Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

(повна назва)

Кафедра програмної інженерії

(повна назва)

**КВАЛІФІКАЦІЙНА РОБОТА**

**Пояснювальна записка**

рівень вищої освіти перший (бакалаврський)

Програмна система для хмарного зберігання особистих файлів.

Бек-енд

(тема)

Виконав:

здобувач 4 року навчання

групи ПЗПІ-21-2

Ігор БУЦУЛІН

(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного\_ \_\_

забезпечення

(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Програмна інженерія

(повна назва освітньої програми)

Керівник ст.викл. кафедри ПІ Олена ОЛІЙНИК

(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри \_\_\_\_\_\_\_\_\_\_\_ \_Кирило СМЕЛЯКОВ

(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 – Інженерія програмного забезпечення

Тип програми Освітньо-професійна

Освітня програма Програмна Інженерія

(шифр і назва)

ЗАТВЕРДЖУЮ:

Зав. кафедри \_\_\_\_\_\_\_\_\_

(підпис)

« » 2025 р.

**ЗАВДАННЯ**

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Буцуліну Ігорю Олександровичу

(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система для хмарного зберігання особистих файлів. Бек-енд

Затверджена наказом по університету від 19.05.2025р. № 397 Ст

2. Термін подання студентом роботи до екзаменаційної комісії \_\_\_\_\_\_\_\_\_ \_

3. Вихідні дані до роботи Розробити серверну частину комплексної програмної системи для хмарного зберігання особистих файлів. Використовувати мову програмування Go, СУБД MongoDB.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проектування програмного забезпечення, опис прийнятих програмних рішень, тестування розробленого програмного забезпечення, висновки, перелік джерел посилань, додатки.

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Назва етапів роботи | Термін виконання етапів роботи | Примітка |
| 1 | Аналіз предметної галузі | 20.05.2025 | *виконано* |
| 2 | Створення специфікації ПЗ | 22.05.2025 | *виконано* |
| 3 | Проектування ПЗ | 24.05.2025 | *виконано* |
| 4 | Розробка ПЗ | 08.05.2025 | *виконано* |
| 5 | Тестування ПЗ | 09.05.2025 | *виконано* |
| 6 | Оформлення пояснювальної записки | 11.06.2025 | *виконано* |
| 7 | Підготовка презентації та доповіді | 13.06.2025 | *виконано* |
| 8 | Попередній захист | 14.06.2025 | *виконано* |
| 9 | Нормоконтроль, рецензування | 15.06.2025 | *виконано* |
| 10 | Здача роботи у електронний архів | 16.06.2025 | *виконано* |
| 11 | Допуск до захисту у зав. кафедри | 19.06.2025 | *виконано* |

Дата видачі завдання 20 травня 2025р.

Здобувач \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ігор БУЦУЛІН\_

(підпис)

Керівник роботи ст.викл. кафедри ПІ Олена ОЛІЙНИК

(підпис) (посада, прізвище, ініціали)

**РЕФЕРАТ/ABSTRACT**

Пояснювальна записка до кваліфікаційної роботи бакалавра містить: 84 с., 16 рис., 16 джерел.

АУТЕНТИФІКАЦІЯ, БЕЗПЕКА, ЗБЕРІГАННЯ ДАНИХ, КОНТРОЛЬ ДОСТУПУ, ОБМІН ФАЙЛАМИ, ОСОБИСТІ ФАЙЛИ, СЕРВЕРНА ЧАСТИНА, УПРАВЛІННЯ ФАЙЛАМИ, ХМАРНЕ СХОВИЩЕ, BACK-END, GO, MONGODB.

Об'єктом розробки є серверна частина програмної системи для хмарного зберігання особистих файлів. Об'єктом дослідження є процеси розробки та тестування серверної частини системи хмарного зберігання, включаючи зберігання даних, аутентифікацію користувачів, управління файлами та контроль доступу.

Метою роботи є реалізація надійної та функціональної серверної частини, здатної забезпечити безпечне зберігання, організацію та обмін особистими файлами користувачів через веб- та мобільний інтерфейси. Для досягнення поставленої мети були використані наступні методи: аналіз предметної області, формування вимог до програмної системи, проектування архітектури програмного забезпечення, розробка з використанням мови програмування Go та бази даних MongoDB, а також тестування розробленого програмного забезпечення.

В результаті виконання було розроблено та протестовано серверний компонент програмної системи для хмарного зберігання особистих файлів. Реалізовано основні функціональні можливості, включаючи завантаження, зберігання, видалення, переміщення, перейменування та копіювання файлів, а також надання доступу до файлів за посиланням з різними рівнями приватності.

ACCESS CONTROL, AUTHENTICATION, BACK-END, CLOUD STORAGE, DATA STORAGE, FILE MANAGEMENT, FILE SHARING, GO, MONGODB, PERSONAL FILES, SECURITY, SERVER-SIDE.

The object of development of the qualification work is the server-side part of a software system for cloud storage of personal files. The object of research is the development and testing processes of the server-side of the cloud storage system, including data storage, user authentication, file management, and access control.

The aim of the work is to implement a reliable and functional server-side capable of ensuring secure storage, organization, and sharing of users' personal files through web and mobile interfaces. To achieve this goal, the following methods were used: analysis of the subject area, requirements engineering for the software system, software architecture design, development using the Go programming language and the MongoDB database, and testing of the developed software.

As a result of the qualification work, the server component of a software system for cloud storage of personal files was developed and tested. The main functionalities were implemented, including uploading, storing, deleting, moving, renaming, and copying files, as well as providing access to files via links with different levels of privacy.

**ЗМІСТ**

[Вступ 7](#_Toc200135260)

[1 Аналіз предметної галузі 9](#_Toc200135261)

[1.1 Аналіз предметної галузі 9](#_Toc200135262)

[1.2 Аналіз ринку 10](#_Toc200135263)

[1.3 Постановка задачі 17](#_Toc200135264)

[2 Формування вимог до програмної системи 19](#_Toc200135265)

[2.1 Формування мети 19](#_Toc200135266)

[2.2 Загальний опис системи 20](#_Toc200135267)

[2.3 Основна функціональність системи 21](#_Toc200135268)

[2.4 Обмеження 22](#_Toc200135269)

[2.5 Припущення та залежності 23](#_Toc200135270)

[3 Архітектура та проєктування 24](#_Toc200135271)

[3.1 UML проєктування ПЗ 24](#_Toc200135272)

[3.2 Проектування розгортання ПЗ 26](#_Toc200135273)

[3.3 Проектування бази даних 28](#_Toc200135274)

[3.4 Проєктування взаємодії між частинами системи 30](#_Toc200135275)

[4 Опис прийнятих програмних рішень 35](#_Toc200135276)

[4.1 Опис програмної реалізації сервісу файлів 35](#_Toc200135277)

[4.2 Опис програмної реалізації сервісу метаданих 44](#_Toc200135278)

[4.3 Опис програмної реалізації сервісу автентифікації 52](#_Toc200135279)

[5 Тестування розробленого програмного забезпечення 57](#_Toc200135280)

[5.1 Обґрунтування вибору видів тестування 57](#_Toc200135281)

[5.2 Опис тестування 59](#_Toc200135282)

[Висновки 65](#_Toc200135283)

[Перелік джерел посилання 67](#_Toc200135284)

**ВСТУП**

У сучасному цифровому світі зростання обсягів особистих файлів, таких як фотографії, відеозаписи та документи, є стрімкою тенденцією. Користувачі все частіше стикаються з необхідністю мати надійний та зручний спосіб зберігання, доступу та обміну цими даними незалежно від використовуваного пристрою та місця знаходження. Існуючі на даний момент рішення для зберігання файлів часто мають певні обмеження, включаючи складність у використанні на різних платформах, недостатню швидкість попереднього перегляду медіаконтенту або ж не завжди інтуїтивно зрозумілий процес обміну файлами з іншими користувачами.

Враховуючи зазначені потреби, актуальність розробки програмної системи для хмарного зберігання особистих файлів є беззаперечною, оскільки користувачі потребують інтуїтивно зрозумілого, безпечного та високоефективного інструменту, який би дозволяв їм легко керувати своїм цифровим контентом, отримувати миттєвий доступ до нього з будь-якого пристрою та зручно обмінюватися файлами з родиною, друзями чи колегами. Відповідно до дослідження, попит на хмарні сервіси зберігання даних продовжує зростати, що підкреслює актуальність розробки ефективних та зручних у використанні систем [3]. Особливо важливим є забезпечення швидкого попереднього перегляду аудіо- та відеофайлів, що значно покращує користувацький досвід при роботі з мультимедійним контентом.

Метою даної кваліфікаційної роботи є розробка серверної частини програмної системи для хмарного зберігання особистих файлів, яка б відповідала сучасним вимогам користувачів щодо зручності, безпеки та швидкодії.

Для досягнення поставленої мети необхідно вирішити наступні завдання:

* провести аналіз предметної області та існуючих рішень для хмарного зберігання даних;
* сформувати вимоги до серверної частини розроблюваної системи;
* розробити архітектуру серверної частини системи, враховуючи необхідність ефективної обробки та зберігання файлів;
* реалізувати основні функціональні можливості серверної частини, включаючи завантаження, зберігання, видалення та обмін файлами;
* провести тестування розробленої серверної частини програмної системи.

Результати даної роботи можуть бути застосовані в якості основи для створення повноцінної програмної системи хмарного зберігання особистих файлів, доступної для широкого кола користувачів. Розроблена серверна частина може використовуватися як самостійний модуль або бути інтегрована з іншими сервісами та платформами, що надають послуги зберігання та обміну даними.

# 1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

## 1.1 Аналіз предметної галузі

У сучасному інформаційному суспільстві спостерігається стрімке зростання обсягів цифрового контенту, що генерується окремими користувачами. До цього контенту належать різноманітні файли, включаючи фотографії, відеозаписи, аудіофайли, текстові документи та інші цифрові матеріали, які мають особисту або професійну цінність. Збільшення кількості та розміру таких файлів зумовлює зростаючу потребу в надійних та доступних рішеннях для їх зберігання, керування та обміну.

Існуючі на сьогоднішній день підходи до зберігання особистих файлів включають локальне зберігання на персональних пристроях, використання зовнішніх носіїв інформації та застосування хмарних сервісів. Локальне зберігання є обмеженим обсягом пам'яті пристрою та створює ризики втрати даних у випадку фізичного пошкодження або втрати пристрою. Зовнішні носії інформації забезпечують більший обсяг зберігання, але їх використання може бути незручним з точки зору мобільності та доступу до файлів з різних пристроїв.

У цьому контексті хмарні сервіси зберігання файлів набувають все більшої популярності, оскільки вони пропонують користувачам значні переваги, включаючи можливість доступу до своїх даних з будь-якого пристрою, підключеного до мережі Інтернет, автоматичне резервне копіювання даних, а також можливість обміну файлами з іншими користувачами. Незважаючи на наявність широкого спектру хмарних сервісів, користувачі продовжують відчувати потребу в більш ефективних та функціональних рішеннях, особливо в частині швидкодії, надійності та можливостей попереднього перегляду мультимедійного контенту.

Зокрема, важливим аспектом є забезпечення швидкого та зручного попереднього перегляду аудіо- та відеофайлів без необхідності їх повного завантаження. Це дозволяє користувачам оперативно оцінювати вміст файлів та економити трафік. Реалізація такої функціональності покладається на ефективну роботу серверної частини системи, яка відповідає за обробку запитів користувачів, керування файлами та забезпечення їхньої цілісності та доступності.

В умовах військового стану в Україні питання надійності та доступності цифрової інформації набуває особливої гостроти. Забезпечення безперебійного доступу до важливих особистих файлів, можливість їх швидкого обміну та відновлення у випадку втрати або пошкодження стають критично важливими для громадян. У зв'язку з цим зростає потреба в розробці та впровадженні стійких та ефективних систем хмарного зберігання даних, здатних функціонувати в умовах нестабільного зв'язку та підвищених ризиків кібербезпеки.

1.2 Аналіз ринку

З метою всеосяжного розуміння поточного запиту на ринку сервісів хмарного зберігання файлів було проведено аналіз конкурентів. Серед питань, щодо яких під час аналізу була виявлена особлива зацікавленість, було визначено кількість користувачів, наявні для користувача функціоналі можливості, загальна швидкодія застосунку, швидкість завантаження-вивантаження даних, стійкість системи до несправностей, кількість виходів з ладу за певний період, фінансова доступність використання зазначених систем.

В першу чергу, було проведено детальний аналіз найбільш популярного представника ринку: хмарне середовище Google Drive корпорації Alphabet Inc.

Google Drive є надзвичайно популярних рішенням для резервного зберігання, обміну та синхронізації файлів. Загальна кількість користувачів сервісу перевищує 1 мільярд [4]. Важливою конкурентною перевагою сервісу є глибока інтеграція сервісу з іншими сервісами Google, що не тільки розширює базу користувачів, а й робить користування продуктом більш зручним.

Google Drive надає користувачу широкі функціональні можливості та зручний веб інтерфейс (рис. 1.1).

Серед стандартних функціональних особливостей системи можна зазначити завантаження файлів, їх вивантаження, поширення, створення, видалення, редагування назв файлів та директорій, переміщення файлів між директоріями.

A screenshot of a computer

Description automatically generated

Рисунок 1.1 – Головний інтерфейс [2]

Надає сервіс також деякі унікальні особливості. Зокрема, завдячуючи глибинній інтеграції з іншими сервісами Google, застосунок надає можливість з попереднього перегляду документів, таблиць, презентацій, зображень, відео та аудіо (рис. 1.2). Також, надається можливість редагувати документи у іншому сервісі компанії: Google Docs, без локального завантаження файлу.

A screenshot of a computer

Description automatically generated

Рисунок 1.2 – Інтерфейс попереднього перегляду документа [2]

Швидкість підключення до систем продукту та швидкість завантаження та вивантаження файлів є високою. За даними аналітиків команди Clowdwards, Google Drive займає друге місце за швидкістю підключення та завантаження даних серед популярних сервісів хмарного зберігання даних [2]. Середньою швидкістю вивантаження даних на сервіс є 1.47 хв/Гб, середньою швидкістю завантаження даних 1.456 хв/Гб.

Загальна швидкодія веб застосунку є посередньою. Важливою проблемою у користуванні веб застосунком постає перевантаженість інтерфейсу інтеграціями з іншими сервісами Google, що хоча й може здатися позитивною рисою, значно погіршує продуктивність роботи програми. Це може значно погіршити досвід користування сервісом на комп’ютерах з невисокими характеристиками продуктивності.

Рівень доступності сервісу є високим. Google Drive, як і інші сервіси даного розробника, використовує власті обчислювальні потужності та потужності зберігання, що мають високу якість та зносостійкість. Також потужності сервісу є сильно децентралізованими, активно застосовується резервне копіювання даних по всьому світу.

Станом на травень 2025 року достеменно відомо лише про один серйозний інцидент відсутності доступу до сервісу за останні два роки [1].

Вартість сервісу є його негативною рисою. Можливість оплати лише послуг сервісу Google Drive відсутня, доступна лише можливість покупки підписки на пакет сервісів Google One, що, окрім Google Drive, включають додатково багато інших, іноді зовсім непов’язаних сервісів Google (рис. 1.3).

A screenshot of a phone

Description automatically generated

Рисунок 1.3 – Поточна вартість на підписки Google One [1]

Такий підхід робить продукт дорогим. Також, ціни для побутових користувачів та компаній досить сильно відрізняються. З позитивних рис, сервіс надає можливість зберігати 15 Гб даних безкоштовно без обмежень в часі та якості зберігання даних.

Також, було проведено аналіз іншого відомого сервісу: Microsoft OneDrive корпорації Microsoft.

Microsoft OneDrive є достатньо популярним рішенням для синхронізації, резервного копіювання та обміну даними. Загальна кількість користувачів перевищує 300 мільйонів [5]. Важливо зазначити, що, як і Google Drive, Microsoft OneDrive має глибинну інтеграцію з сервісами батьківської корпорації: Microsoft, такими як Office, SharePoint, та ін.

Microsoft OneDrive надає повні стандартні функціональні можливості за допомогою веб інтерфейсу (рис. 1.4).

A screenshot of a computer

Description automatically generated

Рисунок 1.4 – Головне меню сервісу Microsoft OneDrive[5]

Окрім стандартного набору можливостей для даної категорії застосунків, сервіс надає також деякі унікальні особливості.

Зокрема, завдяки глибинній інтеграції з сервісами Microsoft Office, надається можливість перегляду та редагування документів форматів Microsoft Office прямо у браузері без локального завантаження файлів або додатків пакету Office на комп’ютер (рис. 1.5).

Швидкість підключення до систем продукту, швидкість вивантаження файлів та загальна швидкодія веб додатку є посередньою, швидкість завантаження даних є високою. За даними аналітиків команди Clowdwards, Microsoft OneDrive займає десяте місце за швидкістю підключення та завантаження даних серед популярних сервісів хмарного зберігання даних [2]. Середньою швидкістю вивантаження даних на сервіс є 1.84 хв/Гб, середньою швидкістю завантаження даних 1.462 хв/Гб.

A screenshot of a computer

Description automatically generated

Рисунок 1.5 – Інтеграція Microsoft OneDrive з Microsoft Office Online [2]

Веб застосунок є перевантаженим та працює досить повільно. Якість роботи сервісу на комп’ютерах з невисокими характеристиками є низькою.

Рівень доступності сервісу є високим. Microsoft OneDrive, як і інші сервіси даного розробника, використовує власті обчислювальні потужності та потужності зберігання, що мають високу якість та зносостійкість. Також, активно застосовується резервне копіювання даних, для збереження даних користувачів.

Станом на жовтень 2024 року достеменно відомо лише про один серйозний інцидент відсутності доступу до сервісу за останні два роки [1].

Вартість сервісу є його негативною рисою. Можливість оплати лише послуг сервісу Microsoft OneDrive відсутня, доступна лише можливість покупки підписки на пакет сервісів Microsoft 365, що, окрім Microsoft OneDrive, включають додатково багато інших, іноді зовсім непов’язаних сервісів Microsoft (рис. 1.6).

Такий підхід робить продукт дорогим. Також, ціни для побутових користувачів та компаній досить сильно відрізняються. З позитивних рис, сервіс надає можливість зберігати 5 Гб даних безкоштовно без обмежень в часі та якості зберігання даних.

A screenshot of a phone

Description automatically generated

Рисунок 1.6 – Вартість підписки Microsoft 365

Окрім цього, було проаналізовано сервіс Koofr.

Koofr є сервісом, що має невисоку популярність. Загальна кількість активних користувачів є невідомою, однак не перевищує мільйона.

Koofr надає повні стандартні функціональні можливості за допомогою веб інтерфейсу (рис. 1.7).

A screenshot of a computer

Description automatically generated

Рисунок 1.7 – Головне меню Koofr [3]

У застосунку доступні інтеграції з іншими відомими хмарними сервісами. Заявлено, що завдяки цьому можна керувати декількома хмарними середовищами одночасно (рис. 1.8).

Однак практичної користі від цього небагато, оскільки унікальні функції різних хмарних сховищ у разі такого формату використання стають недоступні.

Швидкість підключення до систем продукту та швидкість вивантаження файлів є високою, швидкість завантаження даних є посередньою. За даними аналітиків команди Clowdwards, Koofr займає третє місце за швидкістю підключення та завантаження даних серед популярних сервісів хмарного зберігання даних [2]. Середньою швидкістю вивантаження даних на сервіс є 0.9625 хв/Гб, середньою швидкістю завантаження даних 1.66 хв/Гб.

A screenshot of a computer

Description automatically generated

Рисунок 1.8 – Інтеграція з іншими сервісами хмарного зберігання [2]

Загальна швидкодія веб застосунку є високою. Інтерфейс додатку є зручним, простим та не перевантаженим.

Рівень доступності сервісу є високим. Сервіс використовує орендовані потужності європейський провайдерів, забезпечуючи високу надійність та безпеку даних.

Станом на жовтень 2024 року достеменно невідомо про жоден серйозний інцидент з недоступності доступу до сервісу.

Застосунок надає широке різноманіття тарифів, орієновані на різні потреби користувачів (рис. 1.9).

A screenshot of a screen

Description automatically generated

Рисунок 1.9 – Вартість підписки Koofr [2]

Доступні як тарифи лише з оплатою додаткових ресурсів зберігання даних, так і комплексні тарифи з інтеграцією з іншими сервісами, зокрема редакторами документів пакету Office. Також, сервіс надає безкоштовні 10 Гб для зберігання особистих файлів.

Провівши комплексний аналіз існуючої на ринку пропозиції сервісів хмарного зберігання, було зроблено висновок про необхідність надання, окрім загальної функціональності зазначеного типу сервісів, можливості попереднього перегляду широкого типу файлів, забезпечення високої надійності та доступності роботи, високої швидкості завантаження та вивантаження даних, загальної швидкодії сервісу. Для забезпечення цього, визначено, що сервіс має бути децентралізованим.

1.3 Постановка задачі

У контексті зростання обсягів цифрового контенту, що генерується користувачами, та наявності низки вимог користувачів, що залишаються недостатньо задоволеними, виникає об'єктивна потреба у розробці надійної та високоефективної програмної системи для хмарного зберігання особистих файлів. У зв'язку з цим постає задача розробки серверної частини програмної системи, яка б забезпечила користувачам можливість безпечного зберігання, керування та обміну файлами.

Відповідно, постає завдання забезпечити надійне зберігання файлів користувачів та їх метаданих. Користувач зможе завантажувати файли різних форматів до персонального хмарного сховища. Необхідним стає надання можливості керування файлами, включаючи їх видалення, переміщення, перейменування та копіювання. Важливим аспектом є забезпечення контролю доступу до файлів, що дозволить користувачу визначати рівень приватності для кожного файлу або групи файлів.

Зважаючи на потреби користувачів у швидкому доступі до контенту, особливу увагу слід приділити оптимізації обробки та передачі даних. Користувач буде мати змогу швидко отримувати доступ до збережених файлів. Для покращення користувацького досвіду при роботі з мультимедійним контентом, варто забезпечити ефективний механізм для попереднього перегляду аудіо- та відеофайлів.

Окрім основних функцій зберігання та керування файлами, важливою є інтеграція з іншими компонентами програмної системи, зокрема з клієнтською веб-частиною та мобільним застосунком, для забезпечення повноцінного доступу користувачів до своїх файлів з різних пристроїв. Таким чином, розробка серверної частини є ключовим етапом у створенні комплексної програмної системи для хмарного зберігання особистих файлів, що відповідає сучасним вимогам користувачів.

**2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ**

2.1 Формування мети

Метою даної кваліфікаційної роботи є розробка серверної частини програмної системи, призначеної для забезпечення хмарного зберігання особистих файлів користувачів. Основною метою розробки зазначеного застосунку є створення надійного, безпечного та високоефективного інструменту, який дозволить користувачам ефективно керувати своїм цифровим контентом.

У процесі розробки передбачається створення системи, яка надаватиме користувачам можливість безперешкодного доступу до їхніх файлів з будь-якого пристрою, що має підключення до мережі Інтернет. Це дозволить забезпечити зручність зберігання, організації та обміну файлами незалежно від географічного розташування користувача. Розробка системи орієнтована на задоволення потреб широкого кола користувачів у зберіганні різноманітного особистого цифрового контенту.

Однією з ключових цілей розробки є забезпечення високої швидкодії системи при обробці запитів користувачів та передачі значних обсягів даних, включаючи особисті фотографії, відеозаписи, документи тощо. З метою оптимізації швидкості доступу до часто використовуваного контенту передбачається реалізація механізму серверного кешування. Це є критично важливим для забезпечення комфортної роботи з системою, особливо при необхідності попереднього перегляду медіаконтенту [11].

Для забезпечення стабільності та ефективності майбутньої системи, розробка її серверної частини буде здійснюватися з урахуванням необхідності відмови від монолітної архітектури на користь децентралізованого підходу. Планується застосування мікросервісної архітектури, що має підвищити стійкість системи до відмов, забезпечити її гнучке масштабування та спростити подальшу підтримку та модернізацію.

Для забезпечення цілісного та зручного користувацького досвіду, важливим аспектом є досягнення нативної взаємодії між розроблюваною серверною частиною та іншими складовими системи, зокрема з інтерфейсами користувача для веб- та мобільних платформ.

2.2 Загальний опис системи

Розроблена програмна система є клієнт-серверним застосунком, призначеним для організації хмарного зберігання особистих файлів користувачів. Архітектура серверної частини реалізована як мікросервісна, що забезпечує високу відмовостійкість, гнучкість масштабування та спрощує подальшу підтримку й модернізацію системи. Система складається з трьох основних незалежних сервісів: сервісу метаданих файлів, сервісу файлів та сервісу автентифікації.

Серверна частина системи буде нативно інтегрована з іншими компонентами, зокрема з клієнтською веб-частиною та мобільним застосунком. Це забезпечить користувачам можливість зручного та безпечного доступу до їхніх особистих файлів незалежно від пристрою та місця знаходження.

Система реалізує ключові функції для ефективної роботи з файловим контентом. До них належать: перегляд файлів та директорій у вигляді списку, попередній перегляд різних типів файлів, включаючи текстові документи, PDF-файли, зображення, аудіо- та відеоматеріали. Також доступна функціональність отримання детальної інформації про файл, такої як дата завантаження, оновлення та розмір. Система підтримує пошук файлів за назвою, фільтрацію за типом та датою створення, а також сортування за різними критеріями. Користувачі можуть здійснювати управління файлами, що охоплює завантаження файлів з пристрою користувача та вивантаження на пристрій, видалення, переміщення, перейменування та копіювання файлів. Особлива увага приділена наданню доступу до файлів за посиланням з різними рівнями приватності: приватний (лише для власника), обмежений (для визначеного списку користувачів за їхньою електронною поштою) та публічний (для всіх користувачів). Система розрізняє авторизованих та неавторизованих користувачів, надаючи відповідні функціональні можливості, зокрема реєстрацію, автентифікацію та авторизацію для неавторизованих, та повний набір функцій управління файлами для авторизованих.

Для реалізації серверної частини обрано мову програмування Go, яка відрізняється високою продуктивністю, ефективним керуванням пам'яттю та підтримкою паралельних обчислень [15]. Це забезпечує швидку обробку запитів та високу швидкодію системи при роботі зі значними обсягами даних. Сервіси метаданих та автентифікації побудовані з використанням фреймворку go-fiber, який відомий своєю швидкістю та простотою використання для створення вебсерверів [12]. Для сервісу файлів застосовується бібліотека chi, що є легкою та потужною для маршрутизації HTTP-запитів та ефективної обробки файлових операцій [13]. В якості системи керування базами даних для сервісів метаданих та автентифікації використовується MongoDB. Ця документо-орієнтована база даних забезпечує гнучкість схеми даних, що є перевагою при зберіганні різноманітної метаінформації та облікових даних користувачів [14]. Для забезпечення доступу до мікросервісів з боку клієнтських застосунків використовується вебсервер caddy. Caddy автоматично керує сертифікатами TLS, підвищуючи безпеку передачі даних, та виступає єдиною точкою входу до системи [10].

Однією з важливих функцій сервісу файлів є реалізація серверного кешування файлів. Це дозволяє зберігати копії файлів, які часто запитуються, у швидкій пам'яті, що значно прискорює доступ до них, особливо при попередньому перегляді великих медіафайлів, таких як відео та аудіо. Такий підхід суттєво покращує користувацький досвід при роботі з мультимедійним контентом.

2.3 Основна функціональність системи

Розроблювана програмна система розрізняє лише два типи користувачів: авторизований та неавторизований користувач.

Неавторизованому користувачу будуть надаватися наступні функціональні можливості:

* реєстрація нового користувача;
* аутентифікація та авторизація користувача.

Авторизованому користувачу ж буде надано такий набір функціональних можливостей:

* завантаження файлу з сервера;
* вивантаження файлу на сервер;
* зміна назви файлу;
* зміна прав доступу до файлу;
* переміщення файлу;
* пошук файлів;
* фільтрація за типом файлу, датою створення файлу;
* створення директорій;
* видалення директорій;
* зміна назви директорії;
* зміна прав доступу до директорії;
* переміщення директорії;
* пошук директорій;
* сортування за назвою файлу та директорії, датою оновлення, розміром файлу.

Таке розділення дозволяє ефективно керувати доступами до файлів та їх метаданих.

## 2.4 Обмеження

Під час розробки програмної системи для хмарного зберігання особистих файлів були визначені наступні обмеження:

* користувач зможе отримати доступ до файлів лише маючи неперервний доступ до мережі Інтернет;
* користувач може отримати доступ до файлу іншого користувача лише після надання прямого дозволу;
* збереження файлів користувачів можливе на серверах, наданих третіми сторонами.

Відповідні обмеження дозволяють системі залишатися безпечною, підвищує її швидкодію.

2.5 Припущення та залежності

Під час розробки програмної системи для хмарного зберігання особистих файлів були визначені наступні припущення та залежності:

* користувач має мати встановлений браузер останньої версії сімейства Chromium або Firefox Quantum;
* користувач використовує останню версію операційної системи сімейств Microsoft Windows, Apple macOS або iOS, Google Android, GNU/Linux;
* постачальники послуг хмарного розгортання надають послуги цілодобово, без переривань.

# 3 АРХІТЕКТУРА ТА ПРОЄКТУВАННЯ

## 3.1 UML проєктування ПЗ

Розроблена програмна система є клієнт-серверним застосунком, призначеним для хмарного зберігання особистих файлів користувачів. Серверна частина системи спроєктована з використанням REST архітектурного стилю для взаємодії з клієнтськими частинами. В основі серверної архітектури лежить підхід мікросервісів, що передбачає розподіл функціональності між незалежними компонентами: сервісом автентифікації, сервісом метаданих та сервісом файлів. Такий підхід сприяє підвищенню відмовостійкості та гнучкості масштабування серверної інфраструктури [6].

Серверна частина забезпечує фундаментальну функціональність для керування даними користувачів. Реалізовано механізми автентифікації та авторизації, що дозволяють розмежовувати доступ до ресурсів системи, використовуючи автентифікацію Google. Інтеграція автентифікації Google (OAuth 2.0) централізує управління ідентифікацією користувачів, але вимагає ретельної обробки токенів доступу та оновлення токенів сервісом автентифікації для безпечного управління сесіями та авторизації доступу до інших мікросервісів [7].

Ключовою функціональністю є управління файлами та директоріями. Серверна частина дозволяє користувачам завантажувати файли зі свого пристрою до хмарного сховища та вивантажувати їх назад. Передбачено операції створення, видалення, перейменування та переміщення як файлів, так і директорій. Система підтримує копіювання файлів в межах сховища користувача. Для організації даних реалізовано відображення ієрархічної структури файлів та директорій.

Серверна частина надає інструменти для пошуку файлів та директорій за їх назвою у всіх директоріях користувача. Реалізовано можливість фільтрації файлів за їх типом та датою створення. Також система дозволяє сортувати файли та директорії за назвою, датою останнього оновлення та розміром файлу (рис. 3.1).

Важливим аспектом є забезпечення контрольованого доступу до файлів. Користувач має змогу ділитися своїми файлами з іншими, генеруючи посилання з різними рівнями доступу: приватний, що обмежує доступ лише власником; обмежений, що надає доступ визначеному списку користувачів за їх електронною поштою; та публічний, що дозволяє доступ будь-кому, хто має посилання. Відповідно, передбачено функціональність зміни прав доступу як для окремих файлів, так і для директорій.

A diagram of a diagram

AI-generated content may be incorrect.

Рисунок 3.1 – UML діаграма прецедентів (рисунок виконано самостійно)

Для покращення взаємодії користувача з контентом, серверна частина підтримує попередній перегляд файлів певних типів, зокрема текстових документів, PDF-файлів, зображень, аудіо- та відеоматеріалів. З метою оптимізації цього процесу, особливо для медіафайлів великого розміру, сервіс файлів реалізує механізм серверного кешування вмісту [9].

## 3.2 Проектування розгортання ПЗ

Розроблена програмна система являє собою клієнт-серверний застосунок, призначений для хмарного зберігання особистих файлів користувачів (рис. 4.2). Серверна частина системи реалізована з використанням мікросервісної архітектури, що забезпечує високу стійкість до відмов, гнучке масштабування та полегшує подальшу підтримку й модернізацію. В основу архітектури покладено три незалежні сервіси: сервіс метаданих файлів, сервіс файлів та сервіс автентифікації.

Сервіс метаданих файлів відповідає за зберігання та управління інформацією про файли користувачів, включаючи їхні назви, розміри, типи, дати створення та модифікації, а також інформацію про права доступу. Для реалізації цього сервісу використовується мова програмування Go, що відрізняється високою продуктивністю та ефективним керуванням пам'яттю, що є важливим для обробки великої кількості запитів. В якості фреймворку для побудови вебсервера застосовується go-fiber, який характеризується простотою використання та високою швидкістю обробки запитів, що сприяє швидкій роботі сервісу. Для зберігання метаданих використовується система керування базами даних MongoDB. MongoDB є документо-орієнтованою та забезпечує гнучкість схеми даних, що є корисним при роботі з різноманітними типами файлів та їхніми атрибутами [8].

Сервіс файлів відповідає за безпосереднє зберігання файлового контенту користувачів та реалізацію операцій з файлами, таких як завантаження, вивантаження, видалення, переміщення, перейменування та копіювання. Цей сервіс також розроблено на мові програмування Go, що забезпечує ефективну роботу з файловою системою та мережевими протоколами. Для реалізації вебсервера в даному сервісі застосовується бібліотека chi, яка є легкою та потужною, надаючи необхідний функціонал для маршрутизації запитів та обробки файлових операцій. Однією з ключових функцій сервісу файлів є серверне кешування, яке дозволяє зберігати копії часто запитуваних файлів у швидкіcній пам'яті, що значно прискорює їхнє отримання при повторних запитах, особливо при попередньому перегляді великих медіафайлів, таких як відео та аудіо.

Сервіс автентифікації забезпечує ідентифікацію та авторизацію користувачів, надаючи захищений доступ до їхніх файлів. Для розробки цього сервісу також використовується мова програмування Go, що гарантує високий рівень безпеки та швидкодії процесів автентифікації. Вебсервер сервісу реалізовано за допомогою фреймворку go-fiber, який забезпечує зручні інструменти для обробки запитів автентифікації та керування сесіями користувачів. Для зберігання облікових даних користувачів, таких як електронні адреси та хешовані паролі, використовується СУБД MongoDB, що забезпечує надійне та безпечне зберігання цієї критично важливої інформації (див.рис.3.2).

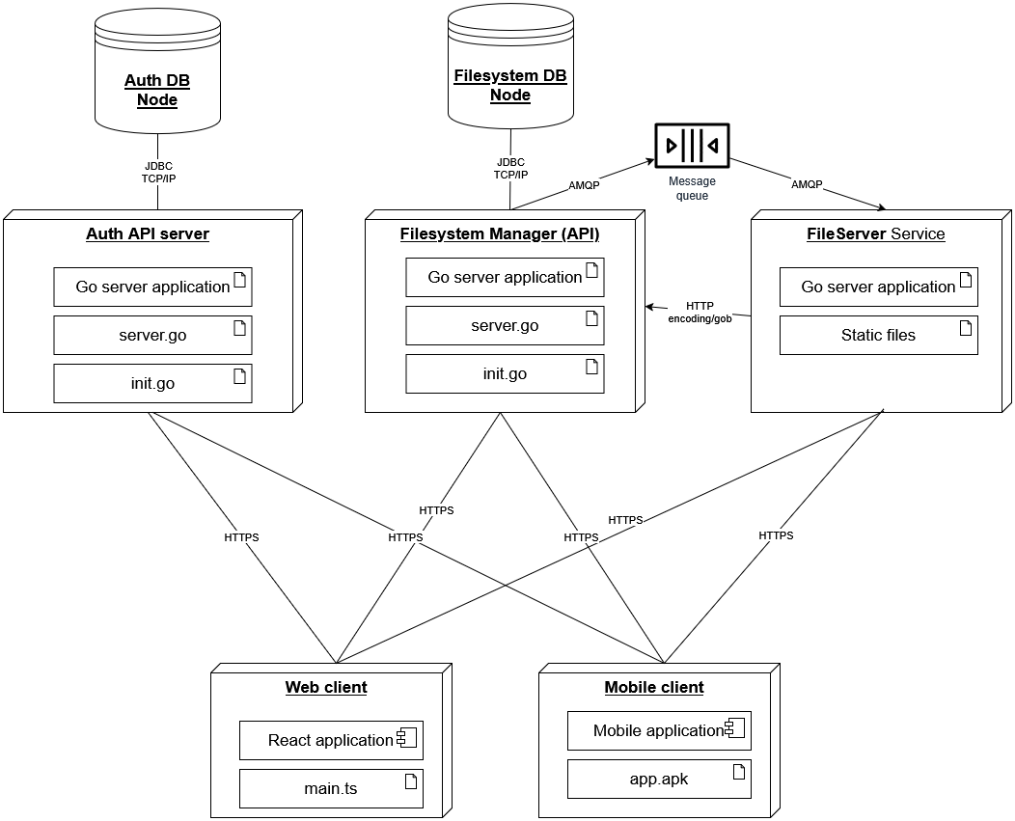


Рисунок 3.2 – UML діаграма розгортання системи (рисунок виконано самостійно)

Для забезпечення доступу до системи з боку клієнтських застосунків (веб-інтерфейсу та мобільного застосунку) використовується вебсервер caddy. Caddy є сучасним вебсервером, який автоматично керує сертифікатами TLS, що підвищує безпеку передачі даних між клієнтом та сервером. Він виступає в якості єдиної точки входу до всіх трьох мікросервісів, забезпечуючи їхню злагоджену роботу та спрощуючи взаємодію з клієнтською частиною.

## 3.3 Проектування бази даних

Проектування сховища даних для програмної системи хмарного зберігання особистих файлів базується на обраній мікросервісній архітектурі, що передбачає розподіл відповідальності між окремими компонентами системи. Зокрема, для сервісу автентифікації та сервісу метаданих було обрано систему керування базами даних MongoDB, тоді як сервіс файлів використовує файлову систему операційної системи для зберігання безпосередньо контенту файлів. Такий підхід дозволяє оптимізувати зберігання різних типів даних відповідно до їхньої структури та специфіки використання.

Сервіс автентифікації відповідає за безпечне зберігання облікових даних користувачів, необхідних для їх ідентифікації та авторизації в системі. Для цієї задачі використовується СУБД MongoDB, яка забезпечує гнучку схему даних, що є перевагою при роботі з інформацією про користувачів, та надає надійні механізми для зберігання чутливих даних, таких як електронні адреси та гешовані паролі. У базі даних цього сервісу зберігається сутність "Користувач", що містить унікальний ідентифікатор, ім'я, посилання на аватар, електронну пошту та часові мітки створення й оновлення запису, як показано на ER-діаграмі. Ця структура підтримує функціональні можливості реєстрації нових користувачів та їх подальшої аутентифікації. Вибір MongoDB для цього сервісу обумовлений також її високою продуктивністю та можливостями масштабування, що важливо для систем з потенційно великою кількістю користувачів (див.рис.3.3).

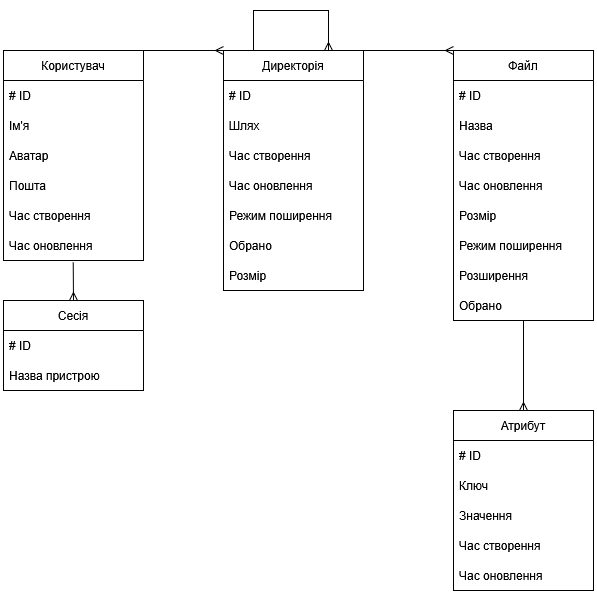


Рисунок 3.3 – UML ER-діаграма (рисунок виконано самостійно)

Сервіс метаданих є центральним компонентом для управління інформаційною структурою сховища. Його база даних, також реалізована на MongoDB, зберігає детальну інформацію про файли та директорії користувачів, включаючи їхні назви, типи, розміри, дати створення та модифікації, інформацію про власника та права доступу. MongoDB ефективно працює з ієрархічними структурами даних, що притаманні файловим системам (вкладені директорії), та дозволяє легко зберігати різноманітні атрибути файлів. Спроектована структура даних включає сутності "Файл" та "Директорія", які пов'язані з сутністю "Користувач". Для файлів зберігаються назва, часові мітки, режим поширення та інші атрибути. Для директорій зберігається шлях, часові мітки та режим поширення. Додаткова сутність "Атрибут", пов'язана з файлом, дозволяє зберігати розширювані пари "ключ-значення" для метаданих. Така організація даних підтримує широкий спектр операцій: відображення вмісту директорій, пошук за назвою, фільтрацію за типом та датою, сортування за назвою, датою та розміром, отримання детальної інформації про об'єкт, завантаження та вивантаження файлів (через взаємодію з сервісом файлів), створення, видалення, перейменування, переміщення, копіювання файлів та директорій, а також управління доступом за посиланням з різними рівнями приватності (приватний, публічний).

Сервіс файлів, хоч і не використовує традиційну базу даних для контенту, тісно взаємодіє з сервісом метаданих. Використано фізичне зберігання файлів у файловій системі сервера шляхом операції безпосереднього читання та запису даних.

## 3.4 Проєктування взаємодії між частинами системи

Дана система реалізована як клієнт-серверний застосунок, що використовує мікросервісну архітектуру та взаємодіє через програмний інтерфейс REST API. Архітектура серверної частини складається з трьох основних незалежно розгорнутих сервісів: сервісу автентифікації, сервісу метаданих, та сервісу файлів. Ефективна та безпечна робота системи залежить від чітко визначених механізмів взаємодії між цими компонентами. Відповідно, важливо детальніше розглянути дві ключові взаємодії: процес автентифікації користувача та обмін даними між сервісом метаданих і сервісом файлів.

Першою важливою взаємодією, що розглядається, є механізм автентифікації користувача. Процес автентифікації ініціюється клієнтським застосунком. Клієнт спочатку звертається до стороннього постачальника послуг ідентифікації, такого як Firebase або Google Identity API, від якого отримує ідентифікаційний маркер (ID-token). Цей ідентифікаційний маркер являє собою підписаний та закодований веб-маркер у форматі JSON (JWT), що засвідчує особу користувача. Отримавши такий маркер, клієнт формує запит до сервісу автентифікації розроблюваної системи, передаючи вказаний ідентифікаційний маркер.

При надходженні запиту від клієнта, сервіс автентифікації здійснює обробку запиту, що включає декілька ключових етапів. Насамперед, він здійснює валідацію наданого ідентифікаційного маркера. Ця перевірка включає контроль цілісності маркера, автентичності його підпису та актуальності терміну дії. У разі успішної валідації, сервіс автентифікації взаємодіє зі своєю внутрішньою базою даних для управління обліковими записами користувачів. Якщо користувач звертається до системи вперше, його дані, отримані з валідованого ідентифікаційного маркера, реєструються в базі даних. Для користувачів, що вже існують в системі, може відбуватися оновлення їхніх облікових даних на основі інформації з маркера (рис. 3.4).

A screenshot of a computer screen

AI-generated content may be incorrect.

Рисунок 3.4 – UML діаграма послідовностей (рисунок виконано самостійно)

Після успішного завершення процедур ідентифікації та збереження або оновлення даних користувача в базі даних, сервіс автентифікації генерує два типи власних веб-маркерів у форматі JWT: маркер доступу (Access Token) та маркер оновлення (Refresh Token). Маркер доступу є короткоживучим і містить інформацію про користувача, необхідні для авторизованої взаємодії з іншими сервісами системи, зокрема з сервісом метаданих. Маркер оновлення, на відміну від маркера доступу, є довгоживучим у межах поточної сесії взаємодії користувача з системою. Він зберігає дані, що стосуються поточної сесії. Основне призначення маркера оновлення полягає у наданні можливості безпечного оновлення маркера доступу після завершення терміну його дії, без необхідності повторного проходження користувачем повного циклу автентифікації через стороннього постачальника послуг ідентифікації. Окрім того, маркер оновлення може бути використаний для ініціювання процедури завершення поточної сесії користувача. Згенерована пара маркерів – доступу та оновлення – повертається клієнтському застосунку. Надалі клієнт використовує отриманий маркер доступу для здійснення авторизованих запитів до сервісу метаданих.

Крім того, ключовим аспектом функціонування розробленої системи є координація між сервісом метаданих, що виконує роль менеджера файлової системи, та сервісами файлів під час виконання операцій створення, редагування, перегляду або видалення файлів. Ця взаємодія побудована на основі асинхронного обміну повідомленнями з використанням брокера повідомлень RabbitMQ та синхронних HTTP-запитів для підтвердження операцій.

Процес взаємодії ініціюється, коли клієнт надсилає запит на виконання операції з файлом (наприклад, створення, модифікація, читання або видалення) до сервісу метаданих. Отримавши такий запит, сервіс метаданих формує відповідне повідомлення, яке містить унікальний ідентифікатор файлу, зворотний шлях до себе (адресу, за якою сервіс файлів може надіслати відповідь) та тип запитуваної операції. Це повідомлення публікується у чергу RabbitMQ. Важливо зазначити, що черга налаштована у режимі "fanout", що означає доставку повідомлення усім активним екземплярам сервісу файлів, підписаним на цю чергу. Такий підхід забезпечує можливість паралельної обробки запиту та вибору найбільш відповідного сервісу файлів для виконання операції.

Кожен екземпляр сервісу файлів, отримавши повідомлення з черги, аналізує його вміст та перевіряє власну спроможність виконати запитувану операцію. Критеріями для такої перевірки можуть бути наявність файлу з указаним ідентифікатором (у випадках операцій читання, редагування чи видалення) або наявність достатнього вільного простору у сховищі (для операції створення файлу). Якщо сервіс файлів визначає, що може обробити запит, він негайно ініціює синхронний HTTP-запит на адресу сервісу метаданих, що була вказана у повідомленні з черги. Цей HTTP-запит містить власну мережеву адресу сервісу файлів, унікальний ідентифікатор з'єднання, який буде використовуватися клієнтом для подальшої прямої взаємодії, та маркер, що індикує відсутність або наявність помилок під час попередньої перевірки.

Сервіс метаданих, отримавши HTTP-відповідь від одного з сервісів файлів, фіксує її. Якщо це перша успішна відповідь для даного запиту, сервіс метаданих надсилає у відповідь HTTP-статус код 204 (No Content). Це сигналізує сервісу файлів, що його пропозицію прийнято, і він може готуватися до безпосередньої взаємодії з клієнтом. Якщо ж сервіс метаданих отримує відповідь від іншого сервісу файлів, коли запит вже взято в обробку першим сервісом, або якщо перший сервіс, що відгукнувся, повідомив про помилку, сервіс метаданих може відповісти статус-кодом 205 (Reset Content). Цей статус інструктує сервіс файлів відхилити запит та, за необхідності, відкотити будь-які попередньо зроблені зміни, наприклад, закрити підготовлене з'єднання (рис. 3.5).

A screenshot of a computer program

AI-generated content may be incorrect.

Рисунок 3.5 – UML діаграма послідовностей (рисунок виконано самостійно)

Після успішного узгодження з одним із сервісів файлів та отримання від нього необхідних даних (його адреси та ідентифікатора з'єднання), сервіс метаданих повертає цю інформацію клієнту, який ініціював операцію. Отримавши ці дані, клієнт отримує можливість встановлювати пряме з'єднання з обраним сервісом файлів для передачі або отримання вмісту файлу, залежно від типу початкового запиту. Таким чином, основне навантаження, пов'язане з передачею файлових даних, розподіляється між екземплярами сервісу файлів, а сервіс метаданих виконує координуючу роль.

# 4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

## 4.1 Опис програмної реалізації сервісу файлів

Програмна реалізація сервісу файлів виконана з дотриманням принципів модульності та чіткого розподілу відповідальності між компонентами. Внутрішня структура сервісу організована у вигляді пакетів, кожен з яких інкапсулює специфічну частину функціональності.

Пакет app є точкою входу до сервісу та відповідає за його коректну ініціалізацію. Він містить єдину функцію Run, основним завданням якої є послідовний імпорт та належне налаштування всіх інших пакетів, що входять до складу сервісу файлів. Таким чином, пакет app забезпечує підготовку середовища та запуск усіх необхідних компонентів для повноцінного функціонування сервісу.

**l.Info("running server", slog.String("url", cfg.URL))**

**g, gCtx := errgroup.WithContext(ctx)**

**g.Go(func() error {**

**return handler.Start(ctx)**

**})**

**g.Go(func() error {**

**filesConnector.StartDisposalRoutine(time.Duration(cfg.GC.SleepInMinutes)\*time.Minute, time.Duration(cfg.GC.KeepAliveInMinutes)\*time.Minute)**

**readersConnector.StartDisposalRoutine(time.Duration(cfg.GC.SleepInMinutes)\*time.Minute, time.Duration(cfg.GC.KeepAliveInMinutes)\*time.Minute)**

**return nil**

**})**

**g.Go(func() error {**

**return queueHandler.Start(ctx)**

**})**

**g.Go(func() error {**

**<-gCtx.Done()**

**err := queueHandler.Stop(ctx)**

**err = errors.Join(err, handler.Stop(ctx))**

**return err**

**})**

**if err := g.Wait(); err != nil {**

**l.Error("server shutdown", slog.String("err", err.Error()))**

**}**

Пакет connector реалізує механізм управління активними підключеннями до файлів. Центральним елементом цього пакету є структура Connector. Вона інкапсулює логіку зберігання та надання актуальної інформації про відкриті з'єднання. Для цього всередині Connector використовується синхронізована асоціативна колекція, де в якості ключа виступає універсальний унікальний ідентифікатор (UUID) конкретного підключення. Значенням у цій колекції є структура, що агрегує дані про останній запит, здійснений за відповідним підключенням, а також вказівник на самі дані, що зберігаються та обробляються в рамках цього підключення. Окрім цього, структура Connector містить метод, призначений для автоматичного виявлення та завершення тих підключень, які тривалий час не демонструють активності. Такий підхід дозволяє оптимізувати використання системних ресурсів. Важливо зазначити, що Connector спроєктований як єдиний екземпляр (singleton) у системі, що централізовано відповідає за повний життєвий цикл усіх підключень у межах сервісу файлів.

**func (c \*Connector[V]) OpenConnection(value V) (uuid.UUID, error) {**

**id := uuid.New()**

**err := c.m.Set(id, Connection[V]{**

**ID: id,**

**ActivityTime: time.Now(),**

**Value: value,**

**})**

**return id, err**

**}**

**func (c \*Connector[V]) Connection(id uuid.UUID) (V, error) {**

**var value V**

**connection, ok := c.m.Get(id)**

**if !ok {**

**return value, os.ErrNotExist**

**}**

**value = connection.Value**

**err := c.m.Set(id, Connection[V]{**

**ID: id,**

**ActivityTime: time.Now(),**

**Value: value,**

**})**

**return value, err**

**}**

Пакет controller несе відповідальність за управління файловою системою на вищому рівні абстракції та за взаємодію з нею. До складу цього пакету входять дві основні структури: Controller та FS. Структура Controller призначена для зберігання метаінформації про файли, що наявні у сховищі, зокрема їхні розміри. Вона також виконує функції підрахунку загального обсягу зайнятого дискового простору, керує процесами виділення (аллокації) та звільнення (деаллокації) місця у сховищі. Окрім того, структура Controller здійснює моніторинг та контроль за дотриманням встановлених лімітів на загальний розмір сховища. Структура FS, у свою чергу, є абстрактною оболонкою над стандартними операціями файлової системи операційної системи. Вона надає програмні інтерфейси для безпосереднього виконання операцій створення, видалення, відкриття файлів та запису даних у них.

**func (c \*Controller) TryAllocateStorage(size int64) error {**

**if c.CurrentSize.Load()+size > c.MaxSize {**

**return ErrMaxSizeExceeded**

**}**

**return nil**

**}**

**func (c \*Controller) AllocateStorage(size int64) error {**

**if c.CurrentSize.Load()+size > c.MaxSize {**

**return ErrMaxSizeExceeded**

**}**

**c.CurrentSize.Add(size)**

**return nil**

**}**

Пакет fileio інкапсулює логіку роботи з окремими файлами на низькому рівні, забезпечуючи безпосередні операції читання та запису. Цей пакет представлений трьома ключовими структурами: File, reader та writer. Структура File є об'єктним представленням конкретного файлу в системі. Вона зберігає детальну інформацію про файл, забезпечує механізми синхронізації для безпечного одночасного доступу до файлу та, за потреби, динамічно створює екземпляри структур reader та writer для виконання відповідних операцій. Структура reader є власною, кастомізованою реалізацією буферизованого файлового дескриптора, оптимізованого для операцій читання. Вона підтримує можливість довільного переміщення курсору читання в межах файлу, що є важливим для потокової передачі даних та доступу до довільних фрагментів файлу. Також ця структура надає вдосконалений механізм для полегшеного та ефективного звільнення пам'яті, зайнятої внутрішнім буфером, використовуючи для цього можливості пакету weak, що став частиною стандартної бібліотеки мови програмування Go, починаючи з версії 1.24.

**func (r \*reader) Seek(offset int64, whence int) (n int64, err error) {**

**if r.file.Closed() || r.closed {**

**return 0, os.ErrClosed**

**}**

**if r.file.version() != r.v {**

**r.Close()**

**return 0, os.ErrClosed**

**}**

**r.file.rwMx().RLock()**

**defer r.file.rwMx().RUnlock()**

**r.mx.Lock()**

**defer r.mx.Unlock()**

**buffer := r.buffer.Value()**

**if buffer == nil {**

**n, err = r.osFile.Seek(offset, whence)**

**r.pos = n**

**return n, err**

**}**

**newOffset, err := r.osFile.Seek(offset, whence)**

**if err != nil {**

**r.pos = newOffset**

**return newOffset, err**

**}**

**if diff := newOffset - r.pos; diff >= 0 && diff < int64(buffer.Buffered()) {**

**\_, err = buffer.Discard(int(diff))**

**off, err2 := r.osFile.Seek(int64(buffer.Buffered()), io.SeekCurrent)**

**r.pos = off - int64(buffer.Buffered())**

**return r.pos, errors.Join(err, err2)**

**}**

**buffer.Reset(r.osFile)**

**r.pos = newOffset**

**return r.pos, nil**

**}**

Принцип функціонування такого механізму полягає у тому, що посилання на об'єкти, що зберігаються в кеші, визначаються як "слабкі". Якщо на кешований об'єкт даних більше не існує сильних посилань, збирач сміття має можливість звільнити пам'ять, зайняту цим об'єктом, під час наступного циклу збирання [15, ст. 390]. Також ця структура реалізує проактивне зчитування з дискового накопичувача блоку даних, розмір якого перевищує запитуваний. Цей "випереджувальний" блок даних негайно розміщується в оперативній пам'яті (RAM), яка виконує функцію високошвидкісного кешу. При надходженні наступних запитів на суміжні або просторово близькі сегменти того ж файлу, відповідні дані можуть бути надані безпосередньо з цього кешу в RAM [15, ст. 391].

**func (r \*reader) Read(p []byte) (n int, err error) {**

**if r.file.Closed() || r.closed {**

**return 0, os.ErrClosed**

**}**

**if r.file.version() != r.v {**

**r.Close()**

**return 0, os.ErrClosed**

**}**

**r.file.rwMx().RLock()**

**defer r.file.rwMx().RUnlock()**

**r.mx.Lock()**

**defer r.mx.Unlock()**

**buffer := r.buffer.Value()**

**if buffer == nil {**

**\_, err = r.osFile.Seek(r.pos, io.SeekStart)**

**buffer = bufio.NewReaderSize(r.osFile, r.bufferSize)**

**r.buffer = weak.Make(buffer)**

**}**

**n, err = buffer.Read(p)**

**r.pos += int64(n)**

**return n, err**

**}**

Структура writer реалізує файловий дескриптор, призначений для операцій запису, та забезпечує синхронізований доступ при модифікації файлового вмісту, запобігаючи конфліктам даних.

**func (w \*writer) Write(b []byte) (n int, err error) {**

**if w.ownFile.Closed() || w.closed {**

**return 0, os.ErrClosed**

**}**

**w.mx.Lock()**

**defer w.mx.Unlock()**

**err = w.ownFile.allocate(int64(len(b)))**

**if err != nil {**

**return 0, err**

**}**

**return w.osFile.Write(b)**

**}**

Пакет handlers об'єднує компоненти, що відповідають за обробку вхідних запитів до сервісу файлів та за його взаємодію з іншими частинами розподіленої системи. Він містить два спеціалізовані підпакети. Перший, handlers/queue, реалізує механізм взаємодії з сервісом метаданих через систему асинхронних черг повідомлень RabbitMQ. Для організації обміну повідомленнями використовується бібліотека ThreeDotsLabs/watermill, яка дозволяє ефективно реалізувати патерн проектування "видавець-підписник" (Pub/Sub). Такий підхід забезпечує надійну та гнучку асинхронну комунікацію між сервісами.

**func (h \*Handler) handle(msg \*message.Message) error {**

**l := h.l.With(slog.String("op", "handle"))**

**var request Request**

**if err := h.g.Unmarshal(msg.Payload, &request); err != nil {**

**l.Error("unable to unmarshal queue payload using gob", slog.String("err", err.Error()))**

**msg.Nack()**

**return fmt.Errorf("unable to unmarshal queue payload using gob: %w", err)**

**}**

**msg.Ack()**

**ctx := context.Background()**

**response, revert, err := h.processRequest(ctx, &request)**

**if err != nil {**

**return err**

**}**

**responseBody, err := h.g.Marshal(response)**

**if err != nil {**

**l.Error("unable to marshal responseBody", slog.String("err", err.Error()))**

**return err**

**}**

**result, err := h.client.R().**

**SetContext(ctx).**

**SetAuthScheme("Bearer").**

**SetAuthToken(h.token).**

**SetBody(responseBody).**

**Post(request.Host)**

**if err != nil {**

**l.Error("unable to make request to fsm", slog.String("err", err.Error()))**

**revert()**

**return err**

**}**

**if result.StatusCode() == http.StatusResetContent {**

**revert()**

**}**

**return nil**

**}**

Другий підпакет, handlers/rest, відповідає за обробку синхронних запитів від клієнтів, що надходять через програмний інтерфейс REST (Representational State Transfer). Для реалізації кінцевих точок (ендпоінтів) сервісу та для маршрутизації HTTP-запитів у цьому підпакеті використовується бібліотека go-chi, відома своєю легкістю та високою продуктивністю.

**func (h \*Handler) Register() {**

**r := h.r**

**r.Use(middleware.RequestID)**

**r.Use(middleware.RealIP)**

**r.Use(middleware.Recoverer)**

**r.Use(middleware.URLFormat)**

**if h.cfg.Env == "dev" {**

**r.Use(cors.Handler(cors.Options{**

**AllowedOrigins: []string{"https://\*", "http://\*"},**

**AllowedMethods: []string{"GET", "POST", "PUT", "DELETE", "OPTIONS"},**

**}))**

**} else {**

**r.Use(cors.Handler(cors.Options{**

**AllowedOrigins: []string{"https://" + h.cfg.CORSOrigin, "http://\*" + h.cfg.CORSOrigin},**

**AllowedMethods: []string{"GET", "POST", "PUT", "DELETE", "OPTIONS"},**

**}))**

**}**

**r.Route("/files", func(r chi.Router) {**

**r.Get("/read", h.ReadFile)**

**r.Post("/write", h.WriteFile)**

**r.Post("/close", h.CloseFile)**

**})**

**}**

Пакет usecases містить опис та реалізацію основних сценаріїв взаємодії всередині сервісу файлів. Цей пакет визначає бізнес-логіку сервісу на абстрактному рівні, незалежно від конкретних протоколів передачі даних чи інтерфейсів взаємодії (REST або черги повідомлень). Він агрегує та координує функціональність, що надається іншими пакетами, зокрема connector та controller, для реалізації комплексних операцій, таких як завантаження файлу, його видалення, читання фрагменту тощо. Функції, визначені у пакеті usecases, використовуються компонентами з обох підпакетів handlers (queue та rest) для виконання відповідних бізнес-задач.

**func (u \*UseCases) OpenFile(ctx context.Context, fileID uuid.UUID) (connectionID uuid.UUID, err error) {**

**file, err := u.StorageController.File(fileID)**

**if err != nil {**

**return connectionID, err**

**}**

**bufferSize := max(min(u.MaxBufferSize, int(file.Size())), u.MinBufferSize)**

**reader, err := file.Reader(bufferSize)**

**if err != nil {**

**return connectionID, err**

**}**

**return u.ReadersConnector.OpenConnection(reader)**

**}**

Пакет libs включає набір допоміжних модулів та бібліотек, призначених для спрощення процесу розробки, підвищення якості коду та забезпечення стандартизованого підходу до вирішення типових завдань. До складу цього пакету входять, зокрема, компоненти для роботи з конфігураційними файлами, що дозволяють гнучко налаштовувати параметри сервісу. Також тут міститься модуль для ведення журналів подій (логгер), який забезпечує збір та збереження інформації про роботу сервісу, що є важливим для моніторингу та діагностики. Окрім цього, пакет libs надає абстракцію для реалізації синхронізованої асоціативної колекції (мапи), яка гарантує потокобезпечний доступ до даних у багатопотоковому середовищі.

У складі пакетів також наявні піддиректорії mocks, які містять тестові імітації (моки) для компонентів пакету. Ці імітації використовуються під час проведення модульного тестування для ізоляції тестованих одиниць та перевірки їхньої коректної взаємодії.

## 4.2 Опис програмної реалізації сервісу метаданих

В основі програмної реалізації сервісу метаданих, аналогічно до сервісу файлів, лежить модульний підхід. Нижче розглянуто сукупність пакетів, кожен з яких інкапсулює окрему, чітко визначену частину функціональності, але які разом формують серверну логіку сервісу метаданих.

Пакет app є центральною точкою ініціалізації сервісу. У ньому міститься функція CreateApp, відповідальна за імпорт та послідовну ініціалізацію всіх інших пакетів системи. Цей процес відбувається з використанням контейнера для впровадження залежностей uber/fx, що забезпечує належне конфігурування та запуск компонентів. Окрім основного функціоналу ініціалізації, пакет app також включає інтеграційний тест. Призначення цього тесту полягає у верифікації коректності взаємодії між різними пакетами програми, а також у перевірці правильності описаного процесу ініціалізації системи, що гарантує стабільність та прогнозованість роботи сервісу на етапі запуску.

**func CreateApp(cfg \*config.Config) fx.Option {**

**return fx.Options(**

**fx.Supply(**

**cfg,**

**),**

**fx.Provide(**

**newValidator,**

**log.New,**

**storage.New,**

**fx.Annotate(storage.NewDirectoryStorage, fx.As(new(directory.Storage))),**

**fx.Annotate(storage.NewFileStorage, fx.As(new(file.Storage))),**

**fx.Annotate(communicator.New, fx.As(new(service.Communicator)), fx.As(fx.Self())),**

**fx.Annotate(directory.New, fx.As(new(handler.DirectoryService))),**

**fx.Annotate(file.New, fx.As(new(handler.FileService))),**

**handler.NewDirectoryHandler,**

**handler.NewFileHandler,**

**handler.New,**

**),**

**fx.Invoke(**

**startHTTPServer,**

**),**

**)**

**}**

Пакет communicator інкапсулює логіку взаємодії сервісу метаданих із сервісом файлів. Основним компонентом цього пакету є структура Communicator, методи якої описують процеси створення, видалення, відкриття та редагування файлів. Ці методи формують та передають необхідні дані до сервісу файлів для подальшого виконання відповідних операцій. Важливою частиною функціональності пакету є реалізація кінцевої точки для отримання відповіді від сервісу файлів. Ця відповідь містить посилання на сервіс файлів та ідентифікатор підключення, що дозволяє користувачеві продовжувати безпосередню взаємодію з файлами. Для ефективного обміну даними між сервісами в пакеті реалізовано механізм кодування та декодування даних у форматі gob. Цей внутрішній бінарний формат, специфічний для мови програмування Go, забезпечує високу швидкість обробки даних, оскільки оптимізований для серіалізації структур цієї мови. Також у пакеті communicator визначено структуру Process. Вона відповідає за управління окремим процесом обміну даними між сервісом метаданих та сервісом файлів. Для забезпечення синхронізації та безпечної передачі даних, структура Process використовує вбудовані в мову Go примітиви, такі як канали та sync.Once. Це дозволяє здійснити однократну передачу даних від сервісу файлів до клієнта та сигналізувати іншим компонентам про результат запиту, фактично реалізуючи власний примітив синхронізації.

**func (r \*Process) Set(response \*Response) bool {**

**var b bool**

**r.once.Do(func() {**

**b = true**

**r.result <- response**

**close(r.result)**

**})**

**return b**

**}**

**func (r \*Process) WaitAndGet(ctx context.Context) (\*Response, error) {**

**select {**

**case <-ctx.Done():**

**return nil, ctx.Err()**

**case result := <-r.result:**

**return result, nil**

**}**

**}}**

Пакет core призначений для зберігання доменних структур, що описують основні сутності предметної області застосунку. Ключовими структурами в цьому пакеті є File та Directory, які представляють відповідно файл та директорію в системі. Окрім визначення цих базових моделей, пакет core також містить оголошення фільтрувальної моделі. Ця модель використовується для формування критеріїв пошуку та фільтрації даних. У пакеті реалізовано метод для перетворення цієї фільтрувальної моделі на відповідний запит, що може бути виконаний базою даних MongoDB для отримання відфільтрованих результатів.

**type Filter struct {**

**Type Type `query:"type" validate:"-"` // uint: 0 for both (file and directory), 1 for files only, 2 for directories only**

**Name string `query:"name" validate:"-"`**

**CreatedAtFrom time.Time `query:"createdAtFrom" validate:"-"`**

**CreatedAtTo time.Time `query:"createdAtTo" validate:"-"`**

**UpdatedAtFrom time.Time `query:"updatedAtFrom" validate:"-"`**

**UpdatedAtTo time.Time `query:"updatedAtTo" validate:"-"`**

**Public \*bool `query:"public" validate:"-"`**

**Size \*uint `query:"size" validate:"-"`**

**Starred \*bool `query:"starred" validate:"-"`**

**// Files only**

**Extensions []string `query:"extensions" validate:"-"`**

**}**

Пакет handler відповідає за конфігурацію та управління маршрутизацією HTTP-запитів у сервісі метаданих. У цьому пакеті відбувається ініціалізація та налаштування маршрутизатора go-fiber, який використовується для обробки вхідних запитів. Тут визначаються шляхи для кожної кінцевої точки сервісу та відповідні їм HTTP-методи (GET, POST, PUT, DELETE тощо). Крім того, для оптимізації та підвищення перевикористання коду в пакеті handler реалізовано власну імплементацію узагальнених обробників веб-запитів (Generic Web Handlers). Такий підхід дозволяє уніфікувати обробку типових запитів, зменшуючи дублювання коду та спрощуючи подальшу підтримку й розширення функціональності сервісу.

**func (h \*Handler[T, V]) handleWithoutResult(c \*fiber.Ctx) error {**

**l := h.l.With(slog.String("op", h.name))**

**data, err := h.processData(l, c)**

**if err != nil {**

**return err**

**}**

**userID, err := GetUserID(l, c)**

**if err != nil {**

**return c.Status(http.StatusUnauthorized).JSON(utils.NewErrorResponse("authentification error"))**

**}**

**err = h.serviceWithoutResult(owncontext.New(c.Context(), userID), data)**

**if err != nil {**

**return utils.ProcessError(l, c, err)**

**}**

**return c.Status(http.StatusOK).JSON(utils.NewOKResponse[any](nil))**

**}**

Пакет service/directory інкапсулює бізнес-логіку, пов'язану з операціями над директоріями. Він визначає правила валідації вхідних даних та координує виклики методів пакету storage для взаємодії з базою даних. Функціональність цього пакету охоплює створення нових директорій, їх видалення, отримання інформації про конкретну директорію, а також перегляд її вмісту. Реалізовано логіку для переміщення директорій в ієрархії файлової системи, їх перейменування та додавання до списку обраних для швидкого доступу. Окрім операцій над окремими директоріями, пакет service/directory також містить логіку для здійснення пошуку директорій та файлів у системі на основі визначених критеріїв, забезпечуючи валідацію запитів пошуку та взаємодію з пакетом сховища даних.

Пакет service/file містить бізнес-логіку для операцій, що стосуються файлів. Аналогічно до пакету service/directory, він відповідає за валідацію даних, що надходять від користувача, та за виклик відповідних методів пакету storage для виконання операцій з файлами у базі даних. До функціональних можливостей, реалізованих у цьому пакеті, належать: створення нових файлів, їх видалення, отримання файлу. Пакет також забезпечує логіку для переміщення файлів між директоріями, зміни параметрів доступу до файлів (приватний, публічний), перейменування файлів та їх відмічання як улюблених для полегшення подальшого доступу.

**type CreateRequest struct {**

**ParentDirID types.ObjectId `json:"parentDirID" validate:"required"`**

**Name string `json:"name" validate:"required"`**

**Extension string `json:"extension" validate:"-"`**

**Size uint `json:"size" validate:"required"`**

**}**

**func (s \*Service) Create(ctx owncontext.Context, data \*CreateRequest) (\*Response, error) {**

**l := s.l.With(slog.String("op", "Create"))**

**\_, err := s.getAndCheckDirectory(ctx, data.ParentDirID)**

**if err != nil {**

**return nil, err**

**}**

**file, err := s.s.Create(ctx, data.ParentDirID, ctx.UserID(), data.Name, data.Extension, data.Size)**

**if err != nil {**

**return nil, service.NewDBError(l, err)**

**}**

**host, connectionID, err := s.c.Create(ctx, file.ID, data.Size)**

**if err != nil {**

**return nil, ownerrors.NewInternalError(l, "unable to communicate with FS", err)**

**}**

**return &Response{**

**File: \*file,**

**Host: host,**

**ConnectionID: connectionID,**

**}, nil**

**}**

Пакет storage є рівнем абстракції для взаємодії з базою даних MongoDB. У цьому пакеті описані всі необхідні запити до бази даних для виконання операцій, пов'язаних зі зберіганням та управлінням метаданими файлів та директорій. Функціонал пакету включає реалізацію запитів для створення, видалення, переміщення, зміни параметрів доступу, перейменування та відмічання файлів і директорій як улюблених. Також тут реалізовані запити для отримання вмісту конкретної директорії та для виконання фільтрації даних на основі наданої фільтраційної моделі. Важливою особливістю пакету storage є реалізація функціоналу перерахунку розмірів файлів та директорій при їх переміщенні, додаванні або видаленні. Використання нереляційної бази даних MongoDB дозволяє оптимізувати запити на отримання вмісту директорії та пошук, оскільки інформація про вміст директорії (вкладені файли та директорії) зберігається безпосередньо в об'єкті самої директорії. Це усуває необхідність у складних з'єднаннях таблиць (joins), характерних для реляційних баз даних. Однак такий підхід накладає додаткові вимоги щодо забезпечення узгодженості даних і змушує виконувати декілька запитів при додаванні чи редагуванні інформації. Тим не менш, це є ефективним рішенням, оскільки оптимізує операції читання, які у файлових системах зазвичай виконуються значно частіше, ніж операції запису.

**func aggregationFilter(userID string, filter bson.D, name string, offset, limit uint, sortByField string, sortOrder int) []bson.D {**

**if name == DirectoryCollection {**

**filter = append(filter, bson.E{"name", bson.M{"$ne": "root"}})**

**}**

**filter = append(filter, bson.E{"userID", userID})**

**result := []bson.D{{{"$match", filter}},}**

**if name == DirectoryCollection {**

**result = append(result, bson.D{{"$project", bson.M{**

**"\_id": 1,**

**"userID": 1,**

**"parentDirectoryID": 1,**

**"path": 1,**

**"name": 1,**

**"createdAt": 1,**

**"updatedAt": 1,**

**"public": 1,**

**"size": 1,**

**"starred": 1,**

**}}})**

**}**

**result = append(result, []bson.D{**

**{{"$facet", bson.D{**

**{"result", []bson.D{{{"$count", name + "Count"}}}},**

**{name, []bson.D{**

**{{"$skip", offset}},**

**{{"$limit", limit}},**

**{{"$sort", bson.D{{sortByField, sortOrder}}}},**

**}},**

**}}},**

**{{"$project", bson.M{**

**name + "Count": bson.M{"$arrayElemAt": []interface{}{"$result." + name + "Count", 0}},**

**name: 1,**

**}}},**

**}...)**

**return result**

**}**

Пакет libs об'єднує набір допоміжних підпакетів, які призначені для спрощення процесу розробки та покращення загального досвіду розробника. До складу цього пакету входять компоненти для управління конфігураційними файлами, що дозволяє гнучко налаштовувати параметри роботи сервісу. Також тут міститься логер, призначений для ведення журналу подій, що є важливим для моніторингу та діагностики роботи системи.

## 4.3 Опис програмної реалізації сервісу автентифікації

Реалізація сервісу автентифікації, як і інших частин системи, ґрунтується на засадах модульного підходу. Внутрішня архітектурна організація сервісу представлена комплексом пакетів, де кожний інкапсулює визначений набір функціональних можливостей, призначених для здійснення процесів автентифікації та адміністрування користувацьких сесій.

Пакет app слугує точкою входу до сервісу та відповідає за його коректну ініціалізацію. Аналогічно до реалізації в інших сервісах системи, він містить функцію CreateApp, основним завданням якої є послідовний імпорт та належне налаштування всіх інших пакетів, що входять до складу сервісу автентифікації. Цей процес відбувається з використанням контейнера для впровадження залежностей uber/fx, що забезпечує належне конфігурування та запуск компонентів. Окрім основного функціоналу ініціалізації, пакет app також включає інтеграційний тест. Призначення цього тесту полягає у верифікації коректності взаємодії між різними пакетами програми та перевірці правильності описаного процесу ініціалізації системи.

Центральним пакетом, що відповідає за основну логіку автентифікації, є пакет auth. Він інкапсулює всю функціональність, пов'язану з управлінням сесіями користувачів, включаючи створення нової сесії з генерацією відповідних маркерів доступу (Access Token) та оновлення (Refresh Token), механізми оновлення цих маркерів, а також процедури завершення сесії, що супроводжуються відкликанням активних маркерів.

**func (s \*Service) CreateSession(ctx Context, claimsUser core.User) (Response, error) {**

**user, err := s.storage.GetUser(ctx, claimsUser.Sub)**

**if err != nil {if err = s.storage.AddUser(ctx, &claimsUser); err != nil {return Response{}, errors.NewInternalError(l, "unable to add user to db", err)}**

**user = &claimsUser**

**} else {**

**user.Name = claimsUser.Name**

**user.Picture = claimsUser.Picture**

**err = s.storage.UpdateUser(ctx, user)**

**if err != nil {**

**return Response{}, errors.NewInternalError(l, "unable to update user data", err)**

**}}**

**session := &core.Session{**

**UserSub: user.Sub,**

**Salt: uuid.New(),**

**DeviceData: ctx.UserAgent(),**

**}**

**err = s.storage.AddSession(ctx, session)**

**if err != nil {**

**return Response{}, errors.NewInternalError(l, "unable to add session to db", err)**

**}**

**return s.generateTokens(user, session)**

**}**

Пакет auth містить два спеціалізовані підпакети: google та firebase. Підпакет auth/google реалізує процес автентифікації з використанням Google Identity API. В рамках цього процесу користувач надає системі ідентифікаційний маркер (ID-токен) у форматі JWT, попередньо сформований сервісом Google. Далі відбувається валідація отриманого маркера за допомогою офіційної бібліотеки google.golang.org/api/idtoken. У разі успішної валідації, з маркера вилучаються дані про користувача, після чого ініціюється стандартний процес створення сесії в системі. Аналогічну функціональність реалізує підпакет auth/firebase, який оперує з ідентифікаційними маркерами, отриманими від Firebase Authentication API, та використовує для їх перевірки офіційну бібліотеку firebase.google.com/go/v4.

**func (s \*Service) Authenticate(ctx Context, loginData LoginData) (auth.Response, error) {**

**client, err := s.fb.Auth(ctx)**

**if err != nil {**

**return auth.Response{}, errors.NewValidationError(l,"unable to get firebase client","service temporary unavailable",err)**

**}**

**payload, err := client.VerifyIDTokenAndCheckRevoked(ctx, loginData.JWT)**

**if err != nil {**

**return auth.Response{}, errors.NewValidationError(l,"unable to parse jwt from CreateSession","invalid access token",err)**

**}**

**payload.Claims["sub"] = payload.Subject**

**claimsUser, err := core.NewUserFromClaims(payload.Claims)**

**if err != nil {**

**return auth.Response{}, errors.NewInternalError(l,"unable to parse claims from CreateSession",err)**

**}**

**return s.auth.CreateSession(ctx, claimsUser)**

**}**

Пакет core призначений для зберігання основних моделей даних системи, що стосуються користувача та його сесії. Основними сутностями, визначеними у цьому пакеті, є структури User та Session. Крім визначення самих структур, пакет core також надає методи для їх перетворення у формат map[string]any та навпаки. Це необхідно для підготовки даних до кодування у JWT-маркери та для отримання даних з них.

За обробку вхідних HTTP-запитів та маршрутизацію відповідає пакет handler. У цьому пакеті відбувається ініціалізація та конфігурація HTTP-маршрутизатора go-fiber, який використовується для визначення кінцевих точок (endpoints) сервісу. Аналогічно до сервісу метаданих, за кожною кінцевою точкою закріплюється відповідний шлях та HTTP-метод, що забезпечує належну обробку запитів від клієнтської частини.

Взаємодія з базою даних MongoDB інкапсульована у пакеті storage. Він містить повний набір функцій для виконання операцій CRUD (Create, Read, Update, Delete) над сутностями користувача та сесії. Усі запити до бази даних, необхідні для створення, видалення, отримання та модифікації даних користувачів та їхніх сесій, реалізовані саме у цьому пакеті.

Пакет tokens містить логіку, пов'язану з генерацією та валідацією JWT-маркерів. Ключовою сутністю пакету є структура JWTWorker, яка слугує обгорткою над відповідними бібліотеками для роботи з JWT. Ця структура надає метод Encode, який приймає на вхід дані у форматі map[string]any (отримані з моделей пакета core) та генерує на їх основі JWT-маркер. У процесі генерації до маркера додаються стандартні поля, такі як iss (issuer), aud (audience), iat (issued at), exp (expiration time). Після цього маркер шифрується та підписується, і готовий JWT-маркер повертається для подальшого використання.

**func (j \*JWTWorker) Encode(claims map[string]any) (string, error) {**

**claims["iss"] = Package**

**claims["aud"] = Audience**

**claims["exp"] = jwt.NewNumericDate(time.Now().Add(j.exp))**

**claims["iat"] = jwt.NewNumericDate(time.Now())**

**t := jwt.NewWithClaims(jwt.SigningMethodHS512, jwt.MapClaims(claims))**

**token, err := t.SignedString([]byte(j.secret))**

**if err != nil {**

**return "", fmt.Errorf("unable to sign token: %w", err)**

**}**

**return token, nil**

**}**

Допоміжні функції та інструменти, що спрощують процес розробки та підвищують надійність системи, зосереджені у пакеті libs. Цей пакет включає компоненти для роботи з конфігураційними файлами, що дозволяє гнучко налаштовувати параметри сервісу. Також тут міститься реалізація логера для ведення журналу подій, що є важливим для моніторингу та діагностики роботи системи.

# 5 ТЕСТУВАННЯ РОЗРОБЛЕНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 5.1 Обґрунтування вибору видів тестування

Для забезпечення якості та надійності розробленої серверної частини програмної системи хмарного зберігання особистих файлів, що є клієнт-серверним застосунком з використанням REST API та мікросервісною архітектурою (сервіс метаданих файлів, сервіс файлів та сервіс автентифікації), було обрано комплексний підхід до тестування. Цей підхід включає модульне тестування, інтеграційне тестування та навантажувальне тестування.

Модульне тестування – це вид тестування програмного забезпечення, який полягає у перевірці коректності окремих, найменших логічно виділених компонентів програми (модулів, функцій, процедур, методів) в ізольованому середовищі. Метою є підтвердження того, що кожен модуль функціонує відповідно до своєї специфікації. В контексті розробки серверної частини системи, побудованої на мікросервісній архітектурі, модульне тестування набуває особливої значущості. Кожен мікросервіс, такий як сервіс метаданих файлів, сервіс файлів чи сервіс автентифікації, складається з множини функціональних блоків, коректна робота яких є запорукою загальної надійності відповідного сервісу. Модульне тестування дозволяє локалізувати помилки на ранніх етапах розробки, перевіряючи логіку кожної окремої функції або методу, наприклад, коректність роботи власної реалізації буферизованого об'єкта читання з функціональністю переміщення курсора в сервісі файлів. Це істотно спрощує процес налагодження та знижує вартість виправлення виявлених дефектів. Застосування табличних тестів забезпечує систематичний підхід до перевірки різноманітних вхідних даних та граничних умов для кожного модуля. Для реалізації модульних тестів буде використано стандартний пакет testing мови програмування Go, бібліотеку stretchr/testify для формулювання розширених тверджень та імітаційні об'єкти (моки). Останні будуть згенеровані використовуючи підхід кодогенерації за допомогою інструменту vektra/mockery, що дозволить ефективно ізолювати тестовані модулі від їхніх зовнішніх залежностей.

Інтеграційне тестування, у свою чергу, – це вид тестування, спрямований на перевірку взаємодії між різними компонентами або системами програмного забезпечення після їх об'єднання. Основна мета полягає у виявленні дефектів, що виникають на стиках інтеграції цих компонентів. Для клієнт-серверного застосунку з мікросервісною архітектурою, де функціональність розподілена між кількома незалежно розгорнутими сервісами (сервіс метаданих файлів, сервіс файлів, сервіс автентифікації), що взаємодіють через REST API, інтеграційне тестування є критично важливим. Наприклад, необхідно пересвідчитись, що компоненти певного окремого мікросервісу правильно сконфігуровані та взаємопов’язані між собою, утворюючи функціонально цілісну одиницю. Це включає перевірку того, що всі залежності всередині сервісу коректно надаються та взаємодіють, забезпечуючи очікувану поведінку сервісу як єдиного модуля. Виявлення проблем на цьому рівні, таких як помилки конфігурації, неправильна взаємодія між внутрішніми складовими одного сервісу, або проблеми при ініціалізації сервісу як цілісної системи, є ключовим для забезпечення стабільності та надійності всієї серверної частини.

Навантажувальне тестування – це вид нефункціонального тестування, призначений для перевірки поведінки системи під очікуваним робочим навантаженням. Головна мета полягає у виявленні будь-яких вузьких місць, які обмежують її продуктивність. Для серверної частини системи, яка призначена для обробки запитів від значної кількості користувачів та виконання операцій з файлами, проведення навантажувального тестування є обов'язковим для забезпечення стабільності та прийнятного рівня обслуговування. У контексті мікросервісної архітектури, що складається з сервісу метаданих файлів, сервісу файлів та сервісу автентифікації, доцільно проводити навантажувальне тестування сервісу як окремої функціональної одиниці. Такий підхід дозволяє детально проаналізувати поведінку мікросервісу під високим навантаженням, наприклад, визначити, яку пропускну здатність демонструє сервіс файлів при завантаженні або вивантаженні великих файлів. Результати такого тестування допомагають ідентифікувати точки відмови, оцінити час відгуку системи при пікових навантаженнях та спланувати заходи для оптимізації чи масштабування окремих сервісів для забезпечення безперебійної роботи всієї системи.

## 5.2 Опис тестування

Для проведення модульного тестування було використано набір спеціалізованих інструментів. Основою слугував стандартний пакет testing, що входить до стандартної бібліотеки мови Go. Цей пакет надає каркас для написання автоматизованих тестів. Ключовим елементом є тип \*testing.T, який передається до кожної тестової функції та слугує для керування станом тестування та повідомлення про його результати.

З метою розширення можливостей стандартного пакета testing та для спрощення написання перевірок було залучено сторонню бібліотеку stretchr/testify. Її пакети assert та require надають великий набір функцій-тверджень, що дозволяють у декларативному стилі перевіряти відповідність отриманих результатів очікуваним. Важливою відмінністю є те, що assert лише фіксує помилку, дозволяючи тесту продовжитись, тоді як require негайно зупиняє його виконання у разі невдачі. Це значно підвищує читабельність та лаконічність тестового коду.

Оскільки розроблена система базується на мікросервісній архітектурі, що передбачає взаємодію між компонентами, виникла об'єктивна необхідність в ізоляції тестованих модулів від їхніх залежностей. Для вирішення цього завдання було використано інструмент vektra/mockery. Він дозволяє автоматично генерувати імітаційні об'єкти (моки) на основі існуючих інтерфейсів. Використання моків дає змогу симулювати поведінку зовнішніх компонентів, таких як файлова система або інші мікросервіси, та повністю контролювати умови виконання тесту, перевіряючи коректність взаємодії між модулями.

Обраною методологією для організації тестів стали табличні тести. Цей підхід полягає у визначенні набору тестових випадків у вигляді зрізу структур, де кожна структура описує вхідні дані та очікуваний результат. Основна логіка тесту виконується в циклі, що ітерується по цьому набору. Такий підхід робить тестування систематизованим, дозволяє легко додавати нові сценарії та покращує загальну структуру коду.

У межах тестування сервісу файлів було перевірено функціональність читання файлів, зокрема, коректність послідовного та довільного доступу до даних.

Допоміжна функція prepareReaderFileMock відповідає за підготовку тестового середовища. Вона створює імітаційний об'єкт File за допомогою mockery та тимчасовий фізичний файл. У мок-об'єкті конфігурується очікувана поведінка, що дозволяє ізолювати логіку читання від реальних операцій вводу-виводу.

Тестова функція TestReader\_ReadSequentiallyPerSector\_ReturnFullFile перевіряє коректність послідовного читання файлу по частинах (секторах). За допомогою табличного підходу тестуються різні варіанти вхідних даних. Спочатку виконується повне зчитування файлу для перевірки базової працездатності. Після цього файл зчитується послідовно по секторах, і результати об'єднуються. Наприкінці виконується твердження, що фінальний результат відповідає початковому вмісту файлу. Це гарантує, що механізм читання працює коректно при послідовному доступі.

Тестова функція TestReader\_ReadWithSeekFromBeginningPerSector\_ReturnFullFile перевіряє більш складний сценарій, що включає довільний доступ до даних файлу за допомогою функції Seek. Ця можливість є критично важливою для таких операцій, як потокове відтворення або завантаження частини файлу. Тест, використовуючи табличний підхід, зчитує сектори файлу в довільному, заздалегідь визначеному порядку. Переміщення курсора читання здійснюється від початку файлу. Після зчитування всіх секторів у довільній послідовності, фінальний результат порівнюється з оригінальним вмістом файлу, що підтверджує коректну спільну роботу операцій Read та Seek (таб.5.1).

Таблиця 5.1 – Тест-кейс №1 (таблиця виконана самостійно)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Інформація про тест-кейс** | | | | |
| **Ідентифікатор тесту:** | | Тест-кейс №1 | | |
| **Опис функції:** | | Читання файлів, коректність послідовного та довільного доступу до даних | | |
| **Власник тесту:** | | Буцулін Ігор Олександрович | | |
| **Дата створення:** | | 05.06.2025 | | |
| **Мета тесту:** | | Перевірити коректність реалізації читання файлів, зокрема, коректність послідовного та довільного доступу до даних | | |
| **Передумова** | | | | |
| **№** | **Опис випадку** | | **Очікуваний результат** | **Висновок** |
| 1 | Встановити пакет розробки для мови прграмування Go | | Пакет розробки встановлено | Пройдено |
| 2 | Скомпілювати тести | | Тести скомпільовано | Пройдено |
| **Створення заявки на фінансову допомогу** | | | | |
| **№** | **Опис випадку** | | **Очікуваний результат** | **Висновок** |
| 1 | Віртуальний файл з вмістом "hello and welcome", сектори для читання визначені межами {4, 9, 13}. | | Зчитані та об'єднані частини файлу повністю відповідають початковому вмісту "hello and welcome". | Пройдено |
| 2 | Віртуальний файл з вмістом "hq", сектор для читання визначений межею {1}. | | Зчитані та об'єднані частини файлу повністю відповідають початковому вмісту "hq". | Пройдено |
| 3 | Віртуальний файл "hello and welcome", межі секторів {4, 7, 10, 13}, порядок читання секторів {2, 1, 4, 0, 3}. | | Зчитані у довільному порядку та зібрані частини файлу відповідають початковому вмісту. | Пройдено |
| 4 | Віртуальний файл з вмістом "hq", межа сектора {1}, порядок читання секторів {1, 0}. | | Зчитані у довільному порядку та зібрані частини файлу відповідають початковому вмісту. | Пройдено |

Таким чином, проведене модульне тестування підтвердило коректність реалізації ключових функцій на рівні окремих компонентів сервісу файлів. Застосування табличного підходу та імітаційних об'єктів дозволило досягти високого рівня покриття коду та надійності перевірок. Це, в свою чергу, закладає міцний фундамент для наступних етапів тестування та забезпечує загальну стабільність системи.

Процес інтеграційного тестування було зосереджено на перевірці правильності конфігурації та взаємозв’язків між сервісами на рівні їх ініціалізації та конструювання. Для цього було застосовано підхід, що полягає у валідації графа залежностей застосунку. Такий підхід дозволяє переконатися, що всі компоненти системи правильно сконфігуровані та здатні отримати необхідні їм для роботи залежності від інших компонентів ще до повного запуску серверних служб. Для реалізації цього завдання було використано фреймворк для впровадження залежностей uber-go/fx, який надає інструменти для автоматичної перевірки структурної цілісності програми.

Було розроблено спеціалізований інтеграційний тест TestValidateApp. Цей тест ініціює створення екземпляра застосунку з порожньою конфігурацією, після чого викликає функцію fx.ValidateApp. Ця функція рекурсивно аналізує всі модулі, що складають застосунок, та їхні залежності. Вона перевіряє, чи кожен компонент, що є частиною одного з трьох мікросервісів, має доступ до всіх оголошених ним залежностей, які повинні надаватися іншими частинами системи. Наприклад, перевіряється, чи може сервіс метаданих коректно отримати залежність від сервісу автентифікації для перевірки прав доступу користувача.

Успішне проходження цього тесту, що підтверджується відсутністю помилки при виконанні твердження require.NoError(t, err), свідчить про те, що архітектурна побудова системи є коректною. Це означає, що всі три мікросервіси належним чином інтегровані на рівні програмних компонентів, їхні залежності правильно визначені та задовольняються, а загальна структура серверної частини є стабільною та готовою до запуску. Даний етап тестування є фундаментальним, оскільки він закладає основу для подальших видів тестування, таких як функціональне та навантажувальне, підтверджуючи коректність внутрішньої будови системи.

Для перевірки продуктивності та стабільності розробленої серверної частини програмної системи було проведено навантажувальне тестування. Метою даного етапу була ідентифікація "вузьких місць" в архітектурі, оцінка використання системних ресурсів під стандартним навантаженням та підтвердження загальної стійкості кожного з мікросервісів, а особливо сервісу файлів.

Процедура тестування виконувалася з використанням вбудованих засобів профілювання інтегрованого середовища розробки GoLand (рис. 5.1). Такий підхід дозволив здійснити глибокий аналіз поведінки системи на рівні виконання коду. Для створення умов, наближених до реальної експлуатації, в режимі профілювання послідовно запускалися набори модульних та інтеграційних тестів, а також уся програмна система в цілому. Це забезпечило комплексний огляд поведінки окремих функцій та їхньої взаємодії в рамках мікросервісної архітектури при обробці значної кількості запитів.

A screenshot of a computer

AI-generated content may be incorrect.

Рисунок 5.1 – Результат навантажувального тестування (рисунок виконано самостійно)

У ході аналізу отриманих даних профілювання було встановлено, що найбільше навантаження на центральний процесор створювалося процесом фонового маркування в рамках роботи вбудованого у середовище виконання мови Go збірника сміття (рис 5.2). Цей результат є цілком очікуваним та логічним, оскільки архітектура розробленої системи орієнтована на інтенсивні операції з оперативною пам'яттю, що включають обробку файлів, кешування даних та управління великою кількістю одночасних з'єднань. Автоматичне управління пам'яттю є фундаментальною частиною середовища виконання Go, і його активна робота свідчить про коректне виділення та звільнення ресурсів під час інтенсивної роботи програми.

A screenshot of a computer

AI-generated content may be incorrect.

Рисунок 5.2 – Результат навантажувального тестування (рисунок виконано самостійно)

За підсумками проведеного навантажувального тестування можна зробити висновок, що серверна частина програмної системи демонструє високий рівень оптимізації та витривалості. Не було виявлено критичних помилок, витоків пам'яті або значного зниження продуктивності, що могли б завадити стабільній роботі сервісів. Система ефективно справляється з поставленими навантаженнями, що підтверджує правильність обраних архітектурних рішень та програмних реалізацій.

# ВИСНОВКИ

У рамках даної роботи було розроблено серверну частину програмної системи для хмарного зберігання особистих файлів, що відповідає сучасним вимогам до продуктивності, надійності та безпеки. Мета кваліфікаційної роботи, що полягала у створенні функціонального та надійного серверного компонента, була повністю досягнута.

Початково було проведено комплексний аналіз предметної галузі та існуючих на ринку аналогів. Дослідження виявило потребу в розробці системи, яка б забезпечувала високу швидкість доступу до даних та ефективні механізми попереднього перегляду медіаконтенту, що і визначило основний напрямок подальшої роботи. На основі проведеного аналізу було сформовано детальні вимоги до серверної частини системи. Було обґрунтовано вибір мікросервісної архітектури як основи для забезпечення високої відмовостійкості, гнучкості масштабування та спрощення подальшої підтримки.

У ході проектування було розроблено архітектуру серверної частини, яка складається з трьох незалежних сервісів: сервісу автентифікації, сервісу метаданих та сервісу файлів. Було спроектовано схеми баз даних для кожного сервісу та визначено протоколи їх взаємодії через асинхронні черги повідомлень та програмні інтерфейси REST. Програмна реалізація серверних компонентів була виконана з використанням мови програмування Go, що дозволило досягти високої продуктивності та ефективного управління пам'яттю.

Наукова новизна роботи полягає у застосуванні унікального методу серверного кешування файлів, який значно прискорює доступ до часто запитуваних даних, зокрема при попередньому перегляді великих медіафайлів, що суттєво покращує швидкодію системи. Для перевірки коректності та надійності розробленого програмного забезпечення було проведено комплексне тестування, що включало модульні, інтеграційні тести та навантажувальне тестування, які підтвердили працездатність та стабільність реалізованих функціональних можливостей.

Дана робота має значний потенціал для подальшого продовження дослідження. Перспективними напрямками є розвиток більш складних механізмів контролю доступу, інтеграція з додатковими зовнішніми сервісами, оптимізація алгоритмів обробки та передачі даних для роботи в умовах нестабільного інтернет-з'єднання, а також впровадження просунутих методів шифрування для підвищення рівня безпеки даних.

Результатом виконаної роботи став готовий до експлуатації, надійний та функціональний серверний компонент програмної системи хмарного зберігання. Розроблена серверна частина є масштабованою та стабільною основою, здатною забезпечити ефективне та безпечне управління файлами для широкого кола користувачів, та може бути використана для побудови повноцінної комерційної системи.

**ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ**

1. Donnellan D., Lawrence A., Weinschenk R. Annual outages analysis 2025. *UII Keynote Report 173* : міжнар. конф., м. Нью-Йорк, 2 трав. 2025 р. Нью-Йорк, 2025. С. 26. URL: https://intelligence.uptimeinstitute.com/resource/annual-outage-analysis-2025 (дата звернення: 10.05.2025).
2. Ginn D. The Fastest Cloud Storage Providers in 2025. *Cloudwards*. URL: https://www.cloudwards.net/fastest-cloud-storage/ (дата звернення: 09.05.2025).
3. Kim B.-H., Yoon Y. Cloud Storage Service Architecture Providing the Eventually Consistent Totally Ordered Commit History of Distributed Key-Value Stores for Data Consistency Verification. *Electronics*. 2021. Т. 10, № 21. С. 2702. URL: https://doi.org/10.3390/electronics10212702 (дата звернення: 09.05.2025).
4. Lardinois F. Google Drive will hit a billion users this week | TechCrunch. *TechCrunch*. URL: https://techcrunch.com/2018/07/25/google-drive-will-hit-a-billion-users-this-week (дата звернення: 09.05.2025).
5. Yoon M. Top 6 Pros and Cons of Using OneDrive. *CrashPlan*. URL: https://www.crashplan.com/blog/top-6-pros-and-cons-of-using-onedrive/ (дата звернення: 10.05.2025).
6. Microservices architecture in cloud-native applications: Design patterns and scalability / Oyekunle Claudius Oyeniran та ін. Computer Science & IT Research Journal. 2024. Т. 5, № 9. С. 2107–2124. URL: https://doi.org/10.51594/csitrj.v5i9.1554 (дата звернення: 11.05.2025).
7. de Almeida M. G., Canedo E. D. Authentication and Authorization in Microservices Architecture: A Systematic Literature Review. Applied Sciences. 2022. Т. 12, № 6. С. 3023. URL: https://doi.org/10.3390/app12063023 (дата звернення: 11.05.2025).
8. Ma R., Zhou W., Ma Z. An Efficient NoSQL-Based Storage Schema for Large-Scale Time Series Data. Journal of Database Management. 2024. Т. 35, № 1. С. 1–21. URL: https://doi.org/10.4018/jdm.339915 (дата звернення: 11.05.2025).
9. The Role of Caching in Future Communication Systems and Networks / G. S. Paschos та ін. IEEE Journal on Selected Areas in Communications. 2018. Т. 36, № 6. С. 1111–1125. URL: https://doi.org/10.1109/jsac.2018.2844939 (дата звернення: 11.05.2025).
10. Vijay Kumar Pasunoori. Ensuring Resilience in Microservices with Cloud-Native API Gateways. International Journal of Scientific Research in Computer Science, Engineering and Information Technology. 2025. Т. 11, № 1. С. 548–555. URL: https://doi.org/10.32628/cseit25111252 (дата звернення: 11.05.2025).
11. Network technology for transmission of visual information / S. Bielievtsov та ін. Інформаційні технології і безпека : Матеріали XVIII Міжнар. науково-практ. конф. ІТБ-2018, м. Kyiv, 18 груд. 2018 р. Kyiv, 2018. С. 359. URL: https://www.researchgate.net/publication/330601501\_Network\_technology\_for\_transmission\_of\_visual\_information (дата звернення: 11.05.2025).
12. Welcome | Fiber. Welcome | Fiber. URL: https://docs.gofiber.io/ (date of access: 11.05.2025).
13. Kieltyka P. go-chi docs. chi. URL: https://go-chi.io/#/README (дата звернення: 11.05.2025).
14. MongoDB, Inc. What is MongoDB? - Database Manual v8.0 - MongoDB Docs. MongoDB: The World’s Leading Modern Database | MongoDB. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 11.05.2025).
15. Буцулін І. Проектування серверної частини програмної системи хмарного зберігання файлів в контексті потреби в серверному кешуванні для потокового отримання великих файлів. Науковий простір: актуальні питання, досягнення та інновації: зб. Наук. Пр. З матеріалами ix міжнар. Наук. Конф., м. М. Вінниця, 23 трав. 2025 р. Вінниця, 2025. С. 728.
16. Буцулін І. Репозиторій застосунку. URL: https://github.com/NureButsulinIhor/2025\_B\_PI\_PZPI-21-2\_Butsulin\_I\_O.git (дата звернення: 11.06.2025).

ДОДАТОК А

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ

A screenshot of a document

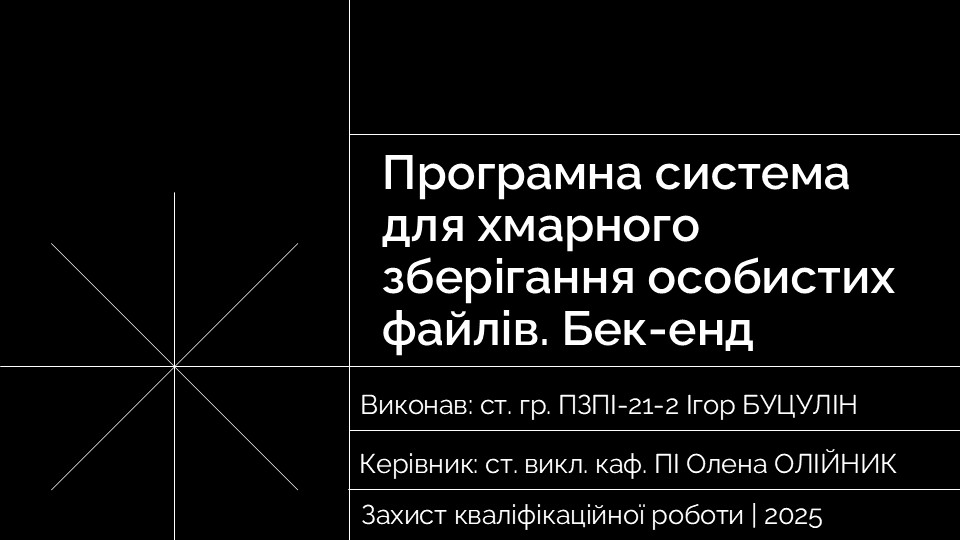
AI-generated content may be incorrect.

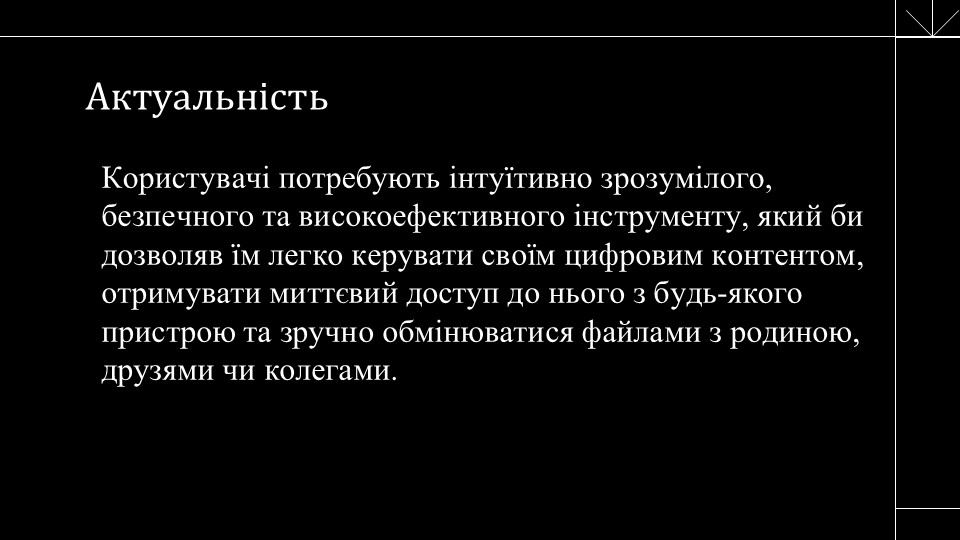
A screenshot of a computer

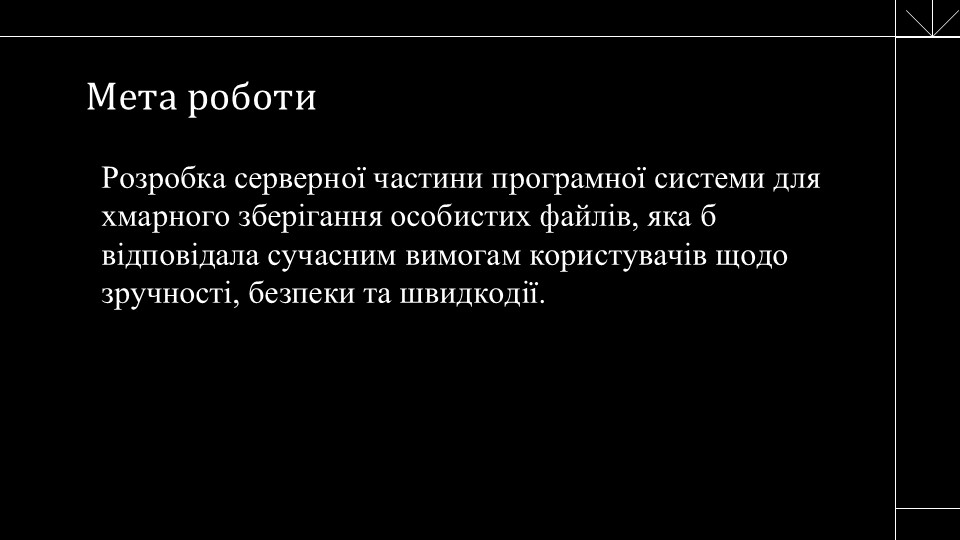
AI-generated content may be incorrect.

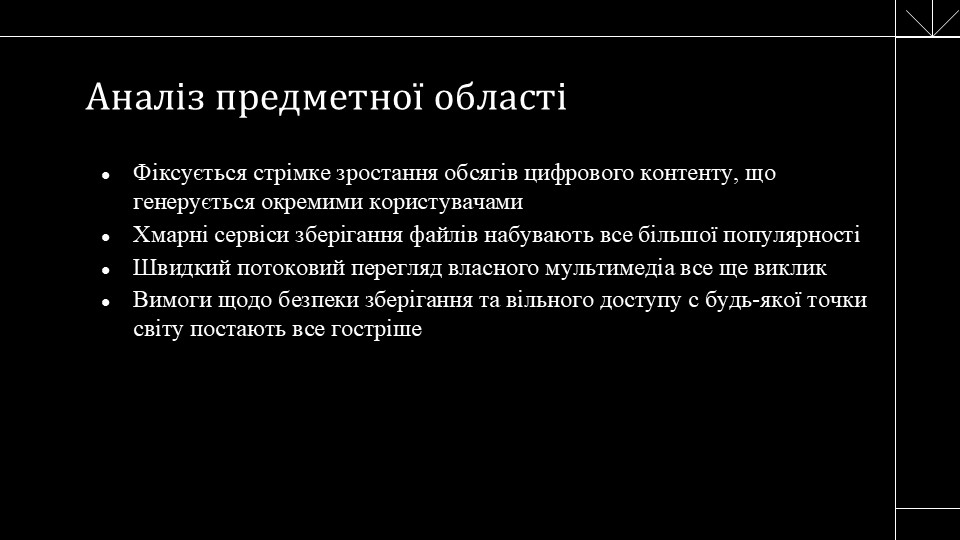
ДОДАТОК Б

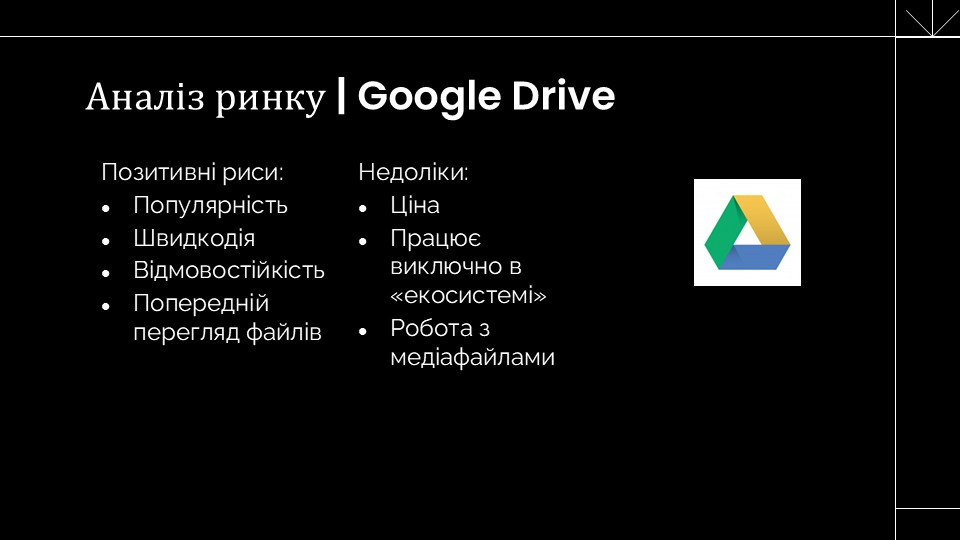
Слайди презентації

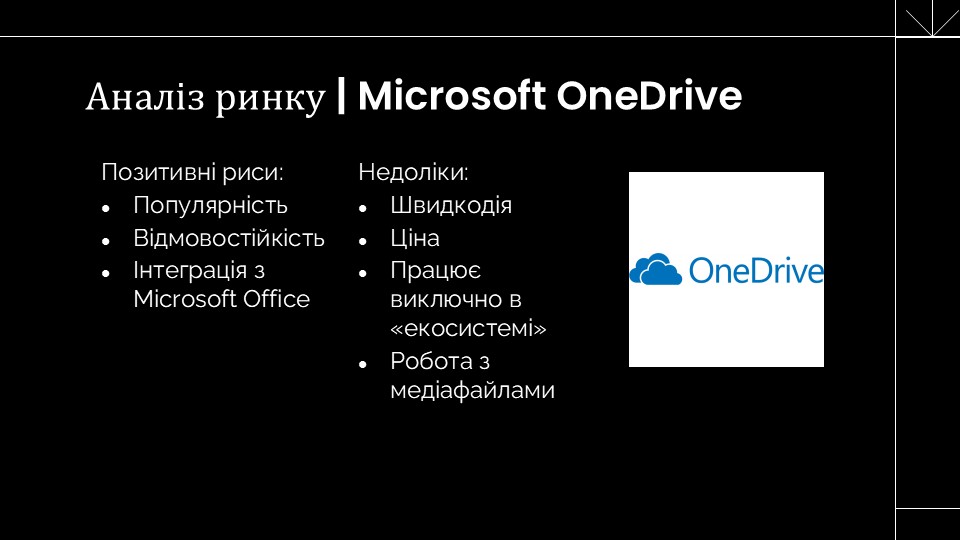


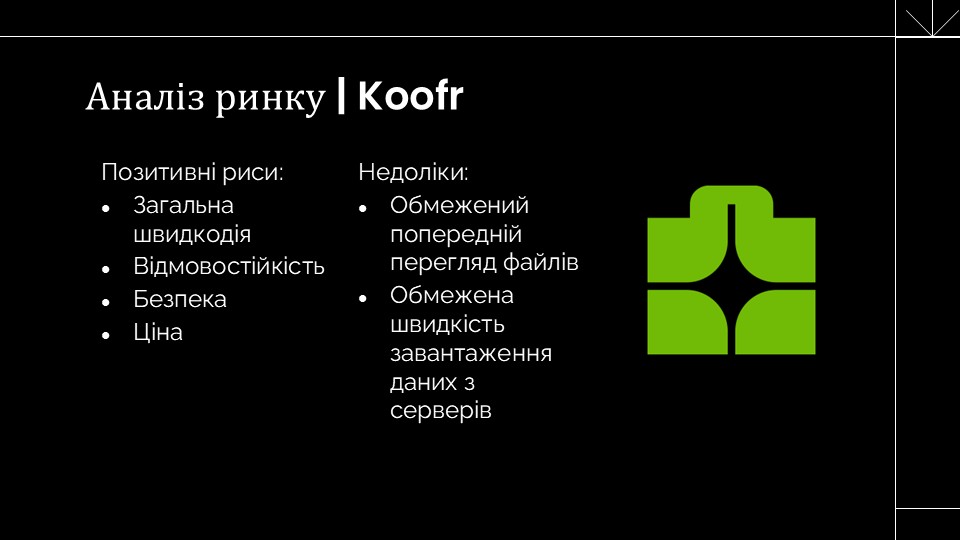


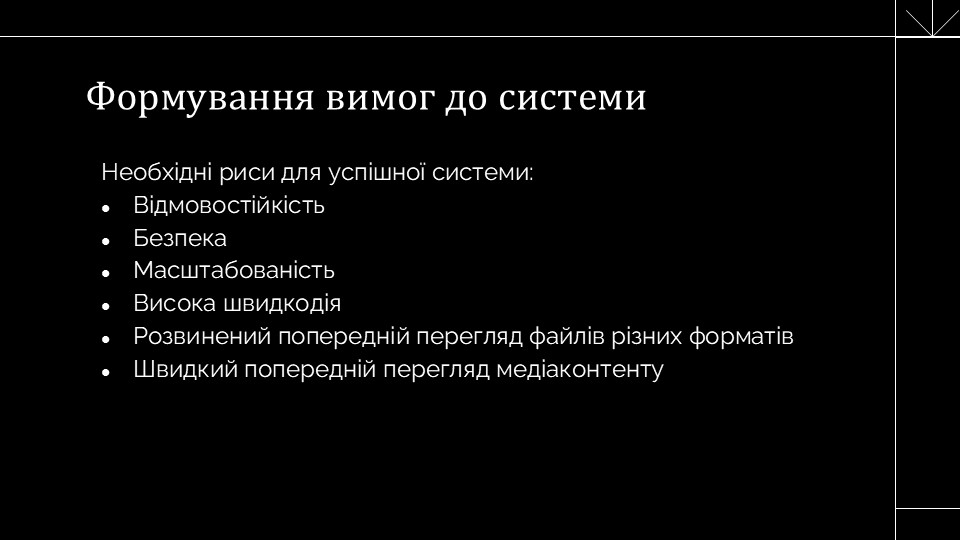












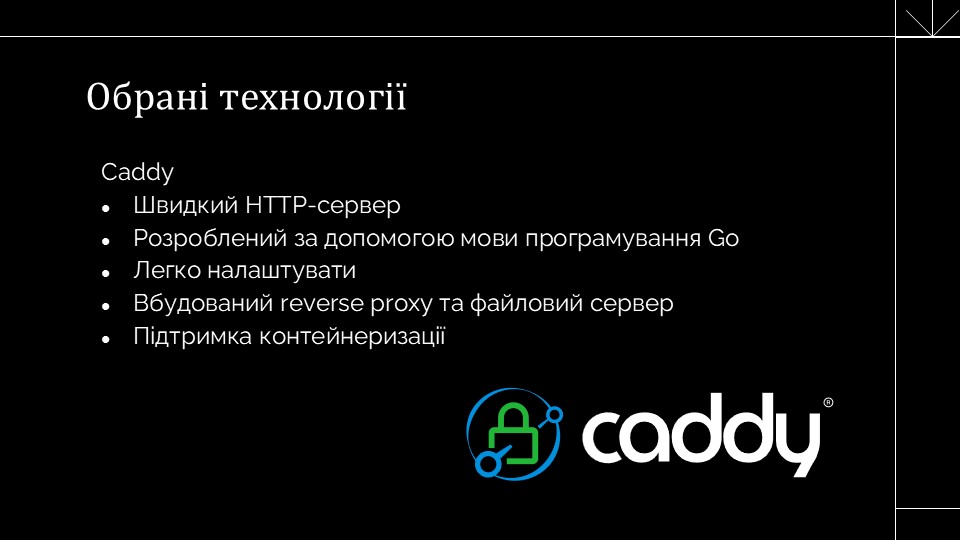








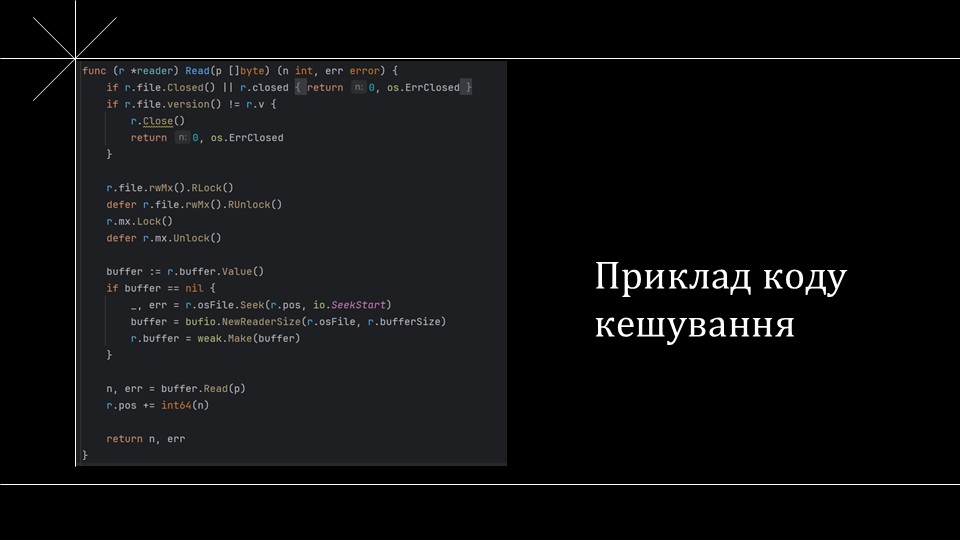


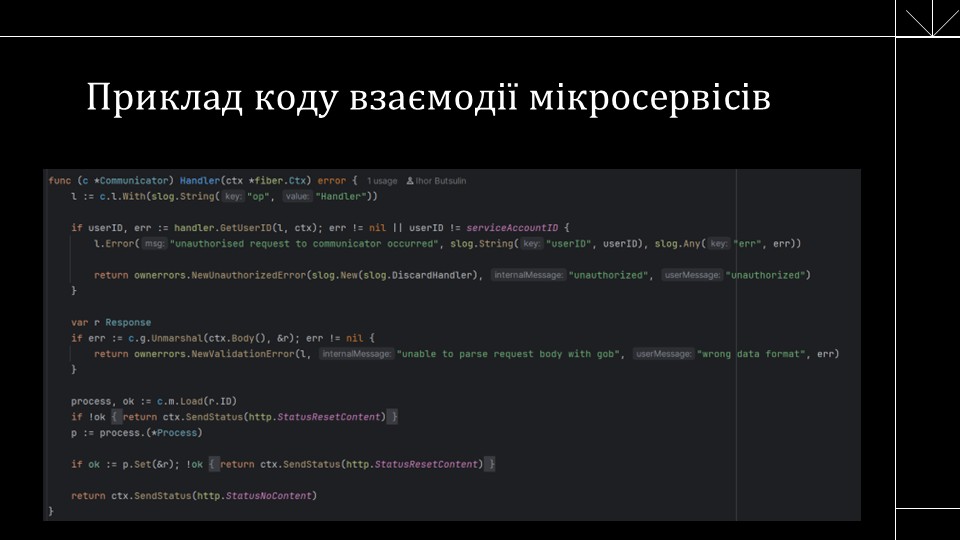


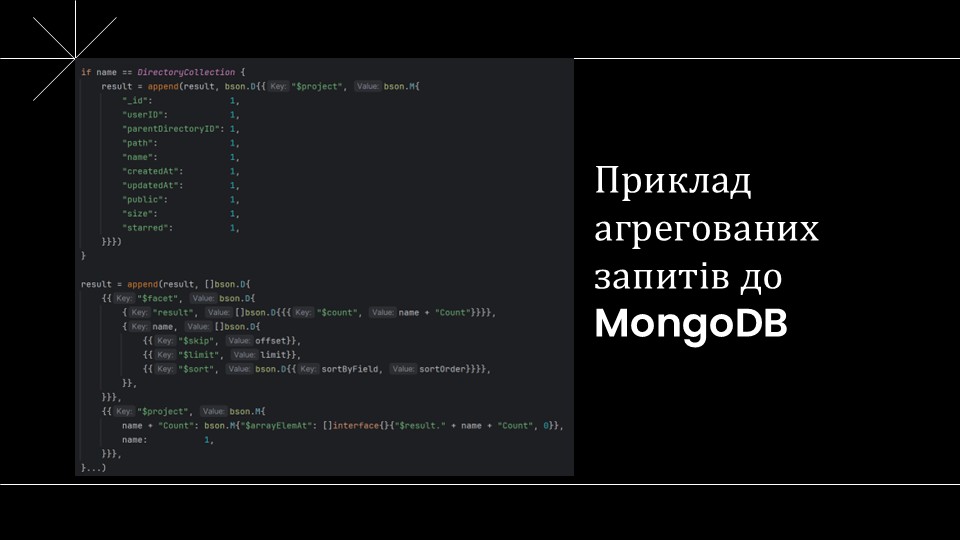




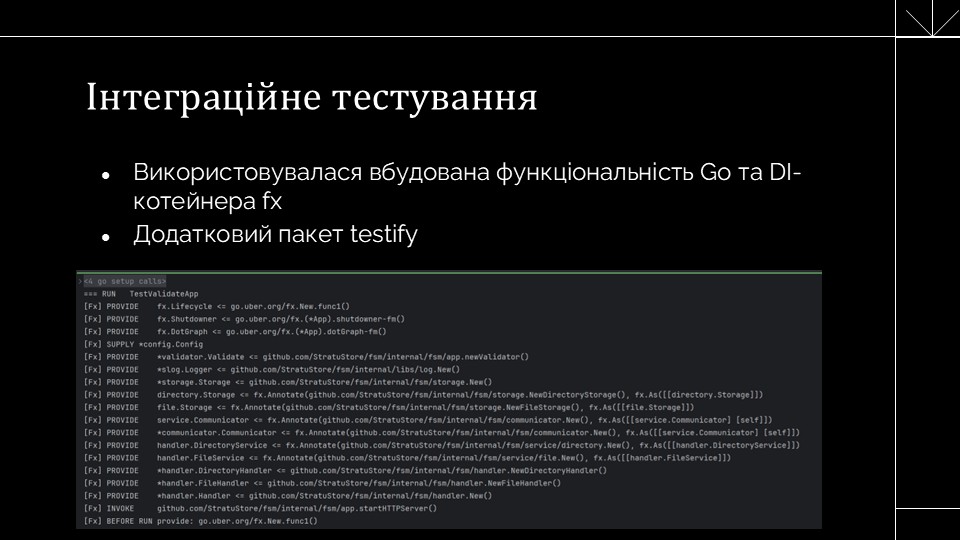


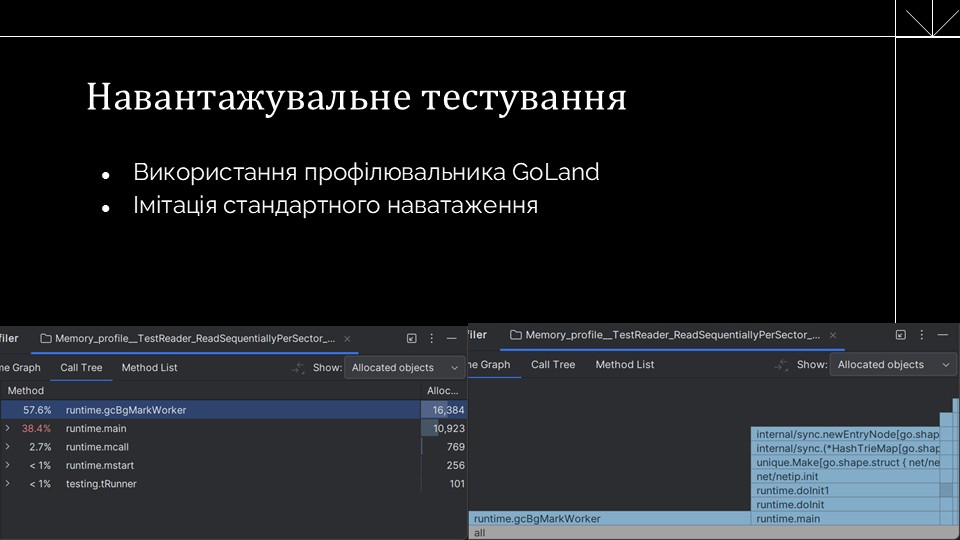




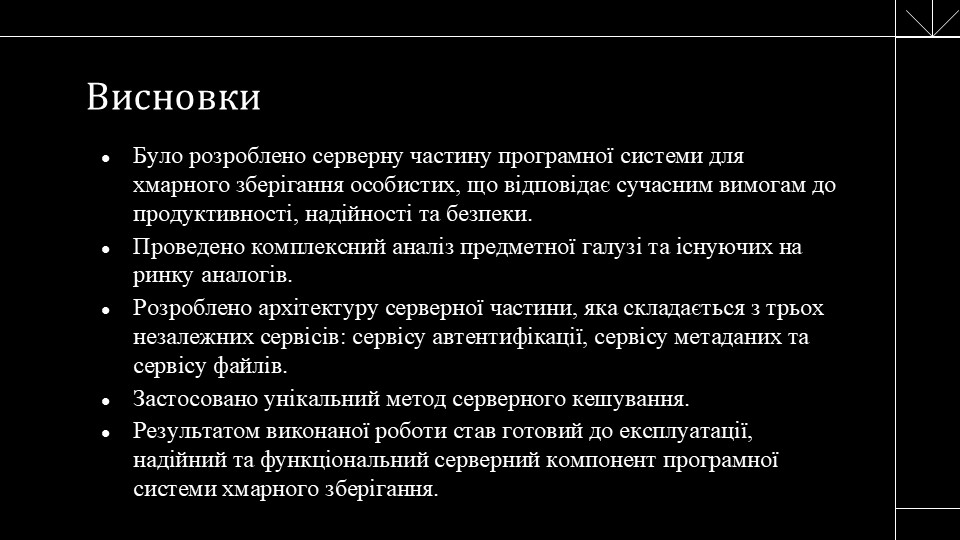


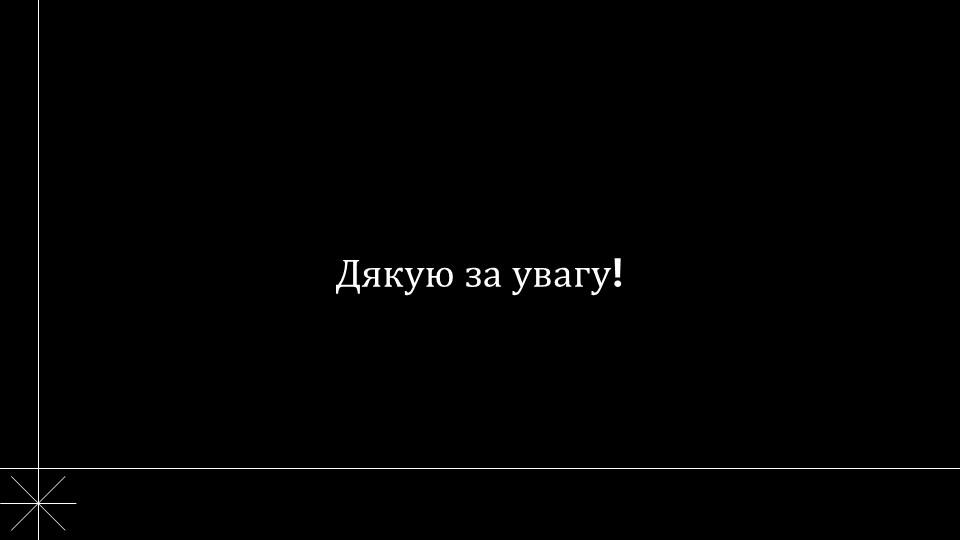












ДОДАТОК В

Тези доповіді

A paper with text on it

AI-generated content may be incorrect.

A close up of a document

AI-generated content may be incorrect.

A close up of a document

AI-generated content may be incorrect.

A close up of a document

AI-generated content may be incorrect.

A paper with text and black text

AI-generated content may be incorrect.

A certificate with text and images

AI-generated content may be incorrect.