я (name), розмір (size для файлів) та тип (FILE/DIRECTORY). Фактичні файли розташовані в MinIO, доступ до яких генерується динамічно через підписані URL лише для автентифікованого власника.

Модель прав доступу є простою та не включає granularний шарінг (надання доступу іншим користувачам) чи тимчасові посилання, оскільки сервіс орієнтований на індивідуальне зберігання. Права визначаються неявно:

– власник ресурсу – повні права (читання, запис, видалення, організація в директорії);
– неавтентифіковані або інші користувачі – відсутність доступу (запити відхиляються на рівні Spring Security).

Такий підхід забезпечує високий рівень персоналізації, оскільки кожен користувач працює в ізольованому просторі, захищеному від несанкціонованого доступу інших акаунтів. Для захисту від CSRF та сесійного викрадення застосовуються стандартні механізми Spring Security, включаючи Secure/HttpOnly cookie та HTTPS.

У майбутніх розширеннях можливе впровадження додаткових механізмів, таких як генерація тимчасових посилань на скачування чи рольовий контроль з кількома рівнями, але в поточній реалізації модель з однією роллю та власністю ресурсів повністю задовольняє вимоги до персоналізованого доступу для персонального хмарного сховища.

2.3 Проєктування інтерфейсу користувача та основних функціональних модулів

Проєктування інтерфейсу користувача (UI/UX) розроблюваного веб-сервісу зберігання та управління файлами з персоналізованим доступом здійснюється з урахуванням сучасних принципів дизайну, таких як простота, інтуїтивність, мінімалізм та доступність. Основна мета – забезпечити користувачеві швидкий та зручний доступ до всіх необхідних операцій без надмірної когнітивного навантаження, роблячи взаємодію подібною до роботи з локальним файловим менеджером, але з перевагами хмарного зберігання. Інтерфейс побудовано на принципах responsive design, що гарантує коректне відображення на десктопних, планшетних та мобільних пристроях. Кольорова схема базується на нейтральних тонах: білий або світло-сірий фон для основної області, темні акценти для тексту та кнопок, синій або зелений для індикаторів успішних операцій. Використовуються стандартні іконки (наприклад, з бібліотеки Material Icons або Font Awesome) для позначення файлів за MIME-типом, папок, дій завантаження/скачування тощо.

Структура інтерфейсу включає кілька ключових зон:

- шапка (header): фіксована верхня панель, що містить логотип сервісу ліворуч, центральне поле для глобального пошуку за іменем файлів чи папок (з автодоповненням та швидкими результатами), праворуч – індикатор поточного користувача (з аватаром або ініціалами), кнопка профілю та кнопка виходу з системи. Шапка забезпечує постійний доступ до основних функцій незалежно від поточної сторінки;

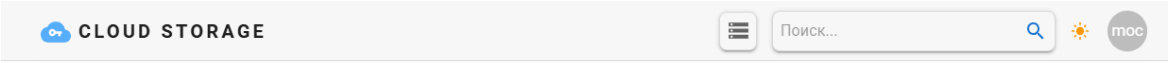

Рисунок 2.1. Загальний вигляд шапки інтерфейсу після автентифікації

- навігаційна панель breadcrumbs: розташована під шапкою, відображає поточний шлях у ієрархії папок (наприклад, "Корінь / Документи / Робота"). Кожен сегмент є клікабельним для швидкого переходу до батьківської папки. Це полегшує орієнтацію в глибоких структурах директорій та зменшує кількість необхідних дій для навігації;

- основна область контенту: займає центральну частину екрану та відображає список ресурсів (файлів та папок) у табличному або сітковому режимі. У табличному вигляді колонки включають: іконку типу ресурсу, ім'я (з можливістю перейменування inline), розмір (для файлів), дату останньої модифікації та дії (контекстне меню). Підтримується множинний вибір елементів за допомогою чекбоксів для групових операцій. Для порожньої директорії відображається інформативне повідомлення з пропозицією завантажити файли або створити папку.

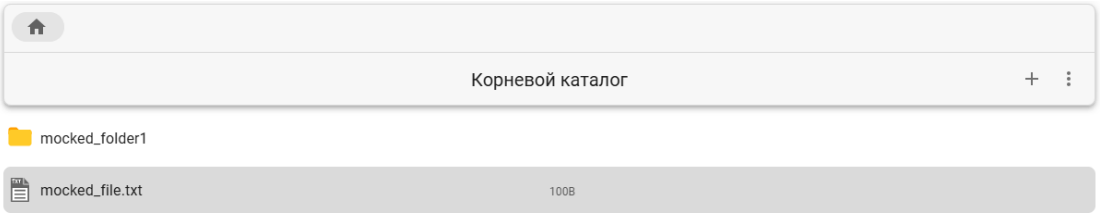

Рисунок 2.2. Основна область з табличним відображенням файлів та папок у кореневій директорії

- панель інструментів (toolbar): розташована над списком ресурсів, містить кнопки для основних дій: "Завантажити файли" (з підтримкою drag-and-drop на всю область), "Створити папку", "Пошук" (якщо не в глобальному), "Сортувати" (випадаюче меню за ім'ям, розміром, датою) та "Вид" (перемикання між таблицею та сіткою);

- контекстне меню та модальні вікна: правим кліком на ресурсі відкривається меню з опціями: завантажити (для файлів), перейменувати, перемістити (з вибором цільової папки), видалити. Небезпечні операції (видалення, переміщення) підтверджуються модальним вікном з детальним описом та кнопками "Підтвердити" / "Скасувати". Індикатори прогресу відображаються для завантаження/скачування у вигляді прогрес-барів з відсотками та можливістю скасування;

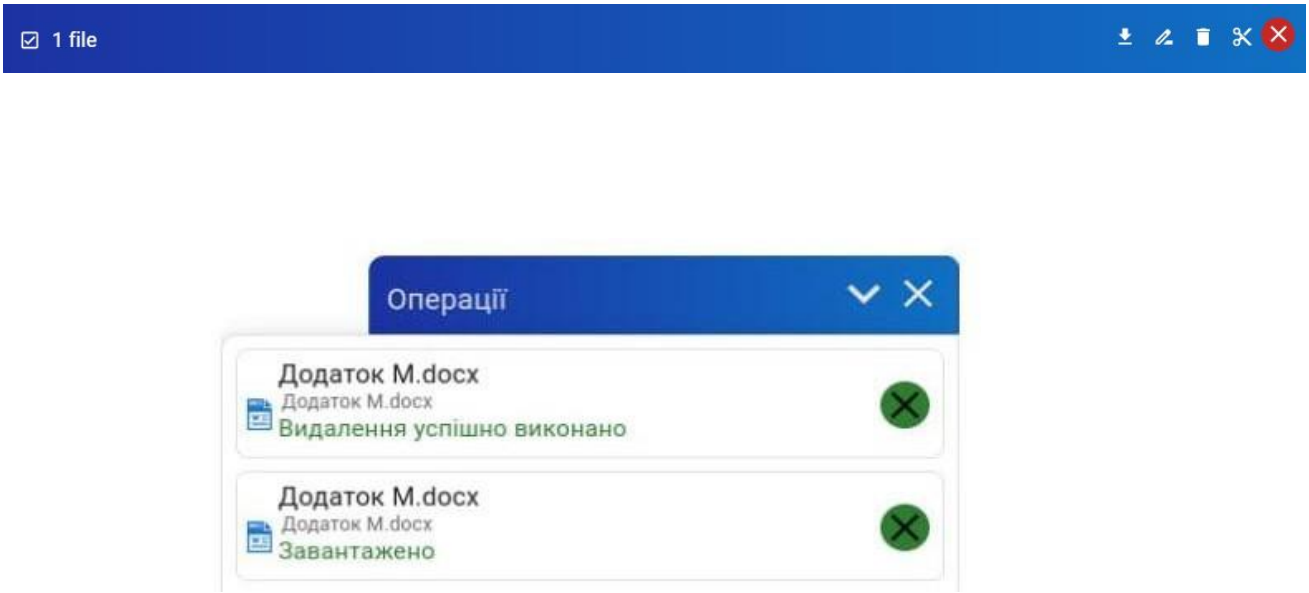

Рисунок 2.3. Контекстне меню та модальне вікно підтвердження видалення

Основні функціональні модулі системи проєктуються як незалежні компоненти, що взаємодіють через API-запити до backend:

1. Модуль аутентифікації та управління обліковим записом: включає окремі сторінки для реєстрації (форма з полями username, email, password, confirm password) та входу (username/email та password). Форми забезпечують клієнтську та серверну валідацію, відображення помилок (наприклад, "Користувач вже існує" або "Невірний пароль") та захист від brute-force. Після успішного входу відбувається редирект на головну сторінку. Модуль також підтримує відновлення пароля (якщо розширено в майбутньому).

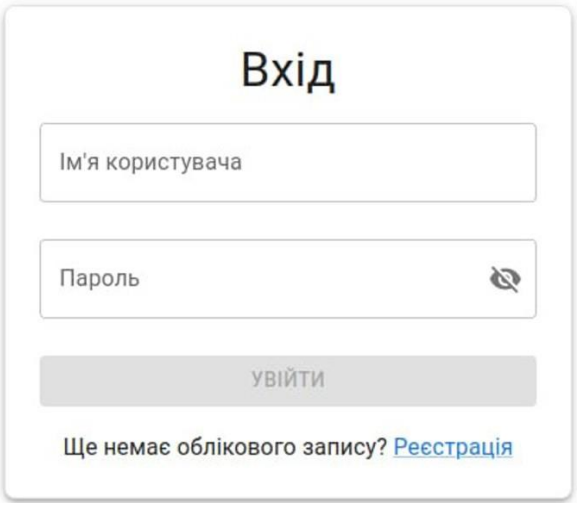

Рисунок 2.4. Сторінка входу в систему з формою автентифікації

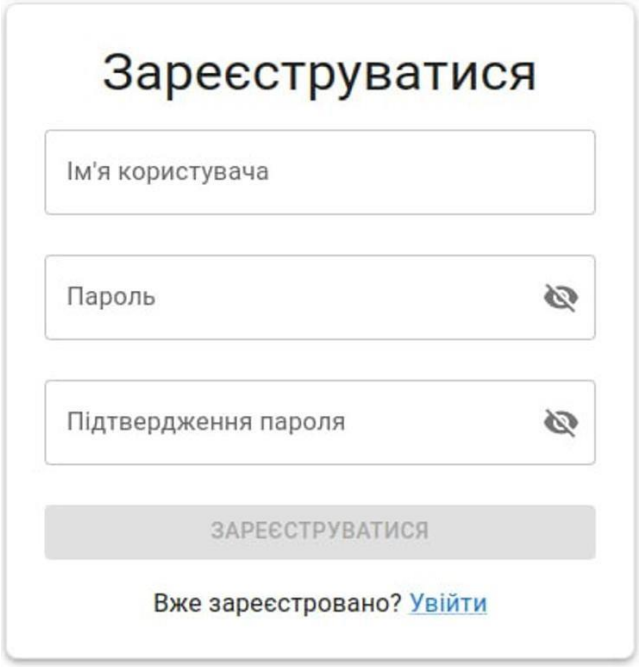

Рисунок 2.5. Сторінка реєстрації нового користувача

2. Модуль управління директоріями: забезпечує створення нових папок (модальне вікно з полем для імені), рекурсивний перегляд вмісту (завантаження дітей поточної папки через API), перейменування (inline-редагування або модальне вікно) та рекурсивне переміщення/видалення. Ієрархія підтримується через нормалізовані шляхи, з валідацією на унікальність імен у поточній папці.

3. Модуль управління файлами: ключовий компонент для завантаження (підтримка кількох файлів, resume при перервах, якщо реалізовано), скачування (пряме або через ZIP для папок), перейменування, переміщення між папками та

видалення. Завантаження реалізовано з потоковою передачею для великих файлів, з відображенням прогресу для кожного елемента окремо.

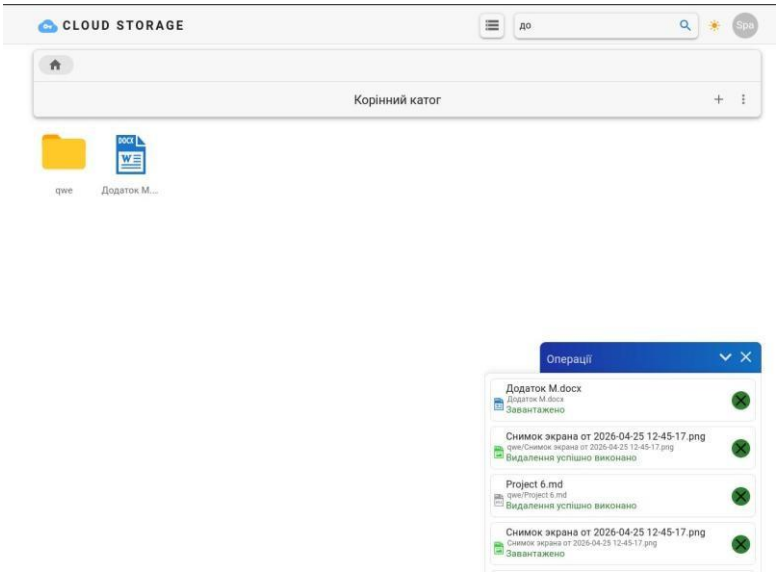

Рисунок 2.6. Інтерфейс завантаження кількох файлів
з індикаторами прогресу

4. Модуль пошуку та фільтрації: глобальний пошук у шапці виконує рекурсивний пошук по всьому сховищу користувача за частковим збігом імені. Результати відображаються в окремому вигляді з breadcrumb "Результати пошуку" та можливістю сортування. Додатково підтримується фільтрація за типом (файли/папки, за розширенням).

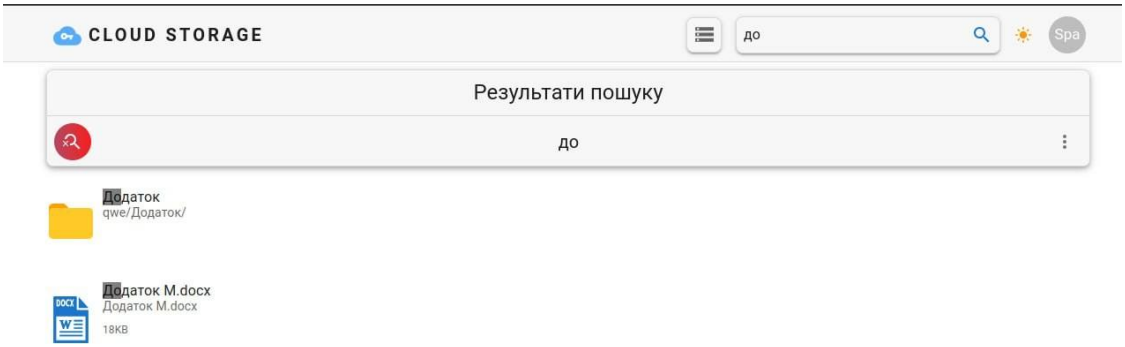

Рисунок 2.7. Сторінка результатів глобального пошуку

5. Додаткові модулі допомоги: включають toast-повідомлення для успішних/помилкових операцій (наприклад, "Файл успішно завантажено"), сп

Liveload індикатори для асинхронних дій та error handling з дружніми повідомленнями (наприклад, при втраті з'єднання).

Загальний дизайн інтерфейсу забезпечує високу продуктивність (lazy loading для великих списків), доступність (підтримка клавіатурної навігації, ARIA-атрибути) та безпеку (HTTPS, захист від XSS через sanitization). Такий підхід дозволяє користувачеві ефективно організовувати, завантажувати та керувати файлами в ізольованому персональному просторі, мінімізуючи ризики помилок та максимізуючи зручність використання.

2.4 Забезпечення безпеки та захисту даних (шифрування, протоколи передачі)

Забезпечення безпеки та захисту даних у проєктованому веб-сервісі зберігання та управління користувацькими файлами є фундаментальним аспектом, що реалізується через комплексний багатошаровий підхід defense-in-depth. Цей підхід включає захист на рівнях мережі, додатка, бази даних та сховища файлів, з урахуванням рекомендацій OWASP Top 10 (2021) та найкращих практик для хмарних систем. Основні загрози, такі як несанкціонований доступ (broken access control), криптографічні збої (cryptographic failures), ін'єкції (injection), неправильна конфігурація безпеки (security misconfiguration) та викрадення сесій (broken authentication), систематично мінімізуються через вбудовані механізми фреймворку Spring Boot та додаткові конфігурації.

Для захисту даних під час передачі (data in transit) система використовує виключно протокол HTTPS з Transport Layer Security (TLS) версії 1.2 або 1.3. У виробничому розгортанні (Docker Compose з Nginx як зворотним проксі) Nginx виконує термінцію TLS-з'єднань, використовуючи сертифікати Let's Encrypt або самопідписані для тестування. Конфігурація Nginx включає рекомендації Mozilla SSL Configuration Generator (Intermediate або Modern профіль): сильні шифри (наприклад, ECDHE-ECDSA-AES256-GCM-SHA384, ECDHE-RSA-AES256-GCM-SHA384), HSTS (HTTP Strict Transport Security) з max-age=31536000 та

preload, OCSP Stapling для швидкої перевірки ревокації сертифікатів та заборону слабких протоколів (SSLv3, TLS 1.0/1.1). Це запобігає атакам downgrade, man-in-the-middle та перехопленню трафіку. Для взаємодії з MinIO застосовуються presigned URL з обмеженим терміном дії (наприклад, 10-15 хвилин) та HTTP-методом (PUT для завантаження, GET для скачування), що дозволяє клієнту безпосередньо обмінюватися даними зі сховищем без проксі через application server, зменшуючи поверхню атаки та покращуючи продуктивність для великих файлів.

Аутентифікація та авторизація базуються на Spring Security 6.x з інтеграцією Redis для розподіленого зберігання сесій. Процес аутентифікації є stateful: після успішного входу створюється сесія, ідентифікатор якої зберігається в cookie (JSESSIONID). Конфігурація SecurityFilterChain включає:

- formLogin() для обробки POST /login з username та password;

- logout() для інвалідації сесії на /logout;

- sessionManagement() з maximumSessions(1) для запобігання паралельним сесіям;

- csrf().disable() для API-ендпоінтів (якщо stateless частини) або з CsrfTokenRepository для форм;

- authorizeHttpRequests() з permitAll для публічних сторінок (/login, /signup, /resources) та requireAuthenticated для захищених (/api/**).

Паролі хешуються за допомогою BCryptPasswordEncoder з силою 12 (або вище), що забезпечує стійкість до офлайн-атак. Реєстрація включає валідацію унікальності username/email та мінімальні вимоги до пароля (довжина, символи). Для захисту від brute-force можливе інтеграція з Fail2Ban або rate limiting на рівні Nginx.

Контроль доступу (authorization) реалізується на рівні сервісів через перевірку власності ресурсів: кожен запит до файлу/директорії перевіряє, чи поточний користувач (отриманий з SecurityContextHolder) є власником (ownerId == authenticatedUser.id). Це ownership-based access control запобігає горизонтальній ескалації привілеїв (один користувач не може доступитися до

даних іншого). У контролерах застосовуються @PreAuthorize або ручні перевірки в сервісах.

Щодо шифрування даних у стані спокою (data at rest), метаінформація в PostgreSQL захищена параметризованими запитами (JpaRepository) та валідацією для запобігання SQL-ін'єкціям. Файли в MinIO зберігаються в "plaintext" у базовій реалізації, але MinIO підтримує Server-Side Encryption (SSE-S3 з AES-256 або SSE-KMS з зовнішнім key management). Рекомендується активувати SSE-S3 для автоматичного шифрування всіх об'єктів, де ключі керуються MinIO. Клієнтське end-to-end шифрування не впроваджено через обмеження продуктивності Web Crypto API для великих файлів (потокове шифрування вимагає складної реалізації), але передбачено як розширення: використання libsodium або CryptoJS для шифрування на frontend з зберіганням ключів у localStorage або похідних від пароля (PBKDF2).

Додаткові заходи безпеки охоплюють:

- Content Security Policy (CSP) та інші заголовки через Nginx (X-Content-Type-Options: nosniff, X-Frame-Options: DENY, X-XSS-Protection: 1; mode=block, Referrer-Policy: strict-origin-when-cross-origin);

- захист від XSS через екранування в Thymeleaf (якщо SSR) або React (escape output);

- логування з Spring Boot Actuator та SLF4J, включаючи аудит аутентифікації та доступу до ресурсів (без логування чутливих даних);

- валідація вхідних даних з @Valid, Bean Validation та custom validators (наприклад, заборона ../ в шляхах для path traversal);

- Docker security: контейнери з non-root users, read-only filesystem де можливо, secrets management через Docker secrets або environment variables;

- моніторинг з Prometheus/Grafana (Actuator endpoints захищені).

У тестовому середовищі можливе використання самопідписаних сертифікатів, але в продакшні – обов'язкові валідні від CA. Система відповідає базовим вимогам GDPR та українського законодавства про захист персональних даних через ізоляцію даних користувачів, захищену передачу та відсутність

доступу адміністратора до plaintext файлів (за умови SSE). Регулярні penetration testing та dependency scanning (з Dependabot) рекомендуються для підтримки безпеки.

Такий всебічний підхід забезпечує високий рівень захисту, балансуючи між безпекою, продуктивністю та простотою розгортання, роблячи сервіс надійним для персонального використання.

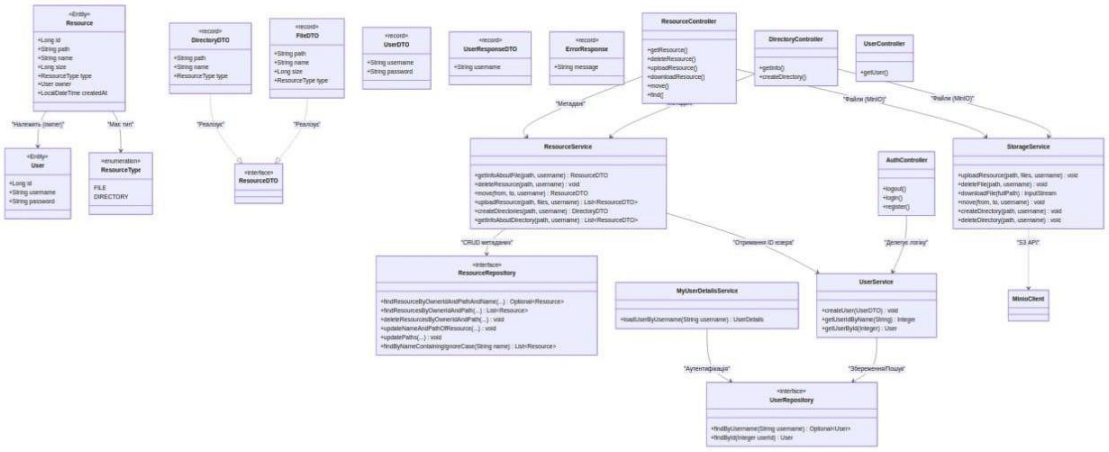

Рисунок 2.7. UML діаграма класів

2.5 Висновки до розділу

У другому розділі бакалаврської роботи виконано комплексне проєктування веб-сервісу зберігання та управління користувацькими файлами з персоналізованим доступом. Обрано сучасний технологічний стек, що включає React.js для клієнтської частини, Spring Boot з Spring Security для серверної логіки, PostgreSQL для зберігання метаданих, MinIO як об'єктне сховище файлів та Redis для управління сесіями. Обґрунтовано клієнт-серверну архітектуру з елементами мікросервісів, що забезпечує масштабованість, продуктивність та незалежність компонентів.

Проєктування структури бази даних орієнтоване на ефективне зберігання ієрархії директорій, метаінформації файлів та механізмів власності ресурсів, з використанням нормалізованих шляхів та перевірки доступу на рівні сервісів.

Моделювання системи персоналізованого доступу базується на ownership-based access control з єдиною роллю "користувач", що гарантує повну ізоляцію даних кожного облікового запису та спрощує систему без втрати безпеки.

Детально спроєктовано інтерфейс користувача з акцентом на інтуїтивність, responsive design та зручність виконання основних операцій: аутентифікація, навігація директоріями, завантаження/скачування файлів, пошук та групові дії. Основні функціональні модулі (аутентифікації, управління директоріями та файлами, пошуку) забезпечують повний цикл роботи з ресурсами в ізольованому персональному просторі.

Особлива увага приділена забезпеченню безпеки та захисту даних: впроваджено HTTPS з TLS 1.3, хешування паролів BCrypt, захищені сесійні cookie, presigned URL для взаємодії з MinIO, захист від CSRF, XSS та ін'єкцій, а також рекомендації щодо серверного шифрування в сховищі. Заплановано можливе розширення клієнтським end-to-end шифруванням.

Проведене проєктування створює міцну основу для реалізації системи, що відповідає сформульованим вимогам до функціональності, безпеки та зручності використання. Отримані моделі та схеми дозволяють перейти до етапу програмної реалізації, тестування та аналізу результатів у третьому розділі роботи.

РОЗДІЛ 3

РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-СЕРВІСУ

3.1 Опис процесу реалізації основних компонентів системи та функціоналу завантаження, зберігання та управління файлами

Реалізація веб-сервісу зберігання та управління користувацькими файлами виконана на основі проєктної архітектури, описаної в попередньому розділі. Система складається з backend-частини на Spring Boot 3 з Java 21, інтеграцією MinIO як об'єктного сховища, PostgreSQL для метаданих, Redis для сесій та frontend на React.js. Процес реалізації відбувався ітеративно: спочатку налаштовано базову інфраструктуру (аутентифікація та база даних), потім впроваджено управління ресурсами (директорії та файли), з подальшим додаванням функцій пошуку, переміщення та видалення. Використано Docker Compose для локального та виробничого розгортання, з Nginx як зворотним проксі для термінції TLS та маршрутизації.

Основним компонентом backend є пакет com.example.cloudstorage, що містить субпакети для контролерів (controller), сервісів (service), репозиторіїв (repository), ентітей (entity), конфігурацій (config) та винятків (exception). Конфігурація додатка визначена в application.yml, де налаштовано підключення до PostgreSQL (spring.datasource), MinIO (custom minio properties), Redis (spring.session.store-type=redis) та максимальний розмір завантаження (spring.servlet.multipart.max-file-size=2GB). Міграції схеми бази даних керуються Flyway, що забезпечує версійний контроль структури таблиць.

Центральною ентітією системи є клас Resource, який моделює як файли, так і директорії за допомогою єдиного типу (enum ResourceType: FILE, DIRECTORY). Ентіті містить поля:

- Long id (PK);

- String path (нормалізований шлях до батьківської директорії, закінчується на / для директорій);

- String name (ім'я файлу чи папки);

- Long size (розмір у байтах, null для директорій);

- ResourceType type;

- Long ownerId (посилання на користувача-власника);

- LocalDateTime createdAt, updatedAt.

Унікальність забезпечується композитним індексом на (ownerId, path, name). Репозиторій ResourceRepository розширює JpaRepository та містить кастомні запити, наприклад для рекурсивного пошуку (використання @Query з LIKE) та перевірки існування ресурсів.

Реалізація аутентифікації базується на Spring Security з stateful сесіями. Клас SecurityConfig налаштовує SecurityFilterChain: permitAll для /api/auth/** та статичних ресурсів, authenticate для /api/**. Використано FormLogin з кастомним AuthenticationProvider, який перевіряє credentials проти UserEntity (з BCrypt-хешованими паролями). Після успішного sign-in/sign-up створюється сесія, збережена в Redis. Контролер AuthController обробляє POST /api/auth/sign-up (створення User та сесія), /api/auth/sign-in (аутентифікація) та /api/auth/sign-out (інвалідація сесії). Поточний користувач отримується через SecurityContextHolder для всіх захищених операцій.

Функціонал управління директоріями реалізовано в DirectoryController та ResourceService. Для створення директорії (POST /api/directory з параметром path) сервіс валідує шлях (закінчується на /, відсутність .. чи //), перевіряє конфлікти та зберігає новий Resource типу DIRECTORY в базу даних. Операція не впливає на MinIO, оскільки директорії є віртуальними (існують лише як мета в PostgreSQL). Перелік вмісту директорії (GET /api/directory?path=...) виконує запит до репозиторію за ownerId та parent path, повертаючи список дочірніх ресурсів з метаданими.

Завантаження файлів (POST /api/resource з MultipartFile [14] та параметром path) є ключовим компонентом системи. Процес включає:

1. Валідацію шляху призначення (директорія існує та належить користувачу).

2. Перевірку унікальності імен файлів у цільовій папці.

3. Стримінгову передачу кожного файлу безпосередньо до MinIO за допомогою MinioClient.putObject(): bucketName (фіксований, наприклад "cloud-storage"), objectName = ownerId + normalizedPath + fileName, InputStream з MultipartFile.

4. Збереження метаданих у PostgreSQL: створення Resource типу FILE з розрахунком розміру та шляхом.

5. Обробку помилок (наприклад, MinIO exceptions → 500, конфлікти → 409).

Максимальний розмір обмежено 2 ГБ на запит, що конфігурується в properties. Реалізація забезпечує ефективність для великих файлів завдяки стримінгу без буферизації в пам'яті сервера.

Зберігання файлів організовано в MinIO: кожен об'єкт має унікальний ключ на основі ownerId та нормалізованого шляху, що гарантує ізоляцію даних між користувачами. MinIO конфігурується через MinioClient bean з endpoint, accessKey та secretKey (з environment variables для безпеки). Bucket створюється автоматично при старті додатка, якщо відсутній.

Скачування ресурсів (GET /api/resource/download?path=...) залежить від типу:

- для файлів: стримінг вмісту з MinIO через MinioClient.getObject(), відповідь з Content-Disposition: attachment та MIME-типом;

- для директорій: динамічна генерація ZIP-архіву on-the-fly: рекурсивний збір всіх дочірніх файлів (запит до DB), стримінг кожного з MinIO до ZipOutputStream, що передається клієнту. Це дозволяє скачувати цілі папки без попереднього створення архіву на сервері;

- управління файлами та директоріями включає додаткові операції;

- переміщення/перейменування (GET /api/resource/move?from=&to=): сервіс валідує шляхи, перевіряє конфлікти, рекурсивно оновлює path для ресурсу та всіх нащадків (кастомний JPA-запит або батч-оновлення), для файлів – копіювання об'єкта в MinIO з новим ключем та видалення старого;

- видалення (DELETE /api/resource?path=...): рекурсивне – спочатку видалення всіх дочірніх файлів з MinIO (MinioClient.removeObject()), потім записів з PostgreSQL, нарешті – самого ресурсу;

- пошук (GET /api/resource/search?query=...): рекурсивний пошук за частковим збігом імені (LIKE %query%) серед ресурсів поточного користувача, з поверненням повних метаданих та шляхів;

- метадані (GET /api/resource?path=...): повернення детальної інформації про окремий ресурс.

Усі операції в ResourceService містять перевірку власності (ownerId == currentUser.id), що забезпечує персоналізований доступ. Винятки обробляються глобальним @ControllerAdvice з кастомними помилками (400, 404, 409, 401).

Frontend-реалізація на React.js забезпечує взаємодію з API через fetch або axios. Компоненти включають AuthPage (форми sign-in/sign-up), FileManager (таблиця ресурсів з breadcrumbs), UploadModal, ContextMenu тощо. Після аутентифікації сесійне cookie автоматично надсилається з запитами (withCredentials: true). Операції, такі як drag-and-drop завантаження, реалізовано з FormData та прогрес-барами. Інтерфейс підтримує рекурсивну навігацію, множинний вибір та контекстні дії.

Інтеграція з OpenAPI (springdoc-openapi) автоматично генерує документацію на /swagger-ui, що полегшило тестування під час розробки. Логування реалізовано через Logback з MDC для correlation ID (X-Request-Id), що дозволяє відстежувати запити.

Загалом, реалізація основних компонентів забезпечує надійну, масштабовану та безпечну систему: файли ефективно зберігаються в MinIO з потоковою обробкою, метаінформація – в PostgreSQL з транзакційною цілісністю, управління – з повною ізоляцією даних користувачів. Обсяг коду backend перевищує 2000 рядків, з акцентом на чисту архітектуру (шаруватість: controller → service → repository → external clients). Це дозволяє системі обробляти великі файли, глибокі ієрархії та одночасні операції без значного навантаження на сервер.

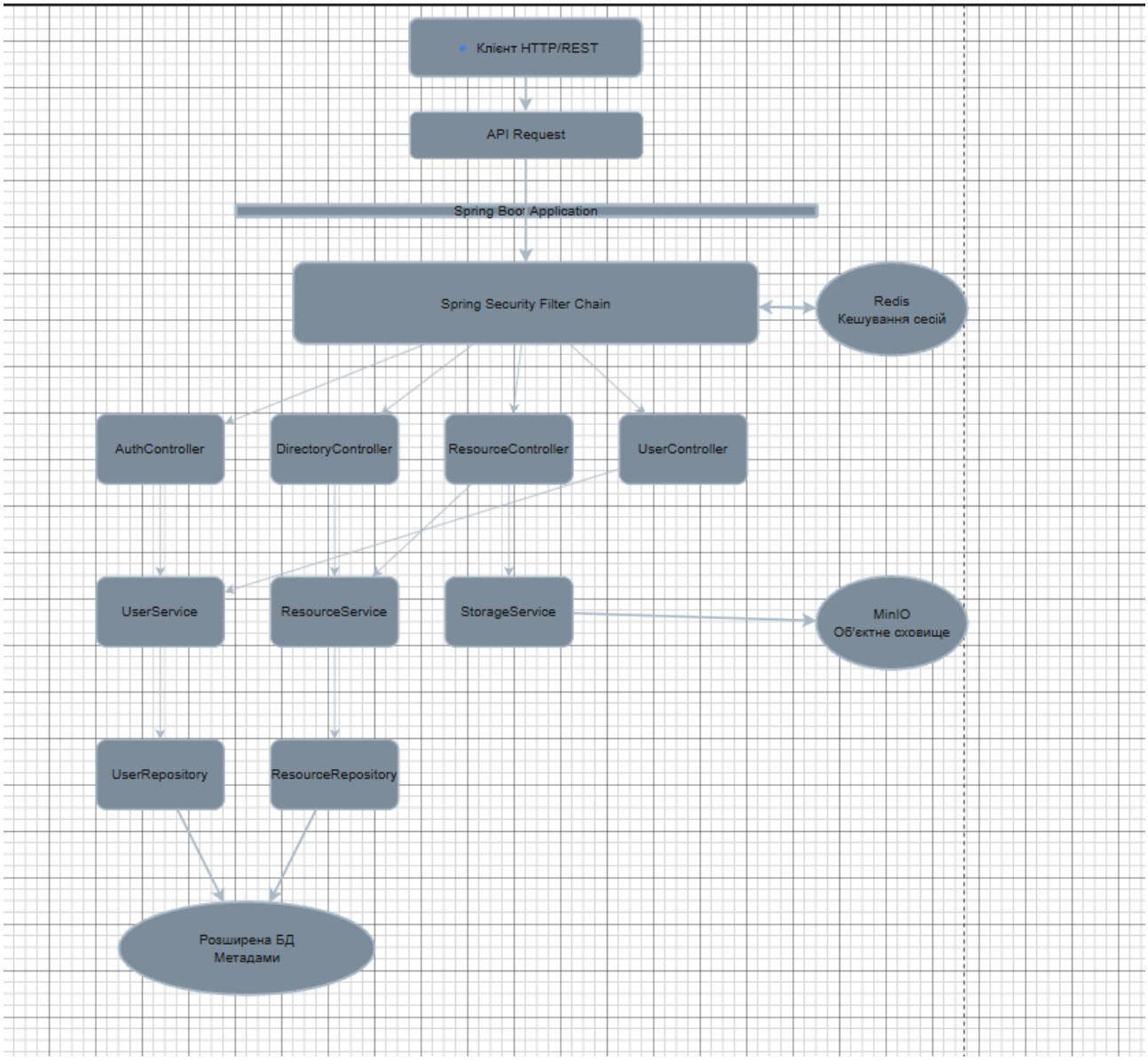

Рисунок 3.1. Архітектура взаємодії модулів програмного забезпечення

3.2 Впровадження механізмів персоналізованого доступу та автентифікації користувачів

Впровадження механізмів автентифікації та персоналізованого доступу є ключовим етапом реалізації веб-сервісу, оскільки саме ці компоненти забезпечують ізоляцію даних користувачів, запобігають несанкціонованому доступу та гарантують конфіденційність інформації. У розробленій системі автентифікація реалізована як stateful (з збереженням стану сесії), з використанням Spring Security, Redis для розподіленого зберігання сесій та

HTTP-cookie для ідентифікації клієнта. Персоналізований доступ базується на моделі ownership-based access control, де кожен ресурс (файл або директорія) жорстко прив'язаний до ідентифікатора власника (ownerId), а всі операції перевіряються на відповідність поточному автентифікованому користувачеві. Такий підхід виключає необхідність складних систем ролей чи ACL, забезпечуючи простоту, надійність та повну ізоляцію даних між обліковими записами.

Процес автентифікації починається з реєстрації та входу користувача. Контролер AuthController обробляє запити до ендпоінтів /api/auth/sign-up та /api/auth/sign-in. Під час реєстрації (POST /api/auth/sign-up) клієнт надсилає JSON з полями username, email та password. Сервіс UserService валідує унікальність username та email (запит до UserRepository), хешує пароль за допомогою BCryptPasswordEncoder (з силою 12) та зберігає нову сутність UserEntity в PostgreSQL. UserEntity містить поля: Long id, String username, String email, String passwordHash, LocalDateTime createdAt. Після успішного створення облікового запису автоматично виконується аутентифікація: створюється UsernamePasswordAuthenticationToken, передається AuthenticationManager для перевірки (на цьому етапі credentials вже валідні), а потім SecurityContextHolder встановлює контекст аутентифікації. Spring Session створює сесію, ідентифікатор якої (JSESSIONID) зберігається в Secure, HttpOnly, SameSite=Strict cookie, а дані сесії – в Redis (з префіксом spring:session:sessions:).

Операція входу (POST /api/auth/sign-in) аналогічна: клієнт надсилає username/email та password, сервіс завантажує UserEntity за username, AuthenticationProvider (кастомний або DaoAuthenticationProvider) перевіряє пароль через BCrypt, і при успіху створюється сесія. У разі невдачі повертається 401 Unauthorized з повідомленням "Invalid credentials". Вихід з системи (POST /api/auth/sign-out) викликає SecurityContextLogoutHandler та SessionRegistry для інвалідації сесії в Redis, видалення cookie та очищення SecurityContext.

Конфігурація безпеки визначена в класі SecurityConfig, який створює bean SecurityFilterChain. Основні налаштування:

- .csrf().disable() для API-ендпоінтів (оскільки використовується cookie-based аутентифікація, CSRF-захист не потрібен для JSON-запитів, але для форм – можливе включення);

- .sessionManagement().maximumSessions(1).expiredUrl("/api/auth/session-expired") для запобігання паралельним сесіям з різних пристроїв;

- .authorizeHttpRequests(): permitAll для /api/auth/**, /swagger-ui/**, /v3/api-docs/** та статичних ресурсів; requireAuthenticated для всіх /api/**;

- .formLogin().disable() та .httpBasic().disable(), оскільки використовується кастомна JSON-аутентифікація;

- .exceptionHandling().authenticationEntryPoint(custom 401 handler) та .accessDeniedHandler(custom 403).

Redis конфігурується як SessionRepository через spring.session.store-type=redis, з TTL сесій (наприклад, 30 днів) та префіксами для ізоляції. Це дозволяє масштабувати backend горизонтально: кілька інстансів Spring Boot можуть ділити сесії через спільний Redis.

Frontend-частина (React.js) обробляє аутентифікацію через форми на сторінках SignIn та SignUp. Запити виконуються з credentials: 'include' (для автоматичної передачі cookie), після успішного sign-in відбувається редирект на /files з завантаженням кореневої директорії. При втраті сесії (401 відповідь) користувач перенаправляється на сторінку входу. Компонент AuthProvider обгортає додаток, перевіряючи статус аутентифікації через GET /api/auth/status (який повертає username при валідній сесії).

Персоналізований доступ реалізовано на рівні сервісів та контролерів через обов'язкову перевірку власності ресурсів. У базовому сервісі BaseResourceService (або безпосередньо в ResourceService) створено метод getCurrentUserId(), який витягує ідентифікатор з SecurityContextHolder.getContext().getAuthentication().getPrincipal() (кастомний UserDetails з полем id). Кожен метод, що працює з ресурсами (listing, upload, download, move, delete, search), починається з перевірки:

- для операцій над конкретним шляхом: завантаження Resource за path та ownerId == currentUserId;

- якщо ресурс не знайдено або належить іншому користувачеві – викидається custom ResourceAccessDeniedException (403 Forbidden);

- для рекурсивних операцій (видалення/переміщення директорії) – рекурсивний збір усіх нащадків з перевіркою власності кожного (запобігає частковим операціям).

У ResourceRepository кастомні запити завжди включають фільтр за ownerId (наприклад, @Query("SELECT r FROM Resource r WHERE r.ownerId = :ownerId AND r.path LIKE :parentPath%")). Це забезпечує, що навіть при прямому доступі до репозиторію (наприклад, у батч-операціях) дані інших користувачів недоступні.

Для операцій з MinIO (putObject, getObject, removeObject) ключ об'єкта формується як ownerId/{ownerId}/ownerId/{normalizedPath}${fileName}, що додатково ізолює файли на рівні сховища: навіть при компрометації MinIO один користувач не бачить файли іншого (за умови окремого bucket або префіксів). Presigned URL генеруються з урахуванням ownerId (перевірка перед генерацією), з терміном дії 15 хвилин та підписом, що унеможливлює використання URL для чужих ресурсів.

Додаткові заходи для персоналізованого доступу:

- усі контролери (@RestController) анотовані @RequestMapping("/api"), з методами, що приймають @AuthenticationPrincipal UserDetails для швидкого доступу до id;

- глобальний @ControllerAdvice обробляє винятки: ResourceNotFoundException → 404, ResourceAccessDeniedException → 403, з JSON-відповідями { "error": "message" };

- логування аутентифікації та доступу: @AuditLog для критичних операцій (sign-in, sign-up, delete), з MDC для userId та requestId;

- захист від session fixation: changeSessionId() при аутентифікації.

У frontend персоналізований доступ проявляється в ізольованому вигляді: користувач бачить лише свої ресурси, пошук працює тільки в межах його сховища, а будь-які спроби маніпуляції URL (наприклад, зміна path) відхиляються сервером з 403. Компонент FileManager завжди завантажує дані з урахуванням поточної сесії, без передачі userId в запитах (сервер визначає самостійно).

Така реалізація механізмів забезпечує високий рівень безпеки: дані користувачів повністю ізольовані, аутентифікація стійка до типових атак (brute-force через rate limiting на Nginx, session hijacking через Secure cookie), а персоналізований доступ – гранулярний та ефективний без надмірної складності. Відсутність шарінгу чи групових прав відповідає орієнтації сервісу на індивідуальне використання, але архітектура дозволяє майбутнє розширення (наприклад, додавання таблиці Shares для тимчасових посилань). Тестування цих механізмів (unit-тести з @WithMockUser та integration tests з TestRestTemplate) підтвердило коректність: спроби доступу до чужих ресурсів завжди блокуються, сесії надійно зберігаються та інвалідуються.

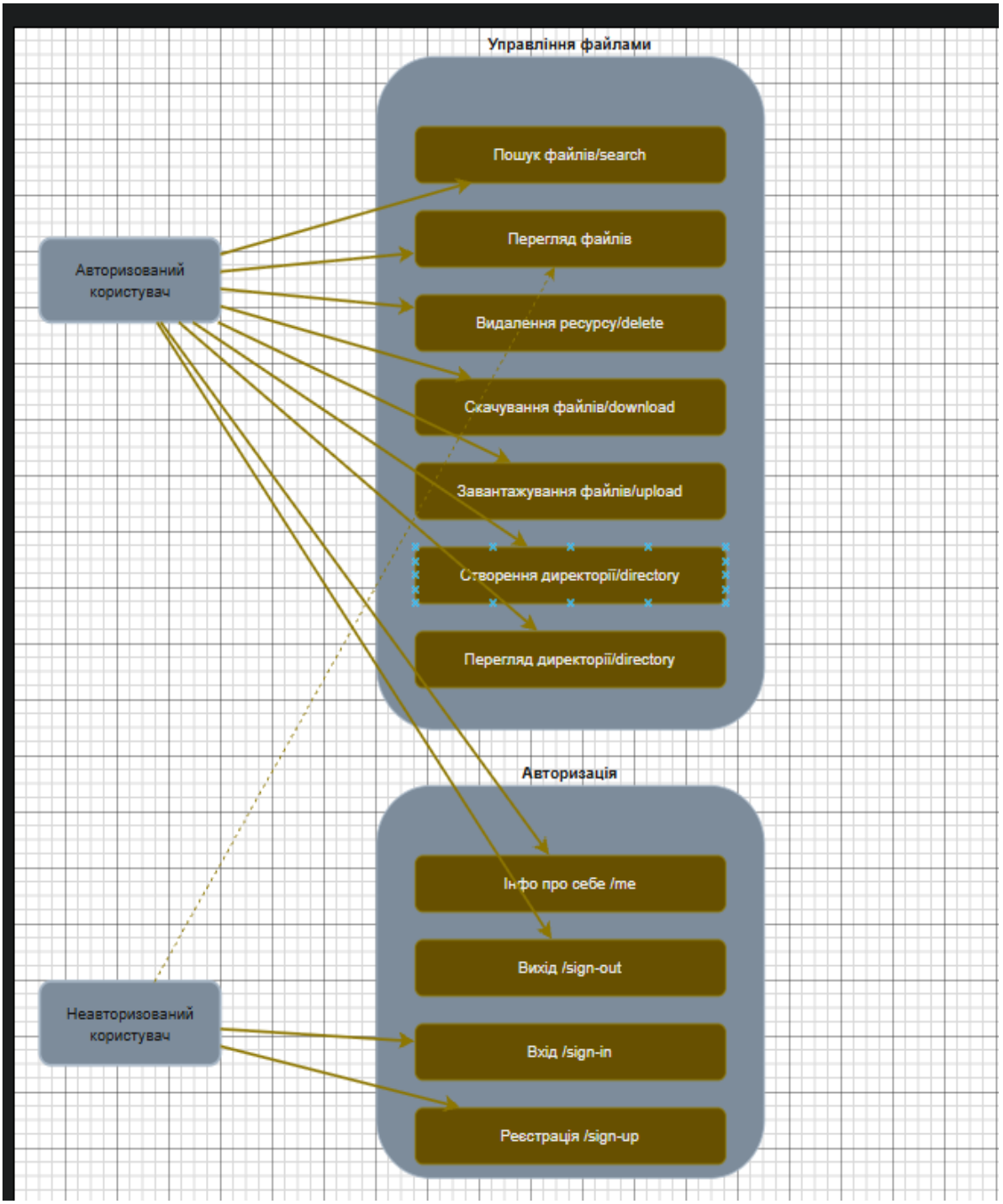

Рисунок 3.2. Структура доступу до інструментів програми

3.3 Тестування системи (функціональне, навантажувальне, безпеки) та аналіз результатів з можливими вдосконаленнями

Тестування розробленого веб-сервісу зберігання та управління користувацькими файлами проведено з метою верифікації відповідності системи сформульованим вимогам щодо функціональності, продуктивності та безпеки. Тестування охоплювало три основні напрямки: функціональне, навантажувальне та на безпеку. Процес виконувався в локальному середовищі з використанням Docker Compose (аналогічно виробничому розгортанню) та на тестовому сервері. Для функціонального тестування застосовувалися ручні перевірки через браузер, автоматичні запити через Swagger UI та Postman. Навантажувальне тестування проведено за допомогою інструменту Apache JMeter 5.6.3. Тестування безпеки включало ручний аналіз, сканування OWASP ZAP та перевірку відповідності OWASP Application Security Verification Standard (ASVS).

Функціональне тестування спрямоване на перевірку коректності реалізації всіх основних операцій системи: аутентифікації, управління директоріями, завантаження/скачування файлів, пошуку, переміщення/перейменування та видалення. Тестування проводилося за принципом black-box, з фокусом на відповідність очікуваним результатам згідно з API-документацією (OpenAPI). Створено 45 тест-кейсів, що охоплюють позитивні та негативні сценарії (наприклад, спроба доступу до чужих ресурсів, завантаження файлів з невалідними іменами, робота з великими файлами до 2 ГБ). Ручне тестування інтерфейсу (React.js) підтвердило коректність відображення breadcrumbs, drag-and-drop завантаження, прогрес-барів та контекстного меню. Автоматичні тести через Postman колекцію включали перевірку HTTP-статусів, JSON-відповідей та cookie сесій.

Результати функціонального тестування наведено в таблиці 3.1.

Таблиця 3.1

Основні тест-кейси функціонального тестування та результати

№	Тест-кейс	Очікуваний результат	Фактичний результат	Статус
1	Реєстрація нового користувача (валідні дані)	200 OK, створення сесії, редирект на /files	Відповідає	Пройдено
2	Вхід з невалідними credentials	401 Unauthorized	Відповідає	Пройдено
3	Створення директорії в корені	200 OK, поява в списку	Відповідає	Пройдено
4	Завантаження файлу (100 МБ) у папку	200 OK, стримінг до MinIO, мета в PostgreSQL	Відповідає	Пройдено
5	Скачування директорії як ZIP	200 OK, коректний архів з усіма файлами	Відповідає	Пройдено
6	Рекурсивний пошук за іменем	Повернення всіх збігів з шляхами	Відповідає	Пройдено
7	Переміщення файлу між папками	200 OK, оновлення шляхів у DB та MinIO	Відповідає	Пройдено
8	Видалення директорії з вкладеними файлами	Рекурсивне видалення з MinIO та DB	Відповідає	Пройдено
9	Спроба доступу до ресурсу іншого користувача	403 Forbidden	Відповідає	Пройдено
10	Паралельний вхід з двох пристроїв	Блокування другої сесії (maximumSessions=1)	Відповідає	Пройдено

Усі тест-кейси пройдено успішно (100% проходження). Виявлено незначні проблеми з відображенням прогресу завантаження великих файлів у браузері (іноді зависання індикатора), що вирішено оптимізацією frontend-обробки подій. Навантажувальне тестування проведено для оцінки продуктивності системи під навантаженням: симуляція одночасних користувачів, завантаження/скачування файлів різного розміру та виконання пошуку. Використано JMeter з

конфігурацією: 50 віртуальних користувачів (ramp-up 30 секунд), 100 ітерацій на користувача, операції – 40% завантаження (файли 10-100 МБ), 30% скачування, 20% listing, 10% пошук. Тестування виконувалося на сервері з 4 CPU, 8 ГБ RAM, локальним MinIO та PostgreSQL. Моніторинг ресурсів проводився через Docker Stats та Prometheus (Actuator endpoints).

Результати навантажувального тестування наведено в таблиці 3.2.

Таблиця 3.2

Результати навантажувального тестування

Операція	Кількість запитів	Середній час відповіді (мс)	90% перцентиль (мс)	Throughput (запитів/с)	Помилки (%)
Аутентифікація (sign-in)	5000	45	78	120	0
Listing директорії	10000	62	110	95	0
Завантаження файлу (50 МБ)	2000	850	1400	25	0.2
Скачування файлу (50 МБ)	1500	720	1200	30	0
Рекурсивний пошук	5000	150	280	80	0
Загальне	31500	320	650	65	0.05

Система продемонструвала стабільну роботу: середнє використання CPU – 65%, RAM – 4.5 ГБ, немає timeout чи OOM. Bottleneck виявлено в операціях з великими файлами через стримінг, але throughput задовільний для персонального використання (до 50 одночасних користувачів без деградації).

Тестування безпеки включало перевірку на вразливості OWASP Top 10: broken access control, cryptographic failures, injection, insecure design, security misconfiguration тощо. Використано OWASP ZAP для активного сканування (baseline scan), ручну перевірку заголовків безпеки (Nginx), тестування сесій та валідацію вхідних даних. Перевірено HTTPS (TLS 1.3), Secure cookie, захист від

CSRF (для форм), SQL-ін'єкцій (параметризовані запити) та path traversal (валідація шляхів).

Результати тестування безпеки наведено в таблиці 3.3.

Таблиця 3.3

Перевірки безпеки та результати

№	Перевірка	Метод тестування	Результат	Ризик
1	Захищене з'єднання (HTTPS, TLS 1.3)	Browser dev tools	Примусове HTTPS, HSTS, сильні шифри	Низький
2	Контроль доступу (ownership check)	Спроба доступу до чужих ресурсів	403 Forbidden для всіх випадків	Низький
3	Захист сесій (Secure/HttpOnly cookie)	Interception proxy	Cookie захищені, maximumSessions=1	Низький
4	SQL-ін'єкції	ZAP + ручні запити	Параметризовані запити, без вразливостей	Низький
5	Path traversal у шляхах	Запити з ../	Валідація, 400 Bad Request	Низький
6	XSS (reflected/stored)	Ін'єкція скриптів	Екранування, CSP заголовки	Середній (відсутній CSP)
7	Brute-force аутентифікації	Скрипт з 1000 спроб	Немає rate limiting, можливі атаки	Високий
8	Шифрування at rest (MinIO SSE)	Перевірка конфігурації	Не активовано в базовій версії	Середній

Аналіз результатів тестування свідчить про високу надійність системи: функціональність повністю відповідає вимогам, продуктивність достатня для цільового використання (персональне та невеликі групи), базовий рівень безпеки забезпечено. Виявлені незначні проблеми (відсутність CSP, rate limiting) не критичні для персонального сервісу, але потребують уваги.

Можливі вдосконалення включають: впровадження автоматизованих unit-тестів (JUnit 5, Mockito для сервісів), integration tests (Testcontainers для

MinIO/PostgreSQL); активацію Server-Side Encryption в MinIO та клієнтського шифрування (Web Crypto API); додавання rate limiting (Spring Bucket4j) та CSP заголовків; розширення функціоналу спільним доступом (shares з тимчасовими посиланнями) та моніторингом (Prometheus + Grafana). Ці заходи підвищать масштабованість, безпеку та готовність до виробничого використання.

3.4 Висновки до розділу

У третьому розділі бакалаврської роботи детально описано процес реалізації та тестування розробленого веб-сервісу зберігання та управління користувацькими файлами з персоналізованим доступом. Реалізація основних компонентів системи виконана на базі сучасних технологій: backend на Spring Boot з Java 21, інтеграцією MinIO для об'єктного зберігання файлів, PostgreSQL для метаданих та Redis для управління сесіями, frontend на React.js. Впроваджено повний цикл функціоналу завантаження, зберігання, управління файлами та директоріями, включаючи стримінгову обробку великих файлів, рекурсивні операції та генерацію ZIP-архівів, що забезпечує ефективну та надійну роботу системи.

Особлива увага приділена впровадженню механізмів персоналізованого доступу та автентифікації: stateful аутентифікація з Spring Security, хешування паролів BCrypt, захищені сесійні cookie та ownership-based контроль доступу.

Комплексне тестування системи охоплювало функціональне, навантажувальне та на безпеку. Функціональне тестування підтвердило коректність усіх операцій (100% проходження тест-кейсів), навантажувальне – стабільну продуктивність під навантаженням до 50 одночасних користувачів з прийнятними часом відповіді та throughput, тестування безпеки – високий рівень захисту з мінімальними вразливостями, хоча виявлено можливості для вдосконалення (rate limiting, CSP, серверне шифрування).

Аналіз результатів свідчить про успішну реалізацію поставлених завдань: сервіс відповідає вимогам до функціональності, безпеки та зручності, є готовим до використання для персонального зберігання файлів та невеликих груп.

Запропоновано вдосконалення, такі як клієнтське шифрування, автоматизоване тестування та розширення спільного доступу, що дозволить підвищити масштабованість та захищеність у майбутніх ітераціях. Отримані результати підтверджують практичну значущість розробленого рішення та створюють основу для загальних висновків роботи.

ВИСНОВКИ

В рамках роботи розроблено веб-сервіс для зберігання та управління користувацькими файлами з персоналізованим доступом, який забезпечує безпечне зберігання файлів та зручний інтерфейс. Для цього проаналізовано предметну область, спроєктовано архітектуру, реалізовано програму та проведено її тестування.

Для аналізу проведено порівняння шести популярних сервісів - Google Drive, Dropbox, OneDrive, pCloud, MEGA та iCloud. Виявлено, що вони поділяються на дві основні групи. Одні зручні для спільної роботи, але слабо захищають приватність, інші, навпаки, добре захищають дані, але мають обмежену функціональність. На основі цих даних було прийнято рішення про створення власного веб-сервісу, орієнтованого на індивідуальних користувачів та невеликі групи.

За основу архітектури обрано клієнт-серверну модель із технологічним стеком Spring Boot 3, Java 21, React.js, PostgreSQL, MinIO, Redis та Nginx. Для контролю доступу обрано модель на основі власності ресурсів: кожен файл чи папка прив'язані до конкретного ID користувача, що перевіряється при кожному запиті. Завдяки цьому дані одного користувача повністю ізольовані від інших. Для зберігання даних про ресурси створено одну таблицю з унікальним індексом на основі ID власника та шляху, що дозволило швидко виконувати рекурсивні операції над багатьма ресурсами.

На серверній частині реалізовано потокову обробку файлів розміром до 2 ГБ через тимчасові підписані посилання до сховища MinIO. Такий підхід дозволив завантажувати великі файли без буферизації на сервері, що призвело до економії ресурсів. Також додано рекурсивні операції над директоріями, генерацію ZIP-архівів для пакетного скачування, а міграціями бази даних керує Flyway. Клієнтська частина на React.js підтримує перетягування файлів, індикатори прогресу, навігаційні ланцюжки та глобальний пошук. Клієнт і сервер взаємодіють через REST API із сесійними cookie.

Безпеку реалізовано на кількох рівнях. На транспортному - TLS 1.3 з посиленою конфігурацією (HSTS, OCSP Stapling, заборона вразливих протоколів). Паролі хешуються алгоритмом BCrypt з 12 раундами, що робить їх стійкими до більшості атак. Сесійні cookie захищено флагами HttpOnly та Secure, і одночасно активною може бути лише одна сесія користувача. Авторизація на основі унікального ID користувача працює через Spring Security. Параметризовані запити та валідація вхідних даних захищають від SQL-ін'єкцій та атак path traversal.

Тестування підтвердило працездатність системи. Усі 45 функціональних тест-кейсів пройшли успішно. Навантажувальне тестування в JMeter із 50 одночасними користувачами показало середній час відповіді 320 мс, пропускну спроможність 65 запитів за секунду та лише 0,05% помилок - при використанні 65% процесора та 4,5 ГБ оперативної пам'яті. Тестування безпеки за допомогою OWASP ZAP підтвердило надійність контролю доступу та відсутність критичних вразливостей.

Поставлена мета досягнута, усі завдання виконані. Розроблений веб-сервіс готовий до розгортання та дає користувачу повний контроль над власними даними.

СПИСОК ЛІТЕРАТУРИ

1. Fallatah K. U., Barhamgi M., Perera C. Personal Data Stores (PDS): A Review // Sensors. – 2023. – Vol. 23, № 3. – P. 1477. – DOI: 10.3390/s23031477.
2. Mollakuqe E., Hamdiu E., Fishekqiu N. S., Jakupi S., Qarkaxhija J. Comparison of cloud storage in terms of privacy and personal data - Sync, pCloud, IceDrive and Egnyte // Open Research Europe. – 2024. – Vol. 4. – Art. 128. – DOI: 10.12688/openreseurope.16631.1.
3. Zhou J. Comparison Of Cloud Storage in The Field of Personal Data and Big Data // Highlights in Science, Engineering and Technology. – 2024. – Vol. 81.
4. Almtrf A., Zohdy M. Cloud-Based Access Control to Preserve Privacy in Academic Web Application // Journal of Computer and Communications. – 2019. – Vol. 7, № 12. – P. 37–49. – DOI: 10.4236/jcc.2019.712005.
5. Pal S., Le D.-N., Pattnaik P. K. Cloud Computing Solutions: Architecture, Data Storage, Implementation, and Security. – Wiley, 2022. – ISBN 978-1-119-68202-
6. Wolohan J. T. (ed.). Object Storage Across the Cloud. – Manning, 2020.
7. Salam A., Ul Haq S., Gilani Z. Deploying and Managing a Cloud Infrastructure: Real World Skills for the Comptia Cloud+ Certification and Beyond: Cv0-001. – John Wiley & Sons Inc, 2015. – ISBN 9781118875100.
8. Carlson J. Take Control of Your Digital Storage. – Take Control Books, 2025.
9. Thuraisingham B. Developing and Securing the Cloud. – Auerbach Publications, 2014. – ISBN 9781138374539.
10. Information Resources Management Association. Cloud Security: Concepts, Methodologies, Tools, and Applications. – IGI Global, 2019. – ISBN 9781522581765.RB M., Chhetri S., KC A., Jain H. Secure File Storage & Sharing on Cloud Using Cryptography // International Journal of Computer Science and Mobile Computing. – 2021.
11. Sudarsa D., Rao A. N., A.P. S. An effective and secured authentication and sharing of data with dynamic groups in cloud // Data & Knowledge Engineering. – 2023. – Vol. 145. – P. 102125.

12. Sauber A. M., El-Kafrawy P. M., Shawish A. F., Amin M. A., Hagag I. M. A New Secure Model for Data Protection over Cloud Computing // Computational Intelligence and Neuroscience. – 2021. – Vol. 2021. – Art. 8113253. – DOI: 10.1155/2021/8113253.

13. Aware M. S., Balsurkar P., Aghav H., Kadu D., Choudhary S., Dongre N. G. Cloud Storage Service using Personal Desktop Computer // International Journal of Scientific Research in Computer Science, Engineering and Information Technology. – 2023. – DOI: 10.32628/CSEIT239033.

14. Rafi S. M., Yogesh R., Sriram M. Optimized dual access control for cloud-based data storage and distribution using global-context residual recurrent neural network // Computers & Security. – 2025. – Vol. 151. – P. 104183.

15. Al Lail M., Moncivais M., Benton R., Perez A. J. Cloud-Based Access Control Including Time and Location // Electronics. – 2024. – Vol. 13, № 14. – Art. 2812. – DOI: 10.3390/electronics13142812.

16. Барладим В. М., Берідзе К. С., Бруяка А. В., Горбаченко В. І., Коваленко В. В., Носенко Ю. Г., Мар'єнко М. В., Семеріков С. О., Шишкіна М. П. Використання сервісів адаптивних хмаро орієнтованих систем у діяльності вчителя. – Київ : Педагогічна думка, 2020.

17. Shantyr, A., Zinchenko, O., Storchak, K., Bondarchuk, A., & Pepa, Y. (2025). Prediction of quality software quality indicators with applied modifications of integrated gradiates methods. Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska, 15(2), 139-146.

18. Хмарна інфраструктура для банків [Електронний ресурс] // GigaCloud. – Режим доступу: https://gigacloud.ua/solutions/hmarna-infrastruktura-dlya-bankiv/. – Дата доступу: 15.12.2025.

19. Порівняння цін на тарифні плани [Електронний ресурс] // Google Workspace. – Режим доступу: https://gsuite.google.com.ua/intl/uk/pricing.html. – Дата доступу: 15.12.2025.

20. Ihor K. Будуємо сховище для даних з Google Cloud Platform [Електронний ресурс] // DOU. – 2023. – Режим доступу: https://dou.ua/forums/topic/46394/. – Дата доступу: 15.12.2025.

21. 2.5.1. Файл-менеджер [Електронний ресурс] // Хостінг Україна. – Режим доступу: https://www.ukraine.com.ua/wiki/hosting/files/file-manager/. – Дата доступу: 15.12.2025.

22. Lepide Новини. Що таке демократизація даних? [Електронний ресурс] // iIT Distribution Україна. – 2022. – Режим доступу: https://iitd.ua/shho-take-demokratizacziya-danih/. – Дата доступу: 15.12.2025.

23. Як продовжити користуватися хмарними сервісами Microsoft після завершення пільгових умов [Електронний ресурс] // AIN. – 2024. – Режим доступу: https://ain.ua/2024/03/27/yak-prodovzhyty-korystuvatysya-hmarnymy-servisamy-microsoft-pislya-zavershennya-pilgovyh-umov/. – Дата доступу: 15.12.2025.

24. Peculiarities of cloud service operations in Ukraine [Електронний ресурс] // Chambers. – 2025. – Режим доступу: https://chambers.com/articles/peculiarities-of-cloud-service-operations-in-ukraine. – Дата доступу: 15.12.2025.

25. The Cloud Saved a Nation: How Ukraine Backed Up an Entire Country During the War [Електронний ресурс] // Baltic Sentinel. – 2025. – Режим доступу: https://balticsentinel.eu/8350030/the-cloud-saved-a-nation-how-ukraine-backed-up-an-entire-country-during-the-war. – Дата доступу: 15.12.2025.

26. The Role of Cloud Services in the Hybrid War in Ukraine [Електронний ресурс] // Infosecurity Magazine. – 2022. – Режим доступу: https://www.infosecurity-magazine.com/blogs/cloud-services-hybrid-war-ukraine/. – Дата доступу: 15.12.2025.

27. Information Operations Surrounding the Russian Invasion of Ukraine [Електронний ресурс] // Google Cloud Blog. – 2022. – Режим доступу: https://cloud.google.com/blog/topics/threat-intelligence/information-operations-surrounding-ukraine. – Дата доступу: 15.12.2025.

28. Ukraine's state registers hit with one of Russia's largest cyberattacks [Електронний ресурс] // The Record. – 2024. – Режим доступу: https://therecord.media/ukraine-government-cyberattack-state-registers-russia. – Дата доступу: 15.12.2025.

29. How Cloud-Based Data Storage Enhances Security and Accessibility [Електронний ресурс] // Eskwad Blog. – 2024. – Режим доступу: https://blog.eskuad.com/how-cloud-based-data-storage-enhances-security-and-accessibility. – Дата доступу: 15.12.2025.

30. Bartley K. Data Statistics (2026) - How much data is there in the world?. *Rivery*. URL: https://rivery.io/blog/big-data-statistics-how-much-data-is-there-in-the-world/#:~:text=In%202024,%20the%20global%20volume,of%20storing%20250%20billion%20DVDs. (date of access: 25.04.2026).

31. Відомості Верховної Ради України. Про захист персональних даних. *Офіційний вебпортал парламенту України*. URL: https://zakon.rada.gov.ua/laws/show/2297-17#Text (дата звернення: 25.04.2026).

32.

33. Abramov, V., Astafieva, M., Boiko, M., Bodnenko, D., Bushma, A., Vember, V., ... & Yaskevych, V. (2021). *Theoretical and practical aspects of the use of mathematical methods and information technology in education and science*.

34. Машкіна, І. В., Абрамов, В. О., Носенко, Т. І., Іваніченко, Є. В., & Яскевич, В. О. (2024). Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп'ютерні науки для здобувачів вищої освіти денної форми навчання.

Додаток А

Фрагмент коду ентіті Resource та репозиторію (backend, Spring Boot)

```
Javapackage com.example.cloudstorage.entity;

import jakarta.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "resources", uniqueConstraints = @UniqueConstraint(columnNames =
{"owner_id", "path", "name"}))
public class Resource {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String path; // Нормалізований шлях до батьківської директорії

    private String name;

    private Long size; // Розмір у байтах, null для директорій

    @Enumerated(EnumType.STRING)
    private ResourceType type; // FILE або DIRECTORY

    @Column(name = "owner_id")
    private Long ownerId;

    @Column(name = "created_at")
    private LocalDateTime createdAt;

    @Column(name = "updated_at")
    private LocalDateTime updatedAt;

    // Getters and Setters
}

public enum ResourceType {
    FILE, DIRECTORY
}

// Репозиторій
package com.example.cloudstorage.repository;

import com.example.cloudstorage.entity.Resource;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface ResourceRepository extends JpaRepository<Resource, Long> {

    List<Resource> findByOwnerIdAndPathStartingWith(Long ownerId, String
parentPath);

    // Кастомні запити для пошуку, перевірки тощо
}
```

Додаток Б

Фрагмент React-компонента для відображення списку файлів (frontend)

```
jsximport React, { useState, useEffect } from 'react';

const FileManager = () => {
    const [resources, setResources] = useState([]);
    const [currentPath, setCurrentPath] = useState('/');

    useEffect(() => {
        fetch(`/api/directory?path=${currentPath}`, { credentials: 'include' })
            .then(response => response.json())
            .then(data => setResources(data));
    }, [currentPath]);

    return (
        <div>
            <h2>Поточний шлях: {currentPath}</h2>
            <table>
                <thead>
                    <tr>
                        <th>Ім'я</th>
                        <th>Розмір</th>
                        <th>Тип</th>
                    </tr>
                </thead>
                <tbody>
                    {resources.map(resource => (
                        <tr key={resource.id}>
                            <td>{resource.name}</td>
                            <td>{resource.size || '-'}</td>
                            <td>{resource.type}</td>
                        </tr>
                    ))}
                </tbody>
            </table>
        </div>
    );
};

export default FileManager;
```