МАГІСТЕРСЬКА РОБОТА

МР. ІПм - 66.00.00.000 ПЗ

Група ІПм-23-3

Пірко Віталій

2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Пірко Віталій Андрійович

(прізвище, ім'я, по батькові)

УДК 004.942

(індекс)

МАГІСТЕРСЬКА РОБОТА

Моделі, методи та алгоритми організації віддаленого доступу та управління файловою системою мобільного пристрою з використанням веб-технологій

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121 - Інженерія програмного

(шифр і назва спеціальності)

Пірко В. А.

(підпис, ініціали та прізвище здобувача освітнього ступеня)

Науковий керівник Зікратий Сергій Вікторович, к.т.н., доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту

Завідувач кафедри

доц. Бандура В.В.

(посада) (підпис) (дата) (ініціали та прізвище)

Нормоконтроль

доц. Вовк Р.Б.

(посада) (підпис) (дата) (ініціали та прізвище)

Робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Івано-Франківськ – 2024

Івано-Франківський національний технічний університет нафти і газу

Інститут інформаційних технологій

Кафедра інженерії програмного забезпечення

Освітній рівень магістр

Спеціальність 121 – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедрою **ІПЗ**
доц. В.В. Бандура
“ 04 ” вересня 2024 р.

ЗАВДАННЯ

НА МАГІСТЕРСЬКУ РОБОТУ СТУДЕНТУ

Пірку Віталію Андрійовичу

(прізвище, ім’я, по-батькові)

1. Тема магістерської роботи **Моделі, методи та алгоритми організації віддаленого доступу та управління файловою системою мобільного пристрою з використанням веб-технологій”**

керівник проекту (роботи) Зікратий Сергій Вікторович, к.т.н., доцент

затверджені наказом закладу вищої освіти від “ 22 ” листопада 2024 р. № 781 /7

2. Строк подання студентом проекту (роботи) 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) Архітектура, формальні моделі та алгоритми організації безпечного віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій в межах локальної мережі.

4. Зміст розрахунково - пояснювальної записки(перелік питань, які потрібно розробити)

1. Дослідження предметної області та аналіз існуючих рішень для віддаленого доступу до файлової системи мобільних пристроїв
2. Проектування системи віддаленого доступу
3. Розробка моделей та алгоритмів взаємодії між мобільним пристроєм та веб-клієнтом
4. Реалізація та аналіз результатів

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

1. Діаграма порядку взаємодії між різними сутностями моделі системи (рис. 2.2)
2. Діаграма послідовності роботи алгоритму передачі даних (рис. 2.8)
3. Діаграма архітектури сервера (рис. 2.9)
4. Діаграма архітектури Android-модулю (рис. 2.10)
5. Діаграма архітектури веб-застосунку (рис. 2.11)

6. Консультанти розділів проекту (роботи)

Розділ	Консультант	Підпис, дата
Перевірка на плагіат	доц., к.т.н. Вовк Р.Б.	

7. Дата видачі завдання 04 вересня 2024 р.

Керівник _____
(підпис)

Завдання прийняв до виконання_____
(підпис)

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назви етапів магістерської роботи	Строк виконання етапів роботи	Примітка
1	Підбір і вивчення літератури по темі магістерської роботи	15.09.2024	виконано
2	Аналіз концепцій та алгоритмів предметної області	29.09.2024	виконано
3	Дослідження предметної області та аналіз існуючих рішень для віддаленого доступу до файлової системи мобільних пристроїв	15.10.2024	виконано
4	Проектування системи віддаленого доступу	08.11.2024	виконано
5	Розробка моделей та алгоритмів взаємодії між мобільним пристроєм та веб-клієнтом	20.11.2024	виконано
6	Реалізація та аналіз результатів	01.12.2024	виконано
7	Затвердження пояснювальної записки роботи завідувачем кафедри	15.12.2024	виконано

Студент – магістр _____
(підпис)

Керівник роботи _____
(підпис)

АНОТАЦІЯ

Магістерська робота: 97 сс., 41 рис., 2 табл., 45 джерел.

Тема: Моделі, методи та алгоритми організації віддаленого доступу та управління файловою системою мобільного пристрою з використанням веб-технологій.

Об'єкт дослідження: процеси віддаленого доступу та управління файловою системою мобільних пристроїв.

Мета роботи: розробка та дослідження моделей, методів та алгоритмів організації віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій.

Предмет дослідження: моделі, методи та алгоритми організації безпечного віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій.

Результати дослідження:

Розроблено архітектуру системи, де мобільний пристрій виконує подвійну роль - як клієнтського додатку та веб-сервера. Реалізовано механізми безпечного віддаленого доступу через веб-інтерфейс з використанням унікальних сесійних ключів та QR-кодів.

Висновок:

В результаті роботи створено програмне рішення для безпечного віддаленого доступу до файлової системи мобільного пристрою, що працює виключно в межах локальної мережі та не вимагає встановлення додаткового програмного забезпечення на клієнтські пристрої.

ВІДДАЛЕНИЙ ДОСТУП, ANDROID-РОЗРОБКА, ВЕБ-ТЕХНОЛОГІЇ, ФАЙЛОВА СИСТЕМА, МОБІЛЬНІ ПРИСТРОЇ, БЕЗПЕКА ДАНИХ, ЛОКАЛЬНА МЕРЕЖА, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, ВЕБ-ІНТЕРФЕЙС.

ABSTRACT

Master's thesis: 97 pages, 41 figure, 2 tables, 45 references.

Topic: Models, methods and algorithms for organizing remote access and management of mobile device file system using web technologies.

Object of research: processes of remote access and management of mobile devices' file system.

Purpose: development and research of models, methods and algorithms for organizing remote access to the mobile device file system using web technologies.

Subject of research: models, methods and algorithms for organizing secure remote access to the mobile device file system using web technologies.

Research results:

The system architecture has been developed where the mobile device performs a dual role - as a client application and a web server. Mechanisms for secure remote access through a web interface have been implemented using unique session keys and QR codes.

Conclusion:

As a result, a software solution has been created for secure remote access to the mobile device file system that works exclusively within a local network and does not require installation of additional software on client devices.

REMOTE ACCESS, ANDROID DEVELOPMENT, WEB TECHNOLOGIES, FILE SYSTEM, MOBILE DEVICES, DATA SECURITY, LOCAL NETWORK, CLIENT-SERVER ARCHITECTURE, WEB INTERFACE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API - прикладний програмний інтерфейс

DOM - об'єктна модель документа

HTTP - протокол передачі гіпертексту

HTTPS - захищений протокол передачі гіпертексту

JSON - формат запису об'єктів JavaScript

QR - швидкий відгук

SDK - набір засобів розробки

SPA - односторінковий додаток

UI - користувацький інтерфейс

Wi-Fi - бездротова локальна мережа

ПЗ - програмне забезпечення

ВСТУП

Актуальність теми.

Стрімкий розвиток мобільних технологій та зростання обсягів даних, що зберігаються на смартфонах, створюють потребу в ефективних інструментах для віддаленого доступу до файлової системи мобільних пристроїв. За даними досліджень, середній обсяг пам'яті на мобільному пристрої зріс з 32 ГБ у 2015 році до 128 ГБ у 2023 році [42], що свідчить про значне збільшення потреби в зручних механізмах управління файлами.

Актуальність теми дослідження обумовлена необхідністю розробки ефективних рішень для віддаленого доступу до файлової системи мобільних пристроїв, які б не вимагали встановлення додаткового програмного забезпечення на клієнтські пристрої та працювали виключно в межах локальної мережі. Існуючі рішення часто залежать від хмарних сервісів, вимагають створення облікових записів або встановлення спеціалізованого програмного забезпечення, що створює додаткові бар'єри для користувачів.

Мета дослідження – розробка та дослідження моделей, методів та алгоритмів організації віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій.

Для досягнення поставленої мети необхідно вирішити такі **завдання**:

1. Проаналізувати існуючі рішення для віддаленого доступу до файлової системи мобільних пристроїв та визначити їх обмеження
2. Розробити архітектуру системи, що забезпечує безпечний віддалений доступ до файлової системи через веб-інтерфейс
3. Реалізувати механізми автентифікації та захисту даних при роботі в локальній мережі
4. Створити зручний веб-інтерфейс для управління файловою системою
5. Провести тестування та оцінку ефективності розробленого рішення

Об'єкт дослідження – процеси віддаленого доступу та управління файловою системою мобільних пристроїв.

Предмет дослідження – моделі, методи та алгоритми організації безпечного віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій.

Методи дослідження базуються на теорії програмування, методах об'єктно-орієнтованого проектування, технологіях розробки веб-додатків та мобільних застосунків, принципах побудови розподілених систем.

Наукова новизна роботи полягає в розробці нового підходу до організації віддаленого доступу до файлової системи мобільного пристрою, який, на відміну від існуючих рішень, не вимагає встановлення додаткового програмного забезпечення на клієнтські пристрої та забезпечує роботу виключно в межах локальної мережі.

Практичне значення отриманих результатів полягає у створенні програмного рішення, яке дозволяє організувати безпечний віддалений доступ до файлової системи мобільного пристрою через веб-інтерфейс, що може бути використано як в особистих, так і в професійних цілях.

Структура роботи.

Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Загальний обсяг роботи становить 97 сторінок.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

1.1. Актуальність проблеми віддаленого доступу до файлової системи мобільних пристроїв

З кожним роком спостерігається стрімке зростання кількості даних, що зберігаються на мобільних пристроях. Користувачі все частіше використовують смартфони не лише як засоби комунікації, але і як основні пристрої для зберігання важливої інформації - від персональних документів до робочих файлів. За даними дослідження компанії Statista, середній обсяг пам'яті на мобільному пристрої зріс з 32 ГБ у 2015 році до 128 ГБ у 2023 році, що свідчить про значне збільшення потреби в зберіганні даних.

Проте організація зручного доступу до файлової системи мобільного пристрою з інших пристроїв залишається актуальною проблемою. Традиційні методи, такі як підключення через USB-кабель, мають суттєві обмеження - необхідність фізичної присутності та використання спеціальних драйверів. Хмарні сховища, які часто розглядаються як альтернатива, також мають ряд недоліків: обмежений безкоштовний простір, залежність від швидкості інтернет-з'єднання та необхідність попереднього завантаження файлів у хмару.

Особливої актуальності ця проблема набуває в контексті роботи з локальною мережею, де існує потреба швидкого обміну файлами між пристроями без використання інтернету. Наприклад, при проведенні презентацій, коли необхідно швидко отримати доступ до файлів на телефоні з ноутбука, або в домашніх умовах при бажанні переглянути фотографії з телефону на великому екрані комп'ютера.

Окрім того, існуючі рішення часто вимагають встановлення додаткового програмного забезпечення на всі пристрої, що ускладнює процес

налаштування та обмежує можливості використання на чужих пристроях. Тому розробка системи, що дозволяє організувати віддалений доступ до файлової системи мобільного пристрою через веб-інтерфейс, є актуальним завданням, що має практичну цінність для широкого кола користувачів.

Важливим аспектом також є питання безпеки даних при організації такого доступу. В умовах зростання кількості кіберзагроз, необхідно забезпечити захищений механізм передачі даних, який би працював виключно в межах довіреної локальної мережі та не створював додаткових вразливостей для пристрою користувача.

Саме тому перспективним напрямком є розробка рішення, що використовує сучасні веб-технології для організації такого доступу. Веб-технології дозволяють реалізувати кросплатформний інтерфейс, доступний з будь-якого пристрою, що має браузер, без необхідності встановлення додаткового програмного забезпечення. При цьому використання локальної мережі як середовища передачі даних забезпечує достатню швидкість для комфортної роботи з файлами та природний периметр безпеки.

Реалізація мобільного пристрою в ролі веб-сервера також відкриває нові можливості для розширення функціоналу - від простого перегляду файлів до повноцінного файлового менеджера з можливістю виконання базових операцій над файлами. Це особливо актуально в контексті розвитку концепції Internet of Things (IoT), де мобільні пристрої все частіше виступають не лише споживачами, але й постачальниками послуг.

Таким чином, розробка системи віддаленого доступу до файлової системи мобільного пристрою з використанням веб-технологій є актуальним завданням, що відповідає сучасним тенденціям розвитку мобільних технологій та потребам користувачів. Вирішення цієї задачі дозволить спростити процес доступу до файлів на мобільному пристрої та підвищити ефективність роботи з ними в локальній мережі.

1.2. Огляд існуючих рішень та їх обмежень

На даний момент існує велика кількість програмних систем, які надають можливості для передачі даних між різними пристроями. Багато з них здатні також зберігати їх на окремих серверах та надавати інші додаткові послуги.

1.2.1. Спеціалізовані додатки для віддаленого доступу

На ринку представлено кілька спеціалізованих рішень для віддаленого доступу до файлової системи мобільних пристроїв. Одним з найвідоміших є AirDroid, який пропонує веб-інтерфейс для повного контролю над Android-пристроєм. До основних переваг цього рішення належать:

- Широкий функціонал для управління файлами;
- Висока швидкість передачі даних (до 20 МБ/с у локальній мережі);
- Інтегровані можливості віддаленого управління.

Проте, AirDroid та подібні рішення мають суттєві обмеження:

- Ліміт на розмір файлів у безкоштовній версії (зазвичай 30 МБ);
- Обов'язкова реєстрація облікового запису;
- Залежність від зовнішніх серверів навіть при локальній роботі;
- Необхідність постійного інтернет-з'єднання.

Альтернативні рішення, такі як Join та AirMore, пропонують схожий функціонал, але також страждають від типових проблем:

- Залежність від хмарної інфраструктури;
- Необхідність встановлення клієнтського ПЗ;
- Обмежена функціональність у безкоштовних версіях.

1.2.2. Файлові менеджери з підтримкою мережевих протоколів

Окрему категорію становлять розширені файлові менеджери, як-от Solid Explorer та ES File Explorer Pro. Їх особливості:

Solid Explorer має наступний перелік можливостей:

- Підтримка FTP, SFTP та WebDAV протоколів;
- Двопанельний інтерфейс;
- Вбудоване шифрування файлів.

Основні обмеження:

- Відсутність веб-інтерфейсу;
- Складність налаштування мережевих з'єднань;
- Платна ліцензія для повного функціоналу.

ES File Explorer Pro має наступний перелік можливостей:

- Вбудований FTP-сервер;
- Підтримка Wi-Fi Direct;
- Глибока інтеграція з мережевими ресурсами.

Проблеми:

- Проблеми з безпекою;
- Агресивна рекламна політика;
- Відсутність активної підтримки.

1.2.3. Рішення на основі хмарних технологій

Хмарні сервіси, такі як Google Drive, Dropbox та OneDrive, часто використовуються для віддаленого доступу до файлів.

Google Drive:

- 15 ГБ безкоштовного простору;
- Глибока інтеграція з Android;
- Розвинена система синхронізації.

Dropbox:

- LAN-синхронізація;
- Надійна система версіонування;
- Численні інтеграції.

Обмеження хмарних рішень:

- Необхідність попереднього завантаження файлів;
- Залежність від інтернет-з'єднання;
- Питання приватності даних;
- Обмежений безкоштовний простір (особливо у Dropbox - 2 ГБ).

1.2.4. Рішення від виробників пристроїв

Провідні виробники мобільних пристроїв пропонують власні екосистемні рішення:

Samsung:

- Samsung Flow;
- Quick Share;
- Samsung DeX;
- Link to Windows.

Huawei:

- Huawei Share;
- Multi-Screen Collaboration;
- EMUI Share.

Основні обмеження:

- Жорстка прив'язка до екосистеми виробника;
- Необхідність використання сумісних пристроїв;
- Закритість рішень.

1.2.5. Протокольні рішення

Стандартні мережеві протоколи забезпечують універсальний спосіб доступу до файлів:

FTP/SFTP:

- Універсальна підтримка;
- Висока швидкість передачі;
- Широкий вибір клієнтів.

WebDAV:

- Інтеграція з HTTP;
- Підтримка HTTPS;
- Робота з метаданими.

Обмеження:

- Складність налаштування на мобільних пристроях;
- Нижча швидкість передачі у WebDAV;
- Проблеми з безпекою при неправильній конфігурації.

1.2.6. Порівняльний аналіз рішень

На таблиці 1.1 зображено зібрані з усіх типів існуючих рішень результати порівняння для наочної оцінки.

Таблиця 1.1

Результати порівняння існуючих програмних рішень

Критерій	**Спеціалізовані додатки**	**Файлові менеджери**	**Хмарні рішення**	**Рішення виробників**	**Протокольні рішення**
Простота використання	Висока	Середня	Висока	Висока	Низька
Безпека	Середня	Середня	Висока	Висока	Змінна
Швидкість передачі	Висока	Висока	Залежить від інтернету	Висока	Висока
Незалежність від інтернету	Ні	Так	Ні	Частково	Так
Кросплатформність	Так	Обмежена	Так	Ні	Так

Продовження таблиці 1.1

Вартість	Платні/Freemium	Переважно платні	Freemium	Безкоштовні	Безкоштовні

Проведений аналіз показує, що існуючі рішення мають суттєві обмеження для повсякденного використання, особливо в контексті локальної мережі. Це підтверджує необхідність розробки нового рішення, яке б поєднувало простоту використання з незалежністю від зовнішніх сервісів та не вимагало встановлення додаткового програмного забезпечення на клієнтські пристрої.

1.3. Формування вимог до системи

Сучасне ПЗ досягло високої швидкості створення, проте слід враховувати, що його гнучкість неухильно зменшується по мірі розробки. Тому дуже важливо заздалегідь визначити основні вимоги, які система повинна задовільняти, щоб на стадії її реалізації направляти її в потрібному напрямку.

1.3.1. Функціональні вимоги

На основі проведеного аналізу предметної області та дослідження потреб користувачів було сформовано комплекс функціональних вимог до розроблюваної системи. Особлива увага приділялася забезпеченню інтуїтивно зрозумілого інтерфейсу та повноцінних можливостей управління файловою системою мобільного пристрою.

Центральним елементом мобільного додатку є система відображення інформації про стан пристрою. Користувач повинен мати можливість в режимі реального часу відслідковувати рівень заповнення пам'яті пристрою через візуальний індикатор, що відображає як загальний обсяг доступного

простору, так і поточний рівень його використання. Система також має надавати механізми швидкого підключення інших пристроїв через генерацію та відображення QR-коду, який містить всю необхідну інформацію для встановлення з'єднання. Для випадків, коли сканування QR-коду неможливе або незручне, система повинна відображати текстовий варіант посилання для підключення.

Ключовим функціональним блоком системи є механізми навігації по файловій структурі пристрою. Користувач повинен мати можливість переглядати повну ієрархію директорій, починаючи з кореневої папки, та здійснювати навігацію між різними рівнями структури. При цьому система має забезпечувати відображення детальної інформації про кожен файл, включаючи його розмір, дату створення та останньої модифікації. Для зручності роботи з великими наборами файлів необхідна реалізація механізмів сортування за різними параметрами, що дозволить користувачам швидко знаходити потрібні файли.

Особливу увагу при формуванні вимог було приділено функціональності для роботи з файлами. Система повинна підтримувати повний спектр базових операцій, включаючи завантаження файлів як з мобільного пристрою на клієнтський пристрій, так і в зворотному напрямку. При цьому процес передачі файлів має супроводжуватися індикацією прогресу та можливістю скасування операції. Користувачі також повинні мати можливість виконувати операції з управління файлами, такі як видалення, перейменування та переміщення між директоріями. Всі ці операції мають виконуватися з відповідними підтвердженнями для запобігання випадковим діям.

Важливим аспектом функціональності є система управління сесіями доступу. Кожне підключення до системи має супроводжуватися генерацією унікального сесійного ключа, який забезпечує безпечний доступ до файлової системи. Для підвищення безпеки система повинна автоматично завершувати

неактивні сесії після визначеного періоду відсутності активності. Користувач мобільного пристрою також повинен мати можливість примусово завершити будь-яку активну сесію у випадку необхідності.

Всі перелічені функціональні вимоги спрямовані на створення зручного та безпечного інструменту для віддаленого доступу до файлової системи мобільного пристрою. Їх реалізація дозволить користувачам ефективно управляти файлами на своєму пристрої без необхідності фізичного підключення або встановлення додаткового програмного забезпечення на клієнтські пристрої.

1.3.2. Нефункціональні вимоги

Розробка ефективної системи віддаленого доступу до файлової системи мобільного пристрою вимагає чіткого визначення нефункціональних вимог, які забезпечують якість та надійність рішення. Особлива увага приділяється аспектам продуктивності, надійності, зручності використання та системним вимогам.

В контексті продуктивності система повинна забезпечувати швидкий та ефективний доступ до файлової структури пристрою. Критичним параметром є час відгуку системи при навігації по директоріях, який не повинен перевищувати 500 мілісекунд для забезпечення комфортної роботи користувача. При роботі з файлами система має забезпечувати швидкість передачі даних не менше 50% від максимальної швидкості Wi-Fi мережі, що дозволить ефективно працювати з файлами різного розміру. Важливою вимогою є також підтримка одночасної роботи до 5 клієнтів без суттєвої деградації продуктивності, що забезпечить можливість групової роботи з файлами.

Надійність системи є критичним фактором для забезпечення довіри користувачів. Система повинна демонструвати стабільну роботу при тривалому використанні без втрати продуктивності та витоків пам'яті.

Особливу увагу слід приділити механізмам відновлення після збоїв мережевого з'єднання – система має автоматично відновлювати роботу при відновленні зв'язку. При виконанні файлових операцій критично важливим є збереження цілісності даних навіть у випадку раптового переривання операції, для чого необхідно реалізувати механізми транзакційності та відкату змін.

З точки зору зручності використання, інтерфейс системи повинен бути інтуїтивно зрозумілим та не вимагати спеціальної підготовки користувачів. Важливою особливістю є відсутність необхідності встановлення додаткового програмного забезпечення на клієнтські пристрої – достатньо наявності веб-браузера. Веб-інтерфейс має бути адаптивним та коректно відображатися на пристроях з різними розмірами екрану, від смартфонів до десктопних комп'ютерів. Процес підключення до системи повинен бути максимально спрощеним та реалізованим через сканування QR-коду, що дозволить уникнути необхідності ручного введення мережевих параметрів.

Системні вимоги визначають технічні обмеження для роботи системи. Мобільний додаток повинен підтримувати пристрої з Android версії 8.0 і вище, що забезпечить широке охоплення користувачів при збереженні можливості використання сучасних API платформи. Особливу увагу слід приділити оптимізації споживання системних ресурсів [27] у фоновому режимі для мінімізації впливу на роботу інших додатків та тривалість роботи від батареї. Система має працювати в стандартних Wi-Fi мережах без необхідності додаткової конфігурації мережевого обладнання, що спростить процес розгортання та використання.

Реалізація всіх зазначених нефункціональних вимог дозволить створити надійне та зручне рішення для віддаленого доступу до файлової системи мобільного пристрою, яке відповідатиме сучасним стандартам якості програмного забезпечення та очікуванням користувачів.

1.3.3. Вимоги до безпеки

Безпека системи віддаленого доступу до файлової системи мобільного пристрою є критично важливим аспектом, що вимагає комплексного підходу до захисту даних та контролю доступу. При проектуванні системи необхідно врахувати всі потенційні вектори атак та забезпечити надійний захист користувацьких даних без надмірного ускладнення процесу використання системи.

Основним елементом системи захисту доступу є механізм автентифікації на основі унікальних сесійних ключів. Кожне нове підключення до системи повинно супроводжуватися генерацією криптографічно стійкого ключа достатньої довжини, який використовується для ідентифікації клієнта протягом всього сеансу роботи. Важливою вимогою є обмеження терміну дії сесійних ключів та автоматичне завершення сесій після визначеного періоду неактивності, що мінімізує ризики несанкціонованого доступу у випадку компрометації ключа.

Архітектура системи повинна забезпечувати роботу виключно в межах локальної мережі, що створює природний периметр безпеки та знижує ризики зовнішніх атак. При цьому необхідно реалізувати механізми визначення та валідації мережевих з'єднань для запобігання спробам доступу з-поза меж локальної мережі. Особлива увага має бути приділена захисту від несанкціонованого доступу до системних файлів та критичних директорій операційної системи Android.

У контексті захисту даних система повинна забезпечувати комплексну валідацію всіх вхідних даних та файлів. Це включає перевірку коректності шляхів до файлів для запобігання path traversal атакам, валідацію типів файлів при завантаженні для уникнення потенційно небезпечного контенту, а також механізми забезпечення цілісності даних при їх передачі між клієнтом та сервером. Особливо важливим є впровадження механізмів відновлення

після збоїв для запобігання пошкодженню файлової системи при перериванні операцій передачі даних.

Система контролю доступу повинна забезпечувати гранульований підхід до управління правами користувачів. Необхідно реалізувати чітке розмежування доступних операцій з файлами, впровадити механізми логування всіх критичних операцій для можливості аудиту безпеки та розслідування інцидентів. Важливою вимогою є наявність можливості термінового завершення будь-яких активних сесій у випадку виявлення підозрілої активності.

Контроль одночасних підключень до системи є критичним аспектом забезпечення її стабільної роботи та безпеки. Система повинна впровадити багаторівневий механізм моніторингу активних сесій, який включає відстеження кількості підключень з унікальних IP-адрес, аналіз поведінки користувачів та виявлення аномальної активності. Важливо встановити динамічні обмеження на кількість одночасних підключень, які можуть адаптуватися залежно від поточного навантаження на сервер та доступних ресурсів. При досягненні критичних показників система має автоматично застосовувати механізми захисту, такі як тимчасове блокування нових підключень або впровадження додаткової автентифікації.

Для забезпечення коректної роботи з файлами при конкурентному доступі необхідно реалізувати механізм блокування на рівні записів та транзакційну модель обробки даних. Це дозволить уникнути конфліктів при одночасному редагуванні та забезпечити атомарність операцій. Система повинна підтримувати черги запитів та механізми відкату змін у разі виникнення помилок чи конфліктів. Особливу увагу слід приділити оптимізації механізмів блокування для мінімізації затримок при масовому доступі до файлів.

Безпека Android-додатку вимагає комплексного підходу до захисту від різних типів атак. Необхідно використовувати обфускацію коду [37, 40],

перевірку цілісності критичних компонентів додатку при кожному запуску, та впровадити механізми виявлення спроб налагодження чи модифікації коду. Для захисту від реверс-інжинірингу слід застосовувати передові техніки заплутування коду та захисту від декомпіляції. Критичні дані повинні зберігатися в зашифрованому вигляді з використанням AndroidKeyStore та інших системних механізмів безпечного зберігання. Важливо також регулярно оновлювати використовувані алгоритми шифрування та механізми захисту відповідно до актуальних рекомендацій з безпеки.

Архітектура системи безпеки повинна базуватися на модульному принципі з чітко визначеними інтерфейсами між компонентами. Це забезпечить можливість легкого додавання нових механізмів захисту та модифікації існуючих без впливу на інші частини системи. Важливо передбачити механізми конфігурації та налаштування параметрів безпеки через централізований інтерфейс адміністрування. Система повинна підтримувати можливість динамічного оновлення правил безпеки та сигнатур загроз без необхідності перезапуску сервісів. Також важливо забезпечити детальне логування всіх подій безпеки та створити механізми автоматичного аналізу логів для виявлення потенційних загроз.

Висновки до розділу

В результаті проведеного дослідження предметної області та аналізу існуючих рішень було встановлено високу актуальність проблеми організації віддаленого доступу до файлової системи мобільних пристроїв. Стрімке зростання обсягів даних, що зберігаються на мобільних пристроях (з 32 ГБ у 2015 році до 128 ГБ у 2023 році), створює потребу в ефективних інструментах управління файлами Аналіз існуючих рішень виявив їх суттєві обмеження. Спеціалізовані додатки, такі як AirDroid, мають значні

обмеження у безкоштовних версіях та вимагають постійного підключення до інтернету. Файлові менеджери з мережевою функціональністю страждають від складності налаштування та необхідності встановлення додаткового програмного забезпечення. Хмарні рішення, хоч і пропонують зручний інтерфейс, вимагають попереднього завантаження файлів та залежать від швидкості інтернет-з'єднання. Рішення від виробників пристроїв обмежені їхніми екосистемами, а протокольні рішення складні в налаштуванні для пересічного користувача.

На основі проведеного дослідження було сформовано комплексні вимоги до системи, що враховують як функціональні аспекти (навігація по файловій системі, управління файлами, генерація посилань доступу), так і нефункціональні характеристики (продуктивність, надійність, зручність використання). Особливу увагу приділено вимогам безпеки, що є критичним аспектом при роботі з файловою системою пристрою.

Результати дослідження підтверджують необхідність розробки нового рішення, яке б поєднувало простоту використання з незалежністю від зовнішніх сервісів та не вимагало встановлення додаткового програмного забезпечення на клієнтські пристрої. Таке рішення має потенціал значно спростити процес управління файлами на мобільних пристроях та підвищити ефективність роботи користувачів.

РОЗДІЛ 2. МОДЕЛІ, МЕТОДИ ТА АЛГОРИТМИ ОРГАНІЗАЦІЇ ВІДДАЛЕНОГО ДОСТУПУ

2.1. Формальні моделі системи

2.1.1. Модель файлової системи та операцій з нею

При проектуванні моделі файлової системи було застосовано патерн "Фасад", схема якого представлена на рисунку 2.1. Цей патерн забезпечує уніфікований інтерфейс для роботи з файловою системою.

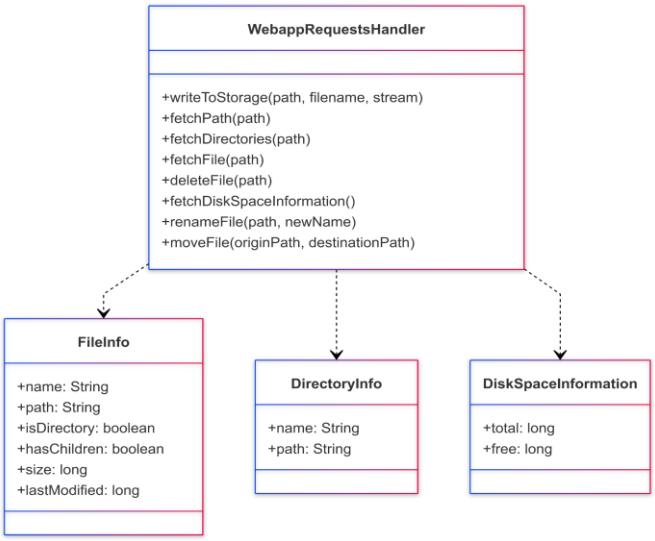


Рис. 2.1. Діаграма залежностей між фасадом та зовнішніми компонентами

Як показано на рисунку 2.1, центральним компонентом є сутність WebappRequestsHandler, яка інкапсулює роботу з файловою системою. Спроектована модель включає наступні функціональні блоки:

1. Операції з файлами через методи сутності WebappRequestsHandler:
    - Читання файлів (fetchFile);
    - Запис файлів (writeToStorage);
    - Видалення файлів (deleteFile);

- Перейменування файлів (renameFile);
- Переміщення файлів (moveFile).

2. Операції з директоріями:
   - Отримання списку файлів через fetchPath з поверненням FileInfo;
   - Навігація по структурі з використанням DirectoryInfo;
   - Отримання метаданих через відповідні поля класів.
3. Інформаційні операції:
   - Отримання інформації про дисковий простір через DiskSpaceInformation;
   - Аналіз властивостей файлів через FileInfo;
   - Управління метаданими директорій через DirectoryInfo.

2.1.2. Модель безпеки та управління сесіями

Для забезпечення безпеки системи було спроектовано багаторівневу модель захисту, процес автентифікації та авторизації якої показано на рисунку 2.2.

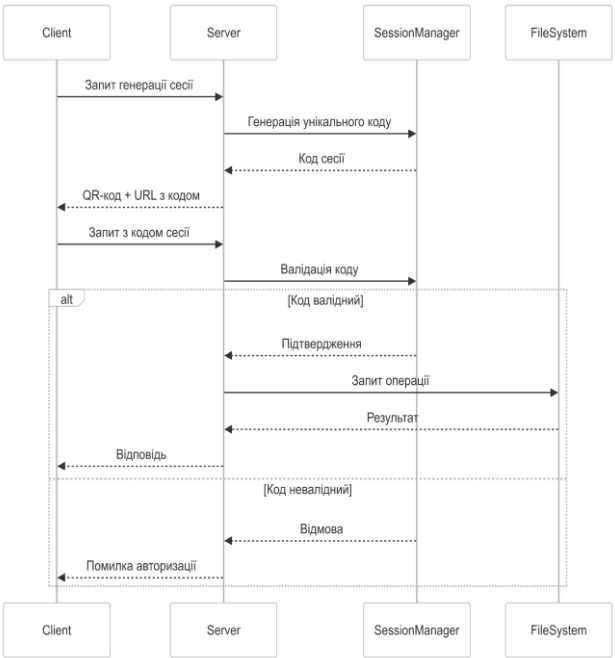


Рис. 2.2. Діаграма порядку взаємодії між різними сутностями моделі системи

Як видно з діаграми на рисунку 2.2, процес автентифікації включає:

1. Управління сесіями:
    - Генерація криптографічно стійких сесійних ключів (як показано на діаграмі 2.3);
    - Обмеження часу життя сесій (5 хвилин);
    - Моніторинг активності сесій через SessionManager;
    - Автоматичне завершення неактивних сесій.
2. Контроль доступу, що реалізується через Security Layer (рисунок 2.1):
    - Валідація всіх вхідних запитів;
    - Перевірка прав доступу до файлів;
    - Санітизація шляхів до файлів;
    - Обмеження доступу до системних директорій.
3. Мережева безпека:
    - Використання локальної мережі;
    - Обмеження доступу за IP-адресою;
    - Валідація параметрів запитів;
    - Захист від CSRF-атак.
4. Аудит та логування:
    - Реєстрація всіх операцій через вбудовані механізми Ktor;
    - Відстеження помилок;
    - Моніторинг активності;
    - Аналіз підозрілої активності.

2.1.3. Модель взаємодії компонентів системи

Комунікація між компонентами системи реалізована через REST API з використанням протоколу HTTP.

На рисунку 2.3 представлено схему взаємодії компонентів.

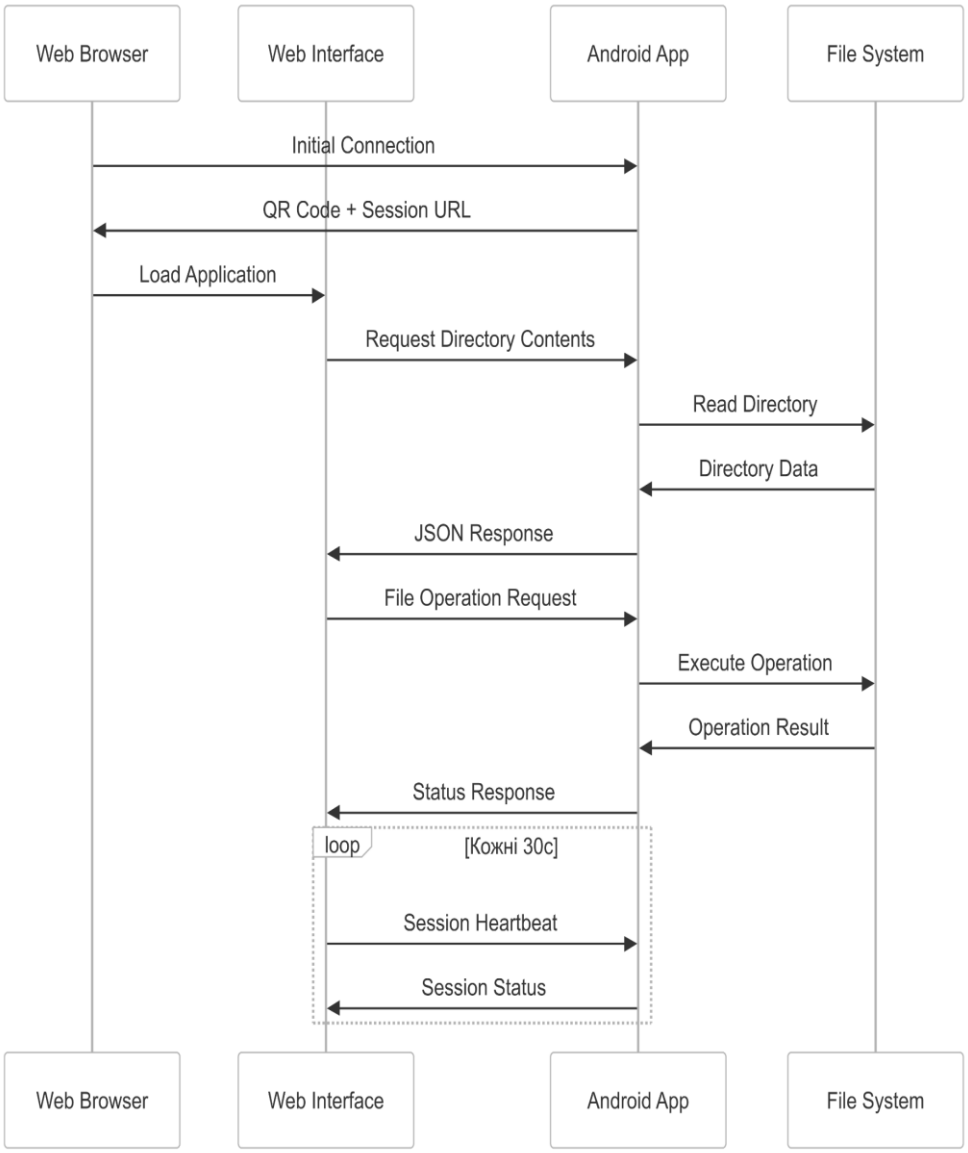


Рис. 2.3. Діаграма мережевої взаємодії між компонентами

Як показано на рисунку 2.3, взаємодія компонентів включає наступні аспекти:

1. Встановлення з'єднання:
    - Генерація сесійного ключа;
    - Створення QR-коду та URL;
    - Ініціалізація веб-інтерфейсу;
    - Валідація з'єднання.
2. Файлові операції:
    - Асинхронна передача команд;

- Обробка результатів операцій;
- Оновлення інтерфейсу;
- Синхронізація стану.

3. Підтримка сесії:
   - Періодичні перевірки активності;
   - Оновлення часу життя сесії;
   - Обробка роз'єднань;
   - Автоматичне відновлення зв'язку.
4. Обробка помилок:
   - Валідація всіх операцій;
   - Логування помилок;
   - Механізми відновлення;
   - Інформування користувача.

2.2 Методи організації віддаленого доступу

2.2.1. Метод генерації та валідації сесійних ключів

Розглянемо метод генерації та валідації сесійних ключів, який є фундаментальним для забезпечення безпечного з'єднання між мобільним пристроєм та веб-клієнтом у системі віддаленого доступу до файлової системи.

Метод базується на концепції одноразових сесійних ключів з обмеженим терміном дії, що забезпечують значний рівень безпеки при встановленні з'єднання між клієнтом та сервером. Такий ключ дозволяє контролювати доступність ресурсів, які надаються серверним компонентом системи тільки для визначеного переліку користувачів, зокрема не тільки за самою його наявністю, а і за часом, протягом якого він вважається дійсним.

Як показано на рисунку 2.4, процес генерації та валідації ключа складається з декількох взаємопов'язаних етапів.

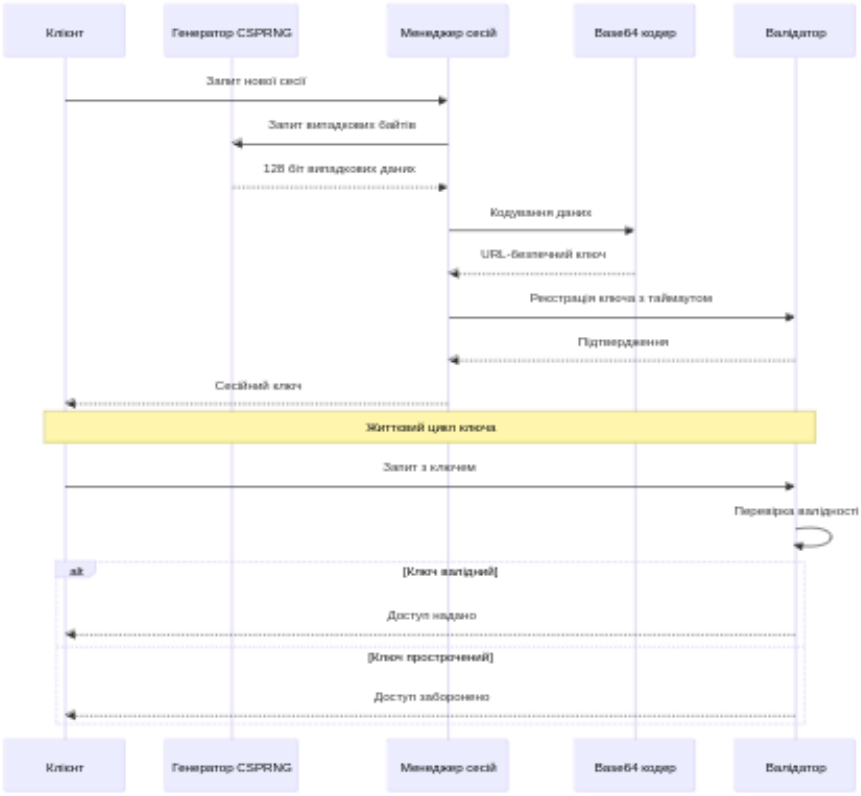


Рис. 2.4. Діаграма послідовності роботи методу генерації та валідації сесійних ключів

Центральним елементом системи є криптографічно стійкий генератор псевдовипадкових чисел, який забезпечує генерацію послідовності випадкових байтів достатньої ентропії. У реалізованій системі використовується 128-бітна послідовність, що забезпечує простір можливих ключів розміром 2^{128}, що робить перебір практично неможливим при сучасному рівні обчислювальних потужностей.

Згенерований ключ проходить процес кодування за допомогою модифікованого алгоритму Base64, спеціально адаптованого для безпечної передачі через URL (URL-safe Base64). Це досягається заміною стандартних символів '+' та '/' на '-' та '_' відповідно, що гарантує коректну обробку ключа всіма компонентами системи передачі даних. Закодований ключ реєструється в системі валідації з прив'язкою до часової мітки створення та встановленим періодом дії.

Валідатор, як показано на рисунку 2.4, здійснює постійний контроль за життєвим циклом ключа, перевіряючи його актуальність при кожному запиті та автоматично відхиляючи запити з простроченими ключами. Це забезпечує додатковий рівень захисту від несанкціонованого доступу навіть у випадку компрометації ключа.

2.2.2. Метод асинхронної передачі даних

Метод асинхронної передачі даних є ключовим компонентом системи, що забезпечує ефективну та надійну взаємодію між клієнтською та серверною частинами при роботі з файловою системою мобільного пристрою. Як показано на рисунку 2.5, метод реалізує багаторівневу архітектуру обробки запитів з використанням черг операцій та асинхронних обробників.

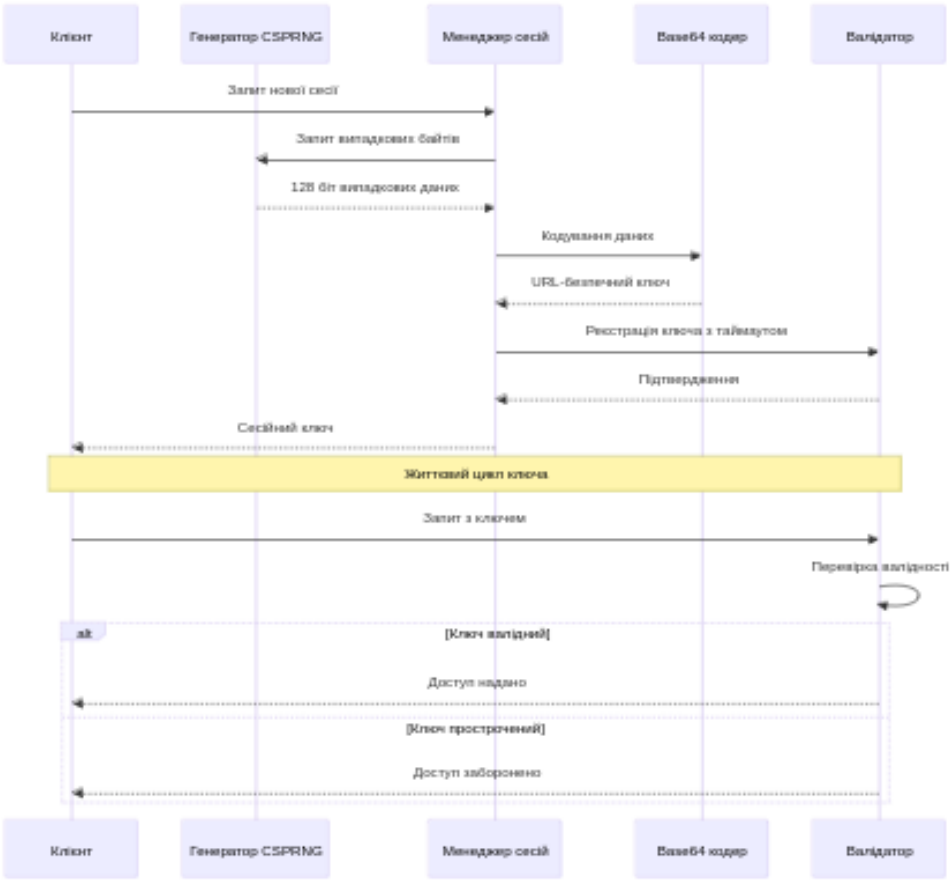


Рис. 2.5. Діаграма послідовності роботи асинхронної передачі даних

Основою методу є принцип неблокуючої взаємодії, при якому операції введення-виведення виконуються паралельно з основним потоком виконання програми. Це досягається шляхом відокремлення процесу ініціації передачі даних від їх фактичної обробки. При надходженні запиту на передачу файлу система створює відповідну операцію в черзі, негайно повертаючи клієнту підтвердження про прийняття запиту до обробки.

Черга операцій виступає в ролі буфера між клієнтськими запитами та системою обробки файлів, забезпечуючи контрольоване навантаження на файлову систему та запобігаючи втраті даних при пікових навантаженнях. Кожна операція в черзі містить метадані про файл, що передається, включаючи його розмір, тип та цільове розташування.

Обробник файлів реалізує механізм поетапної обробки даних з підтримкою відстеження прогресу операції. При роботі з великими файлами дані передаються частинами фіксованого розміру, що дозволяє ефективно управляти використанням пам'яті пристрою та забезпечувати можливість відновлення передачі у випадку збоїв.

Важливим аспектом методу є система зворотного зв'язку, яка забезпечує клієнта актуальною інформацією про стан передачі даних. Це реалізується через механізм подій, які генеруються на кожному етапі обробки файлу: початок передачі, оновлення прогресу, завершення операції або виникнення помилки.

Метод також включає механізми обробки помилок. Помилки поетапно ескалюються компонентами системи для подальшого перетворення в інформативну форму для показу користувачу, або ж для журналювання для аналізу в майбутньому.

Цей метод — незамінний компонент сучасного програмного забезпечення, адже надає можливості для багатозадачності без яких ефективне виконання роботи комп’ютером дуже обмежене.

2.2.3. Метод контролю доступу до файлової системи

Метод контролю доступу до файлової системи представляє собою багаторівневий механізм забезпечення безпеки при роботі з файловою системою мобільного пристрою. Цей метод реалізує принцип глибокого захисту (defense in depth), забезпечуючи послідовну перевірку кожного запиту на різних рівнях системи безпеки.

Цей метод візуалізовано на діаграмі на рисунку 2.6.

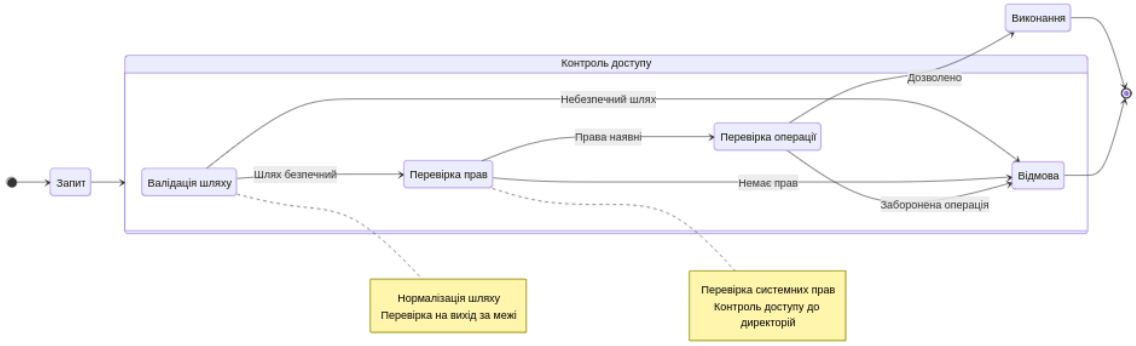


Рис. 2.6. Діаграма етапів доступу до системи

Як видно з рисунка 2.6, процес контролю доступу складається з трьох основних етапів валідації, кожен з яких відповідає за конкретний аспект безпеки. На етапі валідації шляху відбувається нормалізація отриманого шляху до файлу чи директорії та перевірка на спроби виходу за межі дозволеного простору файлової системи. Цей етап є критичним для запобігання атакам типу "path traversal" та захисту системних файлів пристрою.

Другий рівень захисту реалізується через механізм перевірки системних прав доступу. На цьому етапі система верифікує наявність необхідних дозволів для виконання запитуваної операції в контексті поточного користувача. Особлива увага приділяється роботі з системними директоріями та файлами, доступ до яких строго регламентується політиками безпеки операційної системи Android.

Фінальний етап включає аналіз типу операції та її допустимості в поточному контексті. Система оцінює ризики запитуваної операції та приймає рішення про її виконання базуючись на встановлених правилах безпеки. Це дозволяє запобігти потенційно небезпечним операціям, навіть якщо вони формально дозволені на попередніх рівнях перевірки.

У випадку успішного проходження всіх рівнів контролю, система надає доступ до запитуваного ресурсу та виконує необхідну операцію. При цьому здійснюється логування всіх критичних операцій для можливості подальшого аудиту та аналізу безпеки. Якщо на будь-якому етапі виявлено порушення політик безпеки, запит відхиляється з відповідним повідомленням про причину відмови.

Важливою особливістю методу є його адаптивність до різних рівнів привілеїв користувача та типів файлових операцій. Система динамічно адаптує правила контролю доступу залежно від контексту виконання та поточного стану пристрою, що забезпечує оптимальний баланс між безпекою та функціональністю.

Такий комплексний підхід до контролю доступу забезпечує надійний захист файлової системи пристрою при збереженні гнучкості та зручності використання системи віддаленого доступу. Метод враховує специфіку роботи з мобільними пристроями та особливості архітектури операційної системи Android, що робить його ефективним інструментом забезпечення безпеки даних користувача.

2.3 Алгоритми управління файловою системою

2.3.1. Алгоритм валідації файлових шляхів

Алгоритм валідації файлових шляхів є критичним компонентом системи безпеки, що забезпечує контрольований доступ до файлової системи пристрою.

На рисунку 2.7 представлено основні етапи процесу валідації та нормалізації шляхів, що гарантують безпечну роботу з файловою системою.

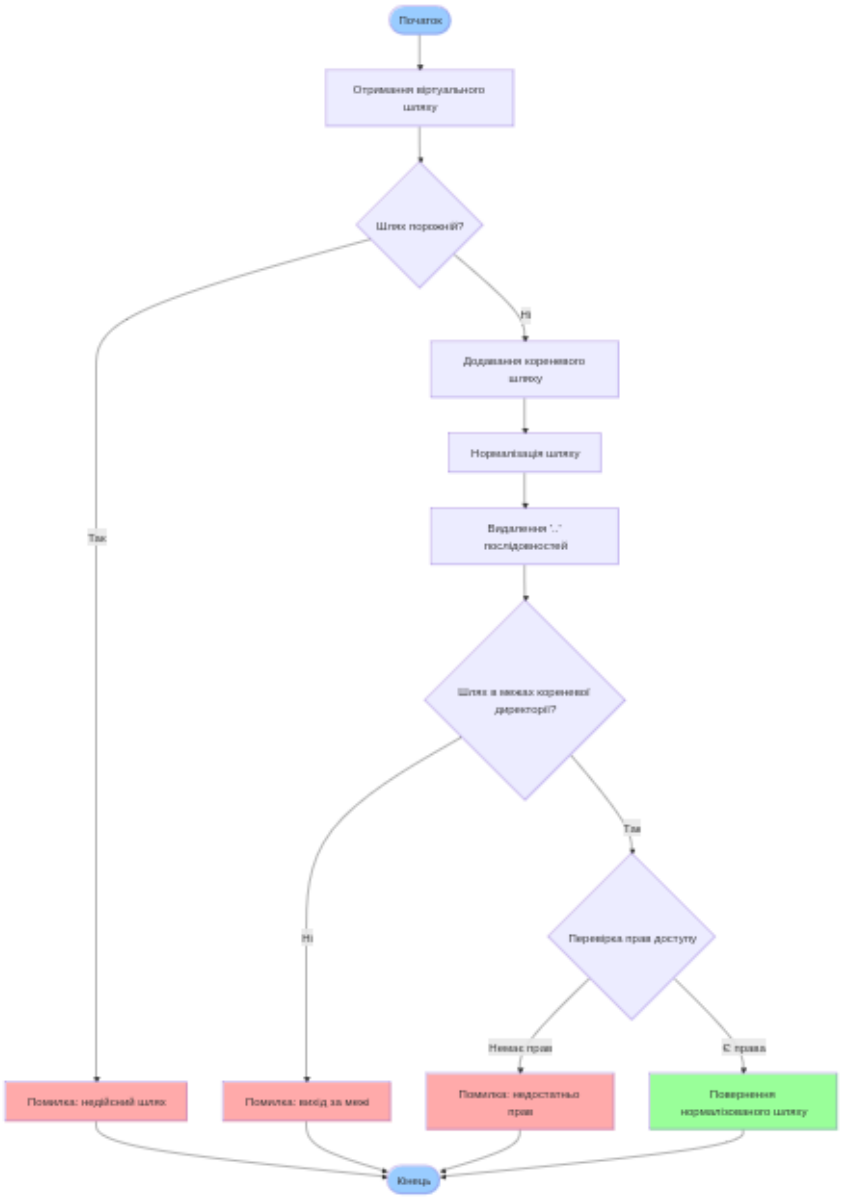


Рис. 2.7. Логічна діаграма процесу валідації шляхів у файловій системі

Процес валідації починається з отримання віртуального шляху від клієнта. Віртуальний шлях представляє собою відносний шлях, який використовується клієнтською частиною системи для навігації по файловій структурі. Перший етап обробки включає перевірку на порожній шлях та базову валідацію формату.

Наступним кроком є додавання кореневого шляху до віртуального. Кореневий шлях визначається конфігурацією системи та обмежує доступний для користувача простір файлової системи. Цей етап є критичним для забезпечення принципу найменших привілеїв, оскільки він фізично обмежує доступ користувача визначеною директорією.

Ключовим елементом алгоритму є процес нормалізації шляху. Під час нормалізації виконується:

- Видалення надлишкових роздільників шляху;
- Обробка спеціальних шляхів навігації;
- Приведення шляху до канонічної форми відповідно до специфіки операційної системи.

Особлива увага приділяється обробці шляхів '..', які можуть бути використані для виходу за межі кореневої директорії. Алгоритм виконує рекурсивну перевірку всіх компонентів шляху, гарантуючи, що результуючий шлях не дозволить доступ до файлів за межами дозволеної області.

Фінальним етапом є валідація прав доступу до отриманого нормалізованого шляху. Система перевіряє наявність необхідних прав читання або запису відповідно до типу операції, що виконується. Це забезпечує додатковий рівень захисту навіть у випадку успішної нормалізації шляху.

У випадку виявлення будь-яких порушень на будь-якому етапі валідації, алгоритм негайно припиняє обробку та повертає відповідну помилку безпеки. Такий консервативний підхід до обробки помилок забезпечує надійний захист від потенційних атак, пов'язаних з маніпуляцією файловими шляхами.

Результатом роботи алгоритму є або нормалізований безпечний шлях, готовий до використання системою, або чітко визначена помилка безпеки, яка може бути належним чином оброблена на рівні програми.

2.3.2. Алогоритм оптимізації передачі файлів

Оптимізація передачі файлів у розробленій системі базується на методі багатопотокового часткового завантаження з попереднім аналізом метаданих. Цей метод дозволяє ефективно працювати як з невеликими, так і з великими файлами, забезпечуючи оптимальне використання мережевих ресурсів та пам'яті пристрою.

Діаграму роботи цього алгоритму ми можемо побачити на рисунку 2.8.

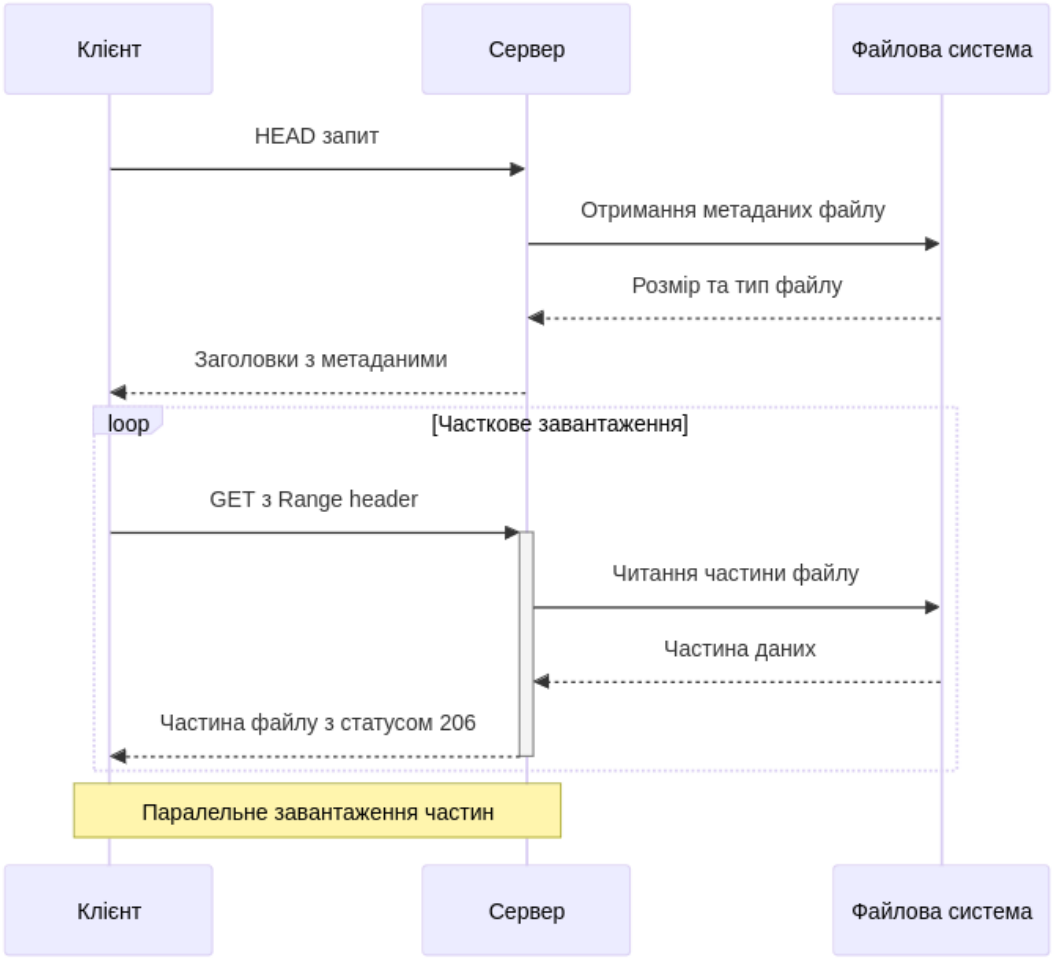


Рис. 2.8. Діаграма послідовності роботи алгоритму передачі даних

Як показано на рисунку 2.8, процес передачі файлу починається з HEAD-запиту, який дозволяє отримати метадані файлу без завантаження його вмісту. Сервер, отримавши такий запит, збирає інформацію про розмір

файлу, тип контенту та підтримку часткового завантаження. Ця інформація передається клієнту через HTTP-заголовки, що дозволяє оптимально спланувати процес завантаження.

Після отримання метаданих, файл розбивається на логічні блоки оптимального розміру. Розмір блоку визначається динамічно, враховуючи загальний розмір файлу та доступні системні ресурси. Для кожного блоку формується окремий GET-запит з вказанням діапазону байтів через заголовок Range. Сервер обробляє такі запити паралельно, відповідаючи статусом 206 Partial Content та відповідною частиною даних.

Для забезпечення цілісності даних, кожен фрагмент файлу верифікується до об'єднання з іншими частинами. У випадку виявлення помилок при передачі, система автоматично повторює завантаження проблемного фрагмента, що гарантує коректність отриманих даних без необхідності повторного завантаження всього файлу.

Використання такого підходу дозволяє досягти оптимального балансу між швидкістю передачі даних та навантаженням на систему, забезпечуючи стабільну роботу навіть при передачі великих файлів через нестабільні мережеві з'єднання.

2.3.3. Алгоритм обробки помилок

Алгоритм обробки помилок та відновлення представляє собою послідовний процес валідації та обробки виключних ситуацій на різних етапах роботи системи.

Першим етапом алгоритму є постійний моніторинг стану мережевого підключення. При виявленні змін у стані мережі (відключення або зміна типу підключення) система автоматично блокує обробку нових запитів до відновлення стабільного з'єднання. Цей етап є критичним для забезпечення коректної роботи всіх компонентів системи.

На другому етапі відбувається валідація сесії для кожного вхідного запиту. Алгоритм перевіряє наявність та валідність сесійного ключа, а також контролює час, що минув з моменту останньої активності. Якщо будь-яка з цих перевірок не проходить, запит відхиляється з відповідним повідомленням про помилку.

Третій етап включає безпосередню обробку операційних помилок, які можуть виникнути під час виконання запитуваної дії. Алгоритм класифікує помилку відповідно до її типу (помилка доступу, відсутність ресурсу, конфлікт) та формує відповідне повідомлення про помилку.

Всі неопрацьовані помилки, що не були перехоплені на попередніх етапах, обробляються загальним обробником помилок, який забезпечує формування уніфікованої відповіді про системну помилку.

Такий підхід забезпечує надійну ідентифікацію та обробку помилкових ситуацій на всіх рівнях роботи системи, гарантуючи коректне інформування клієнта про причини виникнення проблем та можливі шляхи їх вирішення.

2.4 Практична реалізація моделей

2.4.1. Вибір технологічного стеку

У сучасному світі мобільних технологій спостерігається цікава динаміка розвитку операційних систем, де кожна платформа знаходить свою унікальну нішу та аудиторію. Розглянемо основні мобільні операційні системи, які формують цей динамічний ландшафт.

Android, розроблений компанією Google, займає лідируючу позицію на світовому ринку [1] з вражаючою часткою близько 71% користувачів. Ця операційна система відрізняється надзвичайною гнучкістю та відкритістю, що дозволяє виробникам адаптувати її під власні потреби. Користувачі Android отримують доступ до широкого спектру можливостей кастомізації, починаючи від зміни лаунчерів [26] і закінчуючи повною модифікацією

системи. Важливою перевагою є також різноманітність цінових категорій пристроїв - від бюджетних до преміальних моделей, що робить технології доступними для різних верств населення. Система регулярно отримує оновлення безпеки та нові функції, а інтеграція з сервісами Google створює зручну екосистему для користувачів.

iOS від Apple посідає друге місце за популярністю з приблизно 27% ринку. Ця система відома своєю стабільністю та оптимізацією, що забезпечує плавну роботу навіть на старіших пристроях. Apple пропонує закриту екосистему, яка гарантує високий рівень безпеки та конфіденційності, але обмежує можливості кастомізації. Користувачі iOS часто відзначають якість додатків та регулярність оновлень, проте змушені миритися з високою вартістю пристроїв та обмеженим вибором моделей.

HarmonyOS від Huawei, яка з'явилася відносно нещодавно, намагається створити альтернативу Android та iOS. Система демонструє хороші показники продуктивності та оптимізації, але поки що обмежена переважно китайським ринком. Головним викликом для HarmonyOS залишається обмежений доступ до популярних західних сервісів та додатків.

KaiOS займає нішу простих телефонів з базовою функціональністю. Ця система дозволяє запускати полегшені версії популярних додатків на пристроях з обмеженими технічними характеристиками, що робить її популярною на ринках, що розвиваються.

Порівнюючи ці системи, Android виділяється своєю універсальністю та демократичністю. Відкритий характер платформи стимулює інновації та конкуренцію серед виробників, що призводить до постійного вдосконалення користувацького досвіду. Інтеграція з популярними сервісами Google, широкі можливості кастомізації та великий вибір доступних додатків роблять Android привабливим вибором для більшості користувачів.

На відміну від iOS, Android не обмежує користувачів одним виробником та ціновою категорією, дозволяючи знайти оптимальне

співвідношення ціни та якості. Система також надає розробникам більше свободи у створенні додатків, що сприяє появі інноваційних рішень та сервісів.

Важливо відзначити, що Android постійно вдосконалює безпеку та приватність, впроваджуючи нові механізми захисту даних та надаючи користувачам більше контролю над дозволами додатків. Це дозволяє системі конкурувати з iOS у сфері захисту користувацьких даних, зберігаючи при цьому переваги відкритої платформи.

При розробці мобільного застосунку було обрано Android, оскільки вона надає широкі можливості для роботи з файловою системою та мережевими з'єднаннями. Android також має найбільшу частку на ринку мобільних операційних систем [29], що робить застосунок доступним для максимальної кількості користувачів.

Для розробки нативних програмних рішень для платформи Android існує дві популярних мови програмування – Java та Kotlin.

Java, створена компанією Sun Microsystems у 1995 році, продовжує залишатися однією з найпопулярніших мов програмування у світі. Її фундаментальний принцип "Write Once, Run Anywhere" (WORA) забезпечив їй провідну позицію в корпоративній розробці та створенні масштабних систем. Java характеризується надзвичайно розвиненою екосистемою [44], величезною кількістю бібліотек та фреймворків, що пройшли випробування часом. Стабільність, передбачуваність та зворотна сумісність Java роблять її ідеальним вибором для довгострокових проєктів.

Особливої уваги заслуговує потужна система типів Java та її суворий підхід до об'єктно-орієнтованого програмування. Це забезпечує створення надійного та легко підтримуваного коду, що особливо важливо для великих команд розробників. Java має величезну спільноту розробників – за різними оцінками, близько 9 мільйонів активних Java-розробників у світі [1]. Це

означає, що практично будь-яка технічна проблема вже має готове рішення або документовані підходи до її вирішення.

Kotlin, з іншого боку, представляє собою сучасний підхід до програмування на JVM [36]. Розроблена компанією JetBrains у 2011 році, ця мова увібрала в себе найкращі практики сучасного програмування [41], зберігаючи при цьому повну сумісність з Java. Kotlin пропонує більш лаконічний синтаксис, вбудовану підтримку null-safety [22], розумні приведення типів та функціональне програмування [15] "з коробки" [4]. Спільнота Kotlin, хоча й менша за розміром (приблизно 2 мільйони активних розробників), але дуже активна та інноваційна.

У контексті розробки даного Android-застосунку основною мовою було обрано Java, зважаючи на її стабільність, величезну екосистему готових рішень та перевірену часом надійність у розробці Android-застосунків. Java забезпечує надійну основу для реалізації критично важливої бізнес-логіки, роботи з файловою системою та управління даними застосунку [18].

Цей вибір обґрунтований кількома факторами:

- Java має велику екосистему бібліотек та готових рішень для Android-розробки
- Висока стабільність та передбачуваність роботи коду
- Відмінна документація та велика спільнота розробників
- Ефективна робота з потоками та асинхронними операціями
- Зрілі інструменти для відлагодження та профілювання

Частина застосунку, що відповідає за мережеву взаємодію та серверну частину, буде реалізована мовою Kotlin. Це рішення було прийнято через необхідність подальшої інтеграції з фреймворком Ktor, який найкраще працює саме з Kotlin. Kotlin як сучасна мова програмування надає ряд переваг [21, 23]:

- Повна сумісність з Java та Android SDK
- Більш лаконічний та виразний синтаксис

- Вбудована підтримка корутин [11] для асинхронного програмування
- Покращена система типів, що допомагає уникнути null-помилок [33]
- Функціональні можливості мови, які спрощують обробку колекцій та потоків даних.

Таким чином, використання Java та Kotlin у рамках одного проєкту створює потужну синергію: стабільність та надійність Java поєднується з сучасністю та виразністю Kotlin. Це дозволяє максимально ефективно використовувати сильні сторони обох мов, створюючи добре структуровані, надійні та сучасні застосунки. У контексті Android-розробки така комбінація особливо доречна, оскільки обидві мови є офіційно підтримуваними платформою та мають відмінну інтеграцію між собою.

Одним з важливих компонентів майбутньої системи буде Frontend частина, яка працюватиме в браузері користувача. Є дуже велика кількість технологій, які дозволяють її розробляти, зокрема було розглянуто різні мови програмування актуальні на даний момент.

При виборі технології для розробки клієнтської частини веб-застосунку було проведено аналіз найпопулярніших сучасних мов та фреймворків. Основними кандидатами для реалізації фронтенд-частини були JavaScript та TypeScript, які домінують у сфері веб-розробки.

JavaScript, створений компанією Netscape у 1995 році, залишається фундаментальною мовою веб-розробки [9]. Його головною перевагою є універсальність та повсюдна підтримка всіма сучасними браузерами без необхідності додаткових компіляторів чи транспіляторів [19]. JavaScript характеризується надзвичайно розвиненою екосистемою з величезною кількістю бібліотек та фреймворків, найпопулярнішими серед яких є React, Vue та Angular.

Спільнота JavaScript-розробників є найбільшою у світі програмування [34] – за різними оцінками, близько 13.8 мільйонів активних розробників використовують цю мову. Це означає легкий доступ до ресурсів, готових

рішень та швидке вирішення технічних проблем. JavaScript також відрізняється відносно пологою кривою навчання, що дозволяє новим розробникам швидко долучатися до проєкту.

Набирає популярності також мова TypeScript, розроблена Microsoft у 2012 році, вона представляє собою надмножину JavaScript, додаючи статичну типізацію та інші можливості об'єктно-орієнтованого програмування. Її головними перевагами є краща підтримка великих проєктів, зменшення кількості помилок на етапі розробки та покращена підтримка в IDE. Проте TypeScript вимагає додаткового етапу компіляції та має дещо вищий поріг входу для нових розробників.

Інші альтернативи, такі як Elm або ClojureScript, пропонують цікаві підходи до веб-розробки, але мають значно менші спільноти та обмежену екосистему готових рішень. Це може суттєво ускладнити розробку та подальшу підтримку проєкту.

Для реалізації клієнтської частини даного проєкту було обрано JavaScript з кількох ключових причин. По-перше, проєкт не передбачав створення складної архітектури клієнтської частини, де переваги TypeScript були б критично важливими. По-друге, важливим фактором була швидкість розробки та простота внесення змін, що краще забезпечується JavaScript завдяки його динамічній природі та відсутності необхідності в додаткових етапах збірки.

Вибір JavaScript також обумовлений його відмінною підтримкою асинхронних операцій через систему Promise та async/await [16, 5], що критично важливо для взаємодії з серверною частиною застосунку. Наявність великої кількості готових бібліотек для роботи з DOM, обробки подій та управління станом застосунку дозволила значно прискорити процес розробки.

Після вибору мови необхідно обрати також фреймворк для розробки компонентів користувацького інтерфейсу веб-частини нашого програмного

рішення. Розглянемо основні фреймворки та бібліотеки, які домінують на сучасному ринку технологій.

Angular, розроблений та підтримуваний компанією Google, представляє собою повноцінний фреймворк з комплексним підходом до розробки. Він пропонує потужний набір інструментів "з коробки", включаючи маршрутизацію, управління формами, HTTP-клієнт та систему залежностей. Angular використовує TypeScript як основну мову розробки, що забезпечує надійну типізацію та зменшує кількість потенційних помилок. Проте Angular має досить круту криву навчання та може бути надмірним для невеликих проєктів через свою монолітність.

Vue.js позиціонує себе як прогресивний фреймворк, який можна впроваджувати поступово. Він відомий своєю простотою у вивченні та використанні, чіткою документацією та гнучкістю в інтеграції. Vue.js пропонує відмінну продуктивність та легкий процес розробки. Однак, порівняно з React та Angular, Vue.js має меншу екосистему та менше готових компонентів, що може уповільнити розробку складних застосунків.

Svelte представляє собою інноваційний підхід до розробки інтерфейсів, компілюючи компоненти у високооптимізований ванільний JavaScript під час збірки. Це забезпечує відмінну продуктивність та менший розмір фінального бандлу. Проте Svelte є відносно новим гравцем на ринку, має меншу спільноту та обмежену кількість готових рішень.

ReactJS, розроблений Facebook, пропонує компонентний підхід до розробки користувацьких інтерфейсів [7] з акцентом на простоту та гнучкість. Він має найбільшу екосистему серед усіх фронтенд-фреймворків [6], з величезною кількістю готових компонентів, бібліотек та інструментів. React використовує концепцію віртуального DOM [17] для оптимізації оновлень інтерфейсу та пропонує зручний односпрямований потік даних .

Після ретельного аналізу наявних технологій, для розробки веб-інтерфейсу даного проєкту було обрано ReactJS. Цей вибір обумовлений кількома ключовими факторами.

По-перше, проєкт передбачає активну роботу з файлами та динамічне оновлення інтерфейсу в реальному часі, де React's Virtual DOM забезпечує оптимальну продуктивність при частих змінах стану застосунку. По-друге, наявність величезної кількості готових компонентів для роботи з файлами, відображення прогресу завантаження та візуалізації файлової системи значно прискорює процес розробки. Додатково, React пропонує простіший процес інтеграції з нативними можливостями Android через React Native Web [5], що може бути корисним для майбутнього розширення функціоналу.

Таким чином, ReactJS надає оптимальне поєднання продуктивності, гнучкості та зрілості екосистеми [39] для реалізації поставлених завдань, забезпечуючи при цьому можливості для подальшого масштабування та розвитку проєкту.

У сфері розробки серверних компонентів для Android-застосунків існує широкий спектр технологічних рішень. Серед найпопулярніших варіантів можна виділити Spring Boot, який традиційно використовується з Java, легкий вбудований сервер NanoHTTPD, та відносно новий, але потужний фреймворк Ktor від JetBrains.

Spring Boot, будучи частиною екосистеми Spring Framework, пропонує комплексне рішення для створення серверних застосунків [25]. Він надає широкий набір готових компонентів, потужну систему dependency injection, вбудовану підтримку безпеки та зручні інструменти для роботи з базами даних. Однак, для невеликих вбудованих серверів Spring Boot може виявитися надмірним – його повна конфігурація вимагає значних ресурсів, а розмір кінцевого застосунку суттєво збільшується через велику кількість залежностей.

NanoHTTPD представляє собою легкий вбудований веб-сервер для Java, який вирізняється своєю простотою та мінімалістичністю. Він ідеально підходить для базових сценаріїв використання, коли потрібна проста обробка HTTP-запитів. Проте NanoHTTPD має обмежену функціональність, відсутність вбудованої підтримки сучасних веб-протоколів та може вимагати значної кількості ручного кодування для реалізації складніших сценаріїв.

Ktor у поєднанні з Netty виділяється серед інших рішень своєю гнучкістю та сучасним підходом до розробки [24]. Ktor – це легкий та потужний фреймворк, спеціально розроблений для створення асинхронних серверів та клієнтів. Він пропонує модульну архітектуру, де розробник може вибрати лише необхідні компоненти, не обтяжуючи застосунок зайвими залежностями. Netty, як контейнер сервлетів, забезпечує стабільну та перевірену часом основу для роботи веб-застосунків [10, 14].

Після ретельного аналізу доступних варіантів, вибір було зроблено на користь комбінації Ktor + Netty з кількох ключових причин. По-перше, Ktor надає вбудовану підтримку корутин Kotlin, що дозволяє ефективно обробляти асинхронні операції без створення складних callback-структур. Це особливо важливо для мобільних застосунків, де ефективне управління ресурсами є критичним.

По-друге, модульна архітектура Ktor дозволяє мінімізувати розмір кінцевого застосунку [43], включаючи лише необхідні компоненти. Це суттєва перевага порівняно з повноцінними фреймворками як Spring Boot, особливо в контексті мобільної розробки, де розмір застосунку має велике значення.

По-третє, вибір Netty як серверного контейнера забезпечує надійну роботу з мінімальними накладними витратами. Netty має відмінну репутацію в плані продуктивності та стабільності, при цьому залишаючись достатньо легким для вбудовування в мобільний застосунок.

Таким чином, комбінація Ktor та Netty виявилася оптимальним рішенням для реалізації серверного компонента в Android-застосунку, забезпечуючи баланс між функціональністю, продуктивністю та зручністю розробки. Ця комбінація дозволяє ефективно обробляти мережеві запити, зберігаючи при цьому легкість та модульність системи.

При розробці фронтенд частини застосунку критично важливим є вибір UI бібліотеки, яка забезпечить не лише естетичний вигляд, але й функціональність та зручність розробки. Сучасний ринок пропонує широкий спектр UI фреймворків та бібліотек, кожен з яких має свої особливості.

Material-UI (MUI) залишається одним з найпопулярніших рішень для React-застосунків. Ця бібліотека пропонує величезну кількість готових компонентів, які відповідають принципам Material Design від Google. MUI має потужну екосистему, детальну документацію та активну спільноту розробників. Проте, значним недоліком є складність кастомізації компонентів та відносно великий розмір бандлу, що може вплинути на швидкодію застосунку.

Ant Design вирізняється своєю елегантністю та професійним виглядом компонентів. Бібліотека надає широкий набір enterprise-ready компонентів та потужні інструменти для роботи з формами і таблицями. Однак, Ant Design має досить специфічний візуальний стиль, який може бути складно адаптувати під індивідуальні потреби проєкту, а також значний розмір бібліотеки.

Chakra UI набуває все більшої популярності завдяки своїй гнучкості та підходу "accessibility first". Бібліотека пропонує модульну структуру, що дозволяє імпортувати лише необхідні компоненти. Проте, Chakra UI може бути надмірно абстрактною для простих проєктів, а деякі компоненти потребують додаткової конфігурації для досягнення бажаного результату.

Geist UI представляє собою легковісну бібліотеку, що фокусується на мінімалістичному дизайні та високій продуктивності. Особливістю Geist UI є

її орієнтація на сучасні веб-стандарти та оптимізований розмір бандлу. Бібліотека пропонує елегантні компоненти з мінімалістичним дизайном, які легко кастомізуються під потреби проєкту. Хоча спільнота Geist UI менша порівняно з іншими альтернативами, документація бібліотеки достатньо повна та зрозуміла.

Після ретельного аналізу наявних варіантів, вибір було зроблено на користь Geist UI з кількох ключових причин. По-перше, проєкт вимагав легкої та швидкої UI бібліотеки, яка б не обтяжувала застосунок зайвим кодом. Мінімалістичний підхід Geist UI ідеально відповідає цій вимозі, забезпечуючи оптимальний баланс між функціональністю та продуктивністю.

По-друге, специфіка проєкту передбачала створення інтуїтивно зрозумілого інтерфейсу для роботи з файловою системою, де надмірна візуальна складність могла б відволікати користувача. Мінімалістичний дизайн компонентів Geist UI [12] дозволяє створити чистий та функціональний інтерфейс, зосереджений на основних задачах користувача.

Таким чином, вибір Geist UI як основної UI бібліотеки для проєкту дозволив досягти оптимального балансу між функціональністю, продуктивністю та естетикою, забезпечуючи при цьому необхідну гнучкість для подальшого розвитку інтерфейсу застосунку.

Стояв також вибір бібліотеки для зручного завантаження файлів з існуючим відповідним компонентом для графічного інтерфейсу користувача з урахуванням вибраного нами фреймворку ReactJS.

Розглянемо основні бібліотеки для обробки завантаження файлів у веб-застосунках, що були актуальними на момент вибору технологічного стеку проєкту.

Dropzone.js тривалий час залишався стандартом для реалізації функціоналу drag-and-drop завантаження файлів. Бібліотека має велику спільноту, детальну документацію та перевірену часом надійність. Серед її

переваг варто відзначити простоту інтеграції, широкі можливості налаштування зовнішнього вигляду та підтримку адаптивного дизайну. Проте Dropzone.js має обмежені можливості для попередньої обробки файлів та може потребувати додаткового коду для реалізації складних сценаріїв завантаження.

Uppy представляє собою сучасне рішення від команди Transloadit, що пропонує модульну архітектуру та багатий функціонал. Бібліотека підтримує завантаження файлів з різних джерел, включаючи хмарні сховища, має вбудовану можливість попереднього перегляду та редагування зображень. Однак Uppy має досить великий розмір базового пакету та може здаватися надмірною для проєктів, де не потрібен весь її функціонал.

Fine-uploader пропонує комплексне рішення з підтримкою chunked uploads, відновлення перерваних завантажень та розширеними можливостями валідації. Бібліотека добре документована та має стабільну продуктивність. Проте Fine-uploader є комерційним продуктом з платною ліцензією для використання в проєктах, що може бути обмежуючим фактором.

Для реалізації функціоналу завантаження файлів у даному проєкті було обрано саме FilePond. Важливим фактором стала модульна архітектура FilePond, яка дозволяє легко розширювати функціональність через систему плагінів [32], додаючи лише ті можливості, які дійсно потрібні проєкту. Наприклад, вбудована підтримка попередньої обробки зображень та можливість відновлення перерваних завантажень значно спрощують процес розробки та покращують користувацький досвід.

У процесі розробки також активно використовувались інструменти, які де-факто є стандартними для обраних платформ, тому не потребували окремого детального аналізу для вибору:

- Android Studio як основне середовище розробки Android та бекенд компонентів системи

- Code OSS (опенсорс версія популярного VS Code) як основне середовище розробки фронтенд компоненту системи
- Gradle для автоматизації збірки проекту
- Git для контролю версій
- Postman для тестування API
- Chrome DevTools для відлагодження веб-інтерфейсу

Всі обрані технології та інструменти формують цілісний та ефективний стек для розробки програмного рішення, забезпечуючи необхідну функціональність, продуктивність та зручність розробки.

2.4.2. Загальна архітектура системи

При проектуванні системи віддаленого доступу до файлової системи мобільного пристрою основним викликом стала необхідність створення надійного та ефективного рішення, яке б працювало виключно в межах локальної мережі та не вимагало зовнішніх серверів чи хмарних сервісів. Для вирішення цієї задачі було розроблено архітектуру, де мобільний пристрій виконує подвійну роль - як клієнтського додатку для налаштування системи, так і веб-сервера для обслуговування запитів з інших пристроїв.

Система складатиметься з трьох основних модулів:

1. Android-додаток (app module) - відповідатиме за:
    - взаємодію з користувачем
    - відображення стану системи
    - генерацію QR-кодів для швидкого підключення
    - моніторинг дискового простору
    - управління життєвим циклом серверного компонента
2. Сервісний модуль (service module) - реалізує:
    - обробку HTTP-запитів
    - маршрутизацію API
    - операції з файловою системою

- управління сесіями
- механізми безпеки

3. Веб-інтерфейс (web-app module) - забезпечуватиме:

- інтерактивний інтерфейс на базі React та Geist UI
- функціонал завантаження файлів через FilePond
- відображення структури файлової системи
- операції з файлами та директоріями

Особливістю архітектури є використання Android Service для забезпечення безперервної роботи веб-сервера навіть при згорнутому додатку. Це реалізовано через Foreground Service, який підтримує постійне з'єднання та відображає сповіщення про активний стан сервера.

Взаємодія між клієнтською частиною (веб-браузер на іншому пристрої) та сервером буде організована через REST API, що забезпечує уніфікований інтерфейс для всіх операцій з файловою системою. Для обробки запитів використовуватиметься фреймворк Ktor, який забезпечить асинхронну обробку з'єднань та ефективне управління ресурсами.

Система безпеки буде реалізована на кількох рівнях:

- Генерація унікальних URL-адрес для кожної сесії
- Обмеження доступу виключно з пристроїв у локальній мережі
- Валідація всіх вхідних запитів
- Контроль доступу до системних директорій
- Автоматичне завершення неактивних сесій

Для ефективної передачі файлів буде реалізовано механізм часткової передачі даних (partial content), що дозволяє:

- Відновлювати перервані завантаження
- Оптимізувати використання пам'яті
- Забезпечувати стабільну передачу великих файлів

Взаємодія між модулями системи буде організована через чітко визначені інтерфейси, що забезпечує:

- Слабке зв'язування компонентів
- Можливість незалежного тестування кожного модуля
- Гнучкість при внесенні змін
- Можливість розширення функціоналу

Система логування та моніторингу, буде реалізована за допомогою SLF4J та Logback, дозволяє відстежувати:

- Критичні операції з файлами
- Мережеві з'єднання
- Дії користувача
- Помилки та виключні ситуації

Розроблена архітектура забезпечує оптимальний баланс між продуктивністю, безпекою та зручністю використання, при цьому залишаючись достатньо гнучкою для подальшого розширення функціоналу. Модульна структура та чітке розділення відповідальності між компонентами спрощують підтримку та розвиток системи [4].

2.4.3. Проектування компонентів

При проектуванні серверного компоненту системи основна увага приділялася створенню надійної, безпечної та ефективної архітектури, яка б забезпечувала всі необхідні функціональні можливості, визначені в вимогах до системи. Серверний компонент є ключовою частиною системи, оскільки він відповідає за обробку запитів від веб-клієнта, управління файловою системою пристрою та забезпечення безпеки доступу.

При проектуванні серверного компоненту системи було обрано модульну архітектуру [38] з чітким розділенням відповідальності між компонентами. На рисунку 2.1 представлено загальну архітектуру серверної частини системи на базі фреймворку Ktor, який забезпечує асинхронну обробку запитів та високу продуктивність.

Архітектуру серверного компонента зображено на рисунку 2.9.

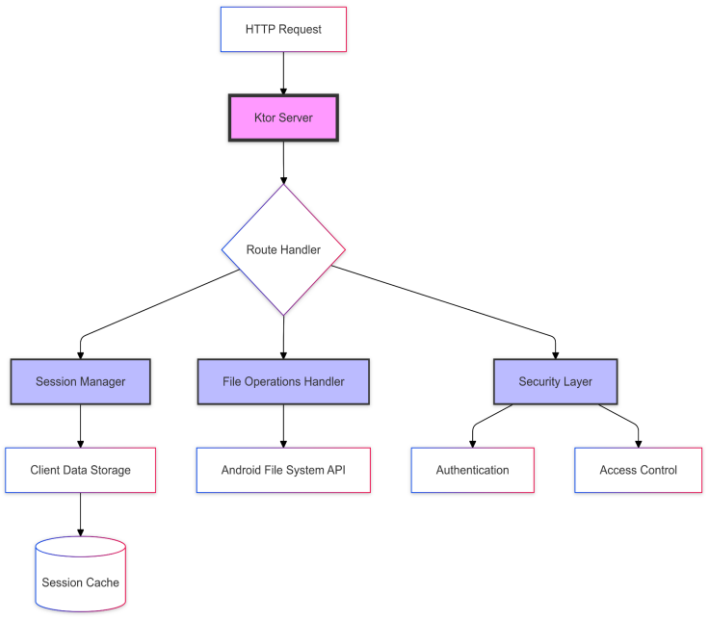


Рисунок 2.9. Діаграма архітектури сервера

Як видно з рисунку 2.9, архітектура включає наступні основні компоненти:

1. Ktor Server Layer:
    - Обробка HTTP-запитів;
    - Маршрутизація через Route Handler;
    - Конфігурація CORS та захисту;
    - Управління життєвим циклом сервера.
2. Session Management:
    - Генерація унікальних сесійних ключів через Session Manager;
    - Валідація сесій з використанням Client Data Storage;
    - Очищення застарілих сесій з Session Cache;
    - Зберігання стану клієнта.
3. Security Layer:
    - Автентифікація запитів через Authentication модуль;
    - Контроль доступу через Access Control;
    - Валідація параметрів;

- Захист від атак.

4. File Operations Handler:
    - Взаємодія з Android File System API;
    - Валідація операцій;
    - Оптимізація доступу;
    - Обробка помилок.

Клієнтська частина системи складається з двох основних компонентів: Android-додатку, який виступає в ролі сервера та надає доступ до файлової системи пристрою, та веб-інтерфейсу, що забезпечує взаємодію з файловою системою через браузер. При проектуванні особлива увага приділялась забезпеченню надійної взаємодії між компонентами та створенню інтуїтивно зрозумілого інтерфейсу користувача.

На рисунку 2.10 представлено загальну архітектуру Android-компоненту системи.

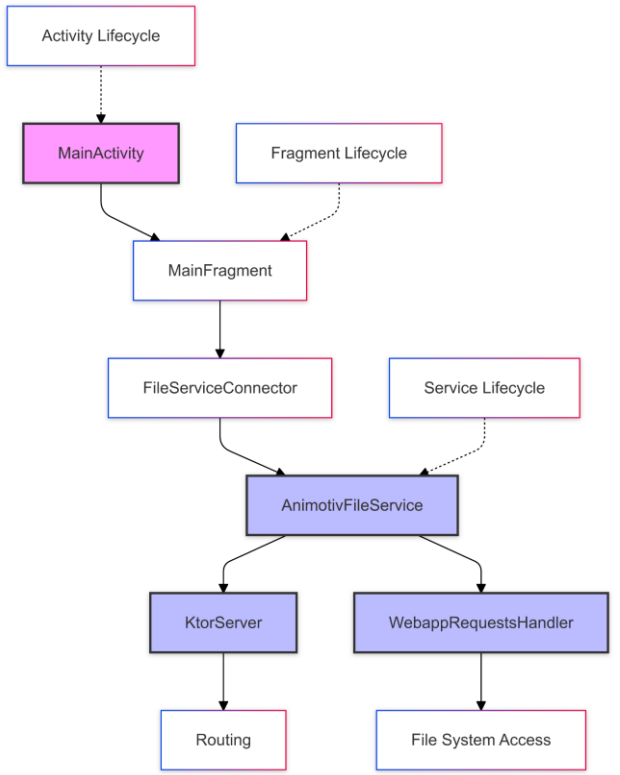


Рис. 2.10. Діаграма архітектури Android-модулю

Основні компоненти Android-додатку, як показано на рисунку 2.10, включають:

1. Компоненти користувацького інтерфейсу:
    - MainActivity для управління основним UI;
    - MainFragment для відображення статусу сервера;
    - Індикатори стану з'єднання та використання пам'яті;
    - Генератор та відображення QR-кодів.
2. Сервісні компоненти:
    - FileService для роботи у фоновому режимі;
    - FileServiceConnector для комунікації між UI та сервісом;
    - KtorServer для обробки HTTP-запитів;
    - WebappRequestsHandler для операцій з файлами.
3. Допоміжні компоненти:
    - Обробники життєвого циклу компонентів;
    - Генератори сесійних ключів;
    - Системи логування та обробки помилок.

Веб-інтерфейс спроектовано як односторінковий застосунок (SPA) на базі React [5, 6, 35] з використанням компонентного підходу. На рисунку 2.11 представлено структуру компонентів веб-інтерфейсу.

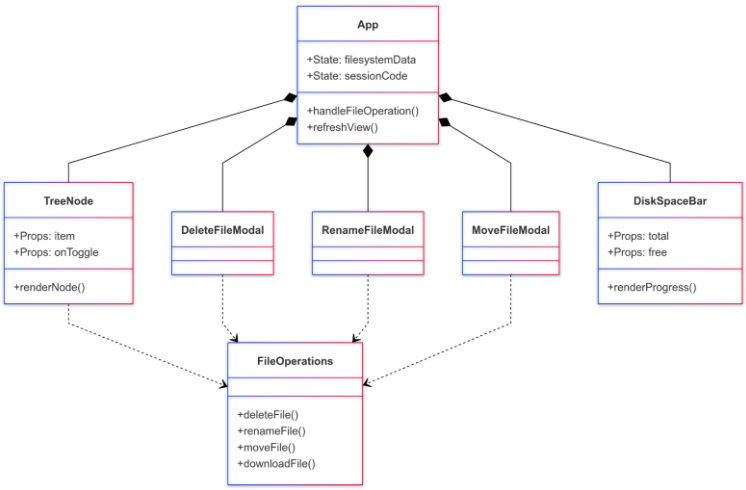


Рис. 2.11. Діаграма архітектури веб-застосунку

Ключові аспекти проектування веб-інтерфейсу включають:

1. Компонентна структура:
    - Деревоподібне представлення файлової системи (TreeNode);
    - Модальні вікна для операцій з файлами;
    - Індикатори стану та прогресу;
    - Компоненти для відображення метаданих.
2. Управління станом:
    - Централізоване зберігання стану додатку;
    - Кешування даних для оптимізації;
    - Асинхронна обробка операцій;
    - Обробка помилок та повторні спроби.
3. Користувацький досвід:
    - Інтуїтивна навігація по файловій системі;
    - Візуальний фідбек для всіх операцій;
    - Адаптивний дизайн;
    - Оптимізація для різних пристроїв.

Висновки до розділу

У другому розділі було розроблено та детально описано формальні моделі, методи та алгоритми для організації віддаленого доступу до файлової системи мобільного пристрою. В ході роботи вдалося створити цілісний комплекс теоретичних та практичних рішень, що забезпечують надійність, безпеку та ефективність системи.

Ключовим досягненням стала розробка трьох взаємопов'язаних формальних моделей: моделі файлової системи, що описує структуру та операції з файлами; моделі безпеки, що визначає механізми захисту та контролю доступу; та моделі мережевої взаємодії, що формалізує

комунікацію між компонентами системи. Особливо вдалим виявилося використання патерну "Фасад" для моделювання файлової системи, що дозволило створити чистий та уніфікований інтерфейс для всіх операцій.

Розроблені методи організації віддаленого доступу, включаючи метод генерації та валідації сесійних ключів, метод асинхронної передачі даних та метод контролю доступу, забезпечують надійний фундамент для безпечної роботи системи. Реалізовані алгоритми управління файловою системою гарантують коректну валідацію шляхів, оптимальну передачу файлів та ефективну обробку помилок.

Таким чином, розроблені теоретичні основи та створюють надійний фундамент для подальшої розробки системи. Модульність та формалізованість запропонованих рішень дозволяють легко розширювати та адаптувати систему під нові вимоги в майбутньому.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1. Реалізація основних компонентів

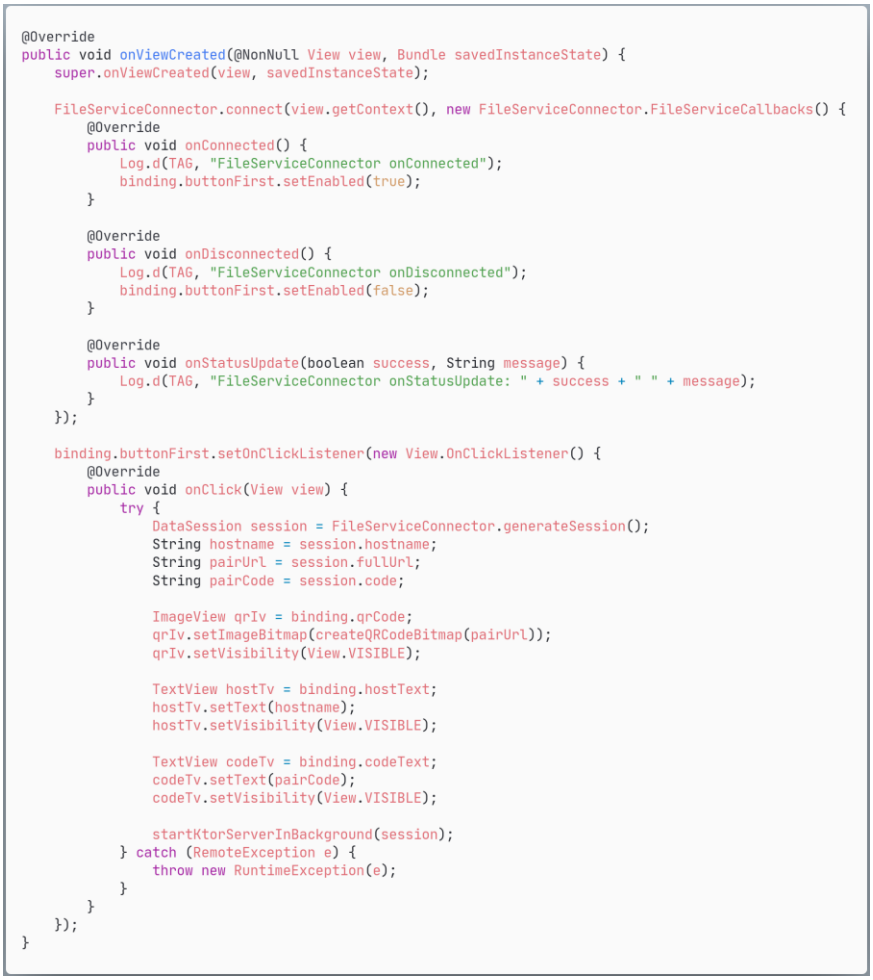
Реалізація системи проводилась з дотриманням принципів об'єктно-орієнтованого програмування [3], використанням сучасних практик розробки та з урахуванням всіх визначених вимог. Система складається з трьох основних компонентів: Android-додатку, вбудованого веб-серверу та веб-інтерфейсу файлового менеджера. Кожен з цих компонентів було реалізовано з використанням відповідних технологій та інструментів, що найкраще відповідають поставленим задачам.

3.1.1. Розробка Android-додатку

Android-додаток є центральним компонентом системи, який забезпечує взаємодію з файловою системою пристрою та надає базовий користувацький інтерфейс. Він виступає основним елементом, через який відбувається керування файлами та взаємодія з користувачем.

Додаток має можливості для повноцінної взаємодії з різними компонентами мобільного пристрою через вбудовані API Android. Це дозволяє отримувати доступ до системних сервісів та функцій пристрою для забезпечення необхідного функціоналу. Важливою функцією є постійний моніторинг стану підключення до мережі інтернет, що дозволяє оперативно адаптувати роботу додатку відповідно до наявності чи відсутності з'єднання. Також система пропонує готовий набір базових графічних компонентів для створення застосунків, що дозволяє швидко почати отримувати видимі результати.

Серцем застосунку є клас MainFragment головний метод onViewCreated якого зображений на рисунку 3.1.



```
@Override
public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    FileServiceConnector.connect(view.getContext(), new FileServiceConnector.FileServiceCallbacks() {
        @Override
        public void onConnected() {
            Log.d(TAG, "FileServiceConnector onConnected");
            binding.buttonFirst.setEnabled(true);
        }

        @Override
        public void onDisconnected() {
            Log.d(TAG, "FileServiceConnector onDisconnected");
            binding.buttonFirst.setEnabled(false);
        }

        @Override
        public void onStatusUpdate(boolean success, String message) {
            Log.d(TAG, "FileServiceConnector onStatusUpdate: " + success + " " + message);
        }
    });

    binding.buttonFirst.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            try {
                DataSession session = FileServiceConnector.generateSession();
                String hostname = session.hostname;
                String pairUrl = session.fullUrl;
                String pairCode = session.code;

                ImageView qrIv = binding.qrCode;
                qrIv.setImageBitmap(createQRCodeBitmap(pairUrl));
                qrIv.setVisibility(View.VISIBLE);

                TextView hostTv = binding.hostText;
                hostTv.setText(hostname);
                hostTv.setVisibility(View.VISIBLE);

                TextView codeTv = binding.codeText;
                codeTv.setText(pairCode);
                codeTv.setVisibility(View.VISIBLE);

                startKtorServerInBackground(session);
            } catch (RemoteException e) {
                throw new RuntimeException(e);
            }
        }
    });
}
```

Рис. 3.1. Метод onViewCreated класу MainFragment

Метод onViewCreated є ключовим компонентом життєвого циклу фрагмента в Android [28] і викликається відразу після створення відповідного представлення. В нашому випадку цей метод виконує декілька важливих функцій для налаштування користувацького інтерфейсу та встановлення з'єднання з сервісом.

На початку методу відбувається підключення до FileService через FileServiceConnector. Для цього викликається статичний метод [13] connect, якому передається контекст представлення та об'єкт зворотного виклику FileServiceCallbacks. Цей колбек містить три методи:

1. onConnected - викликається при успішному підключенні до сервісу. В цьому методі активується кнопка buttonFirst шляхом виклику setEnabled(true);

2. onDisconnected - викликається при втраті з'єднання з сервісом. В цьому випадку кнопка buttonFirst деактивується через setEnabled(false);
3. onStatusUpdate - забезпечує отримання оновлень про стан сервісу, приймаючи булеве значення успішності операції та текстове повідомлення.

Далі в методі налаштовується обробник натискання на кнопку buttonFirst. При натисканні виконується наступна послідовність дій:

1. Генерується нова сесія через FileServiceConnector.generateSession();
2. З отриманого об'єкта DataSession витягуються необхідні дані: hostname (адреса сервера), pairUrl (повне посилання для підключення) та pairCode (код сесії);
3. На основі pairUrl генерується QR-код за допомогою методу createQRCodeBitmap;
4. Оновлюється інтерфейс користувача:
    - QR-код відображається в ImageView;
    - Адреса сервера показується в першому TextView;
    - Код сесії показується в другому TextView.
5. Запускається Ktor сервер у фоновому режимі з параметрами створеної сесії.

Клас MainFragment також містить метод для створення QR-кодів зображений на рисунку 3.2.

```
private static final int QR_WIDTH_HEIGHT = 125;

private static Bitmap createQRCodeBitmap(String qrCodeText) throws IOException, WriterException {
    String charset = "UTF-8";
    BitMatrix matrix = new MultiFormatWriter().encode(
            new String(qrCodeText.getBytes(charset), charset),
            BarcodeFormat.QR_CODE, QR_WIDTH_HEIGHT, QR_WIDTH_HEIGHT
    );

    Bitmap bmp = Bitmap.createBitmap(QR_WIDTH_HEIGHT, QR_WIDTH_HEIGHT, Bitmap.Config.RGB_565);
    for (int x = 0; x < QR_WIDTH_HEIGHT; x++) {
        for (int y = 0; y < QR_WIDTH_HEIGHT; y++) {
            bmp.setPixel(x, y, matrix.get(x, y) ? Color.BLACK : Color.WHITE);
        }
    }
    return bmp;
}
```

Рис. 3.2. Метод createQRCodeBitmap

Метод createQRCodeBitmap приймає текстовий параметр qrCodeText, який містить URL-адресу для підключення до сервісу, та повертає об'єкт типу Bitmap, що представляє згенерований QR-код. Процес генерації QR-коду складається з декількох етапів.

Спочатку визначається кодування символів як UTF-8, яке забезпечує коректне представлення Unicode-символів у вихідному QR-коді. Далі, використовуючи клас MultiFormatWriter з бібліотеки ZXing, створюється матриця точок QR-коду. При створенні матриці вхідний текст конвертується в байти з використанням UTF-8 кодування, що гарантує правильне представлення всіх символів.

Розмір QR-коду визначається константою QR_WIDTH_HEIGHT, яка задає однакові значення для ширини та висоти, створюючи квадратний QR-код. Для створення візуального представлення використовується формат BarcodeFormat.QR_CODE.

Після отримання бітової матриці створюється новий об'єкт Bitmap з форматом RGB_565, який забезпечує оптимальний баланс між якістю зображення та використанням пам'яті. Формат RGB_565 використовує 16 біт на піксель, що достатньо для чорно-білого QR-коду.

Далі відбувається поетапне заповнення створеного растрового зображення. За допомогою вкладених циклів проходимо по кожному пікселю зображення, перевіряючи відповідне значення в бітовій матриці. Якщо значення в матриці true, встановлюємо чорний колір пікселя, інакше - білий. Це створює характерний візерунок QR-коду, який можна зчитати за допомогою камери смартфона чи іншого пристрою.

В результаті отримуємо готове растрове зображення QR-коду, яке встановлюється для відображення в графічному інтерфейсі у випадку натискання користувачем кнопки firstButton, логіку якої було описано до цього.

Для взаємодії з сервісним компонентом, який використовується в якості моста до серверної частини, реалізовано спеціальний клас FileServiceConnector, який забезпечує:

- Підключення до сервісу через механізм прив'язки (binding);
- Обробку подій підключення та відключення;
- Передачу команд від UI до сервісу;
- Отримання зворотного зв'язку про стан операцій;

На рисунку 3.3 можна побачити реалізацію цього класу.

```
public class FileServiceConnector {
    private static final String TAG = "FileServiceConnector";
    private static IFileService fileService;
    private static FileServiceCallbacks clientCallback;

    private FileServiceConnector() { }

    public interface FileServiceCallbacks {
        void onConnected();
        void onDisconnected();
        void onStatusUpdate(boolean success, String message);
    }
}
```

Рис. 3.3 Частина класу FileServiceConnector

Клас FileServiceConnector використовує статичні поля та методи для управління з'єднанням з сервісом, а приватний конструктор застосовується для запобігання створення екземплярів класу, оскільки вся функціональність реалізована через статичні члени. Важливим елементом є інтерфейс FileServiceCallbacks, який визначає методи зворотного виклику для сповіщення клієнтської частини про зміни стану з'єднання.

Для забезпечення взаємодії з сервісом реалізовано об'єкт ServiceConnection зображений на рисунку 3.4.

```
private static ServiceConnection serviceConnection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName className, IBinder service) {
        Log.d(TAG, "onServiceConnected");
        fileService = IFileService.Stub.asInterface(service);

        try {
            fileService.registerCallback(serviceCallback);
        } catch (RemoteException e) {
            throw new RuntimeException(e);
        }

        clientCallback.onConnected();
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        Log.d(TAG, "onServiceDisconnected");
        fileService = null;
        clientCallback.onDisconnected();
    }
};
```

Рис. 3.4. Реалізація екземпляру класу ServiceConnection

ServiceConnection відповідає за управління життєвим циклом зв'єднання з сервісом. При успішному підключенні в методі onServiceConnected відбувається перетворення отриманого IBinder об'єкта в інтерфейс IFileService за допомогою методу Stub.asInterface. Після цього виконується реєстрація колбеку для отримання оновлень від сервісу та сповіщення клієнтського коду про успішне підключення.

Для встановлення з’єднання з сервісом клас FileServiceConnector містить метод connect зображений на рисунку 3.5.

```
public static void connect(Context context, FileServiceCallbacks callbacks) {
    clientCallback = callbacks;
    Intent serviceIntent = new Intent("com.fileservice.app.CONNECT_FILE_SERVICE");
    serviceIntent.setPackage("com.fileservice.app");
    context.bindService(serviceIntent, serviceConnection, Context.BIND_AUTO_CREATE);
}
```

Рис. 3.5. Метод connect

Метод connect приймає контекст Android-додатку та об'єкт колбеків. Він створює Intent з відповідним action та package для ідентифікації сервісу в системі. Індикатор BIND_AUTO_CREATE вказує системі автоматично створювати сервіс при потребі підключення.

3.1.2. Імплементація веб-серверу

Значну частину додатку складає сервісний компонент FileService фрагмент якого зображено на рисунку 3.6. Цей клас відповідає за:

- Моніторинг мережевого підключення
- Генерацію сесій для клієнтів
- Управління життєвим циклом вбудованого серверу
- Обробку запитів до файлової системи.

```
private static final String TAG = "FileService";

private String activeIpAddress = null;

private DataSession activeSession = null;

private Context context;

public FileService(Context context) {
    super();

    Log.d(TAG, "FileService service initializing...");
    this.context = context;

    ConnectivityManager cm = (ConnectivityManager) context.getSystemService(Context.CONNECTIVITY_SERVICE);
    cm.registerDefaultNetworkCallback(new ConnectivityManager.NetworkCallback() {

        @Override
        public void onAvailable(Network network) {
            super.onAvailable(network);
            Log.d(TAG, "Network available. Setting inet services to ready");

            NetworkCapabilities nc = cm.getNetworkCapabilities(network);
            if (nc != null && nc.hasTransport(NetworkCapabilities.TRANSPORT_WIFI)) {
                WifiManager wifiManager = (WifiManager) context.getSystemService(Context.WIFI_SERVICE);
                activeIpAddress = Formatter.formatIpAddress(wifiManager.getConnectionInfo().getIpAddress());
            } else {
                activeIpAddress = null;
            }
        }

        @Override
        public void onLost(Network network) {
            super.onLost(network);
            Log.d(TAG, "Network no longer available. Setting inet services to not ready");
            activeIpAddress = null;
        }
    });
}
```

Рис. 3.6. Частина класу FileService

Як можна побачити на рисунку 3.6 при створенні об’єкту FileService виконується підписка на оновлення стану мережі за допомогою колбеку,

який передається в метод registerDefaultNetworkCallback. В цьому колбеці виконується перевірка на факт наявності активного підключення до локальної мережі Wi-Fi і оновлення значення змінної класу activeIpAddress.

Змінна activeIpAddress використовується для зберігання поточної IP-адреси мобільного пристрою. Ця адреса потрібна для створення активної сесії і посилання на неї для доступу інших пристроїв через веб-інтерфейс системи.

У випадку, якщо мережеве з’єднання зникне, операційна система викличе метод колбеку onLost, де значення змінної activeIpAddress буде скинуто до null.

На рисунку 3.7 зображено створення нової сесії для підключення зовнішніх пристроїв.

```
@Override
public DataSession generateSession() throws RemoteException {
    if (activeIpAddress == null) {
        return DataSession.createError(DataSession.ERROR_NO_INTERNET);
    }
    byte[] codeBytes = new byte[Constants.SESSION_CODE_LENGTH];
    new SecureRandom().nextBytes(codeBytes);
    String code = Base64.encodeToString(codeBytes, Base64.URL_SAFE | Base64.NO_PADDING | Base64.NO_WRAP);
    String hostname = "http://" + activeIpAddress + ":" + Constants.PORT;
    String fullUrl = hostname + "?session=" + code;
    activeSession = new DataSession(fullUrl, hostname, code);

    startKtorServerInBackground(activeSession);

    return activeSession;
}
```

Рис. 3.7. Метод generateSession

Метод generateSession використовує об’єкт SecureRandom для створення захищеного ключа доступу до сесії. Для збільшення універсальності такого ключа він початково генерується з байтів за допомогою методу nextBytes об’єкта SecureRandom, а пізніше за допомогою статичного методу encodeToString класу Base64 з переданими опційними

аргументами перетворюється в тип String. Слід зазначити що методу encodeToString передаються окрім байтів secureBytes отриманих з SecureRandom також бітові значення Base64.URL_SAFE, Base64.NO_PADDING та Base64.NO_WRAP щоб вихідне значення було придатним для застосування в якості аргументу посилання за протоколом HTTP.

На рисунку 3.8 зображено реалізацію методу startKtorServerInBackground, який відповідає за асинхронний запуск серверу Ktor у фоновому режимі.

```
private void startKtorServerInBackground(DataSession session) {

    ExecutorService executorService = Executors.newSingleThreadExecutor();
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            try {
                Log.d(TAG, "Starting Ktor Server...");
                new KtorServer().restartServer(new ClientData(session.code, Instant.now()), context);
                Log.d(TAG, "Ktor Server started successfully");
            } catch (Exception e) {
                Log.e(TAG, "Error starting Ktor Server: " + e.getMessage());
            }
        }
    });
}
```

Рис. 3.8. Метод startKtorServerInBackground

Цей метод приймає об'єкт типу DataSession, який містить інформацію про активну сесію, включаючи унікальний код доступу. Для забезпечення коректної роботи додатку без блокування головного потоку виконання, запуск серверу відбувається в окремому потоці. Це реалізовано за допомогою ExecutorService, який створюється методом newSingleThreadExecutor(). Використання однопотокового виконавця гарантує, що одночасно може

працювати лише один екземпляр серверу [4], що запобігає можливим конфліктам та помилкам при одночасному запуску декількох серверів.

В метод execute передається анонімний клас, що реалізує інтерфейс Runnable [8]. В його методі run виконується безпосередній запуск серверу. Створюється новий екземпляр класу KtorServer та викликається його метод restartServer.

Весь процес запуску серверу обгорнуто в блок try-catch для обробки можливих помилок. У випадку успішного запуску в лог записується відповідне повідомлення. Якщо ж виникає помилка, вона логується з детальним описом через Log.e, що дозволяє розробникам ідентифікувати та виправляти проблеми при розробці та тестуванні.

Серцем серверної частини є Kotlin-файл [14] KtorServer.kt, класи даних та оголошення головного класу цього файлу – KtorServer продемонстровано на рисунку 3.9

```
data class PathRequest(val path: String)
data class RenameRequest(val path: String, val newName: String)
data class MoveRequest(val originPath: String, val destinationPath: String)

class KtorServer {
    private lateinit var handler: WebappRequestsHandler
    private var clientData: ClientData? = null

    companion object {
        private var server: NettyApplicationEngine? = null
    }
```

Рис. 3.9. Класи даних та оголошення класу KtorServer

Клас використовує data-класи [2] PathRequest, RenameRequest та MoveRequest для структурування вхідних даних від клієнта. Ці класи забезпечують типобезпечну обробку JSON-запитів. У самому класі визначено

два основних поля: handler типу WebappRequestsHandler для обробки файлових операцій та clientData типу ClientData для зберігання інформації про поточну сесію. У companion object зберігається статична змінна server, що представляє екземпляр веб-сервера.

Клас KtorServer містить також метод restartServer, показаний на рисунку 3.10.

```
fun restartServer(clientData: ClientData, context: Context) {
    this.clientData = clientData
    handler = WebappRequestsHandler(context)
    applicationEngineEnvironment {
        log = AndroidLogger()
        server?.stop(1000, 1000)
        server = createServer()
        server?.start(wait = true)
    }
}
```

Рис. 3.10. Метод restartServer

Метод restartServer відповідає за перезапуск веб-сервера. Метод приймає об'єкт clientData з інформацією про сесію та context для створення обробника файлових операцій. При виклику методу спочатку зупиняється поточний сервер (якщо він існує) з таймаутом у 1000 мілісекунд, потім створюється новий екземпляр сервера через метод createServer та запускається з параметром wait=true, що означає блокуючий режим очікування завершення запуску. Тобто сервер запущений тут працюватиме у власному потоці і без явного повідомлення буде очікувати на запити невизначену кількість часу. В такому режимі клієнтська частина в будь-який момент зможе здійснити потрібний запит і отримати на нього відповідь.

На рисунку 3.11 зображено частину методу createServer, які відповідає за конфігурацію веб-сервера Netty.

```
private fun createServer(): NettyApplicationEngine =
    embeddedServer(Netty, host = "0.0.0.0", port = Constants.PORT) {
        install(ContentNegotiation) {
            gson()
        }
        install(AutoHeadResponse)
        install(PartialContent)
        install(CallLogging) {
            logger = AndroidLogger()
            level = Level.DEBUG
        }
        install(StatusPages) {
            exception<Throwable> { call, cause →
                call.respond(
                    HttpStatusCode.InternalServerError,
                    "An internal error occurred: ${cause.message}"
                )
            }
        }
```

Рис. 3.11. Код методу createServer відповідальний за конфігурацію сервера

Ця конфігурація можлива завдяки високому ступеню модульності бібліотеки Ktor. Використовується метод ініціалізації embeddedServer, в який передаються фабрика для створення сервера Netty, адреса хоста, та порт. Показана конфігурація включає встановлення таких додаткових плагінів:

- ContentNegotiation з використанням gson для серіалізації/десеріалізації JSON
- AutoHeadResponse для автоматичної обробки HEAD-запитів
- PartialContent для підтримки часткового завантаження файлів
- CallLogging для налагодження та моніторингу запитів
- StatusPages для централізованої обробки помилок

Далі метод createServer містить конфігурацію ресурсів. Його реалізацію можна побачити на рисунку 3.12.

```
routing {
    staticResources("/assets", "/assets/web/assets")
    staticResources("/", "/web")

    singlePageApplication {
        useResources = true
        filesPath = "/assets/web"
        defaultPage = "index.html"
    }
```

Рис. 3.12. Конфігурація статичних ресурсів веб-застосунку

Тут налаштовується обслуговування статичних ресурсів веб-додатку. Директива staticResources вказує серверу звідки брати статичні файли для конкретних URL-шляхів. Конфігурація singlePageApplication налаштовує обробку запитів для Single Page Application (SPA), що дозволяє правильно обробляти навігацію на стороні клієнта.

На рисунку 3.13 зображено початок імплементації маршрутизації.

```
route("/api") {
    post("/browse") {
        failIfUnauthorized(call, clientData)

        val request = call.receive<PathRequest>()
        val path = request.path

        try {
            val filesData = handler.fetchPath(path)
            call.respond(HttpStatusCode.OK, filesData)
        } catch (e: SecurityException) {
            call.respond(HttpStatusCode.Forbidden, "Forbidden route")
        } catch (e1: FileNotFoundException) {
            call.respond(HttpStatusCode.BadRequest, "Bad request")
        }
    }
```

Рис. 3.13. Реалізація маршрутизації шляху /api/browse

Перед обробкою запиту відбувається перевірка авторизації через метод failIfUnauthorized. Запит містить шлях до директорії, вміст якої потрібно отримати. Обробник fetchPath повертає список файлів та папок за вказаним шляхом. У разі виникнення помилок безпеки або відсутності файлів повертаються відповідні HTTP-коди помилок.

На рисунку 3.14 зображено обробку шляху /api/upload, яка відповідає за можливість завантажувати файли на пристрій, на якому виконується сервер.

```
post("/upload") {
    failIfUnauthorized(call, clientData)

    try {
        call.receiveMultipart().forEachPart { part →
            when (part) {
                is PartData.FileItem → {
                    part.streamProvider().use { partStream →
                        handler.writeToStorage(
                            "/Download",
                            part.originalFileName,
                            partStream
                        )
                    }
                }
                else → throw BadRequestException("Unsupported part type")
            }
            part.dispose()
        }
        call.respondText("File uploaded")
        clientData?.lastTransferOperationTime = Instant.now()
    } catch (e: BadRequestException) {
        call.respond(HttpStatusCode.BadRequest, "Bad request")
    }
}
```

Рис. 3.14. Обробка шляху /api/upload

Цей маршрут обробляє multipart-запити, що дозволяє завантажувати файли на сервер. Кожна частина запиту обробляється окремо, і якщо це файл (PartData.FileItem), він зберігається в директорії Download. Після успішного завантаження оновлюється час останньої операції в сесії. Обробка помилок включає перевірку типу контенту та правильності запиту. Також як і з іншими шляхами, які відповідають файловим операціям, змінній, в якій

записано час останньої активності присвоюється нове значення, яке відповідає поточному часу системи шляхом виклику методу now().

Одним із найважливіших методів класу KtorServer є метод failIfUnauthorized зображений на рисунку 3.15. Метод перевіряє наявність активної сесії та валідність коду сесії в параметрах запиту.

```
private suspend fun failIfUnauthorized(
    call: ApplicationCall,
    clientData: ClientData?
) {
    if (clientData == null) call.respond(
        HttpStatusCode.InternalServerError,
        "Session isn't available"
    )
    clientData?.let { data ->
        val timePassedSinceLastTransfer =
            Duration.between(data.lastTransferOperationTime, Instant.now())
        if (
            (!Objects.equals(
                call.request.queryParameters["session"],
                data.code
            ) && !IS_DEBUG) ||
            timePassedSinceLastTransfer.toMinutes() > Constants.SESSION_CODE_EXPIRATION_DELAY_MINUTES && !IS_DEBUG
        ) {
            call.respond(HttpStatusCode.Unauthorized, "Not authorized")
        }
    }
}
```

Рис. 3.15. Реалізація методу failIfUnauthorized

Додатково в failIfUnauthorized перевіряється час, що минув з останньої операції - якщо він перевищує встановлений ліміт, сесія вважається недійсною. Це забезпечує додатковий рівень безпеки при роботі з файловою системою.

Така детальна реалізація серверної частини забезпечує надійну та безпечну обробку файлових операцій, з належним контролем доступу та обробкою помилок.

3.1.3. Реалізація веб-інтерфейсу файлового менеджера

Веб-інтерфейс файлового менеджера реалізовано як односторінковий додаток (SPA) з використанням React та бібліотеки компонентів Geist UI. Основними компонентами інтерфейсу є:

1. Деревоподібна структура файлової системи, яка забезпечує:

- Ієрархічне відображення файлів та папок
- Динамічне завантаження вмісту при навігації
- Візуальні індикатори типів файлів
- Контекстні дії для кожного елемента

2. Система завантаження файлів з такими можливостями:
   - Множинне завантаження файлів
   - Індикація прогресу
   - Обробка помилок
   - Автоматичне оновлення списку файлів
3. Модальні вікна для операцій, що включають:
   - Діалог підтвердження видалення
   - Форму перейменування файлів
   - Інтерфейс переміщення файлів
   - Інформаційні повідомлення
4. Компонент відображення дискового простору з функціями:
   - Візуалізація заповнення пам'яті
   - Оновлення в реальному часі
   - Форматування розмірів у читабельному вигляді

Головним компонентом частини ReactJS застосунку є App.jsx, який реалізує графічну логіку управління файловою системою. Його реалізація розпочинається з визначення основних станів додатку, яка зображена на рисунку 3.16.

```
const [filesystemData, setFilesystemData] = useState(null);
const [sessionCode, setSessionCode] = useState('');
const [isLoading, setIsLoading] = useState(true);
const [loadingFolders, setLoadingFolders] = useState({});
const [expandedFolders, setExpandedFolders] = useState({});
```

Рис. 3.16. Стани застосунку

Стани визначаються за допомогою React-хуку useSate. Змінна filesystemData зберігає деревоподібну структуру файлової системи, sessionCode містить унікальний код сесії для автентифікації запитів до сервера, isLoading відповідає за відображення індикатора завантаження, loadingFolders зберігає статус завантаження для кожної теки окремо, а expandedFolders відслідковує, які теки розгорнуті в інтерфейсі.

На рисунку 3.17 зображено релізацію методу fetchFilesystem.

```
const fetchFilesystem = async (path) => {
    console.log('Fetching filesystem for path:', path);
    setLoadingFolders(prev => ({ ...prev, [path]: true }));
    try {
        const response = await fetch(`/api/browse?session=${sessionCode}`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({ path }),
        });
        if (!response.ok) throw new Error('Failed to fetch filesystem');
        const data = await response.json();
        setIsLoading(false);
        setLoadingFolders(prev => ({ ...prev, [path]: false }));
        return data;
    } catch (error) {
        setToast({ text: error.message, type: 'error' });
        setIsLoading(false);
        setLoadingFolders(prev => ({ ...prev, [path]: false }));
        return [];
    }
};
```

Рис. 3.17. Реалізація fetchFilesystem

Метод fetchFilesystem відповідає за отримання вмісту конкретної теки з сервера. При виклику методу встановлюється прапорець завантаження для вказаного шляху, виконується HTTP-запит до серверного API з передачею коду сесії та шляху. Для здійснення запиту без блокування основного потоку виконання використовується функція await [39]. Після отримання відповіді об’єкт з відповіддю перевіряється, щоб переконатися, був запит успішним, чи

ні. Якщо так, то дані зберігаються у стані компонента. У випадку помилки користувачу показується відповідне повідомлення через компонент Toast.

Фронтенд частина системи містить багато візуально-навантажених методів, проте з точки зору видимої користувачу роботи, одним із найважливіших є handleToggle, реалізацію якого зображено на рисунку 3.18.

```
const handleToggle = useCallback(async (path) => {
    setExpandedFolders(prev => ({
        ...prev,
        [path]: !prev[path]
    }));

    setFilesystemData(prevData => {
        const updateNode = (nodes) => {
            return nodes.map(node => {
                if (node.path === path) {
                    if (node.type === 'directory' && !node.loaded) {
                        fetchFilesystem(path).then(children => {
                            setFilesystemData(currentData => {
                                const updateWithChildren = (nodes) => {
                                    return nodes.map(n => {
                                        if (n.path === path) {
                                            return {
                                                ...n,
                                                children: transformData(children),
                                                loaded: true
                                            };
                                        }
                                        if (n.children) {
                                            return { ...n, children: updateWithChildren(n.children) };
                                        }
                                        return n;
                                    });
                                };
                                return updateWithChildren(currentData);
                            });
                        });
                    }
                    return { ...node, loaded: true };
                }
                if (node.children) {
                    return { ...node, children: updateNode(node.children) };
                }
                return node;
            });
        };
        return updateNode(prevData);
    });
}, [sessionCode]);
```

Рис. 3.18. Метод handleToggle

Метод handleToggle реалізує логіку розгортання/згортання тек у інтерфейсі. При першому розгортанні теки відбувається завантаження її вмісту з сервера. Важливою особливістю є рекурсивне оновлення структури даних - функція updateNode проходить по всьому дереву файлової системи, знаходить потрібний вузол та оновлює його стан і вміст. Використання

useCallback забезпечує оптимізацію продуктивності, запобігаючи зайвим перерендерам компонента.

За допомогою компонентів Geist UI та ReactJS вдалося реалізувати загальний графічний інтерфейс користувача, який зображено на рисунку 3.19.

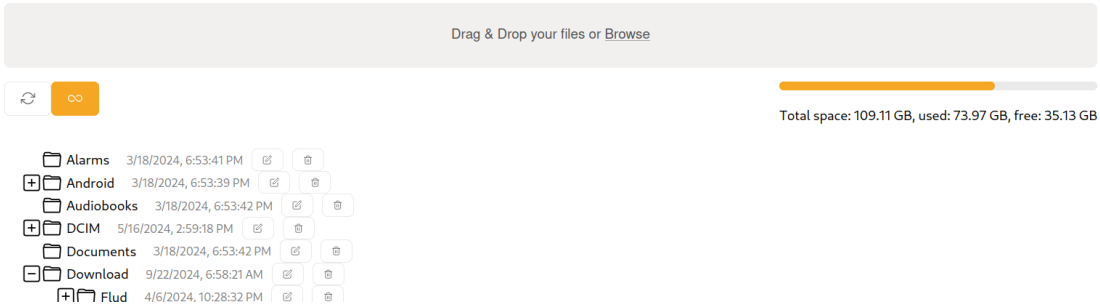


Рис. 3.19. Загальний вигляд графічного інтерфейсу при початковому вході

На рисунку 3.19 можна побачити інтерфейс компонента FilePond для надсилання файлів, в тому числі паралельно декількох, шкалу заповнення дискового простору мобільного пристрою, кнопку для оновлення даних про файли, а також головний компонент – дерево файлової системи. Папки, які містять якісь файли чи інші папки відображаються з іконкою “плюс” зліва від них, що позначає можливість для їх розкриття при натисканні. Якщо ж відображається іконка “мінус”, то папка вже відкрита і є можливість її закрити, щоб приховати її вміст. Справа від папок та файлів відображаються кнопки “Перейменувати” та “Видалити”, які виконують запит на однойменні дії.

Серед іншого під інтерфейсом для завантаження файлів в лівій частині міститься кнопка для примусового оновлення даних, яка змушує клієнт запросити дані про відкриті папки та файли повторно.

На рисунку 3.20 зображено інтерфейс діалогу для переміщення

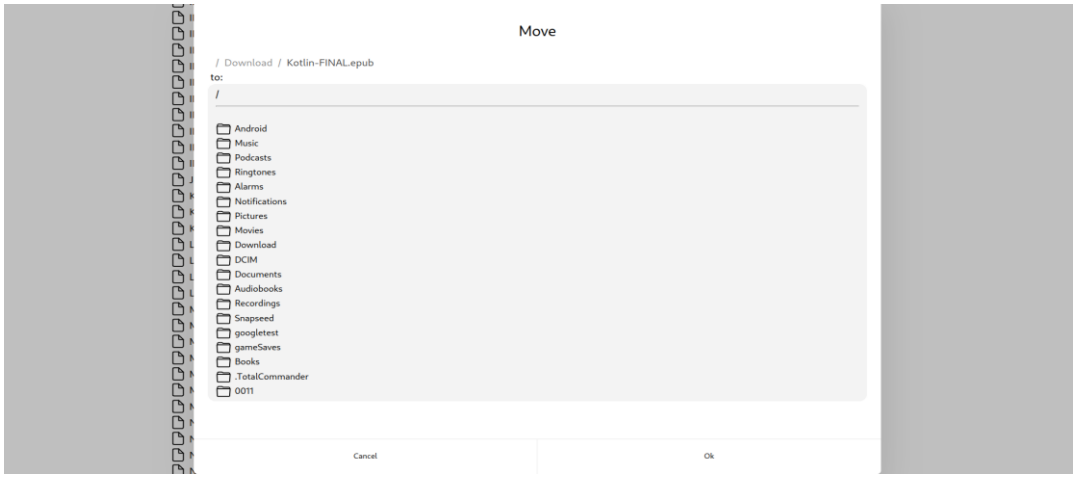


Рис. 3.20. Діалог для вибору папки-призначення для переміщення файлу

Діалог для вибору папки-призначення для переміщення обраного файлу на рисунку 3.20 дозволяє в реальному часі переходити між папками зображеними в сірій його частині, щоб обрати ту, в яку слід перемістити файл. По кліку на елементи файлового шляху в верхній частині сірої зони діалогу можна переміщуватися в попередні позиції.

Створений графічний веб-інтерфейс має сучасний, мінімалістичний вигляд, та дозволяє ефективно виконувати визначені у вимогах до системи операції.

3.2. Тестування системи

Для забезпечення якості та надійності розробленої системи було проведено комплексне тестування, що охоплювало різні аспекти функціонування застосунку.

3.2.1. Методологія тестування

У процесі розробки було застосовано комбінований підхід до тестування, що включав модульне, інтеграційне та end-to-end тестування [37]. Розглянемо кожен вид тестування детальніше.

Модульне тестування (Unit Testing) було спрямоване на перевірку окремих компонентів системи. Для кожного тесту створювалось ізольоване тестове оточення та виконувалась перевірка конкретної функціональності. Цей тип тестування може здаватися спершу не надто обов'язковим, оскільки може не одразу надавати значну користь, оскільки часто на початку програмне рішення може бути достатньо простим для ручного тестування. Проте з плином часу юніт-тести є чудовим інструментом швидкого виявлення будь-яких регресій.

Для тестування отримання детальної інформації про файл було створено тест зображений на рисунку 3.21.

```
@Test
public void testFetchFileInfo() {
    // Створюємо тестовий файл
    File testFile = new File(testDirectory, "test.txt");
    Files.write(testFile.toPath(), "test content".getBytes());

    // Отримуємо інформацію про файл
    FileInfo fileInfo = handler.fetchFile(testFile.getAbsolutePath());

    // Перевіряємо коректність отриманої інформації
    assertEquals("test.txt", fileInfo.getName());
    assertFalse(fileInfo.isDirectory());
    assertEquals(12, fileInfo.getSize());
}
```

Рис. 3.21. Метод для тестування отримання інформації про файл

Цей тест перевіряє коректність роботи методу fetchFile класу WebappRequestsHandler, який отримує базову інформацію про файл - його назву, тип (файл чи директорія) та розмір. Для цієї перевірки використано методи стандартних для Java інструментів для роботи з файлами з пакету java.nio.

Для тестування операцій з файлами було створено окремий тест перейменування зображений на рисунку 3.22.

```
@Test
public void testRenameFile() {
    // Створюємо тестовий файл
    File testFile = new File(testDirectory, "original.txt");
    Files.write(testFile.toPath(), "test content".getBytes());
    String newName = "renamed.txt";

    // Виконуємо перейменування
    boolean renamed = handler.renameFile(testFile.getAbsolutePath(), newName);

    // Перевіряємо результат
    assertTrue(renamed);
    assertFalse(testFile.exists());
    assertTrue(new File(testDirectory, newName).exists());
}
```

Рис. 3.22. Метод для тестування перейменування файлів

Цей тест перевіряє роботу методу renameFile, який відповідає за перейменування файлів. Тест створює файл, перейменовує його та перевіряє, що старий файл більше не існує, а новий створено успішно.

Окремо було протестовано також обробку помилкових ситуацій як зображено на рисунку 3.23, які мають викликати виключення.

```
@Test(expected = FileNotFoundException.class)
public void testFetchNonExistentFile() {
    // Спроба отримати неіснуючий файл має викликати виключення
    handler.fetchFile("/non/existent/path");
}

@Test(expected = SecurityException.class)
public void testAccessDeniedToSystemFiles() {
    // Спроба доступу до системних файлів має викликати виключення
    handler.fetchFile("/system/restricted.file");
}
```

Рис. 3.23. Методи для тестування ситуацій, які мають викликати виключення

Ці тести перевіряють коректність обробки помилкових ситуацій - спроби доступу до неіснуючих файлів та системних директорій. У обох випадках система має генерувати відповідні виключення.

Для підготовки тестового середовища використовувались допоміжні методи, які зображено на рисунку 3.24.

```
@Before
public void setUp() {
    // Створюємо тестове середовище перед кожним тестом
    mockContext = mock(Context.class);
    handler = new WebappRequestsHandler(mockContext);
    testDirectory = Files.createTempDirectory("test_dir").toFile();
}

@After
public void tearDown() {
    // Прибираємо тестові файли після кожного тесту
    FileUtils.deleteDirectory(testDirectory);
}
```

Рис. 3.24. Методи для обладнання тестового середовища

На рисунку 3.24 міститься метод setUp, який створює ізольоване тестове оточення перед кожним тестом, включаючи мок контексту Android та тимчасову директорію для тестових файлів. Метод tearDown забезпечує очищення тестового оточення після кожного тесту, видаляючи всі створені файли.

3.2.2. Аналіз продуктивності

Для оцінки продуктивності системи було розроблено комплексний набір тестів з використанням Apache JMeter. Тестування проводилось на пристрої Pixel 6 з Android 14 у локальній мережі WiFi 6.

Для тестування різних аспектів системи було створено три основні тест-плани в JMeter. Цей інструмент дозволяє симулювати запрограмоване

навантаження на клієнт-серверні системи для перевірки їх продуктивності та загального аналізу поведінки.

Першим є тест навантаження при передачі файлів зображений на рисунку 3.25.

```
<?xml version="1.0" encoding="UTF-8"?>
<jmeterTestPlan version="1.2" properties="5.0">
  <hashTree>
    <TestPlan guiclass="TestPlanGui" testclass="TestPlan" testname="File Upload Test Plan">
      <elementProp name="TestPlan.user_defined_variables" elementType="Arguments">
        <collectionProp name="Arguments.arguments">
          <elementProp name="session" elementType="Argument">
            <stringProp name="Argument.name">session</stringProp>
            <stringProp name="Argument.value">${__P(session)}</stringProp>
          </elementProp>
        </collectionProp>
      </elementProp>
    </TestPlan>
    <hashTree>
      <ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="File Upload Thread Group">
        <elementProp name="ThreadGroup.main_controller" elementType="LoopController">
          <boolProp name="LoopController.continue_forever">false</boolProp>
          <stringProp name="LoopController.loops">100</stringProp>
        </elementProp>
        <stringProp name="ThreadGroup.num_threads">10</stringProp>
        <stringProp name="ThreadGroup.ramp_time">5</stringProp>
      </ThreadGroup>
      <hashTree>
        <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy">
          <elementProp name="HTTPsampler.Files" elementType="HTTPFileArgs">
            <collectionProp name="HTTPFileArgs.files">
              <elementProp name="test_1mb.dat" elementType="HTTPFileArg">
                <stringProp name="File.path">test_1mb.dat</stringProp>
                <stringProp name="File.paramname">file</stringProp>
                <stringProp name="File.mimetype">application/octet-stream</stringProp>
              </elementProp>
            </collectionProp>
          </elementProp>
          <stringProp name="HTTPSampler.path">/api/upload?session=${session}</stringProp>
          <stringProp name="HTTPSampler.method">POST</stringProp>
          <boolProp name="HTTPSampler.use_keepalive">true</boolProp>
        </HTTPSamplerProxy>
        <ResultCollector guiclass="ViewResultsFullVisualizer" testclass="ResultCollector"/>
      </hashTree>
    </hashTree>
  </hashTree>
</jmeterTestPlan>
```

Рис. 3.25. Тест-план для JMeter для тестування завантаження файлів

Цей тест-план, наведений на рисунку 3.25, налаштований на:

- 10 паралельних потоків;
- 100 ітерацій на кожен потік;
- Поступове збільшення навантаження протягом 5 секунд;
- Вимірювання часу відгуку та пропускної здатності.

Він дозволяє побачити поведінку системи згідно поступово зростаючого навантаження, що особливо важливо для рішень, які мають на меті зберігати можливість майбутнього розширення.

Другим типом тестування продуктивності було обрано тестування файлових операцій. План тестування для програми JMeter для якого зображено на рисунку 3.26.

```
<ThreadGroup guiclass="ThreadGroupGui" testclass="ThreadGroup" testname="Sequential Operations Group">
  <elementProp name="ThreadGroup.main_controller" elementType="LoopController">
    <boolProp name="LoopController.continue_forever">false</boolProp>
    <stringProp name="LoopController.loops">1000</stringProp>
  </elementProp>
  <stringProp name="ThreadGroup.num_threads">1</stringProp>
  <stringProp name="ThreadGroup.ramp_time">1</stringProp>
</ThreadGroup>
<hashTree>
  <!— Тест операції перейменування —>
  <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy">
    <elementProp name="HTTPsampler.Arguments" elementType="Arguments">
      <collectionProp name="Arguments.arguments">
        <elementProp name="path" elementType="HTTPArgument">
          <stringProp name="Argument.value">/test.txt</stringProp>
        </elementProp>
        <elementProp name="newName" elementType="HTTPArgument">
          <stringProp name="Argument.value">test_renamed.txt</stringProp>
        </elementProp>
      </collectionProp>
    </elementProp>
    <stringProp name="HTTPSampler.path">/api/rename?session=${session}</stringProp>
    <stringProp name="HTTPSampler.method">POST</stringProp>
  </HTTPSamplerProxy>

  <!— Тест операції переміщення —>
  <HTTPSamplerProxy guiclass="HttpTestSampleGui" testclass="HTTPSamplerProxy">
    <elementProp name="HTTPsampler.Arguments" elementType="Arguments">
      <collectionProp name="Arguments.arguments">
        <elementProp name="originPath" elementType="HTTPArgument">
          <stringProp name="Argument.value">/test_renamed.txt</stringProp>
        </elementProp>
        <elementProp name="destinationPath" elementType="HTTPArgument">
          <stringProp name="Argument.value">/newdir</stringProp>
        </elementProp>
      </collectionProp>
    </elementProp>
    <stringProp name="HTTPSampler.path">/api/moveFile?session=${session}</stringProp>
    <stringProp name="HTTPSampler.method">POST</stringProp>
  </HTTPSamplerProxy>
```

Рис. 3.26. Тест-план для JMeter для тестування файлових операцій

Рисунок 3.26 демонструє тест-план для послідовного виконання файлових операцій з наступною загальною конфігурацією:

- 1000 послідовних операцій
- 50мс затримкою між операціями
- Збором статистики продуктивності.

На основі проведених тестів було отримано результати наведені в таблиці 3.1.

Таблиця 3.1

Результати тестування з JMeter

Операція	**Середній час (мс)**	**90% персентиль (мс)**	**Помилки (%)**	**Використання пам’яті (MB)**
Завантаження файлу 1MB	245	312	0.1	2.1
Завантаження файлу 10MB	1890	2340	0.3	18.5
Завантаження файлу 100MB	16400	19200	1.2	175.0
Перейменування файлу	12	45	0.0	0.8
Переміщення файлу	25	68	0.0	1.1

Аналіз результатів тестування показав:

1. Операції завантаження файлів:
    - Стабільна робота при паралельному завантаженні до 10 файлів;
    - Лінійне зростання часу обробки відносно розміру файлу;
    - Незначний відсоток помилок при роботі з великими файлами.
2. Файлові операції:
    - Висока швидкість базових операцій;
    - Відсутність помилок при послідовному виконанні;
    - Стабільне споживання пам'яті.

Результати тестування показали, що система здатна ефективно обробляти як одиночні операції, так і паралельні запити, зберігаючи при цьому стабільність роботи та прийнятний рівень споживання ресурсів, що особливо важливо в умовах виконання на мобільному пристрої з серйозно обмеженими характеристика.

3.2.3. Оцінка безпеки

Для оцінки безпеки системи було проведено комплексний аналіз з використанням автоматизованих тестів та ручної перевірки. Раннє тестування на стійкість до загроз дозволяє уникнути майбутніх збитків, хоча й зовсім не гарантує успішне відбиття всіх атак, а тільки збільшує таку ймовірність.

На рисунку 3.27 зображено тест генерації ключа сесії.

```
@Test
public void testSessionKeyGeneration() {
    // Тест генерації унікальних ключів
    Set<String> sessionIds = new HashSet<>();
    for (int i = 0; i < 1000; i++) {
        DataSession session = fileService.generateSession();

        // Перевірка унікальності
        assertFalse("Виявлено дублікат сесійного ключа",
            sessionIds.contains(session.code));
        sessionIds.add(session.code);

        // Перевірка довжини ключа
        assertEquals("Некоректна довжина ключа",
            Constants.SESSION_CODE_LENGTH,
            session.code.length());

        // Перевірка формату Base64URL
        assertTrue("Некоректний формат ключа",
            session.code.matches("^[A-Za-z0-9_-]+$"));
    }
}
```

Рис. 3.27. Тест генерації ключа сесії

Цей тест перевіряє основні вимоги до генерації сесійних ключів:

- Унікальність кожного згенерованого ключа;
- Відповідність довжини ключа заданій константі;
- Валідність формату (Base64URL-кодування).

Для перевірки згенерованих сесій на наявність терміну дії було створено тест зображений на рисунку 3.28.

```
@Test
public void testSessionExpiration() {
    DataSession session = fileService.generateSession();

    // Перевірка валідності нової сесії
    assertTrue("Сесія не валідна одразу після створення",
        isSessionValid(session.code));

    // Оновлення часу останньої операції
    session.lastTransferOperationTime =
        Instant.now().minus(Duration.ofMinutes(
            Constants.SESSION_CODE_EXPIRATION_DELAY_MINUTES + 1));

    // Перевірка закінчення терміну дії
    assertFalse("Сесія валідна після закінчення терміну дії",
        isSessionValid(session.code));
}
```

Рис. 3.28. Тест терміну дії сесії

Цей тест перевіряє механізм контролю терміну дії сесій:

- Правильність новоствореної сесії;
- Коректне визначення закінчення терміну дії;
- Правильність роботи з часовими мітками.

Одним із ключових векторів потенційної атаки в системі може бути сам механізм передачі даних. Реалізація доступу до файлів для коректної взаємодії з операційною системою має надавати реальні, або ж абсолютні шляхи до запитуваних файлів, в той час як клієнт не має мати доступу до цієї інформації. Адже така інформація може видавати особисті дані користувача, зокрема небажані назви папок, які в назвах містять паролі. Також серйозною загрозою є потенційна можливість з боку клієнта проводити ін’єкції власних значень в якості шляхів до файлів і вкладати в них рядки для відносного доступу, що може відкрити доступ до будь-яких файлів на пристрої. Такий тип атаки називається Path Traversal [45] і для визначення вразливості до нього, було використано тест зображений на рисунку 3.29.

```
@Test
public void testPathTraversalProtection() {
    WebappRequestsHandler handler = new
WebappRequestsHandler(context);

    // Тестові випадки потенційних Path Traversal
атак
    List<String> maliciousPaths = Arrays.asList(
        "../sensitive_file",
        "../../etc/passwd",
        "%2e%2e%2fetc%2fpasswd"
    );

    for (String path : maliciousPaths) {
        try {
            handler.fetchFile(path);
            fail("Path traversal вразливість
виявлена для: " + path);
        } catch (SecurityException e) {
            assertEquals("Заборонений доступ до
файлу", e.getMessage());
        }
    }
}
```

Рис. 3.29. Тест на захищеність від Path Traversal атаки

Цей тест перевіряє захист від спроб доступу до файлів поза дозволеною директорією:

- Спроби відносної навігації (навігація за непрямими шляхами);
- URL-кодовані шляхи;
- Доступ до системних файлів.

Останній пункт заслуговує окремої уваги, адже Path Traversal атка залежно від реалізації бекенд-частини системи може складатися не тільки зі звичайних текстових рядків, а і містити URL-закодовані частини із зашифрованими в них символами для непрямого доступу до шляхів.

Наступним тестом, testFilenameValidation, зображеним на рисунку 3.30 перевіряється правильність назв файлів, які передаються в систему з веб-інтерфейсу.

```
@Test
public void testFilenameValidation() {
    WebappRequestsHandler handler = new
WebappRequestsHandler(context);

    // Тестові випадки некоректних імен файлів
    Map<String, String> invalidNames = Map.of(
        "file\u0000.txt", "Null-байт в імені",
        "com1", "Зарезервоване системне ім'я",
        "file*.txt", "Заборонені символи",
        "", "Пусте ім'я"
    );

    for (Map.Entry<String, String> entry :
invalidNames.entrySet()) {
        try {
            handler.renameFile("/test.txt",
entry.getKey());
            fail("Прийнято некоректне ім'я: " +
entry.getValue());
        } catch (IllegalArgumentException e) {
            // Очікувана поведінка
        }
    }
}
```

Рис. 3.30. Тест валідації імен файлів

Тест перевіряє систему валідації імен файлів на:

- Наявність null-байтів;
- Використання зарезервованих імен;
- Наявність заборонених символів;
- Граничні випадки (пусті імена).

Успішним результатом виконання цього тесту є перехоплення всіх отриманих виключень, що показує коректну поведінку при валідації перелічених вище типів помилкових імен файлів. Ця перевірка схожа на попередню за логікою вразливості, а саме можливої ін’єкції символів.

Всі результати, а також випливаючі з тестування рекомендації було зібрано і наведено в таблиці 3.2.

Таблиця 3.2

Результати тестування безпеки

Категорія тесту	**Результат**	**Виявлені проблеми**	**Можливі покращення**
Генерація сесій	Успішно	Немає	–
Термін дії сесій	Успішно	Немає	–
Path Traversal	Успішно	Немає	Журналювання спроб атак
Валідація файлів	Задовільно	Відсутня перевірка MIME-типів	Додати валідацію файлів за типом

Тестування безпеки показало, що система забезпечує достатній рівень захисту від зловмисних дій. Є також і простір для покращення. Зокрема є зміст додати журналювання спроб атак та валідацію типів файлів, які можна передавати. Проте, на даному етапі за відсутності значної бази користувачів, такі доповнення вважаю недоречними, оскільки система призначена в першу чергу для особистого користування на власних пристроях в межах власної локальної точки доступу, що робить її складно використовуваною для зловмисних дій.

Висновки до розділу

У третьому розділі було розкрито практичну реалізацію системи віддаленого доступу до файлової системи мобільного пристрою та проведено ґрунтовний аналіз отриманих результатів. В процесі роботи над системою

вдалося досягти гармонійного поєднання різних технологій та підходів до розробки програмного забезпечення.

Особливо важливим аспектом реалізації стало створення надійної та ефективної архітектури Android-додатку, де за допомогою сервісного підходу вдалося забезпечити стабільну роботу системи навіть при тривалому використанні. Використання комбінації мов Java та Kotlin дозволило максимально використати переваги кожної з них: Java забезпечила надійну основу для бізнес-логіки, тоді як Kotlin значно спростив реалізацію асинхронних операцій та мережевої взаємодії.

Вбудований веб-сервер на базі Ktor виявився оптимальним рішенням для забезпечення комунікації між мобільним пристроєм та веб-інтерфейсом. Його асинхронна природа та модульна архітектура дозволили досягти високої продуктивності при мінімальному споживанні ресурсів пристрою. Веб-інтерфейс, реалізований з використанням сучасного стеку технологій React та Geist UI, надає користувачам інтуїтивно зрозумілий та адаптивний інструмент для роботи з файловою системою.

Проведене комплексне тестування системи підтвердило її надійність та ефективність. Особливо цікавими виявилися результати тестування продуктивності, які продемонстрували лінійну залежність часу обробки від розміру файлів та стабільну роботу при паралельному виконанні операцій. Це свідчить про правильність обраних архітектурних рішень та ефективність їх реалізації.

Важливо відзначити, що розроблена система безпеки, хоч і має потенціал для вдосконалення, вже зараз забезпечує надійний захист від найпоширеніших видів атак. Використання криптографічно стійких механізмів генерації сесійних ключів [40] та ретельна валідація всіх операцій створюють надійний фундамент для безпечного використання системи.

Результати цього етапу роботи переконливо демонструють, що розроблена система не лише відповідає всім початковим вимогам, але й має

потенціал для подальшого розвитку та вдосконалення. Створений програмний продукт являє собою практичне рішення актуальної проблеми віддаленого доступу до файлової системи мобільних пристроїв, поєднуючи в собі надійність, безпеку та зручність використання.

ВИСНОВКИ

В ході виконання магістерської роботи було успішно розроблено мобільний застосунок на платформі Android, призначений для організації віддаленого доступу та управління файловою системою мобільного пристрою через веб-інтерфейс. Застосунок надає користувачам можливість зручного доступу до файлів на своєму пристрої з будь-якого іншого пристрою в межах локальної мережі, без необхідності встановлення додаткового програмного забезпечення.

Застосунок реалізовано з використанням комбінації мов програмування Java та Kotlin, де Java забезпечує надійну основу для бізнес-логіки та роботи з файловою системою, а Kotlin значно спрощує реалізацію мережевої взаємодії завдяки інтеграції з фреймворком Ktor. Такий підхід дозволив максимально ефективно використати переваги обох мов та створити надійну та продуктивну систему.

Використання фреймворку Ktor разом із вбудованим сервером Netty забезпечило можливість перетворення мобільного пристрою на повноцінний веб-сервер, здатний обробляти запити та керувати файловою системою в асинхронному режимі. Для створення користувацького веб-інтерфейсу було обрано сучасний стек технологій на базі JavaScript, ReactJS та Geist UI, що дозволило реалізувати зручний та інтуїтивно зрозумілий інтерфейс файлового менеджера.

Особливу увагу в роботі було приділено питанням безпеки. Розроблена система працює виключно в межах локальної мережі та використовує унікальні сесійні ключі для автентифікації, які передаються через QR-коди. Це забезпечує простий та водночас надійний спосіб підключення, створюючи природний периметр безпеки без необхідності додаткової конфігурації.

Розробка даного застосунку дала змогу глибоко вивчити особливості роботи з файловою системою Android, принципи побудови веб-серверів на

мобільних пристроях та сучасні підходи до створення веб-інтерфейсів. Особливо цікавим виявився досвід інтеграції різних технологій та вирішення викликів, пов'язаних із забезпеченням стабільної роботи в умовах обмежених ресурсів мобільного пристрою.

Проведене тестування системи показало високу надійність та продуктивність розробленого рішення. Застосунок демонструє стабільну роботу при тривалому використанні та ефективно справляється з обробкою файлів різного розміру. Результати тестування безпеки підтвердили надійність реалізованих механізмів захисту та стійкість до потенційних атак.

У подальшому перспективи розвитку застосунку можуть включати розширення функціональності файлового менеджера, оптимізацію механізмів передачі великих файлів, впровадження додаткових механізмів безпеки та адаптацію розробленого рішення для інших мобільних платформ.

Отже, виконання цієї магістерської роботи дозволило не лише створити практичне рішення актуальної проблеми, але й отримати цінний досвід у розробці складних програмних систем. Результатом стала успішна реалізація застосунку, який ефективно вирішує поставлені задачі та має значний потенціал для подальшого розвитку. Особливо приємно відзначити, що розроблена система може бути корисною як для особистого використання, так і в професійному середовищі, наприклад, для швидкого обміну файлами під час презентацій або в навчальному процесі.

ПЕРЕЛІК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Murphy, M.: The Busy Coder's Guide to Android Development. CommonsWare, 2022.
2. Murphy, M.: Elements of Android Jetpack. CommonsWare, 2021.
3. Murphy, M.: Exploring Android. CommonsWare, 2020.
4. Wei-Meng, L.: Beginning Android Programming with Kotlin. Wiley Publishing, 2019.
5. Accomazzo, A.; et al.: Fullstack React: The Complete Guide to ReactJS and Friends. Fullstack.io Inc., 2017.
6. Banks, A.; Porcello, E.: Learning React: Functional Web Development with React and Redux. O'Reilly Media, 2020.
7. Bertoli, M.: React Design Patterns and Best Practices. Packt Publishing, 2019.
8. Chakraborty, R.: Reactive Programming with Kotlin. Packt Publishing, 2020.
9. Crockford, D.: JavaScript: The Good Parts. O'Reilly Media, 2008.
10. Dautel, P. M.: Programming Android with Kotlin. O'Reilly Media, 2018.
11. Dinić, I.: Kotlin Coroutines by Example. Packt Publishing, 2021.
12. Nudelman, G.: Android Design Patterns: Interaction Design Solutions for Developers. Wiley Publishing, 2013.
13. Phillips, B.; Hardy, C.: Android Programming: The Big Nerd Ranch Guide (5th Edition). Big Nerd Ranch Guides, 2023.
14. Smyth, N.: Android Studio 4.2 Development Essentials - Kotlin Edition. PayloadMedia Inc., 2021.
15. Smith, C.; Davies, B.: Asynchronous Android Programming. Packt Publishing, 2016.
16. Flanagan, D.: JavaScript: The Definitive Guide (7th Edition). O'Reilly Media, 2020.

17. Freeman, A.: Pro React 16. Apress, 2019.
18. Griffiths, D.; Griffiths, D.: Head First Android Development. O'Reilly Media, 2015.
19. Haverbeke, M.: Eloquent JavaScript: A Modern Introduction to Programming. No Starch Press, 2020.
20. Horton, J.: Android Programming with Kotlin for Beginners. Packt Publishing, 2021.
21. Jemerov, D.; Isakova, S.: Kotlin in Action. Manning Publications, 2017.
22. Jemerov, D.; Isakova, S.: Kotlin for Java Developers. JetBrains, 2017.
23. Karumanchi, N.: Data Structures & Algorithms in Kotlin. CareerMonk Publications, 2021.
24. Kumar, S.; et al.: Building Microservices with Ktor. Packt Publishing, 2021.
25. Leiva, A.: Kotlin for Android Developers. Independently published, 2017.
26. Levin, J.: Android Internals: A Confectioner's Cookbook. Technologeeks Press, 2015.
27. López Manas, E.: Android High Performance Programming. Packt Publishing, 2015.
28. Meier, R.; Lake, I.: Professional Android. Wiley Publishing, 2020.
29. Moskala, M.: Android Development with Kotlin. Packt Publishing, 2018.
30. Phillips, B.; Stewart, C.: Android Programming: The Big Nerd Ranch Guide. Big Nerd Ranch Guides, 2020.
31. Simpson, K.: You Don't Know JS (book series). O'Reilly Media, 2015-2020.
32. Skeen, J.; Greenhalgh, D.: Kotlin Programming: The Big Nerd Ranch Guide. Big Nerd Ranch Guides, 2020.
33. Späth, P.: Kotlin for Android App Development. Apress, 2018.
34. Stefanov, S.: JavaScript Patterns. O'Reilly Media, 2010.
35. Stefanov, S.: React Up & Running: Building Web Applications. O'Reilly Media, 2016.

36. Subramaniam, V.: Programming Kotlin Applications. Pragmatic Bookshelf, 2020.
37. Andersson, H.: A Comparison of the Performance of an Android Application Developed in Native and Cross-Platform. In Journal of Mobile Computing, 2022; pp. 45-52.
38. Cejas, F.: Advanced Android App Architecture. In Android Developer Journal, 2018; pp. 78-92.
39. Johnson, R.; et al.: Evaluating the Performance of JavaScript Frameworks. In International Journal of Web Engineering and Technology, 2023; pp. 112-125.
40. Martin, K.; Smithson, P.: Understanding Jetpack Compose for Modern Android UI. In Journal of Computer Science and Technology Trends, 2023; pp. 67-82.
41. Moskala, M.: How to Build Android Apps with Kotlin. In Proceedings of KotlinConf, 2017; pp. 123-135.
42. Patel, S.; et al.: Cross-Platform Mobile App Development: A Comparative Study. In Journal of Mobile Computing and Application Development, 2022; pp. 89-104.
43. Smith, J.; et al.: A Study on the Performance of Kotlin in Android Development. In Journal of Software Engineering Research and Development, 2022; pp. 156-170.
44. Thompson, R.; Carter, M.: The Evolution of JavaScript Frameworks. In ACM Computing Surveys, 2021; pp. 1-28.
45. Verma, N.; et al.: Development of Native Mobile Application Using Android Studio. In International Journal of Mobile Development, 2023; pp. 234-245.

ДОДАТКИ

Додаток А

Код файлу WebappRequestsHandler.java

```
    public class WebappRequestsHandler {

        private final Context applicationContext;

        public WebappRequestsHandler(Context
applicationContext) {
            this.applicationContext = applicationContext;
        }

        public void saveFileFromStream(String relativePath,
String fileName, InputStream dataStream) throws
IllegalArgumentException, IOException {
            String absolutePath =
convertToSystemPath(relativePath);
            File targetFile = new File(absolutePath, fileName);
            Files.copy(dataStream, targetFile.toPath());
        }

        public List<EntryMetadata> listDirectoryContents(String
relativePath) throws FileNotFoundException,
IllegalArgumentException {
            return scanDirectoryEntries(relativePath);
        }

        public List<FolderMetadata> listSubdirectories(String
relativePath) throws FileNotFoundException {
            return scanFolderStructure(relativePath);
        }

        public File retrieveFile(String relativePath) throws
FileNotFoundException, IllegalArgumentException {
            String absolutePath =
convertToSystemPath(relativePath);
            File targetFile = new File(absolutePath);
            if (!targetFile.exists() || !targetFile.isFile()) {
                throw new FileNotFoundException("Unable to
locate specified file");
            }
            return targetFile;
        }
        public boolean removeEntry(String relativePath) throws
FileNotFoundException, IllegalArgumentException {
            String absolutePath =
convertToSystemPath(relativePath);
            File targetEntry = new File(absolutePath);
```

```
            if (!targetEntry.exists() || !targetEntry.isFile())
{
                throw new FileNotFoundException("Unable to
locate specified entry");
            }
            if (targetEntry.isDirectory()) {
                return cleanDirectory(targetEntry);
            }
            return targetEntry.delete();
        }
        public StorageMetrics getStorageMetrics() {
            StatFs storageStats = new
StatFs(Environment.getDataDirectory().getAbsolutePath());
            long totalCapacity = ((long)
storageStats.getBlockCountLong() * (long)
storageStats.getBlockSizeLong());
            long availableSpace = ((long)
storageStats.getAvailableBlocksLong() * (long)
storageStats.getBlockSizeLong());
            return new StorageMetrics(totalCapacity,
availableSpace);
        }

        public boolean updateEntryName(String relativePath,
String updatedName) throws FileNotFoundException,
FileAlreadyExistsException {
            String absolutePath =
convertToSystemPath(relativePath);
            File sourceFile = new File(absolutePath);
            if (!sourceFile.exists()) {
                throw new FileNotFoundException("Source file
not found");
            }
            File destinationFile = new
File(sourceFile.getParent(), updatedName);
            if (destinationFile.exists()) {
                throw new
FileAlreadyExistsException("Destination name already exists");
            }
            return sourceFile.renameTo(destinationFile);
        }

        public void relocateEntry(String sourcePath, String
destinationPath) throws IOException {
            String absoluteSourcePath =
convertToSystemPath(sourcePath);
            String absoluteDestPath =
convertToSystemPath(destinationPath);
            File sourceFile = new File(absoluteSourcePath);
            if (!sourceFile.exists()) {
                throw new FileNotFoundException("Source entry
not found");
```

```
            }
            File destinationFile = new File(absoluteDestPath,
sourceFile.getName());
            Files.move(sourceFile.toPath(),
destinationFile.toPath(), StandardCopyOption.ATOMIC_MOVE);
        }

        private boolean cleanDirectory(File directory) {
            if (directory.isDirectory()) {
                File[] entries = directory.listFiles();
                if (entries != null) {
                    for (File entry : entries) {
                        if (entry.isDirectory()) {
                            cleanDirectory(entry);
                        } else {
                            if (!entry.delete()) return false;
                        }
                    }
                }
            }
            return directory.delete();
        }
        private List<FolderMetadata> scanFolderStructure(String
relativePath) throws FileNotFoundException {
            String absolutePath =
convertToSystemPath(relativePath);
            File targetDir = new File(absolutePath);
            if (!targetDir.exists() ||
!targetDir.isDirectory()) {
                throw new FileNotFoundException("Directory not
found");
            }
            return Arrays.stream(targetDir.listFiles())
                    .filter(File::isDirectory)
                    .map(folder -> {
                        String virtualPath =
createVirtualPath(folder.getAbsolutePath());
                        return new
FolderMetadata(folder.getName(), virtualPath);
                    }).collect(Collectors.toList());
        }
        private List<EntryMetadata> scanDirectoryEntries(String
relativePath) throws FileNotFoundException {
            String absolutePath =
convertToSystemPath(relativePath);
            File targetDir = new File(absolutePath);
            if (!targetDir.exists() ||
!targetDir.isDirectory()) {
                throw new FileNotFoundException("Directory not
found");
            }
            return Arrays.stream(targetDir.listFiles())
```

```
                    .sorted((entry1, entry2) -> {
                        if (entry1.isDirectory() &&
!entry2.isDirectory()) {
                            return -1;
                        } else if (!entry1.isDirectory() &&
entry2.isDirectory()) {
                            return 1;
                        }
                        return
entry1.getName().compareTo(entry2.getName());
                    })
                    .map(entry -> {
                        boolean isDir = entry.isDirectory();
                        boolean containsItems = false;
                        if (isDir) {
                            String[] items = entry.list();
                            containsItems = items != null &&
items.length > 0;
                        }
                        String virtualPath =
createVirtualPath(entry.getAbsolutePath());
                        return new EntryMetadata(
                                entry.getName(),
                                virtualPath,
                                isDir,
                                containsItems,
                                entry.length(),
                                entry.lastModified()
                        );
                    })
                    .collect(Collectors.toList());
        }

        private String createVirtualPath(String absolutePath) {
            if (absolutePath.startsWith(getBasePath())) {
                return
absolutePath.substring(getBasePath().length());
            }
            throw new SecurityException("Path violates security
constraints");
        }
        private String convertToSystemPath(String relativePath)
{
            String systemPath = getBasePath() + relativePath;
            return systemPath.replace("..", "");
        }

        private String getBasePath() {
            return
applicationContext.getString(R.string.directory_source);
        }
    }
```