МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри інженерії
програмного забезпечення

_____ Євген ДАВИДЕНКО

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА

Веборієнтована платформа хмарного зберігання, управління та розповсюдження даних

Спеціальність 121 Інженерія програмного забезпечення
Освітня програма «Інженерія програмного забезпечення»

Здобувач _____ **Роман КАФТАН**

«__» _____ 20__ р.

Керівник роботи
ст. викладачка

_____ **Марина ФАЛЕНКОВА**

«__» _____ 20__ р.

Миколаїв – 2026

Завдання на виконання кваліфікаційної роботи

Чорноморський національний університет імені Петра Могили

Факультет	Комп'ютерних наук
Кафедра	Інженерії програмного забезпечення
Рівень вищої освіти	Перший (бакалаврський)
Освітній ступінь	Бакалавр
Спеціальність	121 Інженерія програмного забезпечення
Освітня програма	Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри інженерії
програмного забезпечення
_____ Євген ДАВИДЕНКО
«___» _____ 2026 р.

ЗАВДАННЯ

на кваліфікаційну бакалаврську роботу здобувача

Кафтана Романа

1. Тема кваліфікаційної роботи: «Веборієнтована платформа хмарного зберігання, управління та розповсюдження даних» затверджена наказом ректора ЧНУ ім. Петра Могили № _349__ від «_26__» __грудня__ 2025 р.
2. Строк представлення кваліфікаційної роботи: «__» червня 2026 р.
3. Очікуваний результат роботи та початкові дані:

– очікуваним результатом є готовий до застосування вебзастосунок, вихідний код проєкту та результати тестування використовуючи різні методики;

– початковими даними є необхідність забезпечити високий рівень безпеки та конфіденційності для користувачів, аналіз готових рішень.

4. Перелік питань, що підлягають розробці:

– аналіз готових рішень-аналогів для хмарного сховища;

– обгрунтування вибору технологій та фундаментального шаблону «чиста архітектура»;

– створення специфікацій вимог та візуального моделювання з UML;

– розробка класів для API для керування файловою системою та бази даних;

– інтеграція механізмів безпеки для акаунту користувача та його файлів;

– проведення стрес, молульного та на відмовостійкість тестування.

5. Перелік графічних матеріалів: презентація.

6. Консультанти:

Консультант	**Кафедра (організація)**	**Частина роботи**

Дата видачі завдання «____» _____ 20___ р.

КАЛЕНДАРНИЙ ПЛАН

виконання кваліфікаційної роботи

Тема: Веборієнтована платформа хмарного зберігання, управління та розповсюдження даних

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КБР	19.01.2026	23.01.2026	Виконано
2.	Огляд літератури за темою роботи	26.01.2026	06.02.2026	Виконано
3.	Складання календарного плану	09.02.2026	11.02.2026	Виконано
4.	Аналіз предметної області	12.02.2026	06.03.2026	Виконано
5.	Розробка проєктних рішень	09.03.2026	20.03.2026	Виконано
6.	Моделювання та конструювання ПЗ	23.03.2026	03.04.2026	Виконано
7.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування, розробка керівництва користувача	06.04.2026	05.05.2026	Виконано
8.	Відгук керівника КБР	05.05.2026	07.05.2026	Виконано
9.	Оформлення КБР та презентації	08.05.2026	24.05.2026	Виконано
10.	Попередній захист	25.05.2026	25.05.2026	Виконано
11.	Рецензування	15.06.2026	17.06.2026	Виконано
12.	Завершення оформлення презентації	18.06.2026	19.06.2026	Виконано
13.	Захист кваліфікаційної роботи			

Здобувач _____ **Роман КАФТАН**

«__» _____ 20__ р.

Керівник роботи

ст. викладачка _____ **Марина ФАЛЕНКОВА**

«__» _____ 20__ р.

АНОТАЦІЯ

до кваліфікаційної бакалаврської роботи

«Веборієнтована платформа хмарного зберігання, управління та розповсюдження даних»

Здобувач 408 гр.: Кафтан Роман

Керівник: ст. викл. Марина Фаленкова

Актуальність теми: В сучасному технологічно розвиненому світі, який еволюціонує з величезною швидкістю майже кожен день, людство прийшло до того, що зберігання персональних даних виключно на фізичних сховищах являється архаїчним. На заміну цьому з’явились зручні вебсервіси по типу Google drive, Microsoft OneDrive, тощо. Але зазвичай такі застосунки аналізують файли користувачів і використовують їх для свої цілей. Паралельно з ними розвивались і інші технології схожого типу. NextCloud – це представник хмарного сховища, яке можна розгорнути у себе вдома, використовуючи свої фізичні накопичувачі, хоча в цьому і є його мінус, адже таке рішення потребує технічних, адміністративних та більш глибоких знань у сфері інженерії комп'ютерних систем, аніж є у пересічного користувача. Актуальність проєкту полягає в тому, щоб мінімізувати мінуси конкурентів і розробити сервіс, який не буде використовувати персональні дані користувачів для потреб розвитку сервісу, буде мати відкритий код і не потребуватиме складних налаштувань.

Науково-практичним значенням є результатом проблем, які були описані в актуальності, тобто розробленні та застосуванні сучасних рішень для створення безпечного та доступного продукту. З наукової точки зору була задача дослідити вплив і переваги архітектурного підходу «Clean Architecture», дослідженою Робертом Мартіном. Це дозволяє зробити програму більш стійкою та з можливістю легкої масштабованості. З практичного ж огляду це дозволяє розробити повністю гнучку та масштабовану платформу для збереження даних, розгорнуту завдяки Docker та Docker Compose за лічені хвилини. Також вебзастосунок вирішить проблеми закритості, адже матиме відкритий код.

Мета роботи: розробка веборієнтованої платформи хмарного зберігання файлових ресурсів для забезпечення ефективного управління, безпечного збереження та швидкого розповсюдження інформації.

Об’єкт дослідження: процес зберігання, управління та розповсюдження цифрових даних у вебсередовищі.

Предмет дослідження: технології та програмні засоби створення веборієнтованої платформи для хмарного зберігання даних.

Робота складається з чотирьох розділів. У першому розділі проведено аналіз предметної області та існуючих рішень, другий присвячений специфікації вимог, аналізу статей категорії Б та математичну частину проєкту, в третьому розроблено за текстовою специфікацією, візуальне моделювання з стандартом UML, а у четвертому розділі описано методи, написання коду, тестування системи за різними методиками.

КБР викладена на 76 сторінки, вона містить 4 розділи, 19 ілюстрацій, 7 таблиці, 20 джерел в переліку посилань.

Ключові слова: хмарне зберігання, ASP.NET Core, React, безпека даних, JWT, AES-256, Docker, управління файлами, персональне сховище.

ASTRACT

of the Bachelor’s Thesis

" Web-oriented platform of cloud storage, management and distribution of data "

Student of group 408: Kaftan Roman

Supervisor: up. lect. Falenkova Marina

Relevance of the Topic: In today's technologically advanced world, which evolves at a tremendous speed almost every day, humanity has come to the point that storing personal data exclusively on physical storage is archaic. Instead, convenient web services such as Google drive, Microsoft OneDrive, etc. have appeared. But usually such applications analyze user files and use them for their own purposes. In parallel with them, other technologies of a similar type have developed. NextCloud is a representative of cloud storage that can be deployed at home, using your own physical drives, although this is its disadvantage, because such a solution requires technical, administrative and more in-depth knowledge in the field of computer systems engineering than the average user has. The relevance of the project is to minimize the disadvantages of competitors and develop a service that will not use users' personal data for the needs of service development, will have open source and will not require complex settings.

The scientific and practical significance is the result of the problems that were described in the relevance, that is, the development and application of modern solutions to create a secure and accessible product. From a scientific point of view, the task was to investigate the impact and benefits of the architectural approach "Clean Architecture", researched by Robert Martin. This allows you to make the program more stable and with the possibility of easy scalability. From a practical point of view, this allows you to develop a fully flexible and scalable platform for data storage, deployed thanks to Docker and Docker Compose in a matter of minutes. Also, the web application will solve the problems of confidentiality, because it will have open source.

Goal of the Work: To develop a web-oriented cloud storage platform for personal data and file resources to ensure efficient management, secure storage, and rapid information distribution.

Object of Research: The process of storing, managing, and distributing digital data in a web environment.

Subject of Research: Technologies, and software tools for creating a web-oriented cloud data storage platform.

The work consists of four sections. The first section analyzes the subject area and existing solutions, the second is devoted to the specification of requirements, analysis of category B articles and the mathematical part of the project, the third is developed according to the text specification, visual modeling with the UML standard, and the fourth section describes methods, code writing, and system testing using various techniques.

The bachelor's thesis is presented on 76 pages, it contains 4 chapters, 19 illustrations, 7 tables, 20 sources in the reference list.

Keywords: cloud storage, ASP.NET Core, React, data security, JWT, AES-256, Docker, file management, personal storage.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

AES – Advanced Encryption Standard

API – Application Programming Interface

ASP.NET – Active Server Pages для .NET

HTTP – HyperText Transfer Protocol

JWT – JSON Web Token

UML – Unified Modeling Language

ВСТУП

В сучасному технологічно розвиненому світі, який еволюціонує з величезною швидкістю майже кожен день, людство прийшло до того, що зберігання персональних даних виключно на фізичних сховищах являється архаїчним. На заміну цьому з’явились зручні вебсервіси по типу Google Drive, Microsoft OneDrive, тощо. Але зазвичай такі застосунки аналізують файли користувачів і використовують їх для свої цілей. Паралельно з ними розвивались і інші технології схожого типу. NextCloud – це представник хмарного сховища, яке можна розгорнути у себе вдома, використовуючи свої фізичні накопичувачі, хоча в цьому і є його мінус, адже таке рішення потребує технічних, адміністративних та більш глибоких знань у сфері інженерії комп'ютерних систем, аніж є у пересічного користувача. Актуальність проєкту полягає в тому, щоб мінімізувати мінуси конкурентів і розробити сервіс, який не буде використовувати персональні дані користувачів для потреб розвитку сервісу, буде мати відкритий код і не потребуватиме складних налаштувань.

Науково-практичним значенням є результатом проблем, які були описані в актуальності, тобто розробленні та застосуванні сучасних рішень для створення безпечного та доступного продукту. З наукової точки зору була задача дослідити вплив і переваги архітектурного підходу «Clean Architecture», дослідженою Робертом Мартіном. Це дозволяє зробити програму більш стійкою та з можливістю легкої масштабованості. З практичного ж огляду це дозволяє розробити повністю гнучку та масштабовану платформу для збереження даних, розгорнуту завдяки Docker та Docker Compose за лічені хвилини. Також вебзастосунок вирішить проблеми закритості, адже матиме відкритий код.

Мета роботи: розробка веборієнтованої платформи хмарного зберігання файлових ресурсів для забезпечення ефективного управління, безпечного збереження та швидкого розповсюдження інформації.

Для досягнення визначеної мети необхідно вирішити такі завдання:

1. проаналізувати предметну область та існуючі хмарні рішення-аналоги для виділення їхніх переваг та недоліків;
2. спроєктувати архітектуру системи та структуру бази;
3. розробити сервіси для завантаження, зберігання та керування правами доступу до файлів;
4. реалізувати безпекові механізми для захисту облікових записів та шифрування даних;
5. провести функціональне та інтеграційне тестування, виконати емпіричне дослідження стійкості програмного забезпечення шляхом моделювання реальних умов експлуатації.

Об’єкт дослідження: процес зберігання, управління та розповсюдження цифрових даних у веб-середовищі.

Предмет дослідження: технології та програмні засоби створення веборієнтованої платформи для хмарного зберігання даних.

Обґрунтування необхідності нової розробки. Аналіз надав розуміння, що найголовніша зараз ідея в вебзастосунках хмарного збереження, керування та розповсюдження даних є принцип «приватність понад усе». Такі гіганти як Google та Microsoft використовують шифрування даних з ключами, які лежать на їх серверах, що надає дуже потужний рівень захисту, але разом з цим не мають відкритого коду. Саме через це користувачі не можуть бути на сто відсотків впевненні, що їх дані не використовуються третіми особами. З іншого боку NextCloud має відкритий код, але складний у налаштуванні і для роботи потребує фізичного накопичувача. Саме тому розробляється EasyDisk, який поєднує плюси конкурентів і позбавляється їх найбільших мінусів. Це планується реалізувати завдяки відкритості коду, та виділеним серверам-сховищам.

1 АНАЛІТИЧНА ЧАСТИНА. ОБҐРУНТУВАННЯ ПЛАНУ ВИКОНАННЯ ЗАВДАННЯ

1.1 Аналіз предметної області та сучасних систем хмарного зберігання даних

В сучасному світі швидкого технологічного розвитку, поява хмарних сховищ стала наступним етапом. Історично перебрано сотні варіантів збереження даних. Від великогабаритних комп'ютерів, які могли зберігати не більше декількох кілобайтів, що вважалось на той момент справжнім технічним досягненням, що можна на них зберігати, до сьогоднішніх жорстких дисків, де тільки на одному екземплярі може міститися десятки терабайт записів. Проте навіть таких обсягів не може бути достатнім для тієї кількості інформації яка споживається в сучасних реаліях. Розміри таких даних вимірюються вже давно не в гігабайтах чи навіть терабайтах, а в зетабайтах. На рисунку 1.1 можна побачити цю динаміку зростання світових обсягів цифрових даних за версією Statista[1].

Рисунок 1.1 – Графік динаміки зростання світових обсягів цифрових даних

Саме тому, з часом стало зрозуміло, що зберігати інформацію виключно фізично, складно і не таке зручно, оскільки в разі помилок, це призводить до

часткової або навіть повної втрати даних. Через це користувачі почали зберігати резервні копії не тільки на одному накопичувачі, а одразу на декількох. Однак такий підхід прив’язує до одного комп’ютера, або декількох пристроїв одночасно, що для деяких є неприйнятним, бо у них відсутня можливість мати з собою додатковий ноутбук, чи переносний жорсткий диск, чи флеш-накопичувач. Усвідомивши це, великі корпорації, такі як Google, Amazon, Microsoft та Apple в різні проміжки часу вирішили створити сервіси хмарного зберігання [2, 3], до яких кожен зможе миттєво підключатися з будь якої точки планети, не прив’язуючись до свого місцезнаходження. Таким чином з’явились вебзастосунки для хмарного управління, перегляду та розповсюдження даних.

З огляду на глобальні тенденції, об’єктом даного дослідження виступає безпосередньо процес зберігання, управління та розповсюдження цифрових даних у вебзастосунку, а для ефективної реалізації цього проєкту на практиці потрібний сильний інструментарій, тому предметом дослідження є методи, технології та програмні засоби створення веборієнтованої платформи для хмарного зберігання даних.

Для розуміння поточного стану ринку та обґрунтуванню необхідності створення нової платформи, проведено аналіз програм конкурентів, серед яких домінантними є провідні корпорації, представником яких є Google з їх популярним хмарним сховищем Google Drive, яке побудоване на основі архітектури з використання мікросервісів, одразу на декількох мовах програмування: Java, Go, Python та C++. Оскільки компанія має одну з найбільших екосистем в світі, в цей віртуальний диск було інтегровано більшість з рішень для кращого користувацького досвіду, таких як: Documents для роботи з документами будь якого типу, від звичайних текстових, до з розширенням .docx, Presentation для створення, редагування та показ презентацій, Sheets для роботи з різноформатними таблиця, штучний інтелект Gemini та багато іншого. Таку саму стратегію використовує компанія Microsoft і тому має сховище OneDrive в якому також інтегровані схожі рішення, також з ряду офісного програмного забезпечення.

Однак, описанні програмні рішення, мають ряд вагомих недоліків. Більшість рішень від корпорацій мають суворо закритий код, що в свою чергу не дозволяє перевірити, чи не використовуються користувацькі дані третіми особами для неправомірних цілей. «Конфіденційність понад усе» втрачається після такої інформації, і залишається виключно покладатися на заяви цих компаній. Це стосується обох хмарних сховищ, але до цього ще додається мінус OneDrive в його малій безкоштовній квота у всього 5 гігабайтів, що в сучасних умовах, в епоху, коли тільки одна лиш програма може перевищувати обсяг в 10 гігабайтів, є критичним.

Альтернативою великим корпораціям виступають платформи, які позбавлені недоліку «закритого кода», найвідомішою з яких можна вважати NextCloud [4]. Ця клієнт-серверна система написана переважно на PHP та JavaScript і є оптимальним рішенням для користувачів, що прагнуть захистити свої приватні дані, але все ще хочуть уникнути брати з собою важкі і не зручні фізичні накопичувачі. Цей інструмент не має централізованих дата центрів в широкому розумінні, бо для того щоб почати користуватися системою потрібно спочатку її розгорнути на власних серверах користувача, що в свою чергу гарантує йому повний контроль на інформацією та впевненість, що третя особа не може використовувати їх. Однак знову ж, недолік полягає в тому, що можливість самостійного розгортання є не тільки великим плюсом, але ще й мінусом для людей які ніколи не стикались з системним адмініструванням. Високий поріг входження для налаштування вебсервера, бази даних і так далі, вимагає зусиль, і необхідність вивчати специфічні знання, які можуть більше і ні коли не знадобитися користувачеві, а призводить до втрати часу.

Після того як було зроблено аналіз з вище написаними аналогами – створено детальний порівняльний аналіз розглянутих платформ за ключовими критеріями зведено до таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз систем-аналогів хмарного зберігання даних

Характеристика	**Google Drive [2]**	**OneDrive [3]**	**Nextcloud [4]**
Розробник	Google LLC	Microsoft Corporation	Nextcloud GmbH
Архітектура	3-tier Web Application (Microservices)	3-tier Web Application (на базі сервісів Microsoft Azure)	Client-Server (Self-hosted)
Мова реалізації	Java, Go, Python, C++, JS	C#, C++, TypeScript, JavaScript	PHP, JavaScript
Перелік функцій	1. колаборація з Docs, Sheets та Presentation; 2. OCR-пошук; 3. мобільна синхронізація; 4. офлайн-доступ; 5. інтеграція з екосистемою.	1. глибока інтеграція з Microsoft 365; 2. файли за запитом; 3. захищена папка; 4. автоматичне резервне копіювання; 5. Відновлення попередніх версій.	1. наскрізне шифрування; 2. вбудований чат; 3. календар та контакти; 4. керування плагінами; 5. повна приватність даних.

Кінець таблиці 1.1

Переваги	Велика екосистема, 15 ГБ безкоштовно, надійність.	Ідеальна сумісність з Windows/Office, додатковий захист «Personal Vault»	Відкритий код, контроль над даними, відсутність підписки.
Недоліки	Закритий код, ризики приватності з боку розробника , адже проводиться збір метаданих.	Закритий код, збір метаданих корпорацією, сильна прив'язка до платформи Windows, малий обсяг безкоштовно (5 ГБ).	Необхідність мати власний сервер та навички в адмініструванні.

Після проведення аналізу вебзастосунків конкурентів, можна зробити висновок, що сучасний ринок програмного забезпечення пропонує різні рішення для збереження даних. З одного боку є великі екосистеми з багатим функціоналом створені корпораціями для зручності, швидкості та адаптовані для різних користувачів, однак не можуть гарантувати повну конфіденційність через закритий код та збір метаданих, а з іншого боку безпечні і майже повністю відкриті системи, які зберігаються на локальних накопичувачах, мінус яких це ускладнене використання, зумовлена їх архітектурою з необхідністю мати навички адміністрування. Саме тому формуються дуже специфічні проблеми, що потрібно вирішити шляхом розробки нового програмного забезпечення, яке буде поєднувати простоту використання з високим рівнем безпеки користувача і його даних.

1.2 Виявлення недоліків існуючих рішень та обґрунтування необхідності розробки платформи EasyDisk

Піч час написання першого підрозділу з аналізом предметної області та сучасних систем хмарного зберігання даних, виявлено, що ринок характеризується наявністю компромісів, і не існує ідеального вебзастосунка, який задовольняв би вимоги щодо безпеки, зручності та конфіденційності. Користувачеві доводиться погоджуватися на компроміси, що ставить його перед складним вибором. Головним недоліком найпопулярніших платформ від корпорацій Google та Microsoft є те, що вони мають закритий код, що в свою чергу створює недовіру до цих продуктів, адже неодноразово фіксувалися випадки, як ці компанії зберігають метадані, аналізують користувацькі файли і запити в браузерах для того, щоб оптимізувати внутрішні сервіси, або гірше, ґрунтуючись на своїх зборах вставляють нав’язливу, таргетовану рекламу на сайтах та своїх браузерах. Але реклама виступає не єдиним критичним мінусом, в разі того, якщо розглянути сутність терміну «мета дані». Мета дані [5] – інформація про створення елемента, назви, теми, функції тощо, тобто простими словами це «дані про дані». Метадані є ключовими для функціональності систем, що зберігають контент, дозволяючи користувачам знаходити цікаві елементи, записувати важливу інформацію про них та ділитися цією інформацією з іншими. Метадані і їх збір є стандартною технічною процедурою до моменту, поки власники систем не починають ними зловживати. Відомий американський криптограф Брюс Шнаєр зазначав в своїх роботах, що «Метадані розповідають про вас більше, ніж самі дані» [6], що показує критичність такої інформації, бо таким чином можна сформувати точно психологічний та соціальний бік людини. А знедавна, якщо точніше з 2023 року Google змінив свою політику конфіденційності, де прямо вказала, що з цього часу буде використовувати публічні та користувацькі дані для навчання своїх ШІ-моделей, хоча і заявляється, що приватні файли з Drive не будуть братися для цих цілей без прямого дозволу, але саме розуміння того, що перевірити це ніяким чесним

способом не є можливим створює ризики прихованого використання інформації. Саме тому відсутність доступу для всіх до вихідного коду цих систем унеможливлює аудит безпеки повністю, через що знижується рівень довіри з боку користувачів, які критично ставляться до питань цілісності та конфіденційності своїх персональних даних.

На відміну від великих корпорацій, NextCloud позиціонується як альтернатива, яка закриває питання безпеки конфіденційності повністю, тому що вихідний код відкритий і кожен може його модифікувати під себе, щоб переконатися особисто, що дані будуть в цілісності, бо між користувачем та його даними не буде третьої особи, яка буде їх переглядати, аналізувати, тощо. Це досягається завдяки тому, що використовуються не бази даних і дата центри, а свої накопичувачі чи орендовані VPS-сервери, хоча останнє вже менш прийнятне для тих, хто піклується за свою безпеку даних, з причин, зазначених в першій частині розділу про великі компанії та їх вебзастосунки. Тому залишається тільки використовувати свої жорсткі чи твердо тільні диски. Проте, якби цей процес був настільки простим, як здається на перший погляд, то ідеальна платформа була б вже створена і нею був би NextCloud. Однак, для того, щоб почати використовувати своє домашнє хмарне сховище, потрібно мати як мінімум базові знання з системного програмування та адміністрування. Технічною проблемою є те, що для розгортання системи потрібно: вміти працювати з терміналом Linux подібних систем, таких як Ubuntu, Debian тощо; розгорнути базу даних; налаштувати PHP та відкрити порти мережі, і після цього прив’язати домен. Для людини поза сферою інформаційних технологій – це майже неможливо. Окрім цього, з’являється розуміння, що «безкоштовний» застосунок, таким не є, через витрати на вище вказане. Додатковим недоліком є те, що окрім того, треба цю систему налаштувати, під час цього користувач може припуститися непоправних помилок, що призведе до критичних вразливостей. Open Worldwide Application Secutiry Project [7] – головний світовий стандарт безпеки вебдодатків, в рейтингу якого постійно фігурує «небезпечне налаштування», тобто це залишені відкритими порти бази

даних, використання слабких паролів, відсутність HTTPS або ж не правильні права доступу до директорій у Linux. Саме через це звичайному користувачеві буде дуже складно перейти на таку систему без спеціалізованих знань, бо витоки даних через конфігурацію становить більшу загрозу за теоретичне викрадання даних корпораціями в їх хмарних сховищах.

З огляду на вище написані недоліки існуючих рішень, стає очевидним обгунтування необхідності розробки повністю нового вебзастосунка для хмарного зберігання даних під кодовою назвою «EasyDisk Cloud». Головна концепція платформи в тому, щоб уникнути мінусів конкурентів і створити зручний для користування продукт для користувачів, які хочуть отримати поєднання безпеки даних та їх конфіденційної інформації завдяки відкритості коду, і також усунути необхідність вивчення базового розуміння системного програмування та адміністрування. Платформа «EasyDisk Cloud» проєктується як рішення, яке надає: високий рівень захисту, щоб гарантувати користувачу, що його дані будуть залишатися в безпеці, і ніхто інший окрім нього або без його згоди не отримає їх; зручний, сучасний та інтуїтивно зрозумілий інтерфейс, завдячуючи тому, що буде інтегровано вже звичні для всіх стандарти взаємодії, такі як: Drag-and-drop для завантаження або перенесення інформації, швидкий пошук за назвою, датою завантаження та розміром файлів, швидкістю виконання запитів завдяки потужним серверам, які будуть працювати 24 години на добу, 7 днів на тиждень; версійності файлів, що буде гарантувати для користувача 100 відсоткову резервну копію його даних, з можливістю в будь який зручний момент відновлення. Таким чином, це дозволить користувачеві отримати найкомфортніший досвід взаємодії з даними зі всіх можливих готових варіантів на ринку, не турбуючись про налаштування складної серверної інфраструктури, водночас маючи реальну гарантію того, що його файли і конфіденційна інформація не буде в якийсь день отримана третьою особою і використана проти нього самого.

Для того, щоб реалізувати таку ідею, треба вирішити технічні обмеження, які будуть на шляху від створення бази даних, до розгортання готового рішення з

доступом в мережі інтернет, а також забезпечити заявлений рівень приватності та безпеки даних. Саме тому спочатку потрібно вирішити проблему з лімітом на завантаження великих файлів в HTTP/HTTPS протоколах. Сучасні вебсервери, як Kestrel в ASP.NET Core чи Nginx мають такі жорсткі межі на розмір тіла запиту у 5 гігабайт на один суцільний POST-запит, це робиться для того, щоб захиститися від атак типу Denial-Of-Service, або більш знайомий для всіх DoS. Якщо спробувати відправити великий файл всього одним запитом, сервер або відхилить його видавши помилку «413 Payload Too Large», або вичерпає весь обсяг оперативної пам’яті в спробах буферизувати цей файл. Щоб вирішити цю проблему, буде застосовано ефективний алгоритм «чанкінгу» даних. Чанкінг – це процес розбиття великого обсягу інформації на менші та логічні блоки, щоб при завершенні завантаження отримати цілісний файл, що дає в свою чергу можливість обходити обмеження протоколів HTTP/HTTPS [8]. Чанки відправляються на сервер по черзі, і якщо наприклад на 21-му чанку зникне з’єднання з сервером і потім ж відновиться, система не стане завантажувати наступний, а просто зачекає і повторно відправить двадцять перший, а на стороні серверної платформи ці файли будуть додаватися до результуючого файлу і наприкінці зберігатися на диску. Також завдяки цьому в користувацькому інтерфейсі вийде реалізувати точний індикатор прогресу з можливістю паузи чи відміни завантаження та повністю нівелює ліміт на розмір тіла запиту. Паралельно з цим потрібно забезпечити завантажені файли високим рівнем безпеки, що буде реалізовано завдяки шифруванню AES-256. Advanced Encryption Standard – це сучасний стандарт шифрування секретним ключем, який був ухвалений урядом США [9]. Симетричне шифрування означає, що для зашифрування та розшифрування файлу використовується один і той самий криптографічний ключ, довжина якого 256 біт, або рівно 32 символи, що в свою чергу гарантує, що для підбору методом ручного, або машинного перебору всіх варіантів навіть найсучаснішим суперкомп’ютером знадобиться мільйони років. Перевагою серверного шифрування є те, що це знімає обчислювальне навантаження на пристрій користувача, особливо в разі

використання з смартфонів, але все одно гарантує високий рівень захисту, навіть у разі крадіжки жорстких дисків із дата-центрів, бо без ключа, який знаходитиметься в безпечному місці, зловмисник отримає лише набір випадковий байтів.

1.3 Формування стратегії та плану виконання завдань з розробки веборієнтованої платформи

Будь-який процес створення сучасного програмного забезпечення починається з вибору Software Development Life Cycle, тобто з життєвого циклу розробки. Враховуючи архітектурну складність, яка дуже тісно буде взаємодіяти між клієнтською частиною та серверною, разом з інтеграцією криптографічних алгоритмів для збереження даних та приватної інформації, чанкінгу файлів при завантаженні, великої кількості сервісів і розмір проєкту загалом та подальшу контейнеризацію, було вирішено, що використання класичної каскадної моделі «Waterfall» не вважається доречною. Waterfall передбачає жорсткий перехід від спочатку повного проєктування, потім до повного написання коду і так до повного тестування, що є недоцільним і неефективним, адже така складна вебсистема, де використовуватиметься Stripe для керування оплатою і стягнення передплат, докером і стрімінгом дуже ризиковано. Припустимо, що на етапі тестування виникає проблема з Cross-Origin Resource Sharing, тобто спільним використанням ресурсів з різних джерел, або з Kestrel, через що доведеться частково, або навіть повністю перепроєктувати всю архітектуру, що займе занадто велику кількість часу і ресурсів. Саме тому обрано іншу стратегію розробки – ітеративно-інкрементну методологію. Суть цієї методології у тому, що проєкт розбиватиметься на міні-цикли, де кожна ітерація завершуватиметься створення робочої частини системи, що в свою чергу дозволятиме тестувати систему на будь-якому з етапів.

Першим кроком після обрання методології є формування специфікації вимог до програмного забезпечення, оскільки на даному етапі відбувається поділ як на функціональні, тобто логіку застосунка до яких відноситься бізнес процеси

завантаження, видалення, керування даних і генерації посилань на них, так і на нефункціональні, які не є виконуваним, але визначають правила і вимоги до продуктивності, безпеки алгоритмів та відмовостійкості. Для того, щоб ці вимоги перетворились з сухого тексту в візуальну технічну модель, використовуватиметься уніфікована мова моделювання, або як частіше зазначають Unified Modeling Language [10]. UML створена для побудови діаграм прецедентів, діаграм класів та послідовностей, що в свою чергу дозволяє описувати життєвий цикл об’єктів які розглядаються для виявлення логічної межі ще на етапі проєктування.

Після першого кроку наступним логічним етапом буде проєктування інформаційної моделі та бази даних, оскільки файлова система хмарного сховища являє собою складну деревоподібну структуру, де директорії можуть містити інші директорії та файли, а піддиректорії можуть мати таку ж саму структуру. Щоб отримати на виході платформу з високою надійністю та швидкістю виконання запитів обрано Microsoft SQL Server, тому що ця система керування базою даних ідеально дотримується принципів атомарності, узгодженості, ізольованості та довговічності, а це в свою чергу критично важливо для вебзастосунку такого типу.

Одразу після розробки структури даних стратегією передбачається перехід до серверної частини, і для того щоб забезпечити високу продуктивність та модульність створення проєкту обрано ASP.NET Core 10.0 [11] разом з шаблоном «чиста архітектура» [12], яка була розглянута Робертом Мартіним. Цей шаблон робить класи всередині проєкта та код майже повністю незалежними один від одного, таким чином, що одна частина може взагалі не знати про іншу. Завдяки цьому в разі масштабування програму не доведеться переписувати декілька разів, а лише потрібно буде додати нову частину до вже написаної без порушення цілісності. На стороні сервера реалізовуватиметься ключовий функціонал, такий як шифрування даних, модулі авторизації через JWT-токени, керування рольовими моделями доступу, а також управління файлами.

І останніми частинами стратегії є frontend, який буде написаний з використанням React та після цього фінальним етапом весь проєкт буде запаковано в Docker контейнери [13]. React обраний для створення користувацького інтерфейсу, через те, що це дозволить реалізувати односторінковий вебзастосунок найбільш ефективно ніж в існуючих варіантів бібліотек та фреймворків, а також дозволить зробити слабку зв'язаність з сервером. Одразу після того, як весь застосунок буде протестовано, все разом і база даних, і серверна і користувацька платформи будуть контейнеризовані завдяки Docker Compose у контейнери, що вирішить класичну проблему в програмування, коли локально проєкт працює добре, а на сервері з'являються дефекти, а також це дозволить розгорнути «EasyDisk Cloud» на будь- якому апаратному забезпеченні за лічені хвилини.

Висновки до розділу 1

В ході написання першого розділу про аналітику та обґрунтування плану виконання завдання, ознайомлено зі станом ринку. Оглянуто популярні програмні рішення, де з одного боку, комерційні гіганти, які пропонують доступні тарифи на передплати, великий і зручний функціонал, однак мають недоліки в вигляді закритості систем і ризиком втручання третіх осіб в користувацькі дані. З іншого боку, повністю безпечні системи з відкритим кодом, але для початку користування потрібні базові навички адміністрування.

На основі виявлених недоліків конкурентів обґрунтовано необхідність розробки повністю нової веборієнтованої платформи хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud». Визначено, що для того, щоб усунути зазначені недоліки потрібно розробити застосунок відкритим, захищеним та зручним в користуванні. Щоб успішно реалізувати поставлені завдання було сформовано стратегію проєктування взявши за основу ітеративно-інкрементну методологію, а також визначено які інструменти потрібно для цього використовувати.

2 МОДЕЛЮВАННЯ ТА СПЕЦИФІКАЦІЯ ВИМОГ ДО ВЕБОРІЄНТОВАНОЇ ПЛАТФОРМИ ХМАРНОГО ЗБЕРІГАННЯ

2.1 Аналіз сучасних моделей побудови захищених інформаційних систем

В сучасних умовах цифрова трансформація зростає з надзвичайною швидкістю та цим же створює багато нових викликів для захисту користувацької конфіденційної інформації у вебзастосунках, особливо в таких як хмарні сховища, тому, щоб спроєктувати надійну систему потрібно вирішити фундаментальні проблеми безпеки, тобто одночасного забезпечення захисту приватних даних та цілісності інформації без критичних втрат продуктивності. Тому, щоб повноцінно змоделювати захищену, але і цілісну систему було обрано для аналізу публікацію [14], центральною темою якої є саме це питання. Автори наголошують, що «конфіденційність означає обмеження доступу до даних, а анонімність – неможливість ідентифікації суб’єктів обміну» [14]. Саме тому платформа хмарного зберігання «EasyDisk Cloud» проєктується з фокусом на гарантуванні конфіденційності користувацьких даних.

Також згідно з цією ж статтею [14], стає зрозумілим, що існуючі архітектури обміну приватною або чутливою інформацією поділяються на дві категорії: централізовано та децентралізовано. Централізовані системи – це системи, в яких процеси збереження, обробки та керування даними зосереджені на одному потужному сервері, а користувачі взаємодіють з ним не маючи власних обчислювальних ресурсів. Ключовими функціями на таких серверах є: автентифікація користувача в системі шляхом перевірки його токена, наприклад з використанням JWT; зберігання даних на своїх великих сховищах; контроль доступу через перевірки ролей, чи заходить адміністратор, чи звичайний передплатник, чи взагалі гість. Також дослідники зазначають, що незаперечним

плюсом централізації є те, що такі системи мають високу пропускну здатність мережі, що звісно ж є критичним для передачі великих обсягів даних, проте їхнім суттєвим недоліком є вразливість центрального вузла та ризик несанкціонованого доступу з боку адміністраторів системи до користувацьких метаданих [14]. В той час децентралізація використовує інший підхід. Децентралізація – це архітектура, де обробка, зберігання та керування даними розподілені одразу між всіма, або більшістю учасниками мережі, а не сконцентрована в одному центральному сервері. Однак, хоч і рівень анонімності стає дуже високим, такі системи є мало підходящими для хмарних сховищ, і як наголошують дослідники через те що, додається складність інтеграції таких рішень та через обмежену продуктивність. Для кращого розуміння аналізу в таблиці 2.1 наведено порівняння таких систем між собою, яка представлена в статті [14].

Таблиця 2.1 – Порівняння архітектур обміну чутливою інформацією [14]

Критерій	Централізована архітектура	Децентралізована архітектура
Прозорість	Обмежена, залежить від адміністратора	Висока, базується на реєстрах або відкритому аудиті
Централізація	Єдина точка контролю та зберігання	Розподілена між вузлами
Гнучкість доступу	Рольова модель, жорсткі правила	Динамічний доступ, політики за атрибутами
Відповідність нормативам	Простий аудит, труднощі з реалізацією мінімізації та згоди	Privacy-by-design, складність формалізації підзвітності
Стійкість до загроз	Вразливість до атак на центр і адміністраторів	Вища стійкість, складність синхронізації й виявлення загроз

Аналіз даної таблиці дозволяє зробити висновок, що у кожної з категорій є плюси і мінус, і тому при проєктуванні треба зважувати всі компроміси при виборі. Деякі з категорій вже було розглянуто раніше, тому їх пропустимо і зупинимось на деяких з них для ширшого аналізу. Як зазначалося вище, централізовані системи більш вразливі до атак, по тій причині, що всі процеси сконцентровані в одному, головному центральному сервері, і потрібно визнати цю проблему, оскільки для платформи «EasyDisk Cloud» обрано саме цю архітектуру. Хоча це і є великим мінусом, проте специфіка хмарного сховища вимагає високої швидкості в обробці файлів для зручності клієнтів, що в децентралізованих мережах призвело б до критичних затримок. Саме тому і було прийнято стратегічне рішення використовувати централізовану архітектуру, ризик компенсувати на рівні шифрування сервера, і навіть в разі успішної атаки зловмисник не зможе отримати доступ до даних користувачів. Також це нівелює проблему можливого втручання адміністратора в конфіденціальні дані клієнтів, бо як і у випадку несанкціонованого доступу, без ключа шифрування та розшифрування отримані файли будуть лиш набором без зв’язних байт.

Іншою важливою ознакою, визначеною в сучасному дослідженні, є «Тимчасова чутливість» інформації, яка розглядається в таблиці «Класифікаційні ознаки чутливої інформації в міжорганізаційному середовищі» [14]. Відповідно до статті можна визначити, що «тимчасовість – це залежність від етапу життєвого циклу, що формує політики доступу в часі та динамічного управління правами», а «чутлива інформація – це дані, розповсюдження яких може зашкодити правам, інтересам чи безпеці суб’єктів» [14]. Оскільки веборієнтована платформа хмарного зберігання, керування та розповсюдження даних «EasyDisk Cloud», навіть у назві відображає необхідність надання клієнтам можливості ділитися своїми файлами з третіми особами, тому вимогу про «тимчасову чутливість» покладено в основу модуля. Архітектура системи передбачає генерацію захищених публічних URL-посилань з підтримкою «Time-Ti-Live», бо саме цей механізм визначає максимальний, дозволений час життя, або кількості переходів, під час яких дані,

що передаються існують. Наприклад при спробі поділитися файлом користувач може встановити пароль чи таймер «самознищення» на 24 години, або більше. Протягом цього часу посилання працюватиме і всі кому буде надано доступ зможуть переглядати або завантажувати ці дані. Таким чином, платформа буде забезпечена управлінням доступу, яке автоматично заборонятиме право на отримання інформація для третіх осіб в разі завершення життєвого циклу посилання, чи не вірно введеного паролю.

Проте конфіденційність для користувацьких даних при розробці платформи хмарного сховища, це лише одна з складових для успішності виконання, тому окрім цього є інший критичний аспект у вигляді оптимізації зберігання даних та забезпечення високої швидкості системи під час сильного навантаження. Саме тому було обрано другу статтю під назвою «модель інтелектуальної системи управління колекціями в хмарних середовищах» [15]. Згідно з цим дослідженням, для того, щоб ефективно управляти ресурсами, потрібна чітка структура розділення. Автори наголошують, що «у хмарному середовищі зберігається не лише множина об’єктів, але й метаінформація, така як: часові мітки, права доступу…» [15], цим дослідники обгрунтовують доцільність проведення експерименту над досліджуваними моделями. Після експерименту було доведено високу ефективність архітектури, що розділяє реляційну систему управління базою даних для мета даних та окреме сховище, для фізичних користувацьких файлів. Саме з цієї причини для системи хмарного зберігання даних «EasyDisk Cloud», обрано архітектуру яка проєктується за аналогічним принципом, тобто ієрархічні зв’язки, такі як: структура папок, які будуть показуватись користувачеві як фізичні, в інтерфейсі однак будуть знаходитись лише в базі даних, права доступу та час створення, розташовуватимуться в Microsoft SQL Server, тоді як фізичні, зашифровані файли розміщуватимуться в ізольованому віртуальному сховищі Windows Subsystem for Linux [16] завдяки Docker Volumes.

Окрім вищезазначеної структури зберігання, обов’язково має виконуватися ще одна вимога у вигляді швидкодії. Тому дослідники, провівши, що підтвердив

про необхідність правильної структури зберігання даних, отримали результати, які можна вважати еталонними. Взявши вибірку з 50 тисяч об’єктів система відповідала на запит менше ніж за 0.2 секунди, що доводить практичну цінність такої структури для реальних хмарних сховищ [15]. Окремо було проведено ще один експеримент, щоб зрозуміти різницю між ефективною архітектурою та неоптимізованою, де вже при 10 тисячах об’єктів середній час пошуку перевищив всі прийнятні та не тільки ліміти швидкості відповіді у 3 секунди [15]. Таким чином, автори доводять, що відсутність оптимізації призводить до занадто великих затримок, що для користувацького застосунку такого масштабу є неприпустимим. І для того, щоб вебзастосунок хмарного сховища «EasyDisk Cloud» функціонував стабільно і міг відповідати еталонним критеріям продуктивності, у серверному ядрі використано оптимізовану структуру, яку наводили автори статті, та реалізовано асинхронну модель обробки запитів. Більше того, для обходу лімітів протоколів HTTP/HTTPS та уникненню блокування потоків сервера під час завантаження великих файлів, використовується алгоритм «чанкінгу», а це гарантує, система залишатиметься зручною для використання навіть при паралельному завантаженні гігабайтних архівів.

В результаті аналізу фахових публікацій [14, 15], можна стверджувати, що обрана стратегія розробки «EasyDisk Cloud» є практично обгрунтованої так і науково, та, що розробка платформи хмарного зберігання вимагає комплексного підходу, де з одного боку, система повинна мати захист для конфіденційної інформації користувача [14], що реалізується завдяки шифруванню AES-256 та токенізацію з використанням JWT. З іншого боку, архітектура повинна забезпечувати високу швидкодію, з часом відповіді менше ніж 0.2 секунди на запит, мати можливість легко масштабуватися в залежності від нових реалій, та бути оптимізованою, щоб майже не навантажуючи систему, зберігати метадані і користувацькі файли, завдяки алгоритму «чанкінгу», та розділення завдань для бази даних і сервісу асинхронного завантаження даних на фізичний накопичувач.

Саме тому таке наукове обгрунтування дозволяє перейти до опису обраного інструментарію та в подальшому формуванню специфікації вимог.

2.2 Опис інструментарію, технологій та математичного апарату

Після проведеного аналізу сучасного стану моделей та методів, що дозволяють вирішити поставлені завдання на основі двох публікацій з переліку фахових видань категорії Б, що не старші за 5 років, можна переходити до опису інструментарію, який використовуватиметься для розробки платформи «EasyDisk Cloud». Головним інструментом для проєктування було обрано Unified Modeling Language [10]. UML – це мова візуального моделювання загального призначення, яка використовується для визначення, візуалізації, конструювання та документування артефактів програмної системи, яка відповідає міжнародному стандарту ISO/IEC 19505. Це допомагає відійти від конкретних реалізацій різних мов програмування та описати архітектуру в вигляді концепції. Для динамічної поведінки системи, буде використано «діаграми прецедентів», які допомагають у створенні чіткого відокремлення межі застосунку, показують акторів платформи, тобто ролей користувачів системи, а саме: звичайний клієнт, адміністратор та гість, а також варіантів їхньої взаємодії з платформою. Для статичного моделювання ж застосовується інша діаграма, а саме «діаграма класів», яка відображає сутність предметної області та її атрибути, методи та зв’язки, що є критично важливою частиною для хмарного сховища, адже файлова система – це складна структура, де необхідно точно описувати відношення типу «один-до-багатьох», бо користувач може мати не тільки 1 папку чи файл, а одразу багато, та рекурсивні зв’язки, у випадку з папками, які містять вкладені директорії.

Наступним розглядатиметься інструментарій для забезпечення конфіденційності даних через використання надійного криптографічного захисту. Для безпеки фізичних файлів користувачів системи, обрано алгоритм шифрування AES-256 [9], тому що цей стандарт дозволяє реалізувати захист на серверній частині, і не навантажувати зайвий раз клієнтські пристрої. Цей стандарт оперує

даними у вигляді двовимірного масиву байтів, або інакше кажучи, матрицею станів, розміром чотири на чотири, використовуючи ключ довжиною 256 біт, з яким алгоритм робить 14 раундів криптографічні перетворення, де в кожному раунді виконується чотири математичні операції. Першою операцією є «SubBytes», де відбувається нелінійна заміна байтів з використанням підстановок S-box. Друга починається «ShiftRows», тобто циклічне зміщення рядків матриці станів. Третьою операцією «MixColumns» множаться стовпці матриці на фіксований многочлен у скінченому полі Галуа $GF(2^8)$ і завершається раунд четвертим етапом «AddRoundKey», де знаходиться побітова операція XOR матриці станів з раундовим ключем. Такий підхід дозволяє гарантувати стійкість системи до атак типу brute-force та забезпечує концепцію «Encryption at Rest», тобто метод захисту інформації, де дані шифруються саме тоді, коли фінально зберігаються на фізичному накопичувачі.

Щоб захистити користувацький сеанс від зовнішніх спроб отримати доступ до його акаунту використовується JSON Web Token [17], та необов'язковою, але дуже важливою двохфакторною автентифікацією. 2FA, працює як другий пароль, який постійно змінюється, і щоб його підключити потрібно відсканувати qr-код, в якому міститься секретний пароль-ключ, для відновлення. JWT своєю чергою використовується вже тоді коли користувач остаточно ввів свої дані при логуванні, після чого йому видається токен, завдяки якому система буде перевіряти, чи саме цей клієнт звертається до неї, чи це спроба хакера отримати доступ до файлів. Токен складається з трьох частин, а саме з: заголовка, корисного навантаження та криптографічного підпису. Безпека токена базується на хеш функції, де підпис формується за формулою 2.1.

$$Signature = HMACSHA256(base64UrlEncode(header) + \\ + "." + base64UrlEncode(payload), secret\_key) , \quad (2.1)$$

де Signature – результуючий криптографічний цифровий підпис токена;

HMACSHA256 – криптографічна функція хешування, яка використовує алгоритм SHA-256 для генерації підпису;

base64UrlEncode – функція кодування вхідних даних у рядковий формат Base64Url;

header – заголовок токена у форматі JSON;

payload – корисне навантаження у форматі JSON;

secret_key – унікальний секретний криптографічний ключ.

Ця функція гарантує, що будь яка стороння спроба зміни даних у корисному навантаженні токена призведе до помилки і сервер одразу відхилить запит, через невідповідність хеш суми.

Найголовнішою частиною платформи хмарного сховища є завантаження файлів на сервер, і для реалізації цього потрібно нівелювати обмеження HTTP/HTTPS протоколів, де тіло, що більше 5 гігабайт, не можна відправляти одним запитом, тому для цього використовується алгоритм «чанкінгу». Математична модель базується на діленні цілого об’єкта на менші частини. Тому, для наочної демонстрації механізму розглянемо наступне, отже нехай загальний розмір файлу становить S_f байт, а максимальний розмір чанку S_c байт, тоді загальна формула буде виглядати так (2.2):

$$N_C = \left\lceil \frac{S_f}{S_c} \right\rceil, \tag{2.2}$$

де N_C – загальна кількість чанків;

S_f – загальний розмір файлу;

S_c – максимальний розмір одного чанку.

Браузер клієнта обчислює індекси байтів для кожного і-го чанка, де $i \in [0,\ N_C - 1]$ і кожний такий фрагмент містить байти від $i * S_c$ до $\min((i+1) * * S_c - 1, S_f - 1)$. Такий підхід суттєво оптимізовує споживання оперативної пам’яті сервера, тому що замість повного файлу сервер отримує маленькі частинки, що кратно зменшує кількість пам’яті для обробки поточного фрагмента.

2.3 Специфікація вимог до програмного забезпечення

1. Призначення та межі проєкту

1.1 Призначення системи

Платформа «EasyDisk Cloud» призначена для забезпечення найкращого користувацького досвіду в сфері хмарних сховищ, і надає безпечне, приватне та інтуїтивно зрозуміле середовище, також система дозволяє клієнтові управляти, зберігати, видаляти та ділитися клієнтськими файлами, без страху за витік своїх даних, тому що в застосунку реалізовано надійне серверне шифрування та перевірка токени під час кожного запиту.

1.2 Погодження, що ухвалені в програмній документації:

– специфікація безпеки вимагає виконання стандартів безпеки, щодо захисту приватних даних користувачів;

– ухвалено реалізації проєкту на основі фундаментального шаблону «чиста архітектура» для серверної платформи;

– взаємодія клієнстської частини з серверною відбувається тільки у форматі JSON.

1.3 Межі проєкту:

– нова розробка покриває клієнтський вебзастосунок та серверний API для обробки файлових операцій та шифрування;

– система розробляється з подальшим розгортанням в ізольованому, віртуальному контейнеризованому середовищі Docker;

– у межах кваліфікаційної роботи не передбачається розробка нативного мобільного застосунка, але завдяки адаптивності інтерфейсу, користувачі зможуть його використовувати в своїх браузерах з будь- якого девайсу.

2. Загальний опис

2.1 Сфера застосування

Програмне забезпечення орієнтується першочергово на індивідуальне використовування, а саме для розробників одиночок, фрілансерів, дизайнерів і взагалі тим клієнтам, що потребують працювати з великими медіафайлами.

2.2 Характеристики користувачів:

– гість – клієнт, який не зареєструвався, але має можливість відкривати, переглядати та встановлювати файли, які йому надали за посиланням, до іншого ж інструментарію доступ не надається;

– користувач – клієнт, що зареєструвався і має доступ до всіх можливостей цієї ролі на сайті, а тобто: скачувати, ділитися, переглядати, видаляти, тощо.

– адміністратор – супер користувач, який має доступ до тих самих інструментів, що і звичайний, плюс адміністративна панель з аудитом, переглядом користувачів і в цілому трафіку.

2.3 Загальна структура і склад системи

Платформа складається з трьох рівнів:

– підсистема зберігання файлів та СКБД складається з: бази даних Microsoft SQL Server для управління записами та системи зберігання на фізичних томах Docker Volumes зашифрованих байтів;

– серверна частина, яка побудована на ASP.NET Core 8.0, що відповідає за обробку HTTP-запитів, автентифікацію та шифрування даних;

– клієнтська частина, яка побудована завдяки бібліотеці React, що забезпечує взаємодію користувача з інтерфейсом, відображає файлове дерево та нарізає файли на чанки.

2.4 Загальні обмеження:

– система потребує сучасний браузер з підтримкою HTML5 File API, щоб гарантувати коректну роботу всіх інструментів;

– швидкість завантаження та розшифрування файлів обмежена швидкістю мережевого каналу користувача.

3. Функції системи

3.1 Функція системи: Реєстрація та автентифікація користувачів

3.1.1 Опис функції: Процес створення облікового запису, чи безпечного входу у вже існуючий з використання JWT для токенізації сесії.

3.1.2 Вхідна і вихідна інформація:

– вхідна: електронна пошта, пароль, і повторення паролю в разі реєстрації;

– вихідна: токен JWT, базова інформація про профіль користувача.

3.1.3 Функціональні вимоги:

– паролі обов’язково зберігають тільки у хешованому вигляді перед записом у базу даних;

– система повинна завершувати авторизований сеанс користувача з системи, коли вичерпується час життя токену для повторної авторизації та аутентифікації.

3.2 Функція системи: Завантаження великий файлів за алгоритмом «чанкінгу»

3.2.1 Опис функції: Безпечне завантаження користувацьких файлів на сервер шляхом розбиття на менші частини у браузері з подальшою послідовною передачею на сервер, де чанки будуть склеєні воєдино.

3.2.2 Вхідна і вихідна інформація:

– вхідна: чанк, метадані у вигляді ID файлу, індекс та загальна кількісті цих фрагментів, FolderId;

– вихідна: статус успішного збереження чанку та загальний відсоток на прогресбарі.

3.2.3 Функціональні вимоги:

– сервер потрібен перевіряти на цілісність кожний фрагмент;

– розмір одного фрагменту не може бути більше 5 мегабайтів, щоб оптимізувати навантаження на систему.

3.3 Функція системи: Генерація захищених публічних посилань

3.3.1 Опис функції: Створення посилань з обмеженим та необмеженим часом життя та з можливістю відкриття навіть для незареєстрованих користувачів.

3.3.2 Вхідна і вихідна інформація:

– вхідна: ID файлу, пароль та час життя посилання;

– вихідна: URL, статус про успішність..

3.3.3 Функціональні вимоги:

– посилання повинно автоматично видалитися після закінчення його життя;

– користувач обов’язково має ввести пароль, якщо він був встановлений.

3.4 Функція системи: М’яке видалення та Кошик

3.4.1 Опис функції: Безпечне видалення файлу або папки з основного простору користувача, і додавання їх до кошику, з можливістю подальшого повного видалення чи відновлення.

3.4.2 Вхідна і вихідна інформація:

– вхідна: ID файла, чи папки, команда для відправки до кошику, чи відновлення, чи остаточного видалення;

– вихідна: оновлений список файлів в кошику та робочої користувацької області.

3.4.3 Функціональні вимоги:

– при виконанні команди «остаточного видалення», система потрібна фізично видалити файл чи папку з сервера та СКБД

4. Вимоги до інформаційного забезпечення

4.1 Джерела і зміст вхідної інформації

– файли будь- яких форматів, що завантажуються клієнтом;

– метадані файлів генеруються сервером.

4.2 Нормативно-довідкова інформація

– Система використовує стандарт для типів MIME для вірного відображення іконок в користувацькому інтерфейсі;

4.3 Вимоги до способів організації, збереження та ведення інформації

– реляційна модель у MSSQL з нормалізацією до 3-ї нормальної форми;

– фізичні файли зберігаються в ізольованому, віртуальному середовищі у файловій системі сервера через Docker Volumes.

5. Вимоги до технічного забезпечення

Щоб забезпечити стабільну роботи платформи, сервер має відповідати наступним мінімальним системним вимогам:

– процесор: 4 ядра, для забезпечення шифрування AES-256 та обробки запитів від клієнтів;

– оперативна пам'ять: мінімум 8 ГБ, з урахуванням потреб СКБД MSSQL, Docker Container та Kestrel;

– дискова підсистема: SSD накопичувач для швидкого вводу/виводу при операціях збереження файлових чанків;

– клієнтський пристрій: мінімальні вимоги для стабільної роботи сучасних браузерів.

6. Вимоги до програмного забезпечення

6.1 Архітектура програмної системи

Архітектура побудована з використанням фундаментального шаблону «чиста архітектура», де backend розділений на чотири незалежних шари, а саме: Domain, Application, Infrastructure, API.

6.2 Системне програмне забезпечення

– ОС сервера: Linux, або Windows з використанням WSL;

– система контейнеризації: Docker Engine та Docker Compose.

6.3 Мережне програмне забезпечення

– зворотний проксі: Nginx.

6.4 Програмне забезпечення ведення інформаційної бази

– СКБД: Microsoft SQL Server 2022;

– ORM-фреймворк: Entity Framework Core 10.0.

6.5 Мова і технологія розробки ПЗ

– backend: C#, фреймворк ASP.NET Core 10.0 Web API;

– frontend: мова TypeScript, бібліотека React 18, збирач Vite.

7. Вимоги до зовнішніх інтерфейсів

7.1 Інтерфейс користувача

– користувацький інтерфейс повинен бути повністю адаптивним під мобільні та десктопні екрани;

– система повинна підтримувати «Drag-and-Drop» для зручного завантаження файлів з операційної системи, та керування всередині її;

7.2 Апаратний інтерфейс

Не передбачено.

7.3 Програмний інтерфейс

– взаємодія між клієнтом та сервером здійснюється виключно через RESTful API;

– формат обміну метаданими: JSON.

7.4 Комунікаційний протокол

– базовий протокол: HTTP/1.1 та HTTP/2;

– додатковий протокол: TLS 1.2/1.3 для захисту даних під час їх транспортування від клієнта до проксі.

8. Властивості програмного забезпечення

8.1 Доступність: платформа проєктується для цілодобового функціонування з мінімізацією часу простою за рахунок швидкого перезапуску Docker-контейнерів.

8.2 Супроводжуваність: завдяки використанню «чистої архітектури» та впровадженню інструментів логування, система є відкритою до налагодження та розширення функціоналу.

8.3 Переносимість: переносимість забезпечується контейнеризацією. Продукт може бути розгорнутий на будь-якому сервері (Linux/Windows/macOS), де встановлено Docker Engine, без необхідності інсталяції додаткових SDK.

8.4 Продуктивність: архітектура асинхронної обробки та алгоритм чанкінгу, забезпечують час відгуку сервера на базові API-запити менше за 0.2 секунди.

8.5 Надійність: механізми транзакцій у MSSQL запобігають появі файлів, що не мають запису в базі даних.

8.6 Безпека: система стійка до таких вразливостей як: SQL Injection, XSS, CSRF, разом з цим реалізовано шифрування AES-256, що гарантує захист даних від несанкціонованого доступу.

9. Інші вимоги

Логування та аудит всіх дій користувачів.

Висновки до розділу 2

У другому розділі проведено аналіз сучасних фахових досліджень, та обгрунтовано вибір архітектури для створення веборієнтованої платформи хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud». Виконано моделювання об’єкта та предмета розробки і окрім цього, описано інструментарій та математичний апарат, у вигляді алгоритму «чанкінгу» для завантаження великих файлів, поясненню роботи симетричного шифрування даних на стороні сервера AES-256, та алгоритм створення токени JWT. Також сформовано специфікацію вимог до програмного забезпечення, де було чітко окреслено функціональні та не функціональні межі проєкту і тому може виступати фундаментом для реалізації платформи.

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБОРІЄНТОВАНОЇ ПЛАТФОРМИ «EASYDISK CLOUD»

3.1 Розробка архітектури програмного забезпечення та вибір компонентів

В результаті складеної стратегії в першому розділ, та написаним специфікаціям вимог до програмного забезпечення, для фундаментального шаблону було обрано «чисту архітектуру» [12], що зможе забезпечити масштабованість та незалежність від зовнішніх фреймворків, та всередині самого проєкту. Шаблон складається з чотирьох шарів, а саме: Domain, Application, Infrastructure та API. Шар Domain – це по суті ядро системи, який не знає про жодний з інших частин проєкту, бо виконує тільки одну задачу, а саме надати сутності для всього застосунку. Для повного вирішення поставлених задач платформа хмарного сховища «EasyDisk Cloud» потребує такі основні класи як: AuditLogEntity, FileEntity, FileVersionEntity, FolderEntity, ShareLinkEntity та TagEntity. Також в середині цього шару зберігаються ENUMs чи інші константи. Всередині Application Layer лежить вся бізнес логіка застосунку, або його правила, і ділиться на: Data Transfer Objects, інтерфейси та розширення. В DTOs описуються «обрізані» класи сутностей, завдяки яким відбувається керування для оновлення, видалення, додавання та отримання інформації з бази даних, а в інтерфейсах використовуються ці ж DTO для роботи з файлами, алгоритмами підрахунку вільного місця, логіки генерації даних, тощо. В шарі Infrastructure зберігається вже технічна реалізація всього вебзастосунку, адже саме тут імплементується робота з Microsoft SQL Server через Entity Framework Core та фізичною системою зберігання, управління та видалення файлів сервера, як приклад: алгоритм «чанкінгу», шифрування файлів завдяки AES-256, створення директорій, оплати за передплату, авторизація користувача і так далі. Це забезпечується завдяки сервісам та репозиторіям, які напряму зв’язані зі своїми інтерфейсами. Також тут налаштовано роботу міграцій бази даних, для

швидкої зміни її структури, Identity Framework з готовими рішеннями для керування користувацькою інформацією та його захистом. І останнім шаром інфраструктури сервера є API, який відповідає за «вхідну точку» застосунка, бо саме з цією частиною проєкта буде зв’язуватися користувацький. Тут налаштовані контролери ASP.NET Core, які приймають HTTP запити, перевіряють JWT токени та повертають відповіді у форматі JSON. Також для обробки помилок, які будуть видаватися у всьому проєкті, в цьому шарі є Middleware який перехоплюватиме їх та фільтри для запису логів в аудит.

В результаті аналізу та отриманого висновку в минулому абзаці, для того, щоб забезпечити швидкість виконання запитів, було обрано мову програмування C#, адже саме на її основі написаний фреймворк ASP.NET Core. Вибір зумовлений тим, що всередину вже вбудовано підтримку механізму впровадження залежностей, тобто Dependency Injection без якого не представляється можливість побудови застосунку з використанням шаблону «чиста архітектура», адже як вже зазначалося, деякі шари не знають зовсім про решту, а інші тільки про частину з них. Також мова програмування C# та ASP.NET Core надають інструменти для високої продуктивності асинхронного програмування, а саме клас Task. Цей клас із простору імен System.Threading.Tasks, являє собою асинхронну операцію, або задачу, яка виконується в фоновому форматі та повертає результат в майбутньому, працюючи в окремому потоці, тим самим не зупиняє роботу застосунка, що в свою чергу є ідеальним для багатошарового вебзастосунку хмарного зберігання даних [18]. Однак, щоб це працювало без помилок, потрібно також до методів з Task, там де необхідно додавати «синтаксичний цукор» у вигляді async та await [19].

Для frontend частини використовується бібліотека React та TypeSctipt для написання коду. TypeSctipt – мова програмування, яка є підмножиною JavaScript і розроблена компанією Microsoft, з тієї причини, що звичайний JavaScript не має строгої статичної типізації, і тим самим вирішує цю проблему, що в свою чергу дозволило створити вебзастосунок «EasyDisk Cloud» [20]. Також TypeScript має нативну підтримку об’єктно орієнтованого програмування, і тому ці переваги

мінімізують більшість помилок ще на етапі написання коду, таких як валідація формату метаданих, ще до відправки їх на сервер. React обрано через його компонентну структуру, що є ідеальним для створення Single-Page-Application, адже дозволяє впровадити компонентну архітектуру елементів користувацького інтерфейсу, такі як кнопки, модальні вікна та цілі блоки загалом, прибирає необхідність перезавантаження сторінки при зміні шляхів, що в свою чергу дозволяє реалізувати фонове завантаження для файлів клієнта.

Для того, щоб не писати весь код з нуля кожної частини застосунка, використовуються вже готові бібліотеки для забезпечення безпеки, роботи файловими та користувацькими даними на боці серверної та клієнтської частин. Для реалізації алгоритму шифрування AES-256 використано вбудовану бібліотеку .NET System.Security.Cryptography.Aes, замість сторонніх плагінів, що гарантує стабільну та максимально швидку операцію по шифруванню та розшифруванню файлових байтів. Щоб захистити користувацькі паролі для авторизації використовується готове рішення з фреймворку Identity, що дозволяє автоматично хешувати приватну інформацію перед записом її в базу даних. Таким самим методом, але з бібліотекою BCript.Net-Next, хешується опціональний пароль для посилань на файли та директорії. Для роботи з токенами використовується компонент Microsoft.AspNetCore.Authentication.JwtBearer, щоб автоматично валідувати цифрові підписи, які надаються для входу користувача в систему, та відокремлення його даних від інших. Забезпечення роботи з інформацією з бази даних виконується завдяки Microsoft.EntityFrameworkCore, адже цей компонент дозволяє працювати з MSSQL та будь якою іншою СКБД через Language Integrated Query, що надає автоматичний захист від будь- яких SQL-ін’єкцій. Для взаємодії користувацької частини з серверною, обрано бібліотеку Axios, адже вона має вбудований механізм відстеження завантаження файлу, та забезпечує стабільний інструментарій для відправки POST, GET, PUT, DELETE запитів, що в свою чергу є критично важливим для завантаження чанками потоків байтів на сервер.

В результаті вибору вищезазначених рішень та інструментарію, можна стверджувати, що вимоги, які були записані в специфікаціях повністю виконуються. Таким чином платформа «EasyDisk Cloud» має потужну, фундаментальну архітектуру для масштабованості та незалежності всередині backend-у, використовує сучасний стек фреймворків та мов програмування, а також бібліотек, для пришвидшеної розробки. Завдяки цьому можна переходити до візуального моделювання функцій, поведінки акторів і тощо, з використанням UML.

3.2 Моделювання функцій та інформаційних потоків з використанням UML

Після розробки архітектури та вибору компонентів можна перейти від текстової специфікації вимог, до програмної реалізації, і тому перед цим потрібно провести візуальне моделювання, завдяки стандарту UML, адже такий підхід надає можливість деталізувати як статичну структуру системи, так і динамічну, де до першої відноситься база даних, ієрархія класів та компонентів, а до другої поведінки, алгоритми і загальні сценарії взаємодій користувачів з платформою. Це робиться з тієї причини, що текстові вимоги, які розглядались в другому розділі, можуть містити приховані архітектурні недоліки, а UML моделювання в цьому випадку виступає універсальною мовою, яка демонструє правила у вигляді чітких технічних контрактів. Таким чином, візуальне моделювання для багатошарового вебзастосунку хмарного зберігання є критичною необхідністю, адже система де функціонує процес чанкового завантаження та паралельно шифрування даних, і тому, це може допомогти уникнути помилок перед початком етапу написання коду.

Для зображення, як клієнт буде використовувати платформу хмарного збереження даних «EasyDisk Cloud» створюються діаграми прецедентів, або ж Use Case, це тип діаграми, що візуально показує функціональні вимоги до системи, тим самим, хто саме взаємодіє з нею, тобто всіх акторів, та які дії система виконує для цього. Система налічує чотири ролі, а саме: гість, який має тільки права для доступу

до публічних посилань, звичайних користувач, функціонал якого налічує завантаження, редагування, перегляд та видалення файлів, створення посилань, і так далі, а також адміністратор, який має ті ж самі права, що і звичайний клієнт, однак до цього ще може переглядати сторінку аудита, всіх користувачів системи та загальну інформацію про використані дані на сервері, та платіжна система. На рисунку 3.1 можна побачити таку діаграму, де зображено рольову модель з чотирма акторами, та варіантами використання платформи.

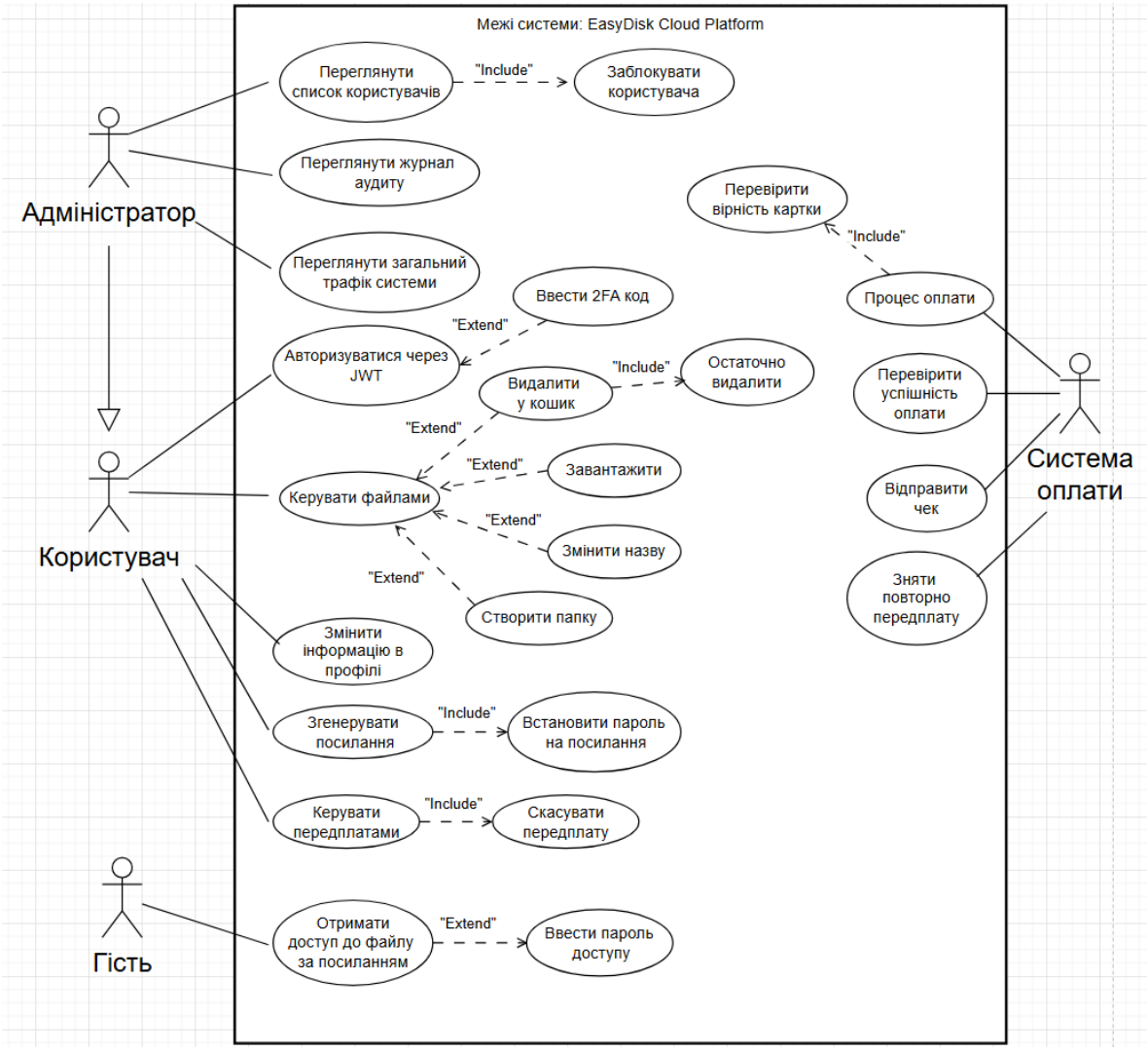

Рисунок 3.1 – Діаграма прецедентів

Діаграма налічує та чотири актори, а саме: адміністратор, користувач, гість та система оплати. Кожна з ролей відображає основні методи і варіанти використання системи, де:

– адміністратор має зв’язок generalization до звичайного користувача, адже наслідує всі його методи, і також вміє переглядати список користувачів, журнал аудиту та загальний трафік в системі;

– користувач налічує методи для керування файлами, а саме: видалення у кошик з подальшою можливістю остаточного знищення, завантаження файлів та папок, зміна назви та створення директорій. Також ця роль передбачає генерацію посилань та керування передплатами;

– гість завжди використовується під час перегляду отриманих посилань з доступом до файлу, в незалежності від того, чи зареєстрований клієнт, чи ні;

– система оплати це єдина не клієнтська роль, адже керування відбувається зовнішньою платформою і тому налічує перевірку успішності оплати, сам процес оплати, відправлення чеку та повторення зняття оплати за передплату. Так спроєктовано для того, щоб вебзастосунок хмарного сховища «EasyDisk Cloud» не потребував зберігати інформацію про банківську таємницю, і таким чином відповідав всім європейським законам.

Після побудови діаграми прецедентів доцільно створити сценарії використання, які будуть використовуватися пізніше для тестування системи. Для цього обрано 2 методи, які будуть найкраще відображати цей процес, а саме: завантаження файлу за алгоритмом «чанкінгу» та авторизація через JWT з активацією 2FA. Спершу розглянути потрібно аутентифікацію, адже це початковий крок для користувача, що продемонстровано в таблиці 3.1.

Таблиця 3.1 – Авторизація через JWT та 2FA

Поле	Опис
ID та Назва	UC-01: Авторизація користувача через JWT та 2FA
Актори	Користувач, Адміністратор
Передумови	Користувач входить в свій обліковий запис в системі та відкрив сторінку аутентифікації

Кінець таблиці 3.1

Основний сценарій	1. користувач вводить свою електронну пошту, та пароль; 2. система на боці користувацького інтерфейсу перевіряє вірність введених даних, та чи залишились пусті рядки вводу; 3. frontend формує запит та передає його на сервер для подальшої обробки; 4. backend формує відповідь та надсилає його назад; 5. користувач потрапляє до наступного меню з вводом коду з мобільного застосунку; 6. система перевіряє вірність коду, створює токен сесії JWT та повертає його користувачеві.
Виключні ситуації	– користувач ввів некоректну або неіснуючу електронну пошту, або пароль; – користувач ввів не вірний код з застосунку двох факторної аутентифікації.
Результат	Користувач отримав токен та перейшов до головної сторінки.

Перший сценарій використання детально демонструє успішний варіант входу клієнта в систему, а також можливі виключні ситуації, які можуть виникнути під час виконання. Як стає зрозуміло з результату в таблиці 3.1, користувач отримує доступ до головної сторінки де знаходяться всі його файли, або пустої, якщо це не було ще виконано. На цій сторінці клієнт окрім перегляду, також може розпочати завантаження нових файлів та папок, завдяки відповідному функціоналу. Для розгляду другого сценарію використання обрано завантаження з алгоритмом «чанкінгу», що відображено в таблиці 3.2.

Таблиця 3.2 – Завантаження файлів з використанням алгоритму «чанкінгу»

Поле	Опис
ID та Назва	UC-02: Завантаження файлів з використанням «чанкінгу»
Актори	Користувач, Адміністратор
Передумови	Користувач авторизований та в нього доступно достатньо вільного місця.
Основний сценарій	1. користувач натискає кнопку «Create Folder» та вводить назву; 2. система перевіряє токен та створює в базі даних директорію; 3. користувач відкриває папку; 4. користувач обирає один, декілька файлів, або папку з системи та переміщує їх у вікно «Drag-end-drop» в браузері; 5. система розпочинає завантаження та відображає прогрес в модальному меню для кожного файлу окремо; 6. після завершення, система генерує попередні мініатюри для графічних та текстових файлів.
Виключні ситуації	– користувач натиснув кнопку «Cancel All» і таким чином зупинив завантаження, або навмисно перезавантажив сторінку; – користувач втратив з’єднання з сервером, чи інтернетом; – розмір файлу чи папки перевищує ліміти передплати.
Результат	Після успішного завантаження система автоматично оновлює інтерфейс, тому одразу з’являються щойно додані файли.

Другий сценарій використання продемонстрував ідеальний варіант при використанні системи користувачем, і моменти де можуть виникнути проблеми,

розриви з мережею і так далі. Таким чином, завдяки таблицям 3.1 та 3.2 стало зрозуміло, як платформа хмарного зберігання даних «EasyDisk Cloud» має себе вести під час роботи з клієнтом.

Також часто створюється діаграма класів на цьому етапі, однак в даному випадку це є не доцільно, через саму структуру застосунку, а саме серверної частини, тому що проєкт складається з: контролерів, що використовуються для обробки запитів від клієнтської частини, сервіси, в який виконується вся бізнес логіка та сутності, які відображають структуру бази даних. Також розмір діаграми є достатньо великим і вмістити її в звіт не можливо.

Останнім етапом, який обов'язково має бути реалізовано в проєкті перед початком частини з кодуванням, є діаграма розгортання системи, яка допомагає спроєктувати фізичну архітектуру для розподілу компонентів по їх обладнаннях, під час виконання роботи. Зазвичай, таку схему створюють тоді, коли застосунок має трирівневу клієнт-серверну архітектуру, як в «EasyDisk Cloud», в якій:

– перший, тобто верхній рівень є клієнстським, бо в цій частині, як зрозуміло з назви, представляється пристрій користувача, тобто його комп'ютер, або ноутбук, або смартфон, на якому виконується застосунок. Frontend в «EasyDisk Cloud» створений за Single Page Application, і тому графічний інтерфейс працює у браузері. Взаємодія з серверною частиною відбувається завдяки захищеним каналам протоколу HTTPS, а обмін даними реалізована з використанням REST API, з передачею інформації у форматі JSON;

– другий рівень є серверним, де виконується обробка клієнтських запитів і працює завдяки фреймворку ASP.NET Core, і має три підрівня, такі як: Kestrel Server, JWT Auth Middleware and ErrorHandler та API. На кожному такому етапі іде перевірка запиту і передача його далі по «конвеєру»;

– третій рівень потрібен для збереження інформації в файловому сховищі Docker Volume та базі даних. Вона розділена на дві незалежні частини, для того, щоб забезпечити найкращу швидкодію системи.

На рисунку 3.2 відображається саме така діаграма розгортання клієнт-серверного вебзастосунку «EasyDisk Cloud».

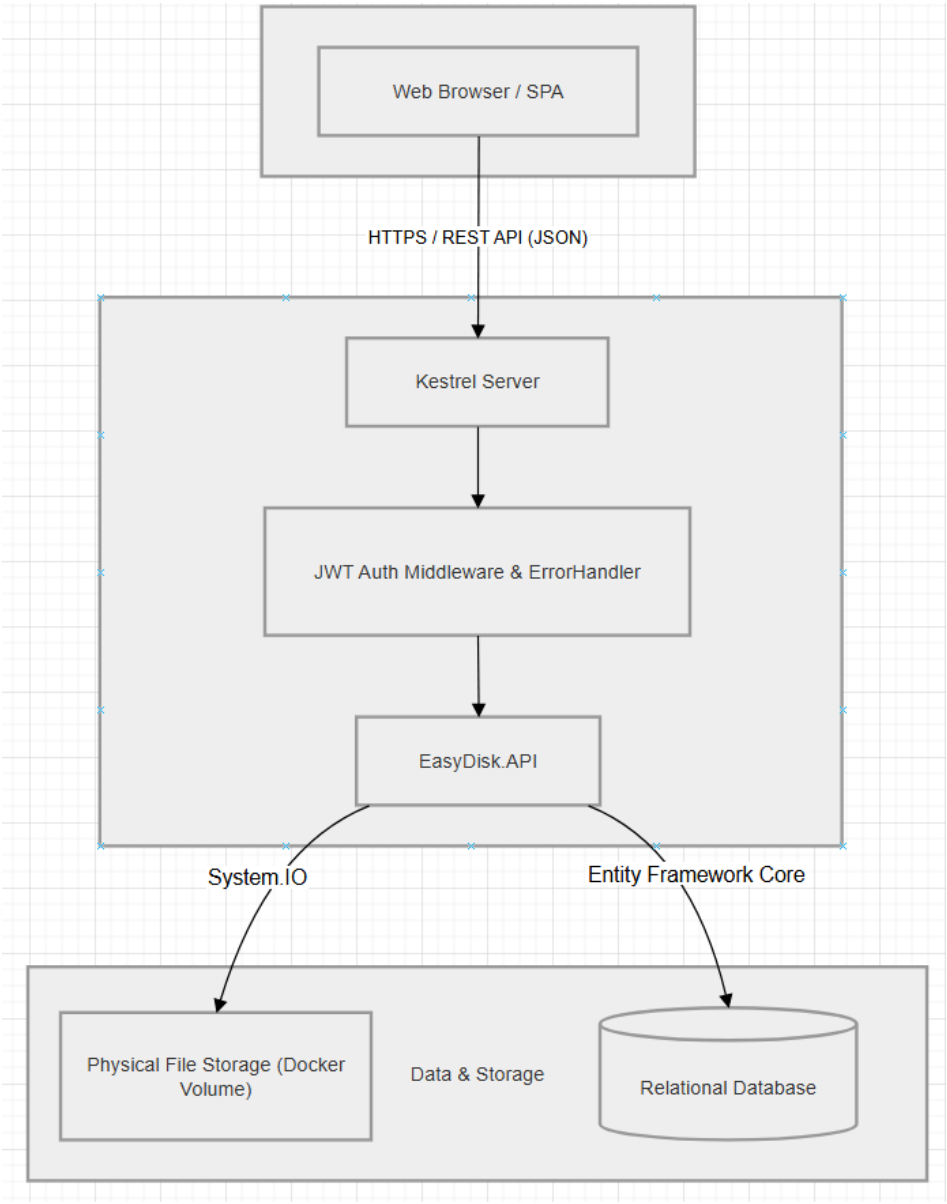

Рисунок 3.2 – Діаграма розгортання

Таким чином проведено візуальне моделювання завдяки стандарту UML, де розроблено діаграми прецедентів, класів та розгортання, що в свою чергу дозволило здійснити ідеальний перехід від текстової специфікації вимог. Спроєктовані моделі забезпечують точність функціональних вимог системи, а це дозволяє перейти до наступного етапу, а саме до опису інтерфейсів та інструкцій користувача.

3.3 Опис інтерфейсів та інструкція користувача

Як зазначалося в минулих підрозділах, а також під час моделювання діаграми розгортання, клієнтська частина платформи «EasyDisk Cloud» реалізована у вигляді Single Page Application, з використанням бібліотеки React, адже з таким підходом сторінки не повинні будуть постійно оновлюватися при зміні шляху, а також, це забезпечує модульність та можливість перевикористовувати вже розроблені елементи без повторного їх написання. Для детального розгляду сторінок створено два макети, щоб продемонструвати загальний вигляд платформи та описати інструкцію, перший з яких, це аутентифікація, що зображено на рисунку 3.3.

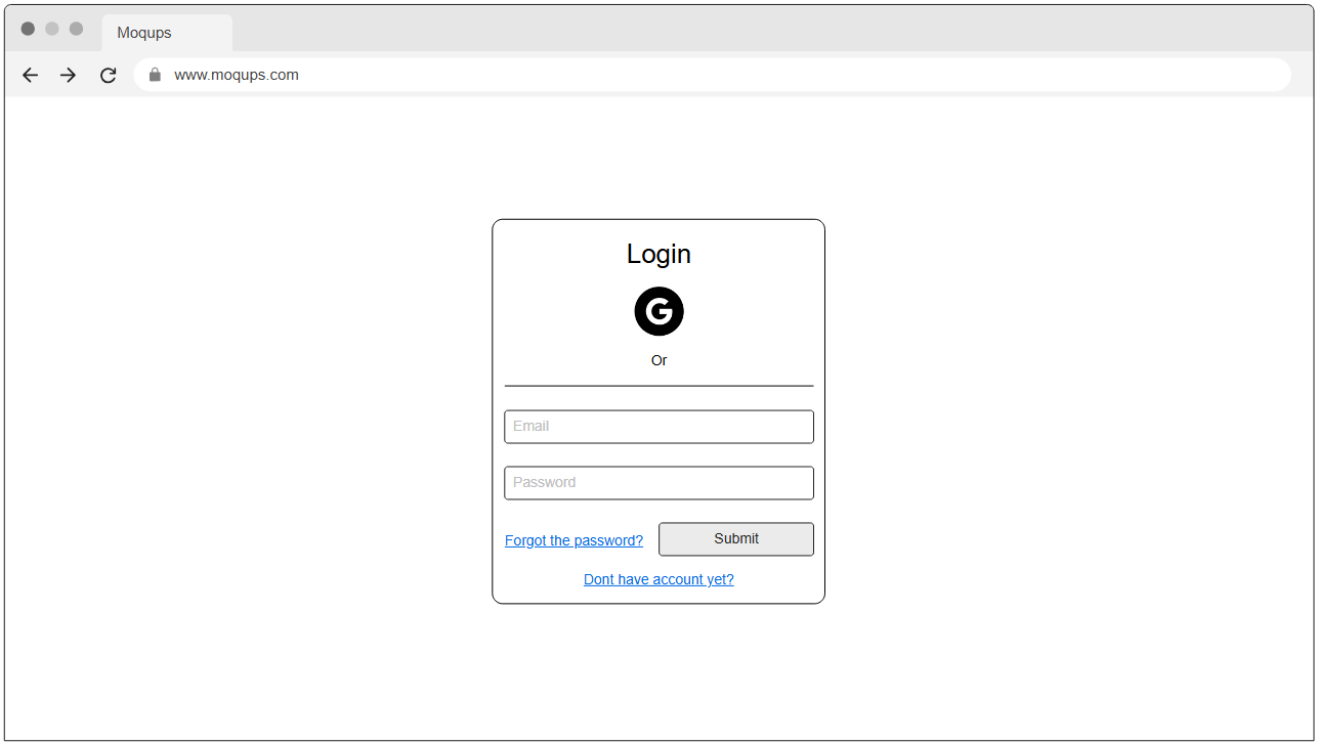

Рисунок 3.3 – Макет сторінки логування

Взаємодія користувача з системою розпочинається зі логування, адже не зареєстрований клієнт не матиме доступу до основних функцій платформи. Сторінка містить форму по середині, де є два поля для вводу електронної пошти та паролю, також є варіант входу завдяки сервісу Google OAuth 2.0, який окрім цього, ще й дозволяє створити аккаунт, якщо раніше, це не було зроблено, а також кнопку «submit» для виконання аутентифікації. У формі передбачено два посилання

«Forgot the password?» та «Don’t have account yet?», які пересилають користувача на форми для відновлення паролю та створення нового акаунту відповідно. При успішній автентифікації клієнтський застосунок отримає від сервера токен сесії, який безпечно зберігатиметься в браузері, і додаватиметься під час кожного запиту для перевірки, чи клієнт збирається виконати цю дію.

Для другої сторінки обрано головну, адже одразу після успішного входу в акаунт, користувача буде перенесено саме сюди. Екран налічує робочу область, де відображається ієрархія файлової системи. На рисунку 3.4 зображений макет головної сторінки, коли клієнт вже має завантажені файли.

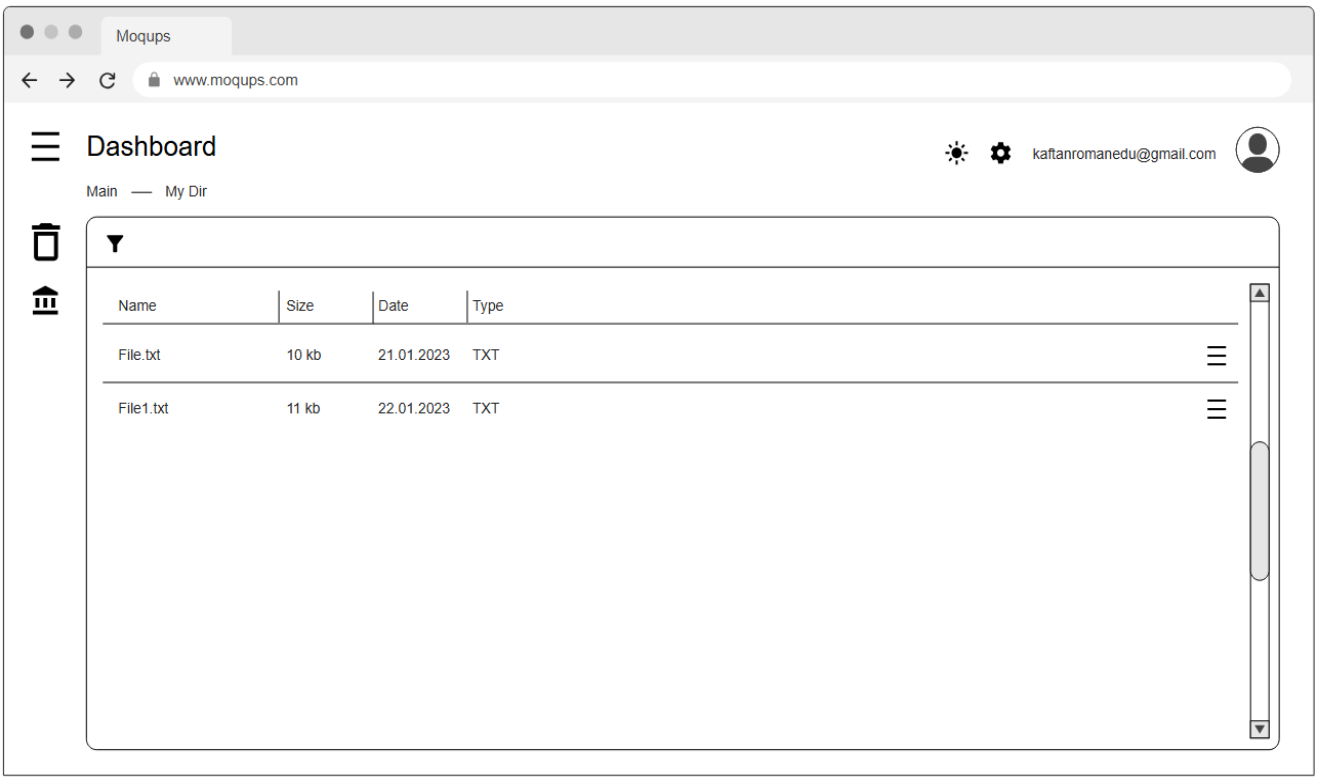

Рисунок 3.4 – Макет головної сторінки

В лівій частині екрану знаходиться навігаційна панель, на якій знаходиться кнопки для переходу на сторінки «сміттєвого кошику» та передплати. В верхній частині відображається поточне місце знаходження в сховищі, під ним є фільтрація, трохи правіше є кнопки для зміни кольорової теми, налаштування профілю та електронна пошта, яка використовується в вигляді імені акаунту. По середині екрану знаходиться робочий простір, де відображені всі завантажені файли та

створені директорії на диску, їх назви, місце яке займають, дату завантаження та тип розширення, а також контекстне меню «бутерброд», при натисненні на яке з’являться варіанти використання. Файли та папки можуть перейменовуватися, встановлюватися, тощо. Також реалізовано функції «Drag-and-Drop» для переміщення та завантаження файлів всередині та поза браузером. Таким чином продемонстровано візуальний інтерфейс користувача завдяки макетам, та описано головні функції сторінок, у вигляді коротких інструкцій.

Висновки до розділу 3

У третьому розділі розроблено архітектуру для веборієнтованої платформа хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud» на базі шаблону «чиста архітектура», а також обгрунтовано вибір технологій, таких як: ASP.NET Core для backend та React для frontend. Виконано моделювання функціональних вимог, що були описані в специфікаях, завдяки використанню стандарту UML, для цього розроблено діаграми прецедентів та розгортання, а також створено сценарії використання для авторизації через JWT та завантаження файлів та директорій за алгоритмом "чанкінгу". Спроєктовано візуальний інтерфейс користувача, а саме два макети для демонстрації та опису інструкцій, сторінки логування та головного екрану, що в свою чергу надає готовий фундамент для переходу до етапу написання коду програми.

4 ПРОГРАМНА РЕАЛІЗАЦІЯ, ТЕСТУВАННЯ ТА КЕРІВНИЦТВО КОРИСТУВАЧА ПЛАТФОРМИ

4.1 Специфікація класів, об'єктів та типів даних

Після завершення текстового та візуального проєктування архітектури веборієнтованої платформи хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud» слід розпочати новий етап, а саме програмування коду застосунку. Для цього першочергово необхідно створити класи сутності, які будуть відображати структуру бази даних, та завдяки яким Entity Framework Core керує. Всього таких класів 7, з яких 4 є основними, а саме:

– ApplicationUser – це сутність користувача, яка наслідується від IdentityUser, і окрім базових полів, має: CreatedAt – поле з датою, коли акаунт користувача був вперше створений, UsedQuotaBytes зберігає в байтах кількість використаної пам’яті з виділеного ліміту, MaxStorageBytes демонструє максимальну куплену кількість місця на користувацькому віртуальному диску, SubscriptionPlan використовується для зберігання назви плану передплати, і має за замовчуванням «Free» статус, StripeCustomerId та StripeSubscriptionId – це поля для платіжної системи, яка безпечно зберігає банківську інформацію клієнта і проводить оплату за передплати, SubscriptionEndDate відповідає за дату закінчення плану та BannedAt – це поле з датою про останнє блокування користувача за порушення правил платформи;

– FileEntity – це клас який відповідає за інформацію про завантажений файл у системі, і містить такі поля, як: Id – це змінна ідентифікатор, яка має тип Guid для уникнення проблем з завантаженням, в разі одночасного додавання двох однакових файлів, з ідентичною назвою, Name для назви, Size для розміру кінцевого файлу, Extension – це поле, що зберігає інформацію про розширення, наприклад .txt або .cs, PhysicalPath для фізичного шляху збережених даних, CreatedAt та DeletedAt – це дати про перше створення та видалення, OwnerId необхідно для прив’язки файлів до окремого користувача, щоб інші не змогли

отримати доступ до сторонніх даних та FolderId – це необов’зкове поле, адже в системі коренева директорія, це є null. Також в цьому класі є поля-зв’язки з Folder, ShareLinks, Versions, Tags, де останні три це колекції;

– FolderEntity – це клас сутності, який відповідає за створені директорії всередині системи, однак на відміну від файлів, тільки в базі даних, а також має такі поля, як: Id – це ідентифікатор папки, який має тип int, а не Guid, бо фізично нічого не створюється, через що таких проблем як у файлів не буде, Name для назви, CreatedAt та DeletedAt – це дати, які використовуються так само як і в сутності FileEntity, тобто для створення та видалення, ParentFolderId, який вказує на папку, що вища в ієрархієї, OwnerId, щоб зберігати власника, а також три поля-зв’язка з ParentFolder, Subfolders та Files, адже директорії можуть мати вкладені файли та папки;

– ShareLinkEntity необхідний, як стає зрозуміло з назви, для генерації та отримання посилань, тому містить поля: Id – змінна ідентифікатор з типом Guid, Token, яке зберігає токен, завдяки якому і отримуватиметься доступ до файлів, PasswordHash – поле хешованого паролю, яке є не обов’язковим, і виставляється тільки з необхідності користувачем, що згенерує посилання, CreatedAt та ExpirationDate змінні дати створення та терміном самознищення, DownloadCount для підрахунку кількості завантажень, OwnerId з власником, FileId та FolderId в разі того, якщо посилання створене на один файл чи цілу директорію, а також зв’язки File та Folder.

Таким чином сутності допоможуть в створенні завдяки міграціям та управлінню бази даних, а разом з інтеграцією EF Core уникнути написання власноруч SQL-запитів, адже замість цього використовуватиметься ApplicationDbContext, в якому і прописуються зв’язки між сутностями. Після цього слід перейти до розгляду основних класів, тобто сервісів, в яких виконуються всі алгоритми системи. Для опису специфікації сервісів обрано два класи, а саме FileStorageService та FileService, адже обидва з них працюють з логікою обробки файлів.

Першим слід розглянути FileStorageService, бо він працює з фізичними даними на диску, які надходять у вигляді чанків від клієнта на сервер, а також за їх тимчасове збереження та врешті формування фінального файлу, та реалізує захист завдяки шифруванню за алгоритмом AES-256. Розбиття файлу на фрагменти дозволяє обійти ліміти протоколу HTTP і зменшує навантаження на сервер. Для того, щоб виконати ці дії створено такі методи:

– AppendChunkAsync, який виконує обробку отриманих фрагментів від клієнта, де створюється тимчасовий файл, якщо прийшов тільки перший чанк, або додається до вже готового, це реалізовується завдяки класу потоків FileStream, з параметром FileMode.Append;

– FinalizeUploadAsync запускається одразу після успішного завантаження останнього фрагменту, і відповідає за злиття та шифрування отриманого файлу. В середину методу в параметрах передається ідентифікатор та тип розширення, і спочатку формується повна назва, перед початком шифрування. З тієї причини, що файл на цьому етапі вже цілісний, він перезаписується в постійне місце дислокації, а тимчасовий файл фізично видаляється.

Кожен чанк має константний розмір в 5 мегабайт, але в байтах, що в свою чергу гарантує, що на сервері ніколи не виділятиметься більше за цей розмір оперативної пам’яті на один потік, що допомагає з уникненням фатальної помилки Out of Memory. Також, перед тим, як остаточно перезаписати файл в нове місце, він шифрується за алгоритмом AES-256, що надає потужний рівень захисту, в разі проникнення з ціллю вкрасти конфіденційну інформацію користувача.

Після успішного розгляду сервісу, який працює з фізичними даними клієнтів, можна переходити до класу FileService, адже він виконує головні алгоритми для платформи хмарного збереження «EasyDisk Cloud», а саме методи встановлення, керування, видалення, перегляду та відновлення файлів. Для найкращого опису класу обрано 5 методи, які дозволять більш детально оглянути виконувані процеси, а саме:

– UploadChunkAsync виконується асинхронно, як і інші, щоб забезпечити клієнтові стабільну систему, яка не буде зависати під час виконання, і що запити які будуть поступати конвеєром, гарантовано обробляться. Роль цього методу у тому, щоб отримати фрагмент з frontend-у та передати його в FileStorageService, також перевіряти, чи останній це чанк, в разі чого метод перейде до фінального етапу злиття;

– DownloadFileAsync використовується тоді, коли користувач потребуватиме завантажити з хмарного сховища на фізичне файл або папку, тому для цього метод приймає ідентифікатор файлу та власника, отримує інформацію з бази даних, після чого створює потік даних та поступово віддає його назад в браузер клієнта;

– SoftDeleteFileAsync та HardDeleteFileAsync найкраще розглядати разом, адже перший метод насправді не знищує дані, бо файл що видаляється вперше, спочатку переміщується до кошику, де користувач за необхідністю може відновити його. Тому спершу в базі даних в полі DeletedAt додається дата, коли користувач видалив файл, через що він більше не показуватиметься на головній сторінці, а тільки в кошику, а вже звідти з’явиться можливість використати другий метод HardDeleteFileAsync, для остаточного очищення фізично;

– RestoreFileVersionAsync – це метод, що дозволяє відновити попередні версії файлів, в разі того, якщо було завантажено такий самий, або з тією ж назвою документ. В файловій системі Windows при перезапису стара версія видаляється, і замінюється на нову, платформа «EasyDisk Cloud» же зберігає всі варіанти таких документів до моменту остаточного видалення їх користувачем.

Таким чином, продемонстровано класи сутностей та сервісів, які виконують різні поставлені задачі, де перші створені лише для відображення структури бази даних, та подальшою змогою керування, а другі виконують всю бізнес-логіку застосунку для завантаження та обробки файлів, папок, тегів, тощо. Використання суворо типізованих змінних для сутностей, дозволяє забезпечити цілісність даних, а в свою чергу сервіси гарантують високу швидкість виконання запитів, з

мінімізацією втрат та забезпечують високий рівень захисту. Саме тому створена структура, дозволяє перейти до розгляду вихідного коду застосунку в наступному підрозділі.

4.2 Лістинги вихідного коду основних компонентів

Після успішного опису методів та основних класів, слід перейти до детального розгляду коду, тому для цього відібрано чотири фрагменти, три з яких з серверної частини і один з клієнтської. Першим таким методом з backend-у є FinalizeUploadAsync, що зображений на рисунку 4.1, тому що саме в цьому фрагменті коду, файли які були завантажені, перед збереженням у сховище, проходять етап шифрування за алгоритмом AES-256.

```
public async Task<string> FinalizeUploadAsync(string uploadId, string extension)
{
    var tempFilePath = Path.Combine(_tempDirectory, $"{uploadId}.tmp");

    if (!File.Exists(tempFilePath))
    {
        throw new StorageFileNotFoundException($"Temp file not found {uploadId}");
    }

    var finalFileName = $"{Guid.NewGuid()}{extension}";
    var finalFilePath = Path.Combine(_uploadDirectory, finalFileName);

    using (Aes aes = Aes.Create())
    {
        aes.Key = _encryptionKey;
        aes.GenerateIV();

        using (var finalFileStream = new FileStream(finalFilePath, FileMode.Create, FileAccess.Write))
        {
            await finalFileStream.WriteAsync(aes.IV, 0, aes.IV.Length);

            using (var encryptor = aes.CreateEncryptor())
            using (var cryptoStream = new CryptoStream(finalFileStream, encryptor, CryptoStreamMode.Write))
            using (var tempFileStream = new FileStream(tempFilePath, FileMode.Open, FileAccess.Read))
            {
                await tempFileStream.CopyToAsync(cryptoStream);
            }
        }
    }

    File.Delete(tempFilePath);

    return Path.Combine("Uploads", finalFileName);
}
```

Рисунок 4.1 – Метод FinalizeUploadAsync

FinalizeUploadAsync використовується для двох цілей, а саме для збереження файлу після його повного завантаження на диск сервера та шифрування даних з

AES-256. Спочатку документ знаходиться в тимчасовій директорії, з якої потрібно перенести до загального сховища, для цього у параметрах функції передається ідентифікатор файлу та розширення, завдяки чому створюється повна назва та остаточний шлях розміщення. Після першого етапу, розпочинається шифрування, де з конфігурації отримується ключ з 32 символів, та генерується IV, тобто унікальний вектор ініціалізації, що записується на початок файлу, потім запускається потік, який перезаписує дані з тимчасового документу у фінальний, зашифрований, де вкінці після успіху, тимчасовий видаляється.

Перед етапом з шифруванням та фіналізацією є асинхронна обробка фрагментів, для того, щоб уникнути проблеми з лімітом на один запит в протоколі HTTP. Тому наступним методом слід розглянути UploadChunkAsync, який зображений на рисунку 4.2, де і виконується поетапне завантаження файлу з нарізаних чанків з клієнтської частини застосунку.

```
public async Task<FileResponseDto?> UploadChunkAsync(UploadChunkDto uploadChunkDto, Stream chunkStream)
{
    var userId = _currentUserService.UserId
        ?? throw new ValidationException("User must be authenticated to upload file");

    if (uploadChunkDto.FolderId.HasValue)
    {
        await _folderRepository.ExistsAsync(uploadChunkDto.FolderId.Value, userId)
            .ValidateExistsAsync(() => $"Folder with id {uploadChunkDto.FolderId.Value} not found.");
    }

    await _fileStorageService.AppendChunkAsync(uploadChunkDto.UploadId, chunkStream);

    if (uploadChunkDto.ChunkIndex < uploadChunkDto.TotalChunks - 1)
    {
        return null;
    }

    return await FinalizeUploadProcessAsync(uploadChunkDto, userId);
}
```

Рисунок 4.2 – Метод UploadChunkAsync

UploadChunkAsync приймає в параметрах потік чанку та загальну інформацію про файл, після чого отримує ідентифікатор користувача з JWT, та перевіряє чи авторизований зараз клієнт, адже це є додатковим захистом на підробку запиту. Потім є перевірка на те, чи існує папка, в яку буде

завантажуватися документ, в разі відсутності видаватиметься помилка. Одразу за перевірками знаходиться асинхронний виклик методу з класу FileStorageService, який розглядався в минулому підрозділі. До запис фрагментів буде до того моменту, поки кількість не становитиме «uploadChunkDto.ChunkIndex < uploadChunkDto.TotalChunks – 1», тобто не залишеться ні одного незавантаженого чанку. Врешті повернеться інформація про файл та айді до асинхронного методу, що виконає фіналізацією з використанням FinalizeUploadAsync. Останнім фрагментом коду, що слід обов'язково розлядати, це метод нарізання з клієнтської частини, а саме uploadFile, що зображений на рисунку 4.3.

```
const uploadFile = useCallback(async (file: File, folderId: number | null) => {
    const uploadId = uuidv4();
    const totalChunks = Math.max(1, Math.ceil(file.size / CHUNK_SIZE));

    setJobs(prev => [...prev, {
        id: uploadId,
        fileName: file.name,
        progress: 0,
        status: 'uploading',
        type: 'upload',
        folderId
    }]);

    try {
        for (let chunkIndex = 0; chunkIndex < totalChunks; chunkIndex++) {
            if (cancelledUploads.current.has(uploadId)) {
                cancelledUploads.current.delete(uploadId);
                return;
            }

            const start = chunkIndex * CHUNK_SIZE;
            const end = Math.min(start + CHUNK_SIZE, file.size);
            const chunk = file.slice(start, end);

            await fileApi.uploadChunk(
                { uploadId, fileName: file.name, chunkIndex, totalChunks, folderId },
                chunk
            );

            const progress = Math.round(((chunkIndex + 1) / totalChunks) * 100);

            setJobs(prev => prev.map(job =>
                job.id === uploadId ? { ...job, progress } : job
            ));
        }

        setJobs(prev => prev.map(job =>
            job.id === uploadId ? { ...job, status: 'completed' } : job
        ));

        queryClient.invalidateQueries({ queryKey: ['files-and-folders', folderId] });
        toast.success(`File "${file.name}" downloaded!`);
```

Рисунок 4.3 – Метод uploadFile

Спочатку в uploadFile генерується унікальний ідентифікатор для нового файлу, а також вираховується загальна кількість фрагментів, які буде відправлено

на серверну частину, після чого реєструється завдання Jobs, що дозволяє одразу відобразити користувачу прогрес-бар та змінювати його динамічно. В блоці Try Catch розпочинається послідовна відправка чанків з перевіркою на обрив з'єднання з мережею. Також реалізовано механізм скасування завантаження, нарізка файлу на рівні частини в 5 мегабайт, завдяки математичним перетворенням, з подальшою відправкою на сервер, де в параметрах знаходиться інформація про файл, та сам чанк. В кінці прогрес-бар заповниться на 100 відсотків та користувач отримає повідомлення про успішне завантаження або помилку під час виконання алгоритму.

Таким чином наведені лістинги кодув цьому підрозділі, а також в додатку Б доводять успішну реалізацію основних методів платформи «EasyDisk Cloud». Наведені фрагменти коду виконують завдання від нарізання та відправки, до об'єднання на шифрування файлів, що надає високий рівень безпеки. Саме тому з'являється можливість перейти до наступної частини розділу, а саме до тестування системи емпіричними методами.

4.3 Тестування програмного застосунку та аналіз результатів

Тестування системи обов'язкове при розробці програмного забезпечення особливо для платформи хмарного збереження, управління та розповсюдження даних, тому обрано такі методи як:

– модульне тестування, тобто Unit Testing для перевірки математичної складової шифрування;

– стрес-тестування, для аналізу споживання системи оперативної пам'яті під час завантаження великого файлу;

– тестування на відмовостійкість, для перевірки поведінки системи при непрогнозованих ситуаціях, таких як обрив мережі або падіння бази даних.

Перший тест, який слід розглянути, це завантаження великого файлу, який детально розписаний в таблиці 4.1.

Таблиця 4.1 – Завантаження великого файлу

№ Т1 : Завантаження великого файлу		
Мета: переконатися, що під час завантаження файлу великого розміру немає витоків пам’яті.		
Тип: Stress		
Приорітет: High	Час на виконання: 2 хв.	
Дата: 01.05.2026	Власник: Кафтан Р.М.	
Передумови: Створено пустий файл обсягом 2 гігабайти, а також запущено команду docker stats.		
Дії	**Очікуваний результат**	**Наявний результат**
1. Відкрити сторінку з завантаженням	1. Сторінка завантажується успішно	Успішно
2. Розпочати завантаження з 2 гб файлом	2. Завантаження розпочинається, прогрес бар починає змінюватись динамічно.	Успішно
3. Відстеження системи на витік пам’яті	3. Витоку пам’яті не має.	Успішно
Післяумови: Файл завантажено в систему і отримано повідомлення про успіх		
Результат: Успішно **Час виконання:** 2 хв **Дата виконання: 01.05.2026**		

В таблиці 4.1 отриманий результат повністю підтверджує, що система може витримувати завантаження великих файлів, і при цьому вільно працювати без зависань. Також, завдяки цьому тесту було перевірено систему на те, чи правильно працює алгоритм чанкінгу, і чи будуть виникати проблеми при обході протокола HTTP. Після успіху в першому тесті, який розглядав метод стресостійкості слід перейти до другого, модульного тестування цілісності файлу, лістинг коду якого можна побачити в додатку А, з алгоритмом AES-256, що описується в таблиці 4.2.

Таблиця 4.2 – Модульне тестування цілісності алгоритму AES-256

№ Т1 : Модульне тестування цілісності алгоритму AES-256	
Мета: перевірити цілісність файлу до та після шифрування і розшифрування.	
Тип: Unit	
Приорітет: High	Час на виконання: 0.5 хв.

Кінець таблиці 4.2

Дата: 02.05.2026		Власник: Кафтан Р.М.
Передумови: Створено unit-тест клас EncryptionTests.		
Дії	**Очікуваний результат**	**Наявний результат**
1. Запустити модульний тест Encrypt_And_Decrypt_ShouldReturnOriginalData	1. Тест розпочався	Успішно
Післяумови: Тест пройдено успішно, цілісність підтверджена, адже файл до шифрування дорівнює тому, що було розшифровано.		
Результат: Успішно **Час виконання:** 0.5 хв **Дата виконання: 02.05.2026**		

Модульний тест Encrypt_And_Decrypt_ShouldReturnOriginalData спочатку ініціалізує змінну originalContent з текстом і перетворюється в набір байтів, який в подальшому зашифрується. Перед цим завдяки Mock створюється конфігурація, яка створює ключ та шлях для файлу. Після успішного шифрування, файл розшифровується і одразу ж звіряються біти до та після, де з тесту видно, що це успішно. Та останнім, обов'язковим модулем, що залишається - це тест на відмовостійкість, що описаний в таблиці 4.3.

Таблиця 4.3 – Тестування стійкості до відмов

№ Т1 : Тимчасова недоступність СКБД		
Мета: перевірити стійкість системи після втрати з'єднання з базою даних.		
Тип: Fault tolerance		
Приорітет: High	Час на виконання: 5 хв.	
Дата: 02.05.2026	Власник: Кафтан Р.М.	
Передумови: Вебзастосунок запущений та успішно працює.		
Дії	**Очікуваний результат**	**Наявний результат**
1. Відкрити головну сторінку	1. Сторінка завантажується успішно	Успішно
2. Вимкнути контейнер з СКБД	2. З'єднання втрачається	Успішно
3. Перезавантажити головну сторінку	3. Система працює, однак інформації немає	Успішно

Кінець таблиці 4.3

4. Увікнути контейнер з базою даних	4. Система успішно працює та з'єднання відновлюється автоматично	Успішно
Післяумови: Вебзастосунок працює штатно та без критичних помилок.		
Результат: Успішно **Час виконання:** 5 хв **Дата виконання: 02.05.2026**		

Останній тест був також успішним, і доказав, що система стійка до відмовостійкості, і може без проблем автоматично перезапускатись, навіть при втраті з'єднання з базою даних. Це працює, завдяки тому, що платформа хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud» є багатошаровою, та розділеною на три рівні системою, що і в свою чергу дозволяє працювати без критичних помилок, навіть при відмові одного з рівнів. Таким чином, розглянуто три методи тестування, однак для більш детального опису, слід також створити графік використання оперативної пам'яті при завантаженні файлу великого об'єму, який зображений на рисунку 4.4.

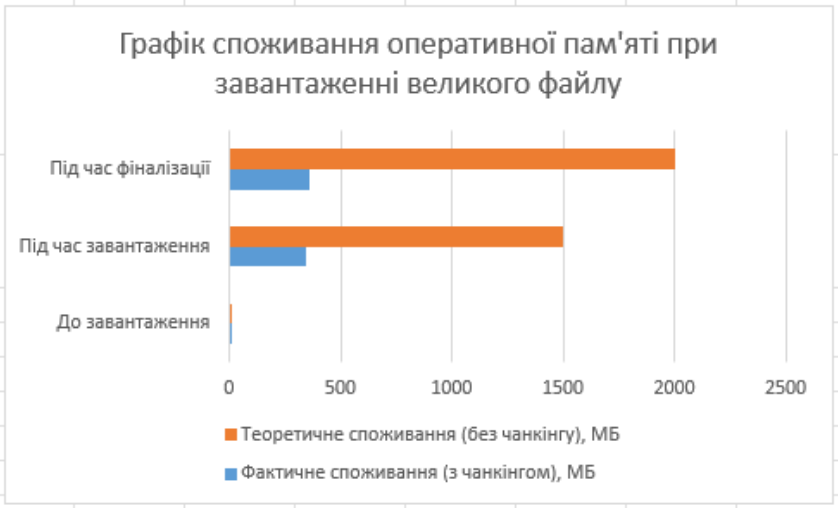

Рисунок 4.4 – Графік споживання оперативної пам'яті

На рисунку 4.4, можна побачити графік споживання оперативної пам'яті при завантаженні файлу з обсягом 2 гігабайти, де сині стовпці відображають реальну картину, а помаранчеві «теоретичну». Теоретичне споживання відображає те, як система використовувала б оперативну пам'ять без алгоритму чанкінгу, однак

окрім того, що витрачання скоротилося до 5,5 разів, це також і дозволило уникнути помилок, які б точно виникли, якщо б цілий файл не розбивався на окремі фрагменти при передачі серверу.

Таким чином проведене тестування доказало, що система є стресо та відмово стійкою, також, що шифрування працює стабільно. Це дозволяє оцінити програмний застосунок, як потужну платформу, яка може виконувати більшість потреб користувача та має можливість легко масштабуватися. Саме тому, доречно переходити до наступного, а вірніше останнього етапу, а саме керівництво користувача.

4.4 Керівництво користувача

Останнім кроком розділу є керівництво користувача, як він має поводитись з вебзастосунком. За макетами з розділу 3, розроблений інтерфейс з використанням Tailwind CSS та Shadcd. Найпершою сторінкою, яку побачить клієнт, є сторінка логування що зображена на рисунку 4.5.

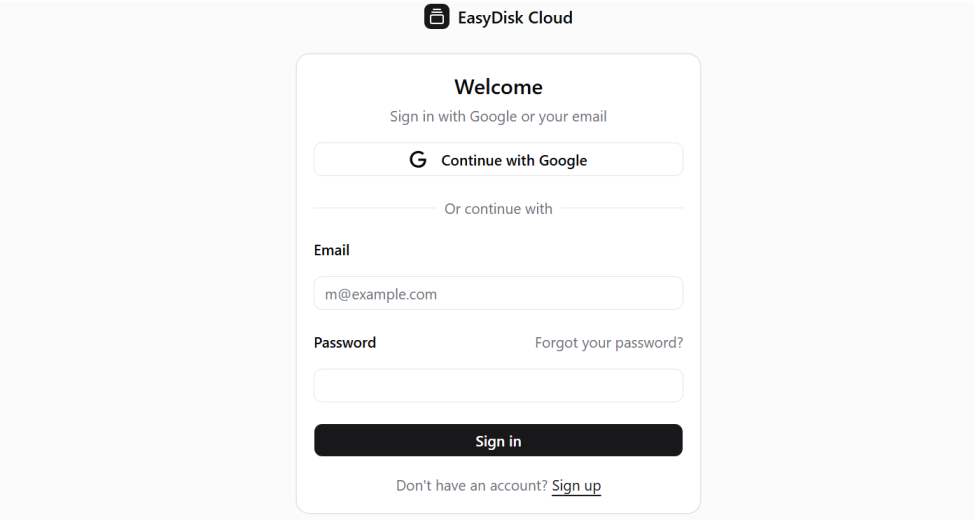

Рисунок 4.5 – Сторівна логування

Сторінка авторизації має можливість входити в акаунт не тільки завдяки звичним полям з електронною поштою та паролем, а також з акаунтом Google. Якщо клієнт досі не має акаунту в системі, йому потрібно натиснути на посилання «Sign up» під кнопкою «Sign in», в іншому випадку користувач повинен ввести свої

дані та ввійти в свій обліковий запис. Одразу після входу, зустрічає головна сторінка, що зображена на рисунку 4.6.

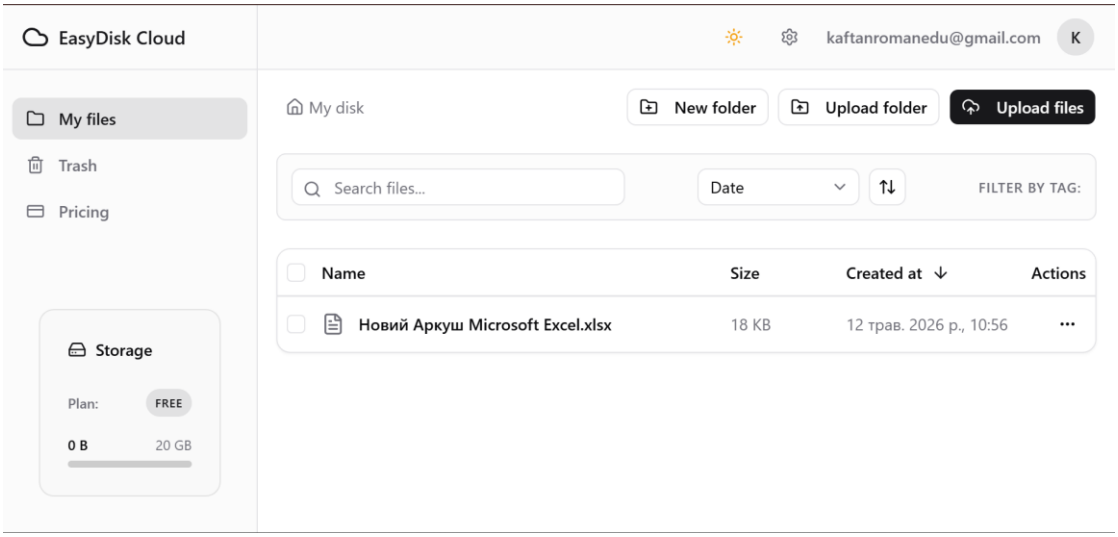

Рисунок 4.6 – Головна сторінка

Головна робоча зона поділена на декілька логічних зон, а саме: ліва частина є навігаційною, зі всіма доступними сторінка, та використаним лімітом пам’яті, згори знаходиться кнопки зміни теми та налаштувань профілю і центральній зоні відображається наявні файли та папки, та кнопки для роботи з файловою системою, завдяки яким користувач може завантажувати нові документи, або перемістити зі своєї системи, цей процес зображений на рисунку 4.7.

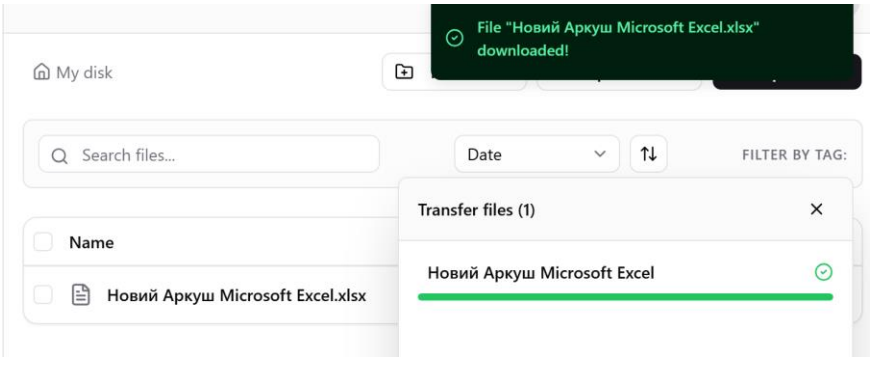

Рисунок 4.7 – Процес завантаження файлів

Процес завантаження розпочинається одразу після переміщення в зону «Drag-and-Drop», а також завдяки цьому в файловій системі є можливість змінювати шлях для кожного файлу та папки. Для цього необхідно затиснути файл

та перемістити в потрібну директорію і навпаки. Також на рисунку 4.7 зображений прогрес-бар, який реалізований завдяки алгоритму чанкінгу. У кожного файлу та папки є можливість створювати публічні посилання, тому для цього в меню з трьома точками є відповідна кнопка, при натисканні якої з'являється меню, що відображено на рисунку 4.8.

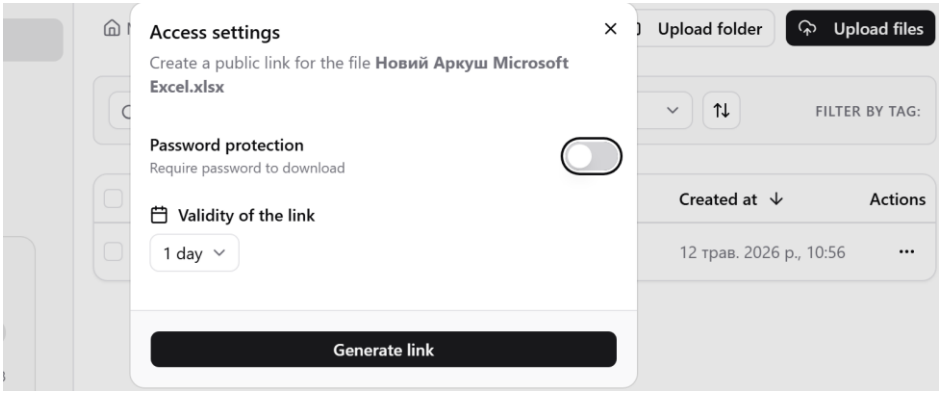

Рисунок 4.8 – Модальне меню створення публічного посилання

В цьому меню є можливість додавання паролю доступу та терміну життя для посилання. Користувач, після налаштування, потрібне натиснути на кнопку «Generate link», яка надасть саме посилання, за яким інші клієнти зможуть отримувати доступ до цього файлу. Також видалені файли, потрапляють до кошику, ця сторінка зображена на рисунку 4.9.

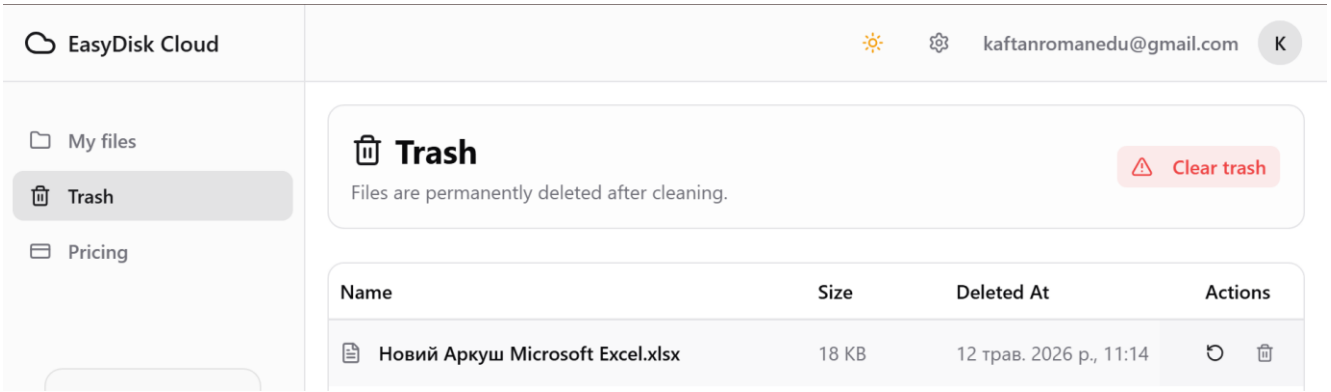

Рисунок 4.9 – Кошик з видаленими файлами

На сторінці з кошиком, відображаються всі видалені предмети з системи, з можливістю повного очищення всього, або окремих папок чи файлів. Також є

можливість відновлення, кнопкою в колонці «Actions», що окрім самого файлу, ще і поверне його версії.

Таким чином розглянуто основні сторінки користувацького інтерфейсу, що демонструє загальний високий рівень виконання вебзастосунку. Завдяки дотриманню сучасних стандартів, платформа хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud», має зручний та адаптований графічний інтерфейс, який працює однаково добре і на мобільних пристроях, і на стаціонарних комп’ютерах. Це дозволяє без додаткових умінь керувати сховищем, завантажувати файли, та отримувати задоволення від користування застосунком.

Висновки до розділу 4

У четвертому розділі розроблено вебзастосунок для хмарного зберігання, управління та розповсюдження даних «EasyDisk Cloud» за трьох рівневою архітектурою описаною в минулих розділах, а саме: frontend, backend та файлове сховище з базою даних. Виконано тестування розробленого програмного забезпечення, використовуючи три методи, такі як: модульне тестування з написання Unit тесту для перевірки шифрування AES-256, стрес-тестування на завантаження файлів великого об’єму та на витоки пам’яті, та тестування на відмовостійкість, при втраті з’єднання з базою даних. Також розроблено користувацький інтерфейс за макетами з третього розділу, що був побудований завдяки React, Tailwind CSS та Shadcn, а також описано інструкцію для користування, де покроково зображені дії з графічних інтерфейсом.

ВИСНОВКИ

В ході виконання кваліфікаційної роботи, проаналізовано галузь та ринок готових рішень хмарних сховищ. Перші, такі як Google Drive Microsoft та OneDrive, що розроблені великими корпораціями, мають велику кількість переваг, однак недоліком є можливі втручання в конфіденційну інформацію клієнта. Другі, такі як NextCloud, вирішують проблеми рішень від централізованих сховищ, завдяки відкритому вихідному коду, однак має також деякий перелік мінусів, через свою структуру, адже попередньо для користування застосунком, потрібно мати хоча б базові навички системного адміністрування. Через що, констатовано те, що ринок переповнений компромісів, тому обрано стратегію по мінімізації недоліків при розробці повністю нової веборієнтованої платформи хмарного збереження, управління та розповсюдження даних «EasyDisk Cloud». Тому одразу після цього, проведено аналіз наукових робіт категорії Б, обрано список необхідних технологій, розглянуто математичну частину та розроблено специфікації вимог. Потім розроблено архітектуру на базі фундаментального шаблону «чиста архітектура», обгрунтовано вибір технологій, виконано візуальне моделювання функціональних вимог з стандартом UML та спроєктовано візуальний інтерфейс користувача з використанням макетів. Це дозволило перейти до написання програмного коду для трьох рівневого клієнт-серверного вебзастосунку. Продемонстровано основні класи платформи, для роботи зі сховищем, та обробкою запитів завантаження з розбиттям файлів на фрагменти, також виконано тестування розробленого програмного забезпечення, використовуючи модульне тестування, стрес-тестування та тестування на відмовостійкість для проведення загальної оцінки застосунку, та розроблено користувацький інтерфейс за макетами з третього розділу. Таким чином, в готовій платформі хмарного сховища даних «EasyDisk Cloud», мінімізовано недоліки конкурентів, забезпечено високий рівень захисту, розроблено адаптований та зручний графічний інтерфейс та гарантовано стабільність роботи під навантаженням.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025. Statista Research Department. URL: https://www.statista.com/statistics/871513/worldwide-data-created/ (Accessed: 21.04.2026)

2. Google Drive. URL: https://drive.google.com (Accessed: 21.04.2026)

3. Microsoft OneDrive. URL: https://onedrive.live.com (Accessed: 21.04.2026)

4. NextCloud. URL: https://nextcloud.com/ (Accessed: 21.04.2026)

5. Riley, Jenn. "Understanding metadata." Washington DC, United States: National Information Standards Organization. 2017. P. 7-10.

6. Schneier, Bruce. Data and Goliath: The hidden battles to collect your data and control your world. WW Norton & Company, 2015. P. 537.

7. Bach-Nutman, Matthew. "Understanding the top 10 owasp vulnerabilities." 2020. P. 1-4.

8. Fielding R., Reschke J. Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing (RFC 7230) / Internet Engineering Task Force (IETF). 2014. URL: https://datatracker.ietf.org/doc/html/rfc7230 (Accessed: 24.04.2026).

9. Selent, Douglas. "Advanced encryption standard." Rivier Academic Journal 6.2. 2010. P. 1-14.

10. Jacobson, Lvar, and James Rumbaugh Grady Booch. "The unified modeling language reference manual." 2021. P. 568.

11. Freeman A. Pro Asp. net core MVC. Apress, 2016. ISBN 1484203976. P. 1017.

12. Martin R. C. Clean architecture: a craftsman’s guide to software structure and design. Prentice Hall Press, 2017. ISBN 0134494164. P. 432.

13. Merkel D. Docker: lightweight linux containers for consistent development and deployment. Linux j. 2014. Vol. 239, № 2. P. 2.

14. Дмитрів Н. С. Анонімність і конфіденційність при обміні чутливою інформацією: порівняння централізованих і децентралізованих підходів. Таврійський науковий вісник. Серія: Технічні науки. 2025. № 4, ч. 1. С. 76–82. DOI: 10.32782/tnv-tech.2025.4.1.8

15. Поперешняк Д. І., В'юннік Ю. О., Поперешняк С. В. Модель інтелектуальної системи управління колекціями в хмарних середовищах. Таврійський науковий вісник. Серія: Технічні науки. 2025. № 4, ч. 1. С. 237–245. DOI: 10.32782/tnv-tech.2025.4.1.25

16. Singh, Prateek. "Linux development on WSL." Learn Windows Subsystem for Linux: A Practical Guide for Developers and IT Professionals. Berkeley, CA: Apress. 2020. P. 131-168.

17. Sakimura N., Jones M., Bradley J. JSON Web Signature (JWS). RFC 7515, May 2015. URL: https://rfc-editor.org/rfc/rfc7515.txt, 2015. (Accessed: 28.04.2026)

18. Richards M., Ford N. Fundamentals of software architecture: an engineering approach. O'Reilly Media, 2020. ISBN 1492043427. P. 432.

19. Sarcar, Vaskaran. "Understanding Tasks." Parallel Programming with C# and. NET: Fundamentals of Concurrency and Asynchrony Behind Fast-Paced Applications. Berkeley, CA: Apress, 2024. P. 1-70.

20. Bierman, G., Abadi, M., & Torgersen, M. (2014, July). Understanding typescript. In European Conference on Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg. P. 257-281.

ДОДАТОК А

Лістинг коду Unit тесту

```
public class EncryptionTests
{
    [Fact]
    0 references
    public async Task Encrypt_And_Decrypt_ShouldReturnOriginalData()
    {
        var originalContent = "This text must be AES-256 encrypted and successfully decrypted!";
        var originalBytes = Encoding.UTF8.GetBytes(originalContent);

        var mockConfig = new Mock<IConfiguration>();
        mockConfig.Setup(c => c["Storage__EncryptionKey"]).Returns("GAuihsd2rASGFbqw94ajKJSFhs7dgfvs");
        mockConfig.Setup(c => c["Storage:BasePath"]).Returns(Path.GetTempPath());

        var storageService = new FileStorageService(mockConfig.Object);
        var uploadId = Guid.NewGuid().ToString();

        using (var inputStream = new MemoryStream(originalBytes))
        {
            await storageService.AppendChunkAsync(uploadId, inputStream);
        }

        var physicalPath = await storageService.FinalizeUploadAsync(uploadId, ".txt");

        string decryptedContent;
        using (var decryptedStream = await storageService.GetFileStreamAsync(physicalPath))
        using (var reader = new StreamReader(decryptedStream))
        {
            decryptedContent = await reader.ReadToEndAsync();
        }

        Assert.Equal(originalContent, decryptedContent);

        await storageService.DeleteFileAsync(physicalPath);
    }
}
```

Рисунок А.1 – Лістинг коду модульного тесту

ДОДАТОК Б

Лістинг коду застосунку

```
protected override void OnModelCreating(ModelBuilder builder)
{
    base.OnModelCreating(builder);

    builder.Entity<FileEntity>().HasQueryFilter(f => f.DeletedAt == null);
    builder.Entity<FolderEntity>().HasQueryFilter(f => f.DeletedAt == null);

    builder.Entity<FolderEntity>()
        .HasOne(f => f.ParentFolder)
        .WithMany(f => f.Subfolders)
        .HasForeignKey(f => f.ParentFolderId)
        .OnDelete(DeleteBehavior.Restrict);

    builder.Entity<FileEntity>()
        .HasOne(f => f.Folder)
        .WithMany(f => f.Files)
        .HasForeignKey(f => f.FolderId)
        .OnDelete(DeleteBehavior.Restrict);

    builder.Entity<FileEntity>()
        .HasMany(f => f.Tags)
        .WithMany(t => t.Files)
        .UsingEntity(j => j.ToTable("FileTags"));

    builder.Entity<ShareLinkEntity>()
        .HasOne(s => s.File)
        .WithMany(f => f.ShareLinks)
        .HasForeignKey(s => s.FileId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<ShareLinkEntity>()
        .HasOne(s => s.Folder)
        .WithMany()
        .HasForeignKey(s => s.FolderId)
        .OnDelete(DeleteBehavior.Cascade);

    builder.Entity<FileVersionEntity>()
        .HasOne(v => v.File)
        .WithMany(f => f.Versions)
        .HasForeignKey(v => v.FileId)
        .OnDelete(DeleteBehavior.Cascade);
}
```

Рисунок Б.1 – Метод OnModelCreating в класі ApplicationDbContext

```
public static async Task SeedRolesAndAdminAsync(IServiceProvider serviceProvider)
{
    var roleManager = serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();
    var userManager = serviceProvider.GetRequiredService<UserManager<ApplicationUser>>();

    var configuration = serviceProvider.GetRequiredService<IConfiguration>();
    var logger = serviceProvider.GetRequiredService<ILoggerFactory>().CreateLogger("DbSeeder");

    string[] roleNames = { "Admin", "User" };
    foreach (var roleName in roleNames)
    {
        if (!await roleManager.RoleExistsAsync(roleName))
        {
            await roleManager.CreateAsync(new IdentityRole(roleName));
            logger.LogInformation($"Role '{roleName}' successfully created.");
        }
    }
    var adminEmail = configuration["AdminSettings:Email"];
    var adminPassword = configuration["AdminSettings:Password"];

    if (string.IsNullOrWhiteSpace(adminEmail) || string.IsNullOrWhiteSpace(adminPassword))
    {
        logger.LogWarning("Credits not found.");
        return;
    }

    var adminUser = await userManager.FindByEmailAsync(adminEmail);

    if (adminUser == null)
    {
        var newAdmin = new ApplicationUser
        {
            UserName = adminEmail,
            Email = adminEmail,
            EmailConfirmed = true
        };

        var creatPowerUser = await userManager.CreateAsync(newAdmin, adminPassword);
        if (creatPowerUser.Succeeded)
        {
            await userManager.AddToRoleAsync(newAdmin, "Admin");
            logger.LogInformation($"Admin '{adminEmail}' successfully created.");
        }
        else
        {
            logger.LogError($"Error creating admin '{adminEmail}':" +
                $" {string.Join(", ", creatPowerUser.Errors.Select(e => e.Description))}");
            foreach (var error in creatPowerUser.Errors)
            {
                logger.LogError($"Error: {error.Code} - {error.Description}");
            }
        }
    }
```

Рисунок Б.2 – Метод SeedRolesAndAdminAsync в класі DbSeader

```
public async Task<IEnumerable<FileResponseDto>> GetFilesAsync(int? folderId = null)
{
    var userId = _currentUserService.UserId
        ?? throw new ValidationException("User must be authenticated to view files.");

    var files = await _fileRepository.GetByFolderIdAsync(folderId, userId);

    return files.Select(f => new FileResponseDto
    {
        Id = f.Id,
        Name = f.Name,
        Size = f.Size,
        Extension = f.Extension,
        FolderId = f.FolderId,
        CreatedAt = f.CreatedAt
    });
}

public async Task<IEnumerable<FileResponseDto>> SearchFilesAsync(FileSearchParametersDto dto)...

public async Task HardDeleteFileAsync(Guid fileId)
{
    var userId = _currentUserService.UserId
        ?? throw new ValidationException("User must be authenticated to delete file.");

    var file = await _fileRepository
        .GetByIdIncludingDeletedAsync(fileId, userId)
        .EnsureExistsAsync(() => $"File with id {fileId} not found.");

    await _shareLinkRepository.DeleteLinksForFileAsync(fileId);

    await _userRepository.UpdateUserQuotaAsync(userId, -file.Size);

    await _fileStorageService.DeleteFileAsync(file.PhysicalPath);

    _fileRepository.Delete(file);

    await _fileRepository.SaveChangesAsync();
}

public async Task SoftDeleteFileAsync(Guid fileId)
{
    var userId = _currentUserService.UserId
        ?? throw new ValidationException("User must be authenticated to delete file.");

    var file = await _fileRepository
        .GetByIdWithTagsAsync(fileId, userId)
        .EnsureExistsAsync(() => $"File with id {fileId} not found.");

    file.DeletedAt = DateTime.UtcNow;

    await _shareLinkRepository.DeleteLinksForFileAsync(fileId);

    await _fileRepository.SaveChangesAsync();
}
```

Рисунок Б.3 – Методи для отримання файлів та їх видалення в класі FileService

```
[HttpPost]
[Route("{id}/versions/{versionNumber}/restore")]
[Audit("File.RestoreVersion", "File")]
public async Task<IActionResult> RestoreFileVersion(Guid id, int versionNumber)
{
    await _fileService.RestoreFileVersionAsync(id, versionNumber);

    return Ok();
}

[HttpGet]
[Route("{id}/download-ticket")]
public IActionResult GetDownloadTicket(Guid id)
{
    var userId = User.FindFirstValue(ClaimTypes.NameIdentifier);
    if (userId == null) return Unauthorized();

    var ticket = Guid.NewGuid().ToString("N");

    _cache.Set($"FileTicket_{id}_{ticket}", userId, TimeSpan.FromSeconds(60));

    return Ok(new { Ticket = ticket });
}

[AllowAnonymous]
[HttpGet]
[Route("download/{id}")]
public async Task<IActionResult> DownloadFile(Guid id, [FromQuery] string ticket)
{
    var cacheKey = $"FileTicket_{id}_{ticket}";
    if (string.IsNullOrEmpty(ticket) || !_cache.TryGetValue(cacheKey, out string? userId))
    {
        return Unauthorized("Invalid or expired download ticket.");
    }

    _cache.Remove(cacheKey);

    var result = await _fileService.DownloadFileAsync(id, userId!);

    return File(result.FileStream, result.ContentType, result.FileName);
}

[HttpGet]
[Route("get-files")]
public async Task<IActionResult> GetFiles([FromQuery] int? folderId = null)
{
    var files = await _fileService.GetFilesAsync(folderId);

    return Ok(files);
}
```

Рисунок Б.4 – Метод контролера FileController