МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю. М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему **Розробка веб-застосунку для зберігання та керування файлами**

Виконав: студент 4 курсу, групи 6.1212-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
М.В. Пащенко
(ініціали та прізвище)

Керівник доцент, к. ф.-м. н. І.А. Скрипник
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Рецензент директор ТОВ «Діскус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ім. Ю. М. Потебні ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення
(код та назва)

Освітня програма Програмне забезпечення систем
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Тетяна КРИТСЬКА

“_01_” __березня_____2026_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Пащенку Максиму Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи Розробка веб-застосунку для зберігання та керування файлами

керівник роботи Скрипник Ірина Анатоліївна, доцент, к. ф.-м. н.
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджена наказом ЗНУ від 23.12.2025 р. № 1830-с

2. Строк подання студентом кваліфікаційної роботи 02.06.2026
3. Вихідні дані бакалаврської роботи
   - комплект нормативних документів ;
   - технічне завдання до роботи.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   - огляд та збір літератури стосовно теми кваліфікаційної роботи;
   - огляд та аналіз існуючих рішень та аналогів;
   - дослідження проблеми зберігання даних у хмарних сховищах;
   - створення програмного продукту та його опис;
   - розробка висновків.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) 15 слайдів презентації

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.03.2026

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалавр-ської роботи	Примітка
1.	Аналіз предметної області	02.03-13.03.26	виконано
2.	Формулювання задачі	16.03-17.03.26	виконано
3.	Аналіз існуючих програмних рішень	18.03-25.03.26	виконано
4.	Дослідження засобів розробки веб-застосунку та алгоритмів шифрування інформації	26.03-01.04.26	виконано
5.	Проєктування бази даних	02.04-10.04.26	виконано
6.	Узгодження дій з науковим керівником	13.04-14.04.26	виконано
7.	Програмна реалізація серверної частини	15.04-24.04.26	виконано
8.	Імплементація алгоритму шифрування	27.04-30.04.26	виконано
9.	Проєктування клієнтської частини	01.05-08.05.26	виконано
10.	Інтеграція API	11.05-15.05.26	виконано
11.	Представлення результатів науковому керівнику	18.05-19.05.26	виконано
12.	Рефакторинг коду	20.05-22.05.26	виконано
13.	Налаштування системи авторизації	25.05-26.05.26	виконано
14.	Тестування веб-застосунку	27.05-28.05.26	виконано
15.	Оформлення звіту	29.05-05.06.26	виконано
16.	Оформлення презентації. Отримання рецензій.	07.06-15.06.26	виконано

Студент _____ _____М.В. Пащенко

Керівник роботи _____ _____І.А. Скрипник

Нормоконтроль пройдено
Нормоконтролер _____ _____І.А. Скрипник

АНОТАЦІЯ

Сторінок — 63

Рисунків — 13

Джерел — 19

Пащенко М.В. Розробка веб-застосунку для зберігання та керування файлами: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник І. А. Скрипник. Запоріжжя : ЗНУ, 2026, 63 с.

Мета полягає у створенні клієнт-серверного веб-застосунку, який поєднує зручність керування файлами з надійним захистом інформації без залежності від сторонніх корпоративних постачальників.

У роботі проведено аналіз існуючих рішень у сфері хмарних сховищ та виявлено їхні недоліки. Для вирішення виявлених проблем спроєктовано незалежну архітектуру системи.

Розроблено серверну частину з використанням сучасних асинхронних технологій, реляційної бази даних та системи кешування в оперативній пам'яті для забезпечення високої швидкості обробки запитів. Імплементовано алгоритм шифрування, що гарантує захист даних на рівні сховища.

Клієнтську частину реалізовано з використанням компонентного підходу, що забезпечує модульність та зручність підтримки інтерфейсу. Для зручності реалізовано динамічне оновлення інтерфейсу без необхідності перезавантаження сторінки, а також механізм фонового завантаження файлів із відображенням поточного стану виконання операцій.

Додатково впроваджено безпечну систему авторизації на базі токенів та гнучкий механізм керування правами спільного доступу до файлів і папок. Усю архітектуру проєкту контейнеризовано, що забезпечує її стабільну та ізольовану роботу в будь-якому середовищі розгортання.

Ключові слова: *хмарне сховище, вебзастосунок, клієнт-серверна архітектура, шифрування даних, база даних.*

ABSTRACT

Pages — 63

Drawings — 13

Sources — 19

Pashchenko M.V. Development of a web application for file storage and management: bachelor's qualification thesis in specialty 121 "Software Engineering" / academic supervisor I.A. Skrypnyk. Zaporizhzhia: ZNU, 2026, 63 p.

The aim is to create a client–server web application that combines convenient file management with reliable information security without dependence on third-party corporate providers.

The work analyzes existing solutions in the field of cloud storage and identifies their shortcomings. To address the detected problems, an independent system architecture is designed.

The server-side is developed using modern asynchronous technologies, a relational database, and an in-memory caching system to ensure high request processing performance. An encryption algorithm is implemented to guarantee data protection at the storage level.

The client-side is implemented using a component-based approach, providing modularity and ease of interface maintenance. For usability, dynamic interface updates without page reloads are implemented, as well as a background file upload mechanism with real-time progress tracking.

Additionally, a secure token-based authorization system is implemented along with a flexible access control mechanism for sharing files and folders. The entire project architecture is containerized, ensuring stable and isolated operation in any deployment environment.

Keywords: *cloud storage, web application, client–server architecture, data encryption, database.*

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 12
1.1 Сучасний стан та тенденції розвитку хмарних сховищ 12
1.2 Аналіз існуючих рішень та аналогів 13
1.2.1 Хмарні сервіси великих компаній 14
1.2.2 Спеціалізовані хмарні сховища 14
1.2.3 Платформи для власного розгортання 15
1.2.4 Висновки щодо існуючих аналогів 16
1.3 Постановка завдання 17
1.4 Висновки до розділу 1 18
2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ВЕБ-ЗАСТОСУНКУ 19
2.1 Проєктування архітектури системи 19
2.2 Проєктування структури бази даних 19
2.3 Проєктування підсистеми безпеки 20
2.4 Проєктування клієнтської частини 21
2.5 Висновки до розділу 2 22
3 РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ 23
3.1 Вибір та обґрунтування технологій розробки 23
3.1.1 Аналіз та розробка серверної частини 23
3.1.2 Обґрунтування вибору СУБД 25
3.1.3 Інтеграція системи кешування 25
3.1.4 Вибір технологій для розробки клієнтського інтерфейсу 26
3.1.5 Вибір допоміжних бібліотек та механізмів безпеки 27
3.1.6 Інфраструктура проєкту 28
3.2 Проектування бази даних та структури збереження інформації 30
3.2.1 ER-діаграма бази даних 30
3.2.2 Опис таблиць бази даних 31
3.3 Розробка серверної частини 33

3.3.1 Реалізація системи автентифікації та безпеки 34
3.3.2 Управління файловою системою..... 36
3.3.3 Організація спільного доступу та управління контактами..... 40
3.3.4 Оптимізація продуктивності за допомогою кешування 42
3.4 Розробка клієнтської частини 43
3.4.1 Проектування користувацького інтерфейсу 45
3.4.2 Реалізація сторінок доступу 46
3.4.3 Розробка головної панелі управління 48
3.4.4 Інтерфейс публічного доступу до файлів 50
3.5 Апробація..... 51
3.6 Висновки до розділу 3..... 59
ВИСНОВКИ..... 61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 62

ВСТУП

Актуальність теми

У сучасну епоху стрімкої цифровізації інформація стала одним із найцінніших ресурсів, а обсяги цифрових даних, що генеруються як окремими користувачами, так і корпоративним сектором, зростають експоненційно. Це зумовлює критичну потребу в надійних, швидких та безпечних системах для зберігання, структурування та обміну файлами.

Сьогодні на ринку домінують глобальні хмарні платформи (такі як Google Drive, Dropbox, Microsoft OneDrive), які пропонують високий рівень зручності. Проте використання цих сервісів супроводжується низкою суттєвих недоліків. По-перше, користувачі стикаються з проблемою прив'язки до постачальника та жорсткими лімітами на безкоштовний обсяг пам'яті, що призводить до постійної комерціалізації додаткових послуг. По-друге, закритий вихідний код цих платформ унеможливлює проведення незалежного аудиту безпеки.

Найголовнішою проблемою існуючих рішень є питання приватності та контролю над даними. У більшості популярних хмарних сховищ ключі шифрування зберігаються на серверах самих корпорацій, що залишає за ними технічну можливість доступу до конфіденційної інформації користувачів і створює ризики масових витоків даних у разі успішних кібератак. Вузькоспеціалізовані захищені сервіси, у свою чергу, часто програють у швидкодії та пропонують перевантажений, незручний інтерфейс.

З огляду на це, розробка власного незалежного клієнт-серверного веб-застосунку для керування файлами є надзвичайно актуальною задачею. Створення системи, яка поєднує сучасний підхід до побудови інтерфейсу користувача із надійною бекенд-архітектурою та імплементацією криптографічного стандарту шифрування для захисту даних, дозволяє вирішити компроміс між зручністю та безпекою.

Мета дослідження

Проєктування та розробка незалежного високопродуктивного клієнт-серверного веб-застосунку для зберігання, структурування та обміну файлами, який поєднує зручний користувацький інтерфейс із гарантованим захистом даних, усуваючи залежність від сторонніх корпоративних хмарних провайдерів.

Завдання дослідження

Для досягнення поставленої мети планується вирішити такі завдання: проаналізувати існуючі хмарні рішення та їхні вразливості; спроєктувати базу даних і клієнт-серверну архітектуру взаємодії за принципами REST API; розробити серверну частину на базі Node.js та Express із використанням системи кешування Redis; імплементувати алгоритм симетричного шифрування файлів стандарту AES-256-CBC; створити клієнтський інтерфейс за допомогою бібліотеки React; впровадити безпечну авторизацію на основі токенів JWT та протоколу OAuth і механізм спільного доступу до файлів, а також контейнеризувати фінальний застосунок за допомогою платформи Docker для легкого розгортання.

Об’єкт дослідження

Об’єктом дослідження є процес безпечного збереження, структурування та обміну цифровими даними у розподіленому веб-середовищі.

Предмет дослідження

Предметом дослідження є моделі, методи та програмні засоби розробки клієнт-серверного веб-застосунку для управління файлами з використанням криптографічного захисту інформації.

Методи дослідження

У процесі виконання роботи проведено аналіз сучасних технологій веб-

розробки та існуючих систем зберігання файлів. Виконано проєктування архітектури веб-застосунку, реалізовано його програмну частину, проведено тестування основних функцій і апробацію отриманих результатів.

Практичне значення одержаних результатів

Практичне значення одержаних результатів полягає у розробці готового до впровадження програмного продукту, який дозволяє організувати незалежне веб-сховище файлів. Створена система забезпечує повну конфіденційність інформації, усуває залежність від сторонніх комерційних провайдерів та може бути швидко розгорнута на будь-яких корпоративних або локальних серверах. Крім того, розроблену архітектуру можна використовувати як надійну основу для подальшого створення та масштабування захищених систем електронного документообігу.

Апробація результатів

Основні положення та результати роботи були представлені у вигляді тез доповіді на науково-практичній конференції студентів, аспірантів і молодих вчених «Молода наука — 2026» [19].

Глосарій

AES-256-CBC — це стандарт симетричного блочного шифрування із довжиною ключа 256 біт, що забезпечує конфіденційність завантажених файлів на рівні файлової системи.

Axios — це спеціалізована бібліотека для виконання асинхронних HTTP-запитів із клієнтської частини до серверного API.

Docker — це платформа для автоматизації розгортання застосунків у ізольованих контейнерах, що гарантує ідентичність робочого середовища на різних серверах.

Express — це фреймворк для середовища Node.js [5], призначений для швидкої розробки серверної частини та обробки мережевих запитів.

JWT — це компактний формат передачі даних у вигляді криптографічно підписаного токена для безпечної авторизації користувачів.

Lucide React — це бібліотека легких та сучасних векторних іконок, інтегрованих як окремі компоненти для інтерфейсу.

Node.js — це середовище виконання коду JavaScript на боці сервера, побудоване на рушії V8, що підтримує асинхронну модель обробки подій.

Nodemailer — це модуль для автоматизації відправки електронних листів безпосередньо із серверної частини застосунку.

OAuth — це протокол безпечної авторизації, який дозволяє користувачам отримувати доступ до системи через сторонні сервіси, зокрема обліковий запис Google.

PostgreSQL — це потужна об'єктно-реляційна система управління базами даних, що забезпечує надійне збереження структурованої інформації та зв'язків між таблицями.

Prisma — це сучасний інструмент об'єктно-реляційного відображення, що надає типізований інтерфейс для взаємодії з базою даних.

React — це декларативна бібліотека для створення динамічних користувацьких інтерфейсів на основі багаторазових компонентів.

Redis — це високопродуктивне сховище структур даних в оперативній пам'яті, яке використовується для кешування сесій та прискорення доступу до інформації.

SPA — це архітектурний підхід до побудови веб-застосунків, за якого вся система завантажується як одна сторінка, а оновлення контенту відбувається динамічно.

Tailwind CSS — це сучасний CSS-фреймворк, що базується на використанні допоміжних класів для швидкої стилізації елементів інтерфейсу без написання коду.

Vite — це інструмент для збірки фронтенд-проєктів, який забезпечує миттєве оновлення модулів під час розробки.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Сучасний стан та тенденції розвитку хмарних сховищ

Стрімкий розвиток інформаційних технологій та загальна цифровізація суспільства призвели до безпрецедентного зростання обсягів цифрової інформації. За даними міжнародних аналітичних агенцій, глобальний обсяг даних генерується експоненційно, що фундаментально змінює підходи до їх зберігання, обробки та захисту.

Історично еволюція систем зберігання пройшла шлях від використання локальних фізичних носіїв (жорстких дисків, оптичних дисків, флеш-накопичувачів) до розподілених мережевих архітектур. Сьогодні абсолютною домінантою на ринку є хмарні обчислення та хмарні сховища даних. Ця парадигма дозволила відв'язати інформацію від конкретного апаратного пристрою користувача, забезпечивши цілодобовий доступ до файлів з будь-якої точки світу за наявності підключення до мережі Інтернет.

Сучасний стан систем зберігання цифрових даних визначається кількома ключовими тенденціями. Насамперед спостерігається стійкий перехід до веб-орієнтованих інтерфейсів та моделі програмне забезпечення як послуга, де головним інструментом взаємодії з даними виступає браузер.

Цей перехід був значною мірою прискорений глобальним впливом віддаленої роботи. Масовий перехід до дистанційного та змішаного форматів роботи зробив хмарні системи критично важливою інфраструктурою. Це сформувало гостру потребу не лише у базовому зберіганні, а й у зручних механізмах поширення даних, керуванні правами доступу та синхронізації змін у реальному часі.

Водночас розвиток галузі супроводжується монополізацією ринку великими IT-корпораціями, такими як Google Workspace, Microsoft Azure/OneDrive та Amazon Web Services. Хоча вони пропонують широкий спектр інтегрованих сервісів у межах однієї платформи, ці компанії

створюють жорстку прив'язку користувача чи бізнесу до своєї екосистеми, що значно ускладнює перенесення даних і дозволяє їм одноосібно диктувати цінову політику.

Поряд із цим, аналіз галузі дозволяє виділити нові тенденції розвитку, які формуються у відповідь на недоліки поточних монопольних рішень. Насамперед значно посилюється фокус на кібербезпеці та криптографії. Через почастішання випадків масових витоків інформації з публічних хмарних сервісів, користувачі та бізнес почали вимагати вищого рівня приватності, що формує стійкий тренд до впровадження концепції нульової довіри та використання надійних стандартів шифрування для захисту даних.

Іншою важливою тенденцією є повернення до суверенітету даних. У зв’язку з жорсткими регуляторними нормами щодо захисту персональної інформації (наприклад, GDPR у Європі), компанії все частіше відмовляються від зберігання чутливих комерційних даних на серверах третіх осіб. Цей фактор активно стимулює розвиток Self-hosted рішень — систем, які можна безпечно розгорнути на власних серверах підприємства.

Отже, сучасні тенденції демонструють поступовий перехід від використання виключно публічних хмарних сервісів до рішень, що поєднують зручний веб-доступ, високу продуктивність та підвищений рівень контролю над конфіденційністю даних. У зв’язку з цим зростає актуальність розробки незалежних веб-застосунків для зберігання файлів, які забезпечують надійний захист інформації за допомогою криптографічних механізмів та ефективне керування даними.

1.2 Аналіз існуючих рішень та аналогів

Для визначення доцільності та основних напрямків розробки веб-застосунку для зберігання файлів необхідно провести аналіз існуючих рішень, представлених на сучасному ринку. На сьогодні доступна велика кількість платформ, які відрізняються за принципами роботи, рівнем безпеки, способом

розгортання та функціональними можливостями. Умовно такі системи можна поділити на три основні категорії: хмарні сервіси великих технологічних компаній, спеціалізовані платформи із підвищеним рівнем захисту даних, а також рішення для самостійного розгортання на власній інфраструктурі.

1.2.1 Хмарні сервіси великих компаній

Ця категорія є однією з найпоширеніших серед приватних користувачів і корпоративного сектору завдяки моделі програмне забезпечення як послуга, та активній підтримці великих технологічних компаній. Такі системи забезпечують високий рівень доступності, зручний та інтуїтивно зрозумілий інтерфейс, а також тісну інтеграцію з офісними сервісами та іншими інструментами екосистеми.

Водночас подібні рішення мають низку недоліків, пов’язаних насамперед із конфіденційністю даних. У більшості випадків ключі шифрування зберігаються на стороні компанії, яка надає послугу, що потенційно створює ризик доступу до інформації як з боку сервісу, так і у випадку компрометації інфраструктури. Додатковими обмеженнями є невеликий обсяг безкоштовного дискового простору та залежність від конкретної платформи, через що перенесення даних до інших сервісів може бути ускладненим.

1.2.2 Спеціалізовані хмарні сховища

Ці системи розробляються як альтернатива масовим хмарним сервісам із основним акцентом на забезпечення конфіденційності та кібербезпеці даних. Вони орієнтовані переважно на користувачів і організації, для яких захист інформації є одним із ключових пріоритетів. Основною особливістю таких платформ є використання архітектури Zero-Knowledge та сучасних алгоритмів шифрування, завдяки чому постачальник послуг не має доступу до ключів користувача і не може отримати доступ до вмісту файлів.

Разом із підвищеним рівнем захисту подібні рішення часто мають певні

обмеження. Використання складних механізмів шифрування може негативно впливати на швидкість обробки та передачі великих файлів, а також ускладнювати взаємодію користувача із системою. Крім того, такі сервіси зазвичай функціонують за платною моделлю підписки та є пропрієтарним програмним забезпеченням із закритим вихідним кодом, що обмежує можливість їх адаптації або розгортання на власній інфраструктурі підприємства.

1.2.3 Платформи для власного розгортання

Ця категорія охоплює програмне забезпечення з відкритим або частково відкритим вихідним кодом, яке розгортається безпосередньо на власних апаратних ресурсах користувача або підприємства. До найбільш відомих платформ такого типу належать Nextcloud, ownCloud та Seafile. Дані системи забезпечують можливість самостійного керування процесом зберігання та обміну файлами без залежності від сторонніх хмарних провайдерів. Основною перевагою таких рішень є повний контроль над інфраструктурою та даними, що дозволяє користувачу самостійно проводити аудит безпеки, налаштовувати серверне середовище та уникати регулярних витрат на оренду хмарного простору.

Разом із цим більшість популярних self-hosted платформ мають низку технічних недоліків. Значна частина таких систем побудована на застарілих монолітних архітектурах, що ускладнює масштабування та підвищує вимоги до серверних ресурсів. Наприклад, Nextcloud та ownCloud, окрім функцій зберігання файлів, містять велику кількість додаткових модулів, зокрема поштові сервіси, календарі, офісні інструменти та засоби відеозв’язку, що може призводити до перевантаження системи та зниження продуктивності. Платформа Seafile вважається більш легкою та продуктивною, однак має менш розвинену екосистему додаткових можливостей. Окремою проблемою є складність розгортання та адміністрування, оскільки підтримка подібних платформ потребує спеціалізованих знань у сфері системного адміністрування

та серверної інфраструктури.

Проведений аналіз існуючих аналогів демонструє наявність певних недоліків у кожній із розглянутих категорій рішень. Масові хмарні сервіси не забезпечують достатнього рівня конфіденційності, спеціалізовані захищені платформи характеризуються високою вартістю та закритістю, а self-hosted системи часто є надмірно складними та ресурсомісткими. У зв’язку з цим актуальним залишається створення легкого, сучасного та незалежного веб-застосунку для зберігання файлів.

Використання сучасного асинхронного середовища Node.js [5], бібліотеки React [4] для побудови клієнтської частини та алгоритму шифрування AES у режимі AES-256-CBC для захисту даних дозволяє створити продуктивну та безпечну систему. Додаткове застосування Docker [11] спрощує процес розгортання та забезпечує гнучкість використання як на локальних серверах, так і у хмарній інфраструктурі, зберігаючи при цьому повний контроль користувача над власними даними [1].

1.2.4 Висновки щодо існуючих аналогів

Проведений аналіз існуючих рішень показує, що сучасний ринок систем зберігання файлів не має універсальної платформи, яка б одночасно забезпечувала високий рівень конфіденційності, продуктивності та простоти використання. Масові хмарні сервіси орієнтовані переважно на зручність і доступність, однак не гарантують повного контролю над даними користувача. Спеціалізовані захищені платформи забезпечують підвищений рівень безпеки, проте часто характеризуються високою вартістю та закритістю програмної архітектури. У свою чергу self-hosted рішення нерідко є складними в адмініструванні, перевантаженими додатковим функціоналом і недостатньо гнучкими з точки зору масштабування.

У зв’язку з цим актуальним є створення сучасного веб-застосунку для зберігання файлів, який поєднуватиме продуктивність, безпечність та незалежність від сторонніх сервісів. Використання середовища Node.js [5],

бібліотеки React [4] та сучасних механізмів шифрування дозволить реалізувати ефективну систему із високою швидкодією та надійним захистом даних. Додаткове використання Docker [11] забезпечить спрощене розгортання та гнучкість використання на різних типах серверної інфраструктури, зберігаючи при цьому повний контроль користувача над власною інформацією.

1.3 Постановка завдання

Метою даної роботи є розробка веб-застосунку для безпечного зберігання та керування файлами, який забезпечуватиме високий рівень конфіденційності інформації та можливість незалежного використання без прив’язки до сторонніх хмарних сервісів. Система повинна поєднувати функціональність сучасних файлових сховищ із надійними механізмами захисту даних та простотою використання.

Для досягнення поставленої мети необхідно реалізувати систему реєстрації та авторизації користувачів із підтримкою підтвердження електронної пошти та входу через сторонні сервіси. Застосунок має забезпечувати можливість завантаження, зберігання, структурування та видалення файлів, підтримувати роботу з папками та надавати функціонал масового завантаження даних.

Важливою складовою системи є забезпечення захисту інформації шляхом автоматичного шифрування файлів під час їх збереження. Також необхідно реалізувати механізми спільного доступу до файлів і папок між користувачами, включаючи можливість створення публічних посилань для доступу до даних.

Окрему увагу необхідно приділити створенню зручного користувацького інтерфейсу, який забезпечуватиме ефективну взаємодію з файловою структурою та підтримку сучасних способів роботи з файлами. Серверна частина застосунку повинна забезпечувати стабільну обробку

запитів, ефективне керування даними та можливість подальшого масштабування системи.

Після завершення розробки необхідно провести апробацію створеного програмного продукту, перевірити працездатність основних функцій системи, оцінити продуктивність, коректність роботи механізмів захисту даних та зручність використання веб-застосунку в реальних умовах експлуатації.

Результатом виконання роботи має стати готовий вебзастосунок для зберігання файлів, який поєднуватиме продуктивність, безпечність та повний контроль користувача над власними даними.

1.4 Висновки до розділу 1

1. Проведено аналіз сучасних веб-орієнтованих систем зберігання файлів та визначено основні тенденції розвитку хмарних сервісів.
2. Досліджено існуючі комерційні, захищені та self-hosted платформи, що дозволило виявити їх основні переваги та недоліки.
3. Визначено ключові проблеми сучасних файлових сервісів, серед яких недостатній рівень конфіденційності даних, залежність від сторонніх провайдерів та складність адміністрування окремих рішень.
4. Обґрунтовано доцільність розробки незалежного веб-застосунку для безпечного зберігання та керування файлами, який поєднуватиме продуктивність, безпечність та простоту використання.
5. Визначено основні технології для реалізації системи, зокрема використання Node.js [5] для серверної частини, React [4] для створення користувацького інтерфейсу, PostgreSQL [6] для зберігання даних та Docker для забезпечення простого розгортання та гнучкості використання системи.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ВЕБ-ЗАСТОСУНКУ

2.1 Проєктування архітектури системи

Проєктування загальної архітектури системи базується на використанні трирівневої моделі, що забезпечує чіткий розподіл відповідальності між інтерфейсом користувача, бізнес-логікою та шаром збереження даних.

Перший рівень — це клієнтська частина, яку планується реалізувати як сучасний Single Page Application на базі React [4] із використанням збирача Vite [9]. Вона відповідатиме за візуалізацію даних, обробку дій користувача та взаємодію із сервером через асинхронні запити за допомогою бібліотеки Axios [15].

Другий рівень — це серверна частина на базі середовища Node.js [5] та фреймворку Express [8]. Цей рівень виконуватиме роль REST API, забезпечуючи безпечну авторизацію через токени JWT [7] і Google Auth 2.0 [13], а також реалізуючи логіку шифрування файлів за стандартом AES-256-CBC.

Третій рівень — це шар збереження даних, що включатиме реляційну базу PostgreSQL [6] для структурування інформації та систему кешування Redis [3] для прискорення доступу до даних. Взаємодія з базою даних здійснюватиметься через Prisma ORM [2].

Для забезпечення цілісності та портативності системи доцільно використовувати Docker, що дозволить розгортати програмні компоненти в ізольованих контейнерах та спростить перенесення застосунку між різними серверами.

2.2 Проєктування структури бази даних

Для забезпечення надійного зберігання та обробки інформації передбачається використання PostgreSQL. Під час проєктування структури бази даних буде реалізовано логічну організацію таблиць і зв’язків між ними

для забезпечення цілісності даних та уникнення дублювання інформації. Для взаємодії серверної частини із базою даних планується використання Prisma ORM [2], що спростить роботу з моделями даних та забезпечить зручне керування структурою бази даних.

Логічна структура бази даних включатиме такі основні сутності:

- Користувач — зберігатиме ідентифікатори, дані для автентифікації, включаючи хешовані паролі та унікальні маркери Google Auth 2.0 [13].
- Файл — міститиме метадані об'єктів, а саме назву, розмір, тип контенту, шлях до фізичного місця збереження зашифрованого блоку та параметри ініціалізації для алгоритму шифрування.
- Папка — забезпечуватиме ієрархічну структуру сховища, пов'язуючи файли та вкладені директорії з їхніми власниками.
- Права доступу — описуватиме зв’язки між об’єктами та іншими користувачами для реалізації функціоналу спільного використання даних.

Окремим рівнем архітектури даних виступатиме система Redis [3]. Її планується використовувати як високошвидкісне сховище для тимчасових об'єктів, зокрема сесій користувачів, активних токенів доступу та кешованих результатів частих запитів до бази даних PostgreSQL. Це дозволить суттєво знизити навантаження на основний диск і прискорити процес авторизації та навігації по файловій системі.

2.3 Проєктування підсистеми безпеки

Для гарантування конфіденційності та цілісності інформації в межах проєкту планується впровадження багаторівневого механізму захисту. На етапі входу в систему передбачено реалізацію автентифікації за допомогою токенів JWT [7], що дозволить безпечно передавати дані про сесію між клієнтом та сервером без ризику перехоплення відкритих паролів. Також заплановано інтеграцію протоколу Google Auth 2.0 для забезпечення швидкої

авторизації користувачів

Безпека збереження локальних облікових записів базуватиметься на використанні бібліотеки Bcrypt, яка забезпечить незворотне хешування паролів із додаванням криптографічної солі. Для захисту від несанкціонованих міждоменних запитів на рівні сервера буде налаштовано політики CORS та механізми валідації вхідних даних, а всі критичні параметри конфігурації планується винести у змінні середовища за допомогою Dotenv.

Центральною частиною підсистеми стане механізм шифрування контенту за стандартом AES-256-CBC. Шифрування виконуватиметься на боці сервера безпосередньо під час завантаження файлу через Multer [12] , що гарантуватиме збереження даних на диску у нечитабельному вигляді. Кожен об'єкт отримає власний унікальний вектор ініціалізації, що дозволить забезпечити високий рівень криптостійкості та захистити приватну інформацію користувача від доступу на рівні системного адміністратора сховища.

2.4 Проєктування клієнтської частини

Клієнтська частина системи розробляється як сучасний Single Page Application на базі бібліотеки React [4] та інструменту збірки Vite [9]. Основний акцент при проєктуванні робиться на створенні модульної архітектури, де інтерфейс розділений на багаторазові компоненти, що забезпечує легкість підтримки та масштабування коду.

Для стилізації елементів інтерфейсу обрано фреймворк Tailwind CSS [10], який дозволяє реалізувати адаптивний дизайн за допомогою набору утилітарних класів. Навігація між основними розділами застосунку, зокрема головною сторінкою та сторінками авторизації, здійснюватиметься за допомогою React Router DOM, який гарантує швидкий перехід між екранами без повного перезавантаження сторінки в браузері.

Особлива увага приділяється механізмам взаємодії з користувачем:

- Оптимістичний інтерфейс — імплементація цього патерну дозволить миттєво відображати зміни у файловій структурі ще до отримання фінального підтвердження від сервера.
- Завантаження даних — використання бібліотеки React Dropzone [16] для реалізації інтуїтивного механізму перетягування файлів та Axios [15] для стабільної передачі потоків даних на бекенд.
- Графічне оформлення — інтеграція бібліотеки Lucide React [17] для відображення сучасних векторних іконок та інструментів React Avatar Editor [18] для персоналізації профілю користувача.

Взаємодія з серверним API буде організована через централізований сервісний шар, що забезпечить коректну передачу токенів авторизації та обробку метаданих за допомогою jsmediatags, дозволяючи користувачеві бачити детальну інформацію про свої медіафайли ще до їх відкриття.

2.5 Висновки до розділу 2

1. Спроєктовано загальну архітектуру веб-застосунку, що включає клієнтську частину, серверний API та систему зберігання даних.
2. Розроблено логічну структуру бази даних та визначено основні сутності системи, необхідні для роботи з користувачами, файлами, папками та правами доступу.
3. Сформовано підсистему безпеки, яка включає механізми автентифікації, захисту облікових записів та шифрування файлів для забезпечення конфіденційності даних.
4. Визначено принципи побудови клієнтської частини застосунку, орієнтованої на зручну взаємодію користувача з файловою системою та динамічне оновлення інтерфейсу.
5. Обґрунтовано використання технологій Docker, PostgreSQL, Redis та React для забезпечення продуктивності, масштабованості та стабільної роботи системи.

3 РОЗРОБКА ПРОГРАМНОГО ЗАСТОСУНКУ

3.1 Вибір та обґрунтування технологій розробки

Етап вибору технологічного стеку є фундаментальним кроком у життєвому циклі розробки програмного забезпечення. Від ухвалених на цьому етапі рішень залежить не лише швидкість та зручність написання коду, але й подальша масштабованість, безпека, продуктивність системи, а також вартість її підтримки. Для реалізації веб-застосунку хмарного сховища було проведено аналіз сучасних архітектурних рішень та обрано стек технологій, що найкраще задовольняє вимоги до систем з високим навантаженням операцій введення-виведення (Рис. 1).

3.1.1 Аналіз та розробка серверної частини

При виборі технології для розробки серверної частини розглядалися такі альтернативи, як Django та FastAPI, Spring Boot та Node.js [5].

Для реалізації даного проєкту було обрано платформу Node.js. Основним аргументом на користь Node.js є його архітектура. Платформа побудована на базі високопродуктивного рушія V8 від Google і використовує асинхронну, подієво-орієнтовану структуру. Оскільки розроблюваний застосунок є файловим сховищем, основним навантаженням на сервер є не обчислення, а операції читання/запису файлів та робота з БД. У таких умовах Node.js дозволяє обробляти тисячі з'єднань в одному потоці.

В якості серверного фреймворку було обрано Express.js [8]. У порівнянні з іншими фреймворками, Express надає розробнику повний контроль над архітектурою. Він має велику екосистему модулів, що дозволило легко інтегрувати Multer [12] для завантаження файлів та модулі безпеки

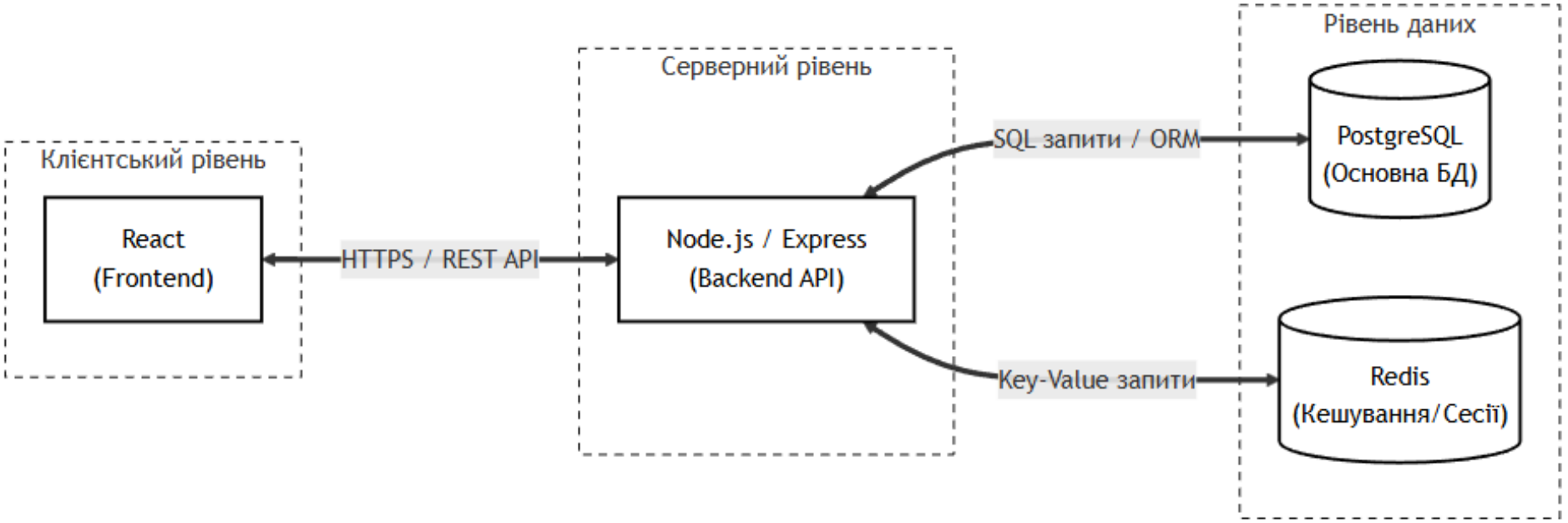


Рисунок 1 — *Загальна архітектура та взаємодія компонентів обраного технологічного стеку*

3.1.2 Обґрунтування вибору СУБД

Оскільки проєкт передбачає створення складної ієрархічної структури (користувачі мають папки, папки містять файли або інші папки, файли мають права доступу), використання реляційної бази даних є обов'язковим для збереження цілісності даних.

Вибір було зроблено на користь PostgreSQL [6] — надійної об'єктно-реляційної СУБД, яка відмінно справляється зі складними зв'язками та гарантує, що ієрархія файлової системи користувача не буде порушена.

Для оптимізації роботи з базою даних на рівні коду було інтегровано Prisma ORM [2]. Prisma кардинально змінює підхід до взаємодії з БД. Вся структура описується в єдиному декларативному файлі, після чого автоматично генерується клієнт доступу.

3.1.3 Інтеграція системи кешування

Сучасні веб-застосунки вимагають високої швидкості відгуку. Для зменшення навантаження на основну базу даних PostgreSQL [6] було використано Redis [3]. Це надшвидке сховище даних типу "ключ-значення", що працює в оперативній пам'яті (Рис. 2).

У межах проєкту Redis [3] застосовується для тимчасового зберігання сесій користувачів, токенів та кодів верифікації електронної пошти. Використання Redis зменшило час відповіді сервера на повторювані запит.

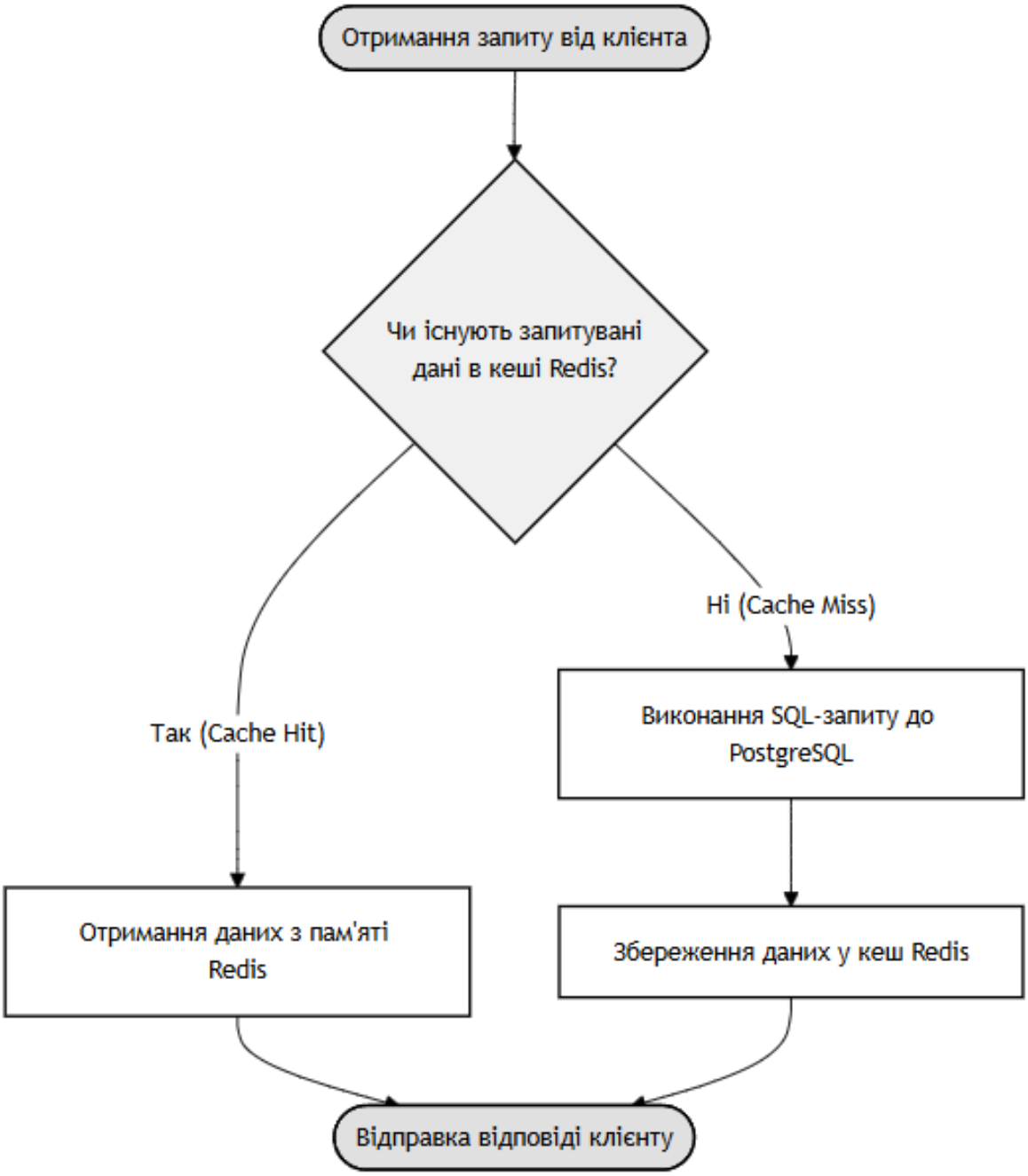


Рисунок 2 — *Алгоритм обробки запиту з використанням системи кешування Redis*

3.1.4 Вибір технологій для розробки клієнтського інтерфейсу

Для розробки візуальної частини застосунку було обрано бібліотеку React.js. Порівняно з Angular чи Vue.js, React пропонує гнучкий компонентний підхід на основі віртуального DOM. При зміні стану (наприклад, при

завантаженні нового файлу), React перемальовує лише той елемент, який змінився. Це критично важливо для Single Page Application.

В якості інструменту збирання проєкту було відкинуто класичний Webpack на користь сучасного Vite [9].

Для стилізації інтерфейсу використано Tailwind CSS [10]. Усі стилі задаються безпосередньо в класах React-компонентів, що значно прискорює розробку візуальної частини сторінок та зменшує розмір кінцевого CSS-файлу.

3.1.5 Вибір допоміжних бібліотек та механізмів безпеки

Для реалізації повноцінного функціоналу застосунку використано спеціалізовані бібліотеки:

Зокрема, використано пакет jsonwebtoken для генерації JWT-токенів доступу та Bcrypt для хешування паролів. Приклад генерації JWT-токена наведено в лістингу 1.

Лістинг коду 1 — Генерація JWT-токена в контролері автентифікації

```
const jwt = require('jsonwebtoken');

const token = jwt.sign(
    {
        id: user.id,
        email: user.email
    },
    process.env.JWT_SECRET,
    {
        expiresIn: '24h'
    }
);
```

Для роботи з мережею інтегровано HTTP-клієнт axios з налаштуванням інтерцепторів для додавання токенів авторизації.

Натомість для реалізації функції перетягування файлів обрано react-dropzone.

3.1.6 Інфраструктура проєкту

Важливим аспектом розробки також є організація процесу керування вихідним кодом. Для контролю змін і ведення історії розробки передбачається використання розподіленої системи контролю версій Git.

З метою уніфікації середовища виконання було застосовано технологію контейнеризації Docker [11] . Фронтенд, бекенд, база даних PostgreSQL та кеш-сервер Redis [3] упаковуються в контейнери (Рис.3). Це забезпечує ідентичність середовищ розробки та розгортання. Приклад вмісту файлу Dockerfile наведено в лістингу 2.

Лістинг коду 2 — Вміст файлу Dockerfile для контейнеризації серверної частини застосунку

```
FROM node:22-alpine

WORKDIR /src

COPY package*.json ./

RUN npm install

COPY . .

RUN npx prisma generate

EXPOSE 5000

CMD ["npm", "start"]
```

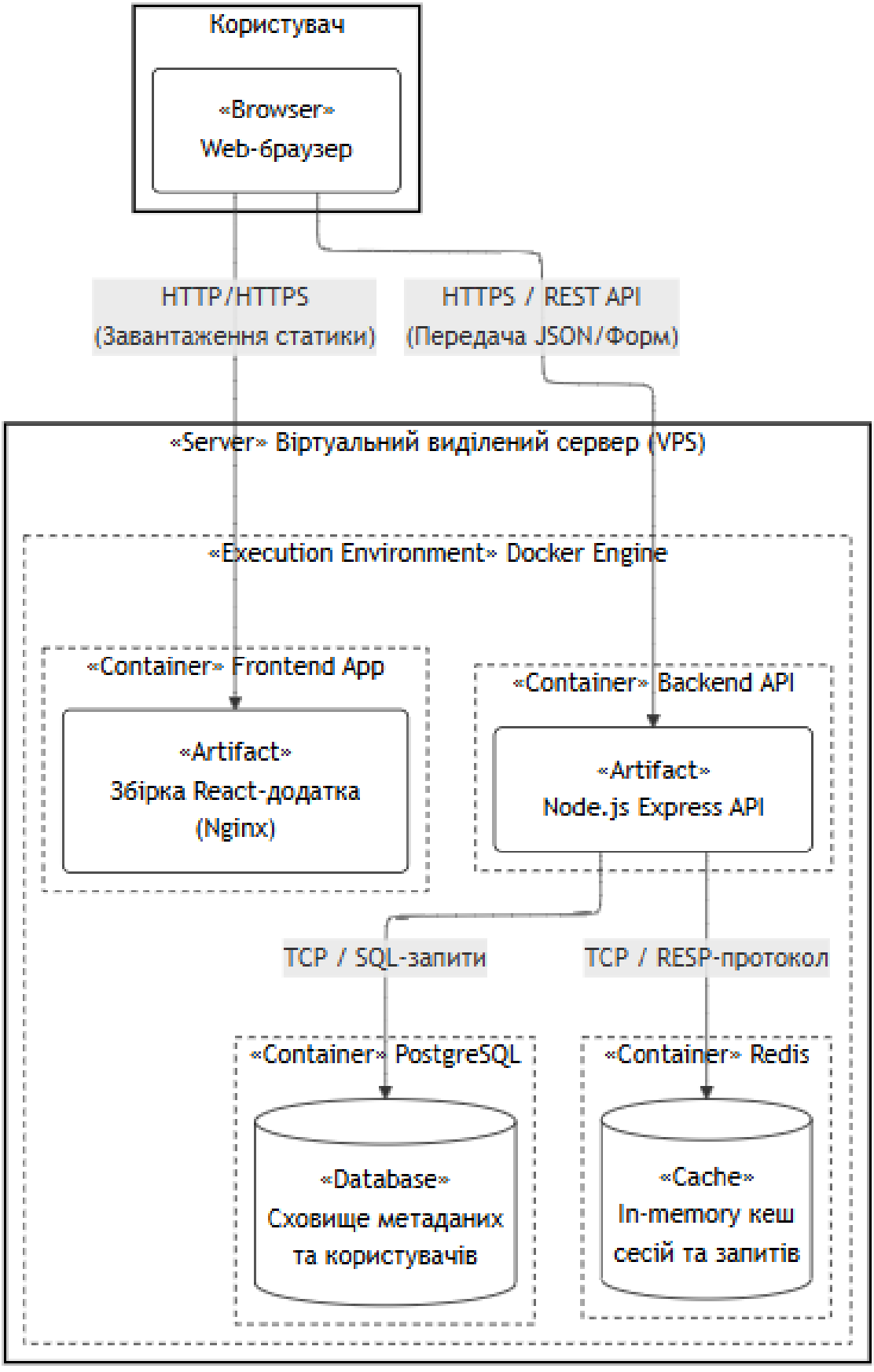


Рисунок 3 — *Розгортання інфраструктури застосунку у Docker*

3.2 Проектування бази даних та структури збереження інформації

Основою будь-якого інформаційного веб-застосунку є база даних, яка забезпечує надійне зберігання, швидкий пошук та підтримку цілісності даних. У процесі проектування архітектури хмарного сховища було розроблено реляційну схему бази даних, що складається з 6 основних таблиць: User, File, Folder, Contact, SharedFile та SharedFolder.

Зв'язки між таблицями налаштовані таким чином, щоб гарантувати цілісність файлової ієрархії. Наприклад, використано каскадне видалення ON DELETE CASCADE, завдяки якому при видаленні папки автоматично видаляються всі вкладені в неї файли та записи про спільний доступ.

3.2.1 ER-діаграма бази даних

Логічну структуру бази даних та зв'язки між сутностями найкраще ілюструє ER-діаграма (Рис. 4).

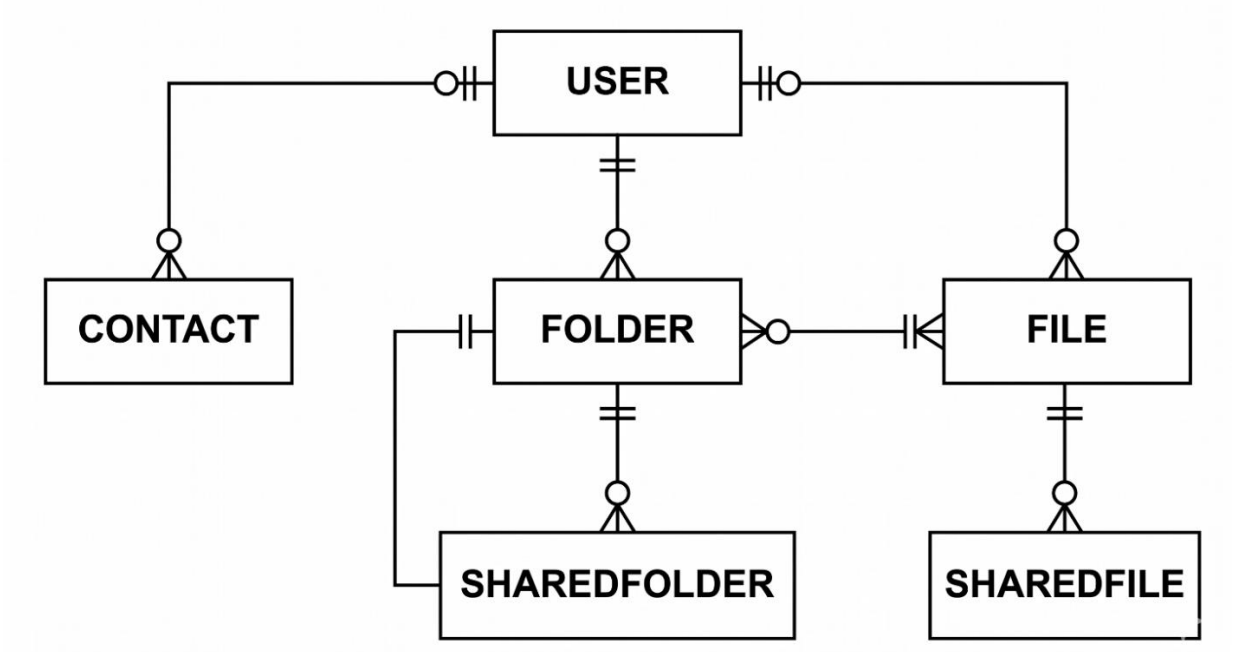


Рисунок 4 — *ER-діаграма бази даних*

3.2.2 Опис таблиць бази даних

Нижче наведено детальний опис кожної таблиці бази даних із зазначенням типів даних полів та їхнього призначення в системі.

База даних містить в собі наступні сутності(Табл.1 — 5):

Таблиця 1 – Таблиця “User”

User
id: унікальний рядок (UUID), що є первинним ключем ідентифікації користувача.
email: унікальний рядок, що містить електронну адресу користувача (використовується для логіна).
password: рядок, що містить захешований пароль користувача для безпечної авторизації.
firstName: рядок, ім'я користувача (необов'язкове поле).
lastName: рядок, прізвище користувача (необов'язкове поле).
dob: поле дати/часу, дата народження користувача (необов'язкове поле).
avatarUrl: рядок, посилання на зображення профілю користувача.
isEmailVerified: булеве значення, яке вказує, чи підтвердив користувач свою електронну пошту (за замовчуванням false).
verifyToken: унікальний рядок, що містить токен для підтвердження email.
storageUsed: ціле число, яке зберігає об'єм використаного дискового простору користувачем у байтах (за замовчуванням 0).
createdAt: поле дати/часу, що фіксує момент створення облікового запису (за замовчуванням поточний час).

Таблиця 2 – Таблиця “Folder”

Folder
id: унікальний рядок (UUID), що є первинним ключем папки.
name: рядок, що містить назву папки.
userId: рядок (зовнішній ключ), ідентифікатор користувача, який є власником папки.
parentId: рядок (зовнішній ключ), ідентифікатор батьківської папки, в якій знаходиться дана папка (дозволяє створювати вкладені підпапки).
createdAt: поле дати/часу, момент створення папки.

Таблиця 3 – Таблиця “File”

File
id: унікальний рядок (UUID), первинний ключ файлу.
originalName: рядок, оригінальна назва файлу під час завантаження.
storageKey: унікальний рядок, унікальне ім'я файлу на фізичному носії або в хмарному сховищі.
mimeType: рядок, MIME-тип файлу (наприклад, image/png або application/pdf).
size: ціле число, розмір файлу в байтах.
iv: рядок, вектор ініціалізації, що використовується для криптографічного шифрування файлу.
isPublic: булеве значення, яке вказує, чи доступний файл за публічним посиланням (за замовчуванням false).
publicToken: унікальний рядок, токен доступу для публічного скачування файлу.
userId: рядок (зовнішній ключ), ідентифікатор власника файлу.
folderId: рядок (зовнішній ключ), ідентифікатор папки, в якій знаходиться файл.

Таблиця 5 – “SharedFile”

SharedFile
id: унікальний рядок (UUID), первинний ключ доступу.
fileId: рядок (зовнішній ключ), ідентифікатор файлу, до якого надається доступ.
userId: рядок (зовнішній ключ), ідентифікатор користувача, який отримує доступ.
role: рядок, рівень прав доступу (наприклад, VIEWER або EDITOR).

Таблиця 4 – Таблиця “Contact”

Contact
createdAt: поле дати/часу, момент завантаження файлу.
id: унікальний рядок (UUID), первинний ключ запису.
userId: рядок (зовнішній ключ), ідентифікатор користувача, який ініціює контакт.
contactId: рядок (зовнішній ключ), ідентифікатор користувача, якого додають у контакти.
status: рядок, поточний статус запиту.
createdAt: поле дати/часу, момент створення запиту.

3.3 Розробка серверної частини

Серверна частина застосунку реалізована у вигляді RESTful API на базі Node.js та фреймворку Express [8]. Архітектура бекенду побудована за принципом слабкої зв'язності, де кожен модуль відповідає за чітко визначену бізнес-логіку. Основні модулі включають автентифікацію, управління файловою системою, організацію доступу та систему кешування (Рис.5).

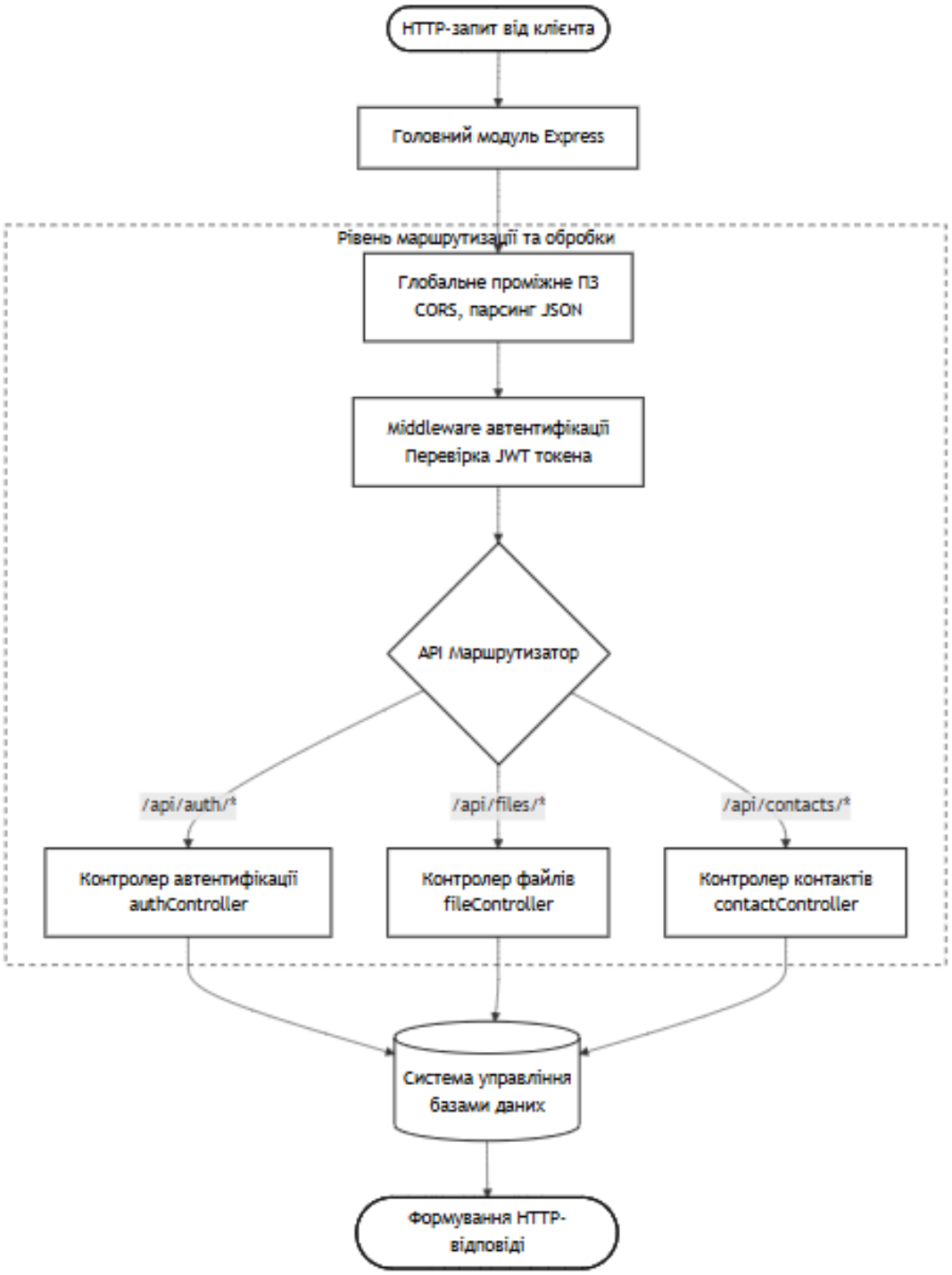


Рисунок 5 — *Архітектура серверної частини RESTful API та маршрутизація запитів*

3.3.1 Реалізація системи автентифікації та безпеки

За безпеку та керування сесіями користувачів відповідає модуль authController.js. Оскільки застосунок зберігає конфіденційні файли, було

реалізовано багаторівневу систему автентифікації.

При створенні нового акаунта відкритий пароль користувача ніколи не зберігається в базі даних. Замість цього використовується бібліотека bcrypt, яка застосовує алгоритм адаптивного криптографічного хешування з використанням солі. Навіть у разі компрометації бази даних, відновлення оригінального пароля методом перебору є математично нерентабельним.

Для захисту від спаму та фейкових акаунтів інтегровано модуль nodemailer [14]. При реєстрації генерується 6-значний код, який відправляється на пошту. Додатково впроваджено механізм checkCooldown, який блокує повторні спроби відправки листів частіше ніж раз на 60 секунд.

Після успішної авторизації сервер не створює класичну сесію, а генерує бездержавний JWT-токен, що містить ID користувача та час життя 24 години. Цей токен підписується секретним ключем JWT_SECRET і передається в HTTP-заголовку Authorization при кожному наступному запиті. Приклад реалізації функції авторизації наведено у лістингу 3.

Додатково реалізовано метод googleAuth, який за допомогою google-auth-library дозволяє користувачам безпечно входити в систему без створення пароля, використовуючи обліковий запис Google.

Лістинг 3 — Реалізація функції авторизації з перевіркою хешу пароля та генерацією JWT-токена у файлі authController.js

```
    const login = async (req, res) => {
        try {
            const { email, password } = req.body;

            const user = await prisma.user.findUnique({
where: { email } });
            if (!user) {
                return res.status(400).json({ message:
'Неправильний email або пароль' });
            }

            const validPassword = await
bcrypt.compare(password, user.password);
            if (!validPassword) {
                return res.status(400).json({ message:
```

```
'Неправильний email або пароль' });
            }


            const token = jwt.sign(
                { id: user.id, email: user.email },
                process.env.JWT_SECRET,
                { expiresIn: '24h' }
            );

            res.status(200).json({
                token,
                user: { id: user.id, email: user.email }
            });
        } catch (error) {
            console.error('Помилка логіну:', error);
            res.status(500).json({ message: 'Помилка
сервера' });
        }
    };
```

3.3.2 Управління файловою системою

Найскладнішою частиною системи є fileController.js та folderController.js, які забезпечують завантаження, зберігання, переміщення та видалення даних.

Безпека збереження фізичних файлів є пріоритетом. Файли не зберігаються на сервері у відкритому вигляді. Під час завантаження за допомогою middleware Multer [12] , файл перехоплюється у вигляді потоку читання і шифрується динамічно під час передачі алгоритмом AES-256-CBC за допомогою вбудованого модуля crypto. Для кожного файлу генерується унікальний вектор ініціалізації , що гарантує, що навіть два однакові файли матимуть різний зашифрований вигляд. Лише після успішного шифрування оригінальний тимчасовий файл асинхронно видаляється з диска за допомогою fs.promises.unlink (Рис. 6). Приклад реалізації функції шифрування наведено у лістингу 4.

Лістинг 4 — Процес шифрування файлу під час завантаження

```
    const uploadFile = async (req, res) => {
        try {
            if (!req.file) return res.status(400).json({
message: 'Файл не знайдено' });

            const originalName =
Buffer.from(req.file.originalname,
'latin1').toString('utf8');
            const userId = req.user.id;
            const fileSize = req.file.size;
            const user = await prisma.user.findUnique({
where: { id: userId } });

            const fsPromises = require('fs').promises;

            if (user.storageUsed + fileSize > MAX_STORAGE)
{
                await
fsPromises.unlink(req.file.path).catch(() => {});
                return res.status(400).json({ message:
'Перевищено ліміт сховища (1 ГБ)' });
            }

            const iv = crypto.randomBytes(16);
            const cipher = crypto.createCipheriv('aes-256-
cbc', Buffer.from(ENCRYPTION_KEY), iv);

            const plainTextPath = req.file.path;
            const encryptedFilename = req.file.filename +
'.enc';
            const encryptedPath = path.join(__dirname,
'../../uploads', encryptedFilename);

            const input =
fs.createReadStream(plainTextPath);
            const output =
fs.createWriteStream(encryptedPath);

            await new Promise((resolve, reject) => {
                input.pipe(cipher).pipe(output)
                    .on('finish', resolve)
                    .on('error', reject);
            });

            await
fsPromises.unlink(plainTextPath).catch(err =>
console.log('Помилка видалення тимчасового файлу:', err));
```

```
            const newFile = await prisma.file.create({
                data: {
                    originalName: originalName,
                    storageKey: encryptedFilename,
                    size: fileSize,
                    mimeType: req.file.mimetype,
                    userId: userId,
                    iv: iv.toString('hex'),
                    folderId: req.body.folderId &&
req.body.folderId !== 'null' ? req.body.folderId : null
                }
            });

            await prisma.user.update({
                where: { id: userId },
                data: { storageUsed: { increment: fileSize
} }
            });

            await prisma.user.update({
                where: { id: userId },
                data: { storageUsed: { increment: fileSize
} }
            });

            await
redisClient.del(`cache:/api/users/profile_${req.user.id}`);
            await
redisClient.del(`cache:/api/files_${req.user.id}`);

            await
redisClient.del(`cache:/api/files?folderId=null_${req.user.
id}`);
            if (req.body.folderId && req.body.folderId !==
'null') {
                await
redisClient.del(`cache:/api/files?folderId=${req.body.folde
rId}_${req.user.id}`);
            }

            res.status(201).json(newFile);
        } catch (error) {
            console.error('Помилка завантаження:', error);

            if (req.file && req.file.path) {

require('fs').promises.unlink(req.file.path).catch(() =>
{});
```

```
            }
            res.status(500).json({ message: 'Помилка
сервера' });
        }
    };
```

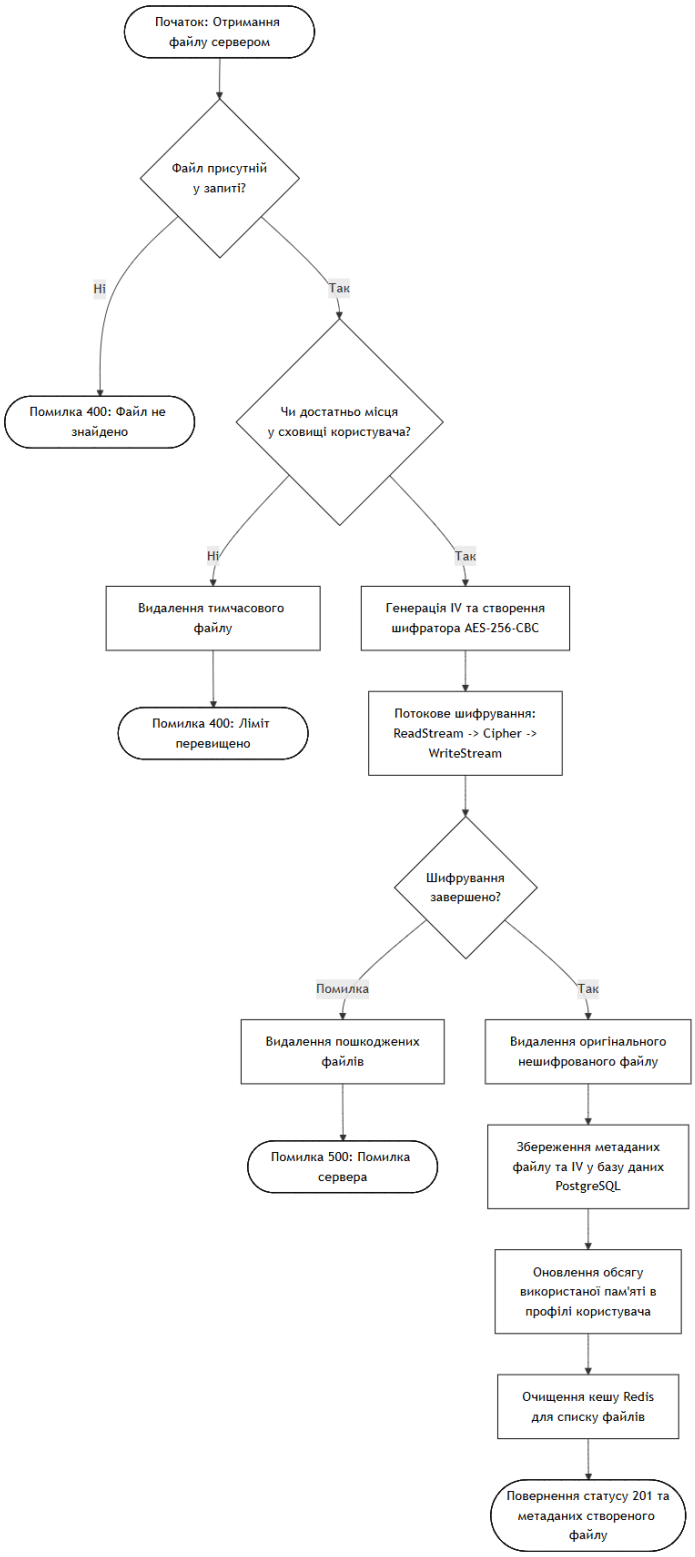


Рисунок 6 — *Блок-схема алгоритму завантаження, шифрування та збереження файлу*

Для організації ієрархії реалізовано метод moveFile, який просто оновлює зовнішній ключ folderId у базі даних. Видалення файлів також оптимізовано: спочатку асинхронно видаляється фізичний файл з диска, потім каскадно видаляються всі записи про права доступу до нього в БД, і наостанок перераховується доступна квота дискового простору storageUsed для власника.

3.3.3 Організація спільного доступу та управління контактами

Хмарне сховище дозволяє користувачам ділитися своїми файлами, що реалізовано через contactController.js та відповідні методи fileController.js. Передбачено два механізми доступу: публічне посилання та спільний доступ файлу.

Якщо файл потрібно показати будь-кому в Інтернеті, генерується унікальний криптографічний рядок publicToken. Логіку зміни статусу публічності та генерації токена наведено в лістингу 5.

Лістинг коду 5 — Генерація публічного токена доступу до файлу

```
const togglePublicLink = async (req, res) => {
        const fileId = req.params.id;
        const file = await prisma.file.findUnique({ where:
{ id: fileId } });

        // Перевірка прав автора або співавтора
        if (file.userId !== req.user.id) {
            const shared = await
prisma.sharedFile.findFirst({
                where: { fileId, userId: req.user.id, role:
'EDITOR' }
            });
            if (!shared) return res.status(403).json({
message: 'Немає прав' });
        }

        const isPublicNow = !file.isPublic;
        const publicToken = isPublicNow ?
crypto.randomBytes(16).toString('hex') : null;
```

```
        await prisma.file.update({
            where: { id: fileId },
            data: { isPublic: isPublicNow, publicToken:
publicToken }
        });

        res.status(200).json({ isPublic: isPublicNow,
publicToken });
    };
```

Якщо інший неавторизований користувач переходить за цим токеном downloadPublicFile, система розшифровує файл і віддає його. Це показано в лістингу 6.

Лістинг коду 6 — Розшифрування та завантаження файлу за публічним посиланням

```
    const downloadPublicFile = async (req, res) => {
        const { token } = req.params;
        const file = await prisma.file.findUnique({ where:
{ publicToken: token } });

        if (!file || !file.isPublic) {
            return res.status(404).json({ message:
'Посилання недійсне' });
        }

        const encryptedPath = path.join(__dirname,
'../../uploads', file.storageKey);
        const iv = Buffer.from(file.iv, 'hex');
        const decipher = crypto.createDecipheriv('aes-256-
cbc', Buffer.from(ENCRYPTION_KEY), iv);

        res.setHeader('Content-Disposition', `attachment;
filename="${encodeURIComponent(file.originalName)}"`);
        res.setHeader('Content-Type', file.mimeType);

        const input = fs.createReadStream(encryptedPath);
        input.pipe(decipher).pipe(res);
    };
```

Також користувач може надати доступ до файлу чи папки конкретній людині, вказавши її роль (VIEWER — лише читання, EDITOR — право на перейменування/видалення). При кожному запиті на завантаження сервер

здійснює складну багаторівневу перевірку доступу: чи є користувач власником файлу, чи має він доступ конкретно до цього файлу, або чи є він співавтором батьківської папки. Алгоритм перевірки наведено у лістингу 7.

Лістинг 7 — Надання індивідуального доступу користувачеві

```
let hasAccess = false;

if (file.userId === req.user.id) {
    hasAccess = true;
} else if (file.isPublic) {
    hasAccess = true;
} else {
    const sharedFile = await
prisma.sharedFile.findFirst({
        where: { fileId: file.id, userId: req.user.id }
    });

    if (sharedFile) {
        hasAccess = true;
    } else if (file.folderId) {
        const folder = await prisma.folder.findUnique({
where: { id: file.folderId } });

        if (folder && folder.userId === req.user.id) {
            hasAccess = true;
        } else {
            const sharedFolder = await
prisma.sharedFolder.findFirst({
                where: { folderId: file.folderId,
userId: req.user.id }
            });
            if (sharedFolder) hasAccess = true;
        }
    }
}

if (!hasAccess) return res.status(403).json({ message:
'Немає доступу до файлу' });
```

3.3.4 Оптимізація продуктивності за допомогою кешування

Оскільки перевірка прав доступу та побудова дерева папок вимагають складних SQL-запитів, у систему було інтегровано in-memory базу даних Redis.

Було застосовано патерн Cache-Aside. Наприклад, коли користувач запитує свій профіль або список файлів, сервер спочатку перевіряє наявність цих даних у Redis [3]. Якщо дані є — вони віддаються миттєво минаючи запит до PostgreSQL [6].

Для забезпечення актуальності даних реалізовано строгу систему інвалідації кешу. Будь-яка мутація, наприклад: створення файлу, переміщення в іншу папку, видалення викликає метод redisClient.del(). Приклад інвалідації Redis-кешу наведено в лістингу 8.

Лістинг 8 — Багаторівнева перевірка прав доступу при скачуванні файлу

```
            await
redisClient.del(`cache:/api/users/profile_${req.user.id}`);
            await
redisClient.del(`cache:/api/files_${req.user.id}`);
            await
redisClient.del(`cache:/api/files?folderId=null_${req.user.
id}`);
            if (req.body.folderId && req.body.folderId !==
'null') {
                await
redisClient.del(`cache:/api/files?folderId=${req.body.folde
rId}_${req.user.id}`);
            }
```

3.4 Розробка клієнтської частини

Клієнтська частина застосунку є єдиною точкою взаємодії користувача з системою хмарного сховища. Вона реалізована за концепцією Single Page Application, що гарантує відсутність перезавантажень сторінки під час переходів між директоріями та забезпечує досвід використання, наближений до нативних десктопних застосунків. Для побудови архітектури було обрано бібліотеку React.js (Рис. 7).

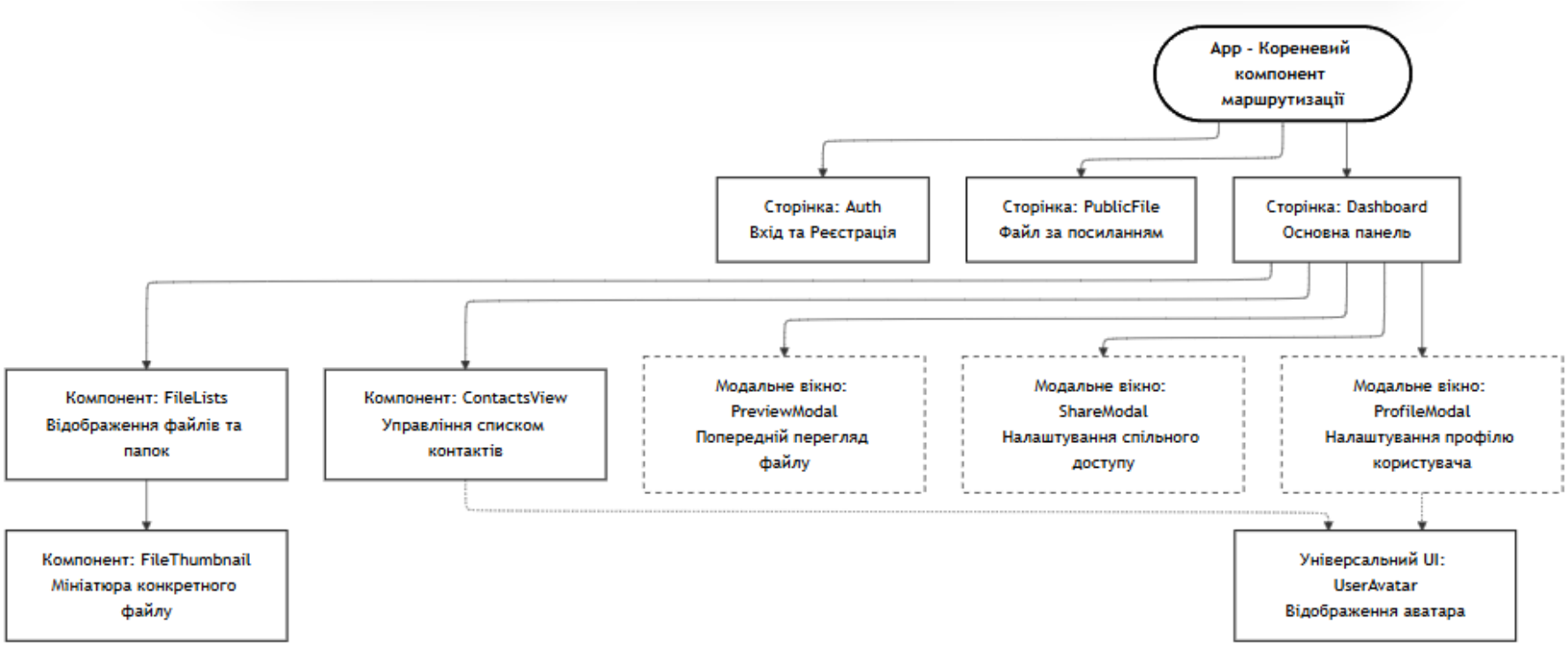


Рисунок 7 — *Архітектура компонентів клієнтської частини React-застосунку*

3.4.1 Проектування користувацького інтерфейсу

Інтерфейс застосунку спроєктований з акцентом на мінімалізм та інтуїтивну зрозумілість. Для візуального оформлення використано utility-first CSS фреймворк Tailwind CSS [10]. Це дозволило створити цілісну дизайн-систему, такі як стандартизовані кольори, відступи, типографіка без написання об'ємних CSS-файлів. Усі графічні піктограми, наприклад іконки папок, файлів, налаштувань реалізовані за допомогою оптимізованої SVG-бібліотеки Lucide React [17].

Навігація в застосунку реалізована за допомогою бібліотеки react-router-dom. Для захисту приватних сторінок від неавторизованого доступу було створено механізм приватних маршрутів. Якщо користувач не має валідного JWT-токена, він автоматично перенаправляється на сторінку авторизації. Приклад такого налаштування маршрутизації наведено в лістингу 9.

Лістинг 9 — Налаштування маршрутизації в App.jsx з використанням механізму PrivateRoute для захисту Dashboard

```
    import { BrowserRouter, Routes, Route, Navigate } from
'react-router-dom';
    import Auth from './pages/Auth';
    import Dashboard from './pages/Dashboard';
    import PublicFile from './pages/PublicFile';
    import VerifyEmail from './pages/VerifyEmail';

    const ProtectedRoute = ({ children }) => {
      const token = localStorage.getItem('token');

      if (!token) {
        return <Navigate to="/auth" replace />;
      }

      return children;
    };

    function App() {
      return (
        <BrowserRouter>
          <Routes>
```

```
            <Route path="/shared/:token"
element={<PublicFile />} />

            <Route path="/verify-email/:token"
element={<VerifyEmail />} />

            <Route path="/auth" element={<Auth />} />

            <Route
              path="/dashboard"
              element={
                <ProtectedRoute>
                  <Dashboard />
                </ProtectedRoute>
              }
            />

            <Route path="*" element={<Navigate
to="/dashboard" replace />} />
          </Routes>
        </BrowserRouter>
      );
    }

    export default App;
```

Взаємодія з бекендом винесена в окремий шар логіки за допомогою HTTP-клієнта Axios. Для Axios налаштовано глобальний перехоплювач, який автоматично додає JWT-токен у заголовок Authorization до кожного запиту, що робить код компонентів чистим і позбавляє необхідності вручну передавати токен при кожній дії.

3.4.2 Реалізація сторінок доступу

Сторінка авторизації Auth.jsx є точкою входу в систему. Вона поєднує в собі форми входу та реєстрації, перемикання між якими реалізовано через зміну локального стану компонента без зміни URL.

Форми мають клієнтську валідацію введених даних, зокрема перевірка довжини пароля, формату email.

На сторінці реалізовано кнопку швидкого входу через Google за допомогою бібліотеки @react-oauth/google. При натисканні генерується

OAuth-токен, який відправляється на бекенд для верифікації (Рис. 8).

Після заповнення всіх полів реєстрації у користувача з’являється форма для введення коду безпеки, який було надіслано на пошту, яку він вводив у полі email. Якщо пошта підтверджена, користувач автоматично допускається до панелі управління і відбувається створення акаунту.

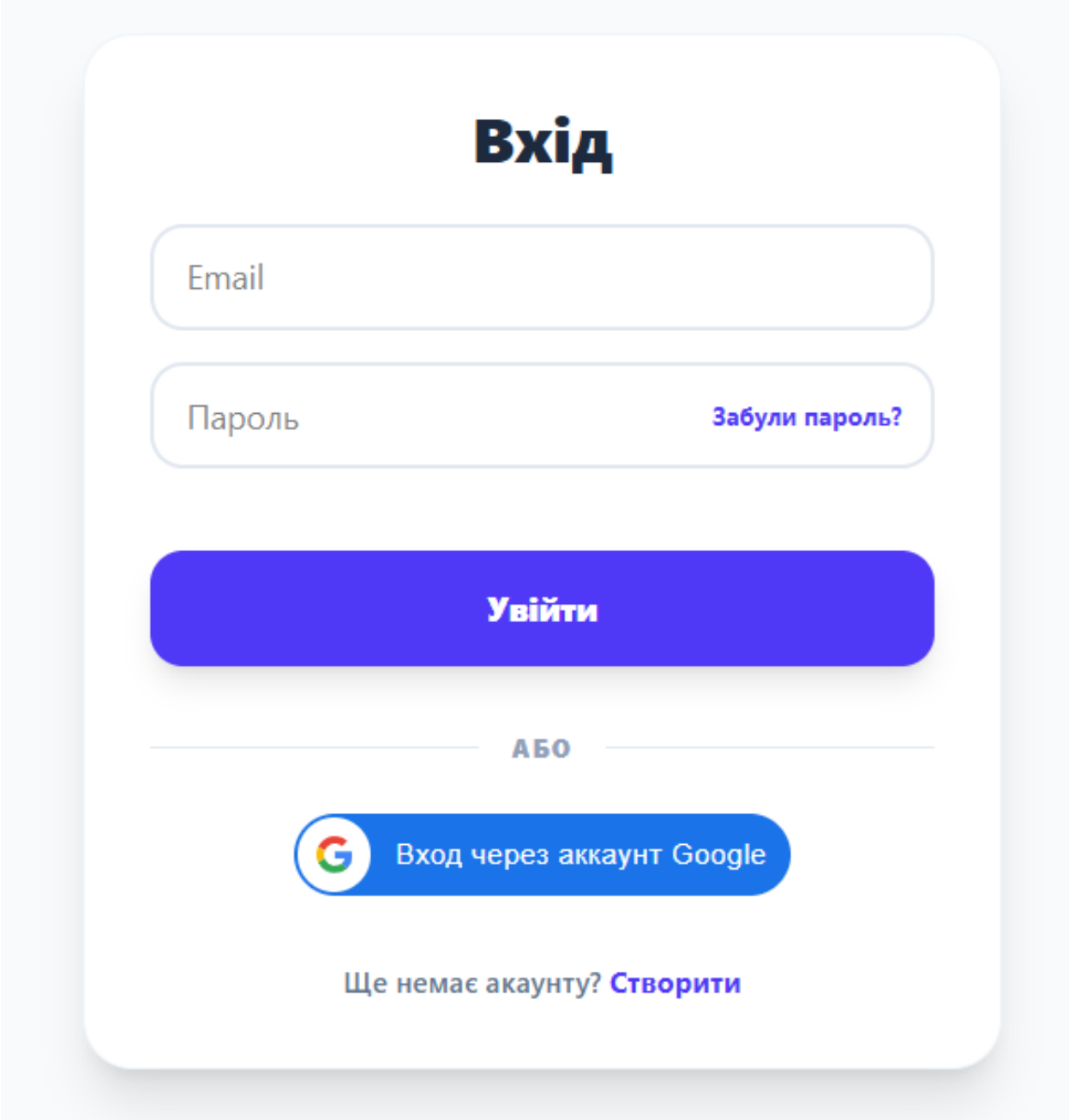


Рисунок 8 — *Інтерфейс сторінки авторизації*

3.4.3 Розробка головної панелі управління

Компонент Dashboard.jsx є ядром клієнтської частини. Саме тут реалізовано логіку файлового менеджера. Цей компонент має складний внутрішній стан: поточна папка, масив файлів, масив підпапок, стан завантаження та модальні вікна.

Ієрархія папок відображається у вигляді breadcrumbs, що дозволяє користувачеві легко повернутися на рівень вище або в кореневу директорію.

Для підвищення зручності роботи із системою реалізовано підтримку завантаження файлів за допомогою механізму Drag-and-Drop. Користувач може просто перетягнути необхідні файли з комп'ютера безпосередньо у вікно веб-застосунку, після чого вони автоматично передаються на сервер для подальшого збереження у вибраній папці. Такий підхід спрощує взаємодію з файловою системою та робить процес завантаження більш швидким і зрозумілим для користувача. Приклад реалізації такого хука наведено в лістингу 10.

Лістинг 10 — Реалізація хука налаштування області Drag-and-Drop для завантаження файлів у файловий менеджер

```
import { useDropzone } from 'react-dropzone';
import { useCallback } from 'react';

const onDrop = useCallback((acceptedFiles) => {
  setIsUploadModalOpen(false);

  acceptedFiles.forEach(async (file) => {
    const usedSpace = user?.storageUsed || 0;
    const availableSpace = MAX_STORAGE - usedSpace;

    if (file.size > availableSpace) {
      showToast(`Недостатньо місця для ${file.name}!`,
'error');
      return;
    }
  });
}, [currentFolder, fetchMyStorage, fetchBaseData,
user]);
```

```
    const { getRootProps, getInputProps, isDragActive } =
useDropzone({
      onDrop,
      noClick: true
    });

    return (
      <div
        className="min-h-screen bg-white w-full text-slate-
900 relative flex flex-col"
        {...getRootProps()}
      >
        <input {...getInputProps()} />
      </div>
    );
```

Для кожного файлу та папки реалізовано взаємодію через модальні вікна. Користувач може перейменувати файл, видалити його, поділитися ним через систему контактів, або згенерувати публічне посилання. Інтерфейс оновлюється динамічно завдяки використанню підходу Optimistic UI — це зміна відображається в браузері ще до того, як прийде підтвердження від сервера, що створює ілюзію миттєвої роботи застосунку. Інтерфейс програми зображено на Рис. 9

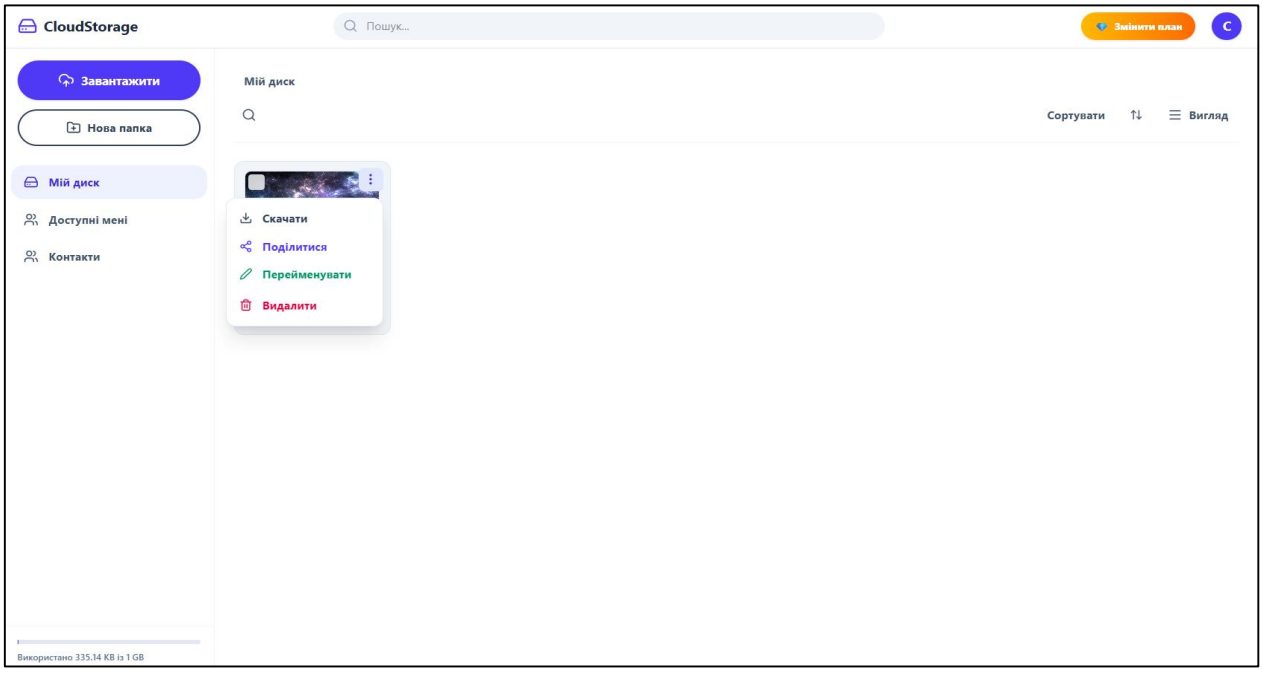


Рисунок 9 — *Інтерфейс головної панелі управління з відкритим меню дій для*

обраного файлу

3.4.4 Інтерфейс публічного доступу до файлів

Для забезпечення зручного обміну файлами з користувачами, які не мають облікового запису в системі, реалізовано окрему сторінку публічного доступу до файлів (Рис. 10). Доступ до такого файлу здійснюється за унікальним посиланням, яке містить спеціальний токен, що автоматично зчитується із URL-адреси під час відкриття сторінки.

При рендері компонента відправляється GET-запит до публічного API-ендпоінту бекенда для отримання метаданих у вигляді: назва файлу, розмір у мегабайтах, MIME-тип, автор. Завдяки цьому користувач бачить прев'ю того, що він збирається завантажити.

Збереження у своє сховище: Якщо неавторизований користувач вирішив увійти в систему (або вже авторизований), на сторінці стає активною кнопка "Зберегти на мій диск". Це викликає відповідний метод бекенду savePublicFile, який копіює зашифрований файл у внутрішнє сховище без необхідності його повторного скачування та завантаження.

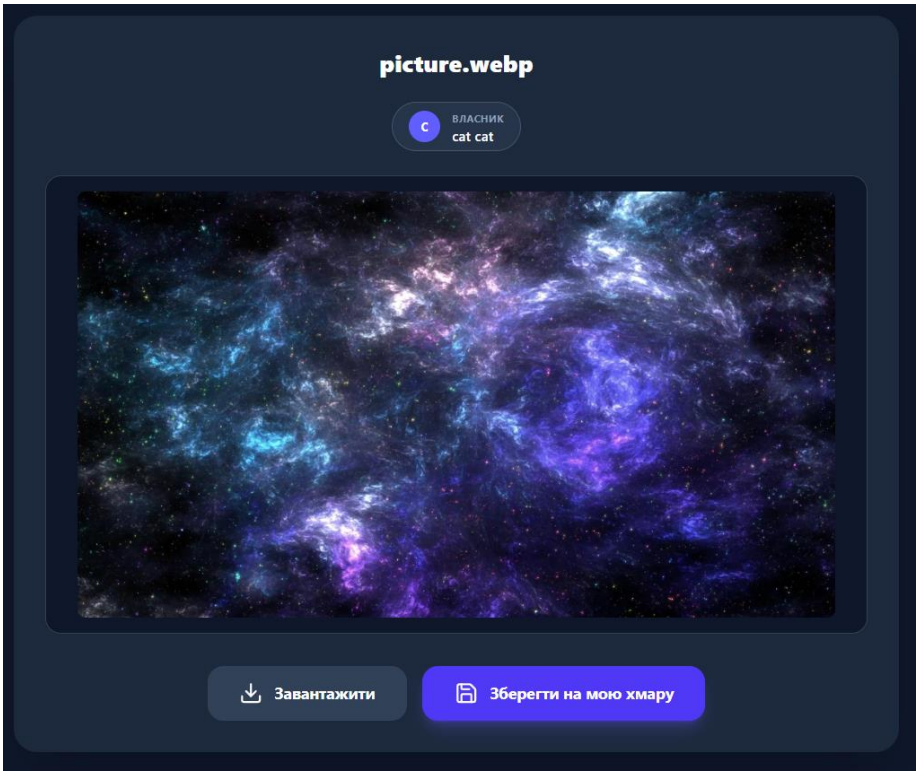


Рисунок 10 — *Інтерфейс сторінки файлу з публічним доступом*

3.5 Апробація

Апробація розробленого програмного забезпечення є важливим етапом, що дозволяє підтвердити працездатність системи, її відповідність висунутим вимогам та коректність взаємодії між компонентами. Оскільки розроблене хмарне сховище орієнтоване на безпечне зберігання та обмін даними кінцевих користувачів, основним інструментом апробації було обрано проведення комплексного інтеграційного тестування на основі реальних сценаріїв використання.

Для проведення повноцінної перевірки серверної частини, взаємодії з базою даних та алгоритмів розмежування доступу було спеціально розроблено набір автоматизованих тестів у середовищі Postman. Створена тестова колекція запитів дозволила повністю імітувати взаємодію двох користувачів та слугує практичним доказом працездатності системи. Цей автоматизований сценарій перевіряє повний життєвий цикл: реєстрацію нових акаунтів із підтвердженням email, авторизацію, завантаження файлів, а головне — успішне блокування спроб несанкціонованого доступу до чужих даних та коректне очищення даних після завершення тестів. Результати успішного проходження розробленого набору тестів наведено на Рис. 11.

На лістингу 11 представлено специфікацію API, описану у форматі JSON відповідно до стандарту OpenAPI



Рисунок 11 — *Результати автоматизованого інтеграційного тестування API*

Лістинг 11 — Специфікація API, описана у форматі JSON

```
{
  "openapi": "3.0.0",
  "info": {
    "title": "Tests",
    "version": "1.0.0",
    "description": "Повний цикл тестування безпеки
(реєстрація, верифікація, ізоляція доступу, очищення)."
  },
```

```
"servers": [
  {
    "url": "http://localhost:5000"
  }
],
"paths": {
  "/api/users/register": {
    "post": {
      "summary": "1.1 Реєстрація User A",
      "responses": {},
      "tags": [
        "1. Користувачі"
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "type": "object",
              "properties": {
                "email": {
                  "type": "string"
                },
                "password": {
                  "type": "string",
                  "format": "color"
                },
                "firstName": {
                  "type": "string"
                },
                "lastName": {
                  "type": "string"
                }
              }
            },
            "example": {
              "email": "",
              "password": "123456",
              "firstName": "User",
              "lastName": "A"
            }
          }
        }
      }
    }
  },
  "/api/users/verify-code": {
    "post": {
      "summary": "1.2 Підтвердження Email User A",
      "responses": {},
```

```
      "tags": [
        "1. Користувачі"
      ],
      "requestBody": {
        "content": {
          "application/json": {
            "schema": {
              "type": "object",
              "properties": {
                "email": {
                  "type": "string"
                },
                "code": {
                  "type": "string"
                }
              }
            },
            "example": {
              "email": "",
              "code": ""
            }
          }
        }
      }
    },
    "/api/users/login": {
      "post": {
        "summary": "1.3 Логін User A",
        "responses": {},
        "tags": [
          "1. Користувачі"
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "type": "object",
                "properties": {
                  "email": {
                    "type": "string"
                  },
                  "password": {
                    "type": "string",
                    "format": "color"
                  }
                }
              },
              "example": {
```

```
            "email": "",
            "password": "123456"
          }
        }
      }
    }
  },
  "/api/files/upload": {
    "post": {
      "summary": "2.1 Завантаження файлу (User A)",
      "responses": {},
      "tags": [
        "2. Файли"
      ],
      "security": [
        {
          "BearerAuth": []
        }
      ]
    }
  },
  "/api/files/{fileIdA}/download": {
    "get": {
      "summary": "2.2 Читання свого файлу (User A)",
      "responses": {},
      "tags": [
        "2. Файли"
      ],
      "security": [
        {
          "BearerAuth": []
        }
      ]
    },
    "parameters": [
      {
        "name": "fileIdA",
        "in": "path",
        "required": true,
        "schema": {
          "type": "string"
        },
        "example": ""
      }
    ]
  },
  "/api/files/{fileIdA}/share/public": {
    "post": {
```

```
            "summary": "2.4 Надання доступу за посиланням
(User A)",
            "responses": {},
            "tags": [
              "2. Файли"
            ],
            "security": [
              {
                "BearerAuth": []
              }
            ]
          },
          "parameters": [
            {
              "name": "fileIdA",
              "in": "path",
              "required": true,
              "schema": {
                "type": "string"
              },
              "example": ""
            }
          ]
        },
        "/api/files/{fileIdA}": {
          "delete": {
            "summary": "3.1 Спроба видалення чужого файлу
(User B)",
            "responses": {},
            "tags": [
              "3. Очищення даних"
            ],
            "security": [
              {
                "BearerAuth": []
              }
            ]
          },
          "parameters": [
            {
              "name": "fileIdA",
              "in": "path",
              "required": true,
              "schema": {
                "type": "string"
              },
              "example": ""
            }
          ]
```

```
    },
    "/api/users/profile": {
      "delete": {
        "summary": "3.3 Видалення акаунта (User A)",
        "responses": {},
        "tags": [
          "3. Очищення даних"
        ],
        "security": [
          {
            "BearerAuth": []
          }
        ],
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "type": "object",
                "properties": {
                  "confirmationWord": {
                    "type": "string"
                  }
                }
              },
              "example": {
                "confirmationWord": "ВИДАЛИТИ"
              }
            }
          }
        }
      }
    },
    "components": {
      "securitySchemes": {
        "BearerAuth": {
          "type": "http",
          "scheme": "bearer"
        }
      }
    }
  }
```


Для перевірки клієнтської частини системи було проведено апробацію графічного інтерфейсу, побудованого на базі бібліотеки React. На Рис. 12 зображено головне вікно розробленого додатка після успішного входу

користувача в систему. Інтерфейс коректно відображає ієрархію папок, список завантажених файлів та статистику використання дискового простору, динамічно отримані з бекенду.

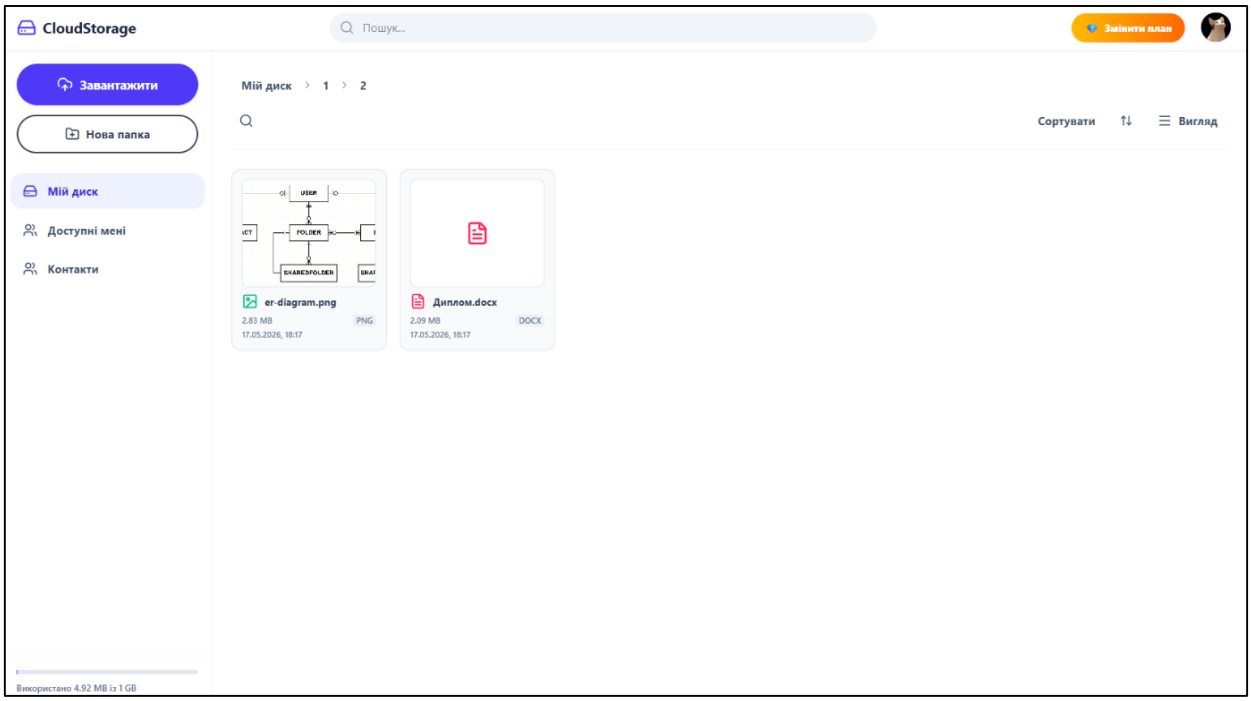


Рисунок 12 — *Головна сторінка після успішного входу і завантаження файлів*

Одним із ключових сценаріїв роботи є безпечне завантаження користувацьких файлів. Для перевірки цього функціоналу користувач взаємодіє з відповідним інтерфейсом завантаження. Система здійснює валідацію обраного файлу та перевірку наявності вільного місця у сховищі на стороні клієнта перед відправкою на сервер. Процес взаємодії користувача з цією формою продемонстровано на Рис. 13.

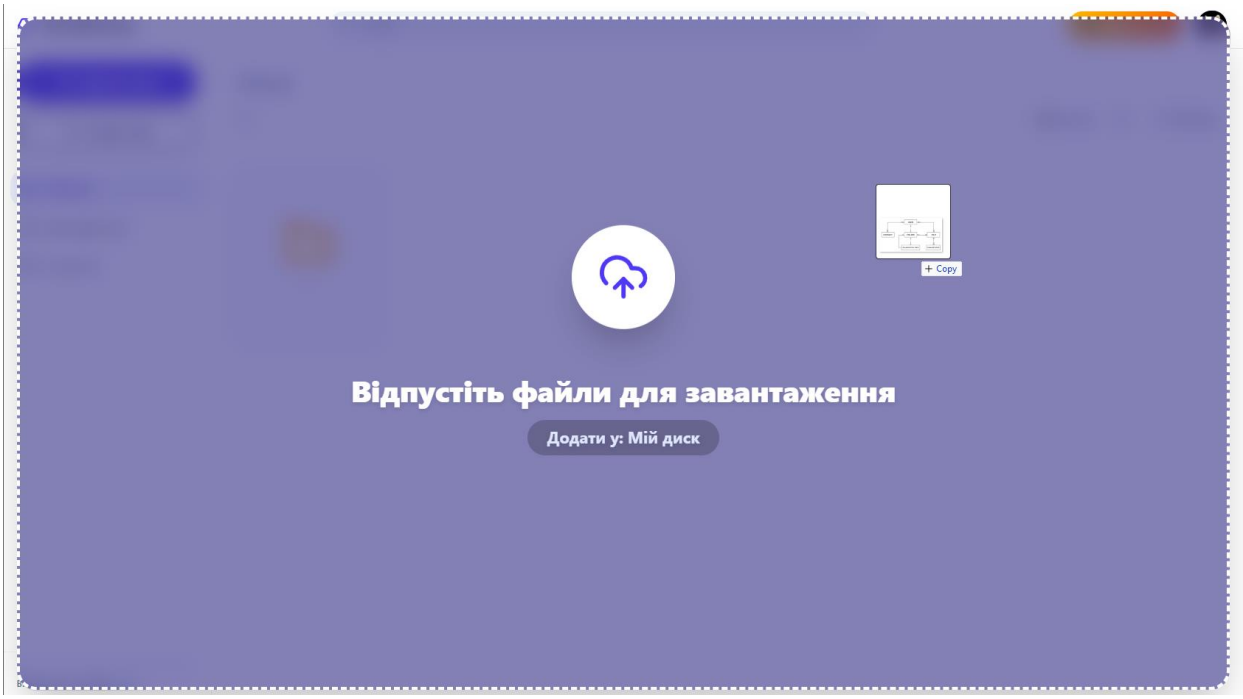


Рисунок 13 — *Інтерфейс завантаження нового файлу в систему*

Після підтвердження дії система успішно обробляє інформацію, відправляє файл на сервер, де відбувається його шифрування та оновлення стану інтерфейсу без перезавантаження сторінки. Під час виконання цієї операції також було перевірено вкладку "Network" у панелі розробника браузера для підтвердження відсутності помилок на рівні мережевих запитів та перевірки статус-коду 201 Created.

Крім того, під час апробації була перевірена адаптивність веб-інтерфейсу, реалізована за допомогою фреймворку Tailwind CSS. Додаток було протестовано на різних роздільних здатностях (мобільний телефон, планшет, десктоп). Елементи управління, навігаційне меню та сітка файлів коректно масштабуються та змінюють своє розташування, що забезпечує зручність використання системи на будь-яких пристроях.

3.6 Висновки до розділу 3

1. Обґрунтовано вибір технологічного стеку для розробки високонавантаженого хмарного сховища. Для бекенду обрано середовище

Node.js із фреймворком Express, для збереження даних — PostgreSQL із Prisma ORM та кешуванням Redis, а для клієнтської частини — бібліотеку React та збирач Vite.

2. Спроєктовано реляційну структуру бази даних, що складається з шести взаємопов'язаних таблиць User, File, Folder, Contact, SharedFile. Вона забезпечує надійне збереження ієрархії файлової системи та підтримує цілісність даних завдяки каскадному видаленню.
3. Розроблено серверну частину у вигляді RESTful API. Імплементовано багаторівневу систему автентифікації з використанням JWT-токенів і Google OAuth 2.0, а також створено механізм потокового криптографічного шифрування файлів алгоритмом AES-256-CBC безпосередньо під час їх завантаження на сервер.
4. Створено клієнтську частину. Інтерфейс включає систему захищених маршрутів, підтримку Drag-and-Drop для файлів, динамічне оновлення стану сторінки та адаптивний дизайн на базі утилітарних класів Tailwind CSS.
5. Проведено апробацію розробленого програмного забезпечення.

ВИСНОВКИ

1. Досліджено ринок існуючих хмарних сховищ, виявлено їхні вразливості щодо приватності даних, зокрема зберігання ключів шифрування на серверах корпорацій, та обґрунтовано необхідність розробки незалежного власного рішення.
2. Розроблено архітектуру системи, що включає клієнтську частину, серверний REST API та рівень збереження даних.
3. Створено реляційну структуру бази даних PostgreSQL для підтримки складної ієрархії файлової системи за допомогою Prisma ORM. Для прискорення обробки запитів та зменшення навантаження на основну базу додано систему кешування Redis.
4. Розроблено механізм шифрування файлів за стандартом AES-256-CBC, який відбувається під час завантаження. Впроваджено систему авторизації на базі JWT-токенів, хешування паролів Bcrypt та інтеграцію з Google OAuth 2.0.
5. Розроблено клієнтську частину за архітектурою Single Page Application з використанням бібліотеки React та стилізації Tailwind CSS. Для підвищення зручності взаємодії з системою реалізовано підтримку Drag-and-Drop, фонове завантаження файлів та динамічне оновлення інтерфейсу без необхідності перезавантаження сторінки.
6. Реалізовано гнучку систему керування правами доступу, що дозволяє користувачам генерувати публічні посилання або надавати доступ до файлів конкретним особам зі свого списку контактів із визначенням ролей.
7. Виконано контейнеризацію інфраструктури проєкту засобами Docker, що забезпечило портативність і простоту розгортання системи. Працездатність застосунку підтверджено за результатами інтеграційного тестування в середовищі Postman та його практичної апробації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cantelon M., Harter M., Holowaychuk T. Node.js in Action. 2nd ed. Manning Publications, 2017. 432 p.
2. Офіційна документація Prisma ORM. URL: https://www.prisma.io/docs (дата звернення 04.06.2026).
3. Офіційна документація Redis. URL: https://redis.io/docs (дата звернення 04.06.2026).
4. Офіційна документація React. URL: https://react.dev/learn (дата звернення 04.06.2026).
5. Офіційна документація Node.js. URL: https://nodejs.org/docs/latest/api/ (дата звернення 04.06.2026).
6. Офіційна документація PostgreSQL. URL: https://www.postgresql.org/docs/ (дата звернення 04.06.2026).
7. Офіційна документація JSON Web Token (JWT). URL: https://www.jwt.io/introduction#what-is-json-web-token-structure (дата звернення 04.06.2026).
8. Офіційна документація Express.js. URL: https://expressjs.com/en/ (дата звернення 04.06.2026).
9. Офіційна документація Vite. URL: https://vite.dev/guide/ (дата звернення 04.06.2026).
10. Офіційна документація Tailwind CSS. URL: https://tailwindcss.com/docs/installation/using-vite (дата звернення 04.06.2026).
11. Офіційна документація Docker. URL: https://docs.docker.com/guides/ (дата звернення 04.06.2026).
12. Multer. Github. URL: https://github.com/expressjs/multer/blob/main/README.md (дата звернення 04.06.2026).
13. Офіційна документація Google OAuth 2.0. URL: https://developers.google.com/identity/protocols/oauth2 (дата звернення

04.06.2026).

14. Офіційна документація Nodemailer. URL: https://nodemailer.com/ (дата звернення 04.06.2026).

15. Офіційна документація Axios. URL: https://axios.rest/pages/getting-started/first-steps.html (дата звернення 04.06.2026).

16. Офіційна документація React Dropzone. URL: https://react-dropzone.js.org/ (дата звернення 04.06.2026).

17. Офіційна документація Lucide. URL: https://lucide.dev/guide/ (дата звернення 04.06.2026).

18. Офіційна документація React Avatar Editor. URL: https://github.com/mosch/react-avatar-editor/blob/main/README.md (дата звернення 04.06.2026).

19. Пащенко М. Розробка веб-застосунку для зберігання та керування файлами. *Збірник наукових праць студентів, здобувачів PhD, докторантів і молодих вчених, університетська конференція «МОЛОДА НАУКА-2026» Том V «Секції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету»* : м. Запоріжжя, 20–24 квітня 2026 р. Запоріжжя, 2026. С. 112–113.

Декларація
академічної доброчесності
здобувача ступеня вищої освіти ЗНУ

Я, Пащенко Максим Володимирович, студент 4 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальність 121 Інженерія програмного забезпечення освітньої програми «Програмне забезпечення систем», адреса електронної пошти makspm22@gmail.com, — підтверджую, що написана мною кваліфікаційна робота на тему «Розробка веб-застосунку для зберігання та керуванням файлами» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 12.06.2026 _____ Пащенко Максим Володимирович
(студент)

Дата 12.06.2026 _____ Скрипник Ірина Анатоліївна
(керівник)