

**Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення**

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА СИСТЕМИ ОРГАНІЗАЦІЇ ФАЙЛІВ І ФОТО
ВИКОРИСТОВУЮЧИ PYTHON ТА SQLITE З ТЕГОВОЮ МОДЕЛЛЮ**

**DEVELOPMENT OF A FILE AND PHOTO ORGANIZATION SYSTEM
USING PYTHON AND SQLITE WITH A TAG MODEL**

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗ-44
Соколов Станіслав Олександрович

Керівник:
Денисюк Віктор Юрійович

Кваліфікаційну роботу
допущено до захисту
«__» _____ 2026 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
ЗАВІДУВАЧ КАФЕДРИ

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Соколову Станіславу Олександровичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка системи організації файлів і фото використовуючи Python та SQLite з теговою моделлю»

Керівник роботи: к.т.н., доцент Денисюк В. Ю.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: Python, PyQt6, SQLite, SQLAlchemy, Visual Studio Code, файлова система користувача, вимоги до організації файлів і фото з використанням тегової моделі, методичні вказівки до виконання кваліфікаційної роботи бакалавра.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): аналіз сучасного стану проблем автоматизації обліку персоналу та управління задачами, постановка завдання, формування вимог до розроблюваної інформаційної системи, обґрунтування вибору технологій та засобів розробки, практична реалізація об'єкта проєктування, розробка структури бази даних, обґрунтування принципів захисту інформаційно-комп'ютерної системи, тестування та налагодження програмного забезпечення.

5. Перелік графічного матеріалу: 16 рисунків, 8 лістингів, 1 таблиця.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Денисюк В. Ю.</i>		
Специфікація вимог до розробленої системи	<i>Денисюк В. Ю.</i>		
Розробка об'єкта проектування	<i>Денисюк В. Ю.</i>		
Нормоконтроль	<i>Суринович О. М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	<u> </u> %		
Академічна доброчесність	<i>Денисюк В. Ю.</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедрі	до 29.05.2026 р.	

Здобувач вищої освіти

Станіслав СОКОЛОВ

Керівник кваліфікаційної роботи

Віктор ДЕНИСЮК

АНОТАЦІЯ

Соколов С. О. Розробка системи організації файлів і фото використовуючи Python та SQLite з теговою моделлю. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності 121 «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків.

У роботі розроблено систему організації файлів і фотографій з використанням Python, SQLite та тегової моделі класифікації. У першому розділі проаналізовано проблему впорядкування локальних цифрових матеріалів, досліджено наявні засоби керування файлами та визначено завдання кваліфікаційної роботи. У другому розділі сформовано вимоги до системи, описано модель тегів, структуру бази даних, сценарії пошуку, фільтрації та перегляду файлів. У третьому розділі наведено реалізацію програмного продукту, описано роботу з файловою системою, збереження метаданих у SQLite, механізми захисту даних і результати тестування. Розроблена система забезпечує зручне групування, пошук і перегляд файлів та фото за тегами.

Ключові слова: організація файлів, фотографії, Python, SQLite, тегова модель, база даних, пошук.

ABSTRACT

Sokolov S. O. Development of a File and Photo Organization System Using Python and SQLite with a Tag Model. Manuscript. Qualification work for bachelor's degree in Software Engineering, specialty 121 Software Engineering. Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification work consists of an introduction, three chapters, conclusions, a list of references, and appendices.

The work presents the development of a file and photo organization system using Python, SQLite, and a tag-based classification model. The first chapter analyzes the problem of organizing local digital materials, examines existing file management tools, and defines the tasks of the qualification work. The second chapter specifies the system requirements and describes the tag model, database structure, and scenarios for searching, filtering, and viewing files. The third chapter presents the implementation of the software product, file system operations, metadata storage in SQLite, data protection mechanisms, and testing results. The developed system provides convenient grouping, searching, and viewing of files and photos by tags.

Keywords: file organization, photos, Python, SQLite, tag model, database, search.

ЗМІСТ

ВСТУП	7
РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ РОЗРОБКИ системи організації файлів і фото ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ	9
1.1 Аналіз сучасного стану проблеми	9
1.2 Постановка завдання на кваліфікаційну роботу бакалавра	12
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ ..	14
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення	14
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання ...	20
РОЗДІЛ 3 системи організації файлів і фото з теговою моделлю	25
3.1 Практична реалізація об'єкта проєктування	25
3.2 Розробка бази даних	33
3.3 Захист інформаційно-комп'ютерної системи	35
3.4 Тестування та налагодження інформаційно-комп'ютерної системи	37
ВИСНОВКИ	40
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	42
ДОДАТКИ	43

ВСТУП

Актуальність теми. У процесі роботи з навчальними, проектними, фото- та мультимедійними матеріалами користувачі часто зберігають файли у різних каталогах без єдиної системи класифікації. З часом це ускладнює пошук потрібного документа або зображення, створює дублікати, приховує корисні метадані та змушує витратити час на ручне переглядання папок. Актуальним є розроблення локальної системи організації файлів і фото, яка індексує вибрані каталоги, зберігає відомості про файли в базі даних, підтримує тегову модель і надає зручний графічний інтерфейс для пошуку, фільтрації та перегляду деталей.

Для побудови настільного інтерфейсу в системі використано PyQt6, який надає прив'язки Python до фреймворку Qt і містить засоби створення вікон, таблиць, списків, кнопок та інших елементів взаємодії з користувачем [1]. Зберігання інформації про каталоги, файли, теги та зв'язки між ними реалізовано через SQLAlchemy ORM, що дає змогу описувати структуру даних у вигляді моделей Python і працювати з реляційною базою на рівні об'єктів [2]. Для роботи з графічними файлами та отримання властивостей зображень застосовано можливості бібліотеки Pillow [3].

Мета кваліфікаційної роботи полягає у розробленні системи організації файлів і фото з використанням Python та SQLite, яка забезпечує індексацію каталогів, збереження метаданих, призначення тегів, пошук, фільтрацію, перегляд деталей і виявлення дублікатів.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- здійснити аналіз процесів локального зберігання файлів і фото та визначити проблеми пошуку, групування, дублювання й ручної класифікації матеріалів;
- сформулювати вимоги до настільної системи організації файлів з підтримкою каталогів індексації, метаданих, тегів, фільтрів і дій над вибраними файлами;

- спроектувати структуру бази даних для зберігання каталогів, файлів, тегів і зв'язків між файлами та тегами;
- обґрунтувати вибір Python, PyQt6, SQLite, SQLAlchemy та Pillow для реалізації системи;
- реалізувати модулі сканування каталогів, пошуку, тегування, перегляду деталей файлів, роботи з фото та пошуку дублікатів;
- перевірити працездатність системи на сценаріях додавання каталогів, фільтрації файлів, призначення тегів, відкриття матеріалів і виявлення однакових файлів.

Об'єктом дослідження є процеси локальної організації, класифікації та пошуку файлів і фото на персональному комп'ютері.

Предметом дослідження є методи та програмні засоби розроблення настільної системи індексації файлів із теговою моделлю, локальною базою даних і графічним інтерфейсом користувача.

Практична цінність розробки полягає у створенні застосунку, який допомагає користувачу швидко впорядковувати файли з різних папок, знаходити потрібні матеріали за назвою, типом або тегами, переглядати властивості файлів і фото, а також позначати дублікати без перенесення всієї інформації до хмарних сервісів.

РОЗДІЛ 1

АНАЛІЗ МЕТОДІВ РОЗРОБКИ СИСТЕМИ ОРГАНІЗАЦІЇ ФАЙЛІВ І ФОТО ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Програми для організації файлів і фотографій розв’язують задачу швидкого пошуку матеріалів у великих локальних колекціях. Для розроблюваного застосунку важливими є індексація каталогів, збереження відомостей про файли у базі даних, підтримка тегів, фільтрація за типами, перегляд зображень і виявлення дублікатів за хешем.

Одним із близьких аналогів є TagSpaces – кросплатформовий менеджер файлів, який дозволяє організовувати локальні файли за допомогою тегів без обов’язкової хмарної синхронізації [4]. На рисунку 1.1 показано інтерфейс TagSpaces.

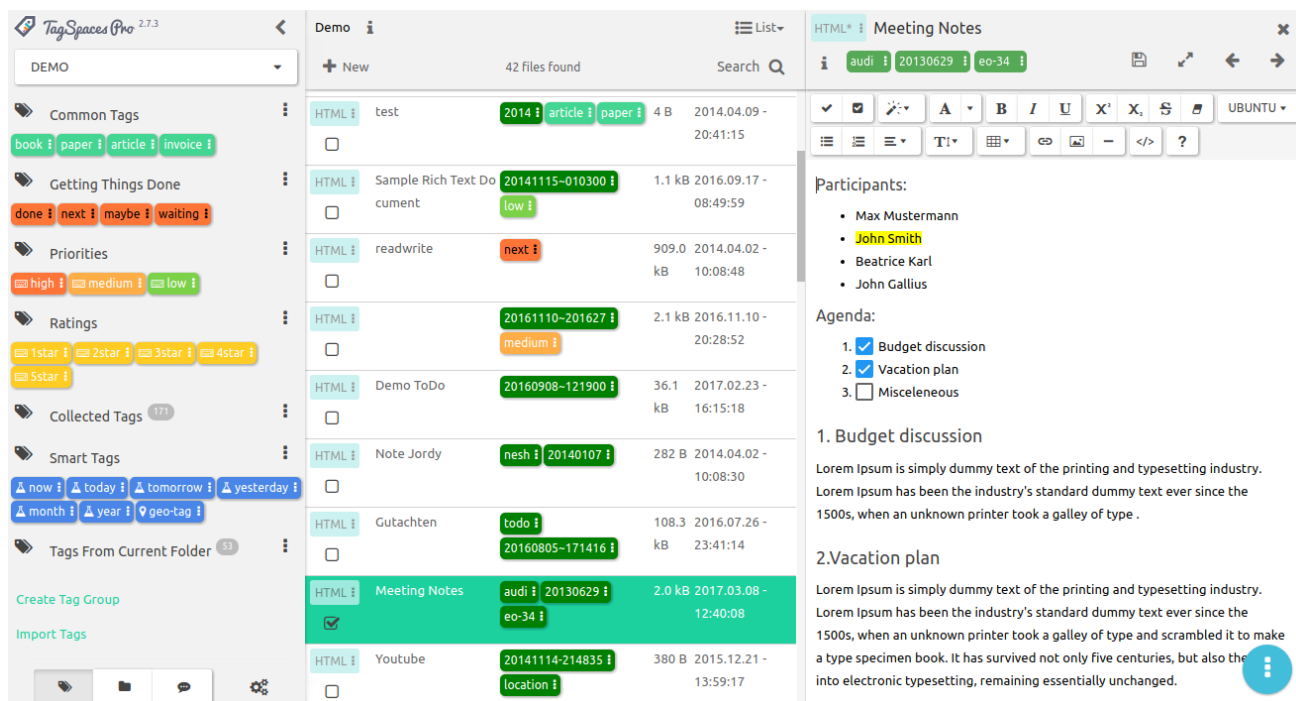


Рисунок 1.1 – Інтерфейс файлового менеджера TagSpaces [5]

Перевагою TagSpaces є зручна концепція тегування і робота з локальними файлами без прив'язки до конкретного сервісу. Недоліком у контексті розроблюваної системи є те, що готовий застосунок має власну модель взаємодії та ширший набір функцій, тоді як кваліфікаційна робота орієнтована на демонстрацію власної реалізації індексації, пошуку, перегляду зображень і перевірки дублікатів.

Для фотоархівів поширеним рішенням є digiKam – програма керування цифровими фотографіями з підтримкою альбомів, тегів, метаданих, пошуку та інструментів роботи із зображеннями [6]. На рисунку 1.2 наведено приклад екрана digiKam з режимом роботи з тегами.

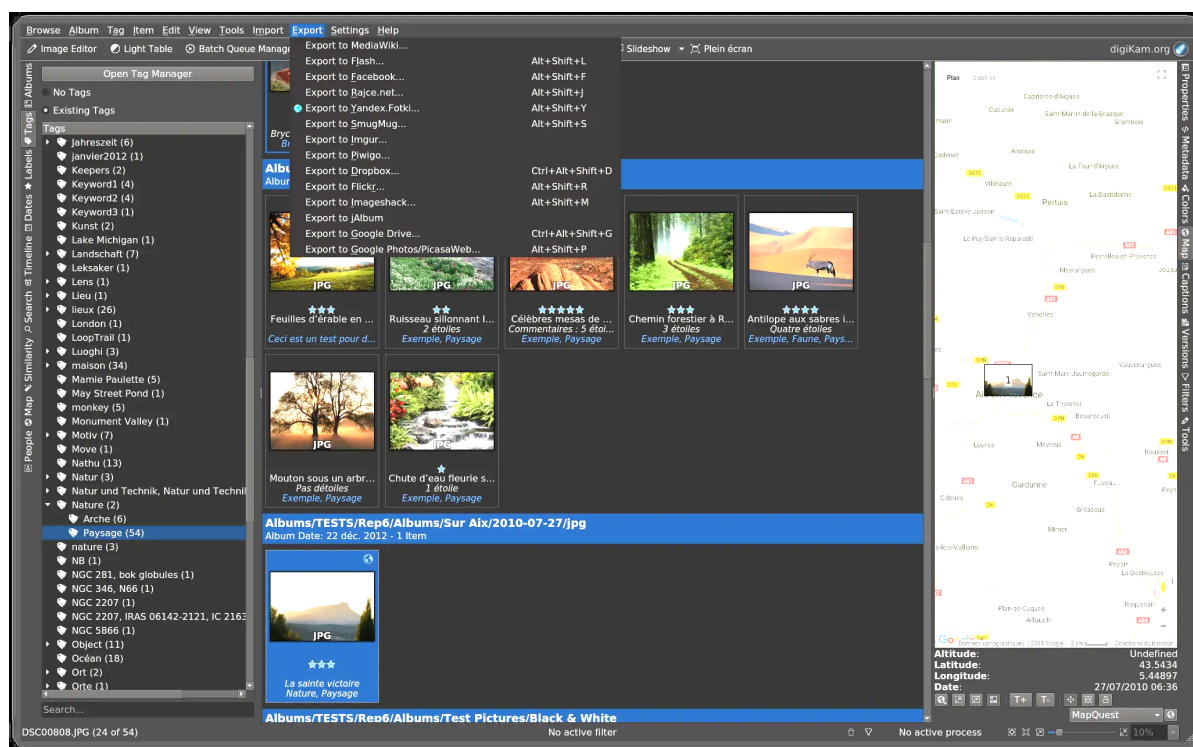


Рисунок 1.2 – Екран роботи з тегами у digiKam [6]

digiKam має потужні можливості для професійного ведення фотоколекцій, зокрема роботу з метаданими, групуванням і розширеним пошуком. Водночас система зосереджена переважно на фотографіях, а розроблений застосунок повинен однаково індексувати різні типи файлів, підтримувати загальні теги, операції перейменування, відкриття й видалення файлів.

Ще одним аналогом є Adobe Bridge – менеджер цифрових ресурсів для перегляду, пошуку, впорядкування і пакетної роботи з файлами у творчих проєктах [7]. Його сильна сторона полягає у професійному перегляді медіафайлів і зв'язку з іншими продуктами Adobe, однак для звичайної локальної організації файлів таке рішення є важким, комерційним і залежним від екосистеми виробника.

Порівняльну характеристику аналогів наведено в таблиці 1.1.

Таблиця 1.1 – Порівняння аналогів системи організації файлів і фотографій

Система	Переваги	Недоліки щодо поставленої задачі
TagSpaces	локальне тегування файлів, зручна навігація, кросплатформовість	готова система має власний підхід до зберігання тегів і не демонструє реалізацію індексації у навчальному проєкті
digiKam	потужна робота з фото, метаданими, альбомами і тегами	орієнтація передусім на фотоколекції, а не на універсальний файловий каталог
Adobe Bridge	професійний перегляд медіафайлів, фільтрація, пакетні операції	комерційність, залежність від екосистеми Adobe, надмірність для локального інструмента
Розроблюваний застосунок	проста індексація каталогів, база SQLite, теги, пошук, попередній перегляд і виявлення дублікатів	обмежений набір професійних функцій редагування та обробки медіа

Отже, аналоги демонструють різні підходи до впорядкування локальних файлів: тегування, фотоальбоми, професійне керування медіаресурсами. Розроблювана система займає проміжну позицію між простим файловим

менеджером і фотокаталогом, оскільки поєднує індексацію файлової структури, тегову модель, пошук, фільтрацію та базові операції з файлами у єдиному настільному інтерфейсі.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

В результаті аналізу процесів локального зберігання файлів, фото та супровідних метаданих було сформовано наступні завдання:

- здійснити аналіз процесів локального зберігання файлів і фото та визначити проблеми пошуку, групування, дублювання й ручної класифікації матеріалів, зосередивши увагу на ситуаціях, коли користувач працює з багатьма папками, різними форматами файлів і повторюваними копіями;

- сформулювати вимоги до настільної системи організації файлів з підтримкою каталогів індексації, метаданих, тегів, фільтрів і дій над вибраними файлами, передбачивши додавання та пересканування папок, відображення списку файлів, пошук за назвою, фільтрацію за типом і роботу з кількома вибраними елементами;

- спроектувати структуру бази даних для зберігання каталогів, файлів, тегів і зв'язків між файлами та тегами, визначивши таблиці folders, files, tags і file_tags, унікальні шляхи до файлів, ознаку зображення, розмір, розширення, тип, дати створення та змінення;

- обґрунтувати вибір Python, PyQt6, SQLite, SQLAlchemy та Pillow для реалізації системи, враховуючи потребу у локальному настільному інтерфейсі, компактному зберіганні даних, об'єктному доступі до таблиць і отриманні характеристик графічних файлів;

- реалізувати модулі сканування каталогів, пошуку, тегування, перегляду деталей файлів, роботи з фото та пошуку дублікатів, щоб користувач міг не лише переглядати індексовані матеріали, а й виконувати практичні дії: відкривати, перейменовувати, видаляти, позначати тегами та знаходити однакові файли;

– перевірити працездатність системи на сценаріях додавання каталогів, фільтрації файлів, призначення тегів, відкриття матеріалів і виявлення однакових файлів, окремо звернувши увагу на оновлення індексу після повторного сканування та коректність відображення деталей вибраного файла.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Під час аналізу предметної області встановлено, що система організації файлів і фотографій повинна поєднувати індексацію файлової структури, збереження метаданих, тегову класифікацію, пошук, фільтрацію, попередній перегляд і допоміжні операції з файлами. Основним завданням є створення локального каталогу, який дає змогу швидше знаходити потрібні матеріали без ручного перегляду багатьох папок. Для цього система повинна зберігати шлях до файла, назву, розширення, тип, розмір, дату створення, дату зміни, ознаку зображення, хеш файла, примітку та зв'язки з тегами.

Основні варіанти використання системи наведено на рисунку 2.1.

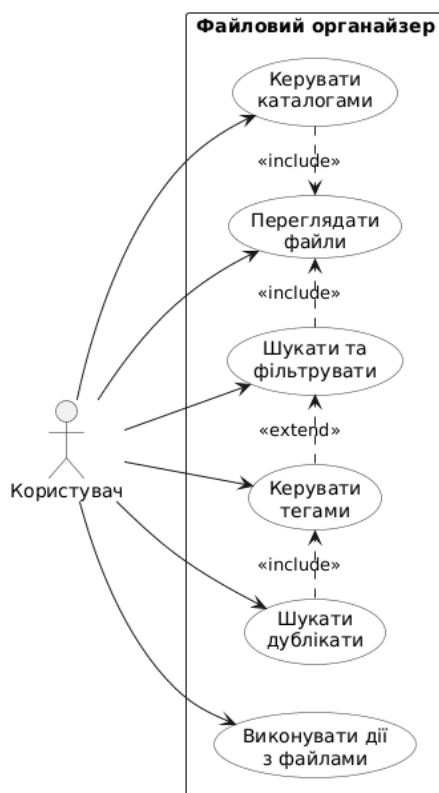


Рисунок 2.1 – UML-діаграма варіантів використання системи організації файлів і фотографій

Користувачем системи є власник локальної файлової колекції, який обирає папки для сканування, переглядає знайдені файли, створює теги, призначає їх одному або кільком файлам, фільтрує результати, переглядає деталі й виконує операції відкриття, перейменування або видалення. Окремий сценарій пов'язаний із пошуком дублікатів: система групує файли за розміром, обчислює SHA256-хеш для кандидатів і позначає знайдені дублікати тегом «Дублікат».

Функціональні вимоги до системи охоплюють сканування папок, повторне сканування збережених каталогів, індексацію файлів, пошук, фільтрацію, створення тегів, призначення тегів, видалення тегів, попередній перегляд і пошук дублікатів. Після вибору папки система повинна обійти її файлову структуру, знайти файли, визначити їхні метадані та створити або оновити записи в базі даних. Користувач повинен мати змогу додати нову папку до індексу або замінити попередньо проскановані каталоги новим набором даних.

Для роботи з тегами система повинна підтримувати створення тегу з назвою, кольором і описом, призначення тегу одному або кільком вибраним файлам, видалення зв'язку між тегом і файлом, а також повне видалення тегу. Пошук має виконуватися за частиною назви файла, активним тегом і типом файла. Додатково система повинна показувати список усіх тегів, список просканованих каталогів, таблицю файлів і панель деталей вибраного файла.

Нефункціональні вимоги визначають зручність, локальність і стійкість роботи застосунку. Інтерфейс повинен бути реалізований як настільний застосунок PyQt6 з окремими областями для каталогів індексації, пошуку, тегів, таблиці файлів і панелі попереднього перегляду [1]. Дані індексу повинні зберігатися у локальній SQLite базі через SQLAlchemy, що дає змогу працювати з моделями даних як з Python-об'єктами [2]. Тривалі операції сканування, завантаження списку файлів і пошуку дублікатів не повинні блокувати інтерфейс, тому вони виконуються у фонових потоках QThread.

Проектування системи виконано з розділенням на інтерфейс, сервіси, моделі та рівень доступу до бази даних. Головне вікно MainWindow відповідає за взаємодію з користувачем, відображення списків, запуск фонових операцій і

оновлення панелі деталей. ScannerService виконує обхід файлової системи та створює або оновлює записи файлів. SearchService формує запити для пошуку й фільтрації. TagService відповідає за створення тегів і керування зв'язками між файлами та тегами. Узагальнену компонентну структуру системи наведено на рисунку 2.2.

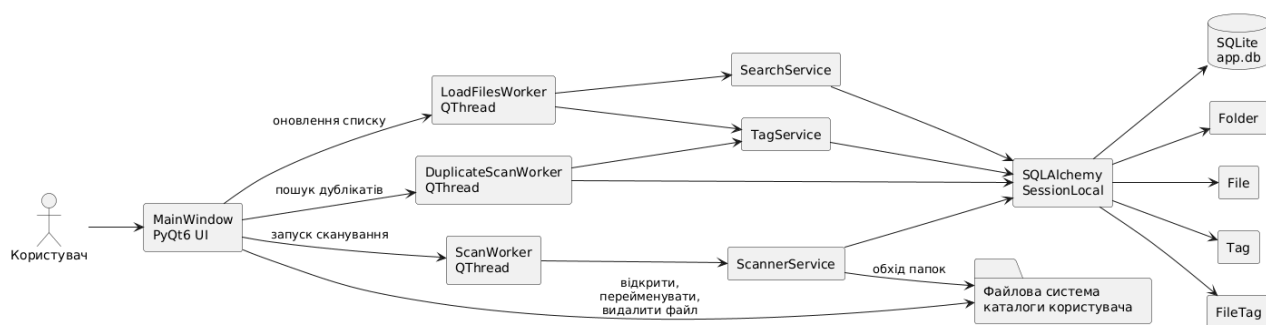


Рисунок 2.2 – UML-діаграма компонентів системи організації файлів і фотографій

Проектна модель даних складається з чотирьох основних сутностей: Folder, File, Tag і FileTag. Така структура відображає основну логіку файлового органайзера: користувач індексує папки, система знаходить у них файли, а потім ці файли можуть класифікуватися за допомогою довільних тегів. Folder зберігає шлях до просканованої папки та дату останнього сканування. Це дає змогу повторно оновлювати окремі каталоги без повної перебудови всього індексу. File описує знайдений файл і містить шлях, назву, розширення, тип, розмір, дати створення та зміни, хеш, ознаку зображення й примітку. Наявність цих полів дозволяє не звертатися до файлової системи для кожної операції пошуку, а використовувати вже збережені метадані.

Tag зберігає назву, колір та опис тегу, тобто задає користувацьку категорію для групування файлів. Завдяки тегам один і той самий файл може належати не лише до певної папки або типу, а й до змістової групи, визначеної самим користувачем. FileTag реалізує зв'язок багато до багатьох між файлами і тегами, оскільки один файл може мати декілька тегів, а один тег може бути призначений

багатьом файлам. Для цієї сутності задано унікальне обмеження на пару file_id і tag_id, щоб один і той самий тег не дублювався для одного файла. UML-діаграму класів проєктної моделі наведено на рисунку 2.3.

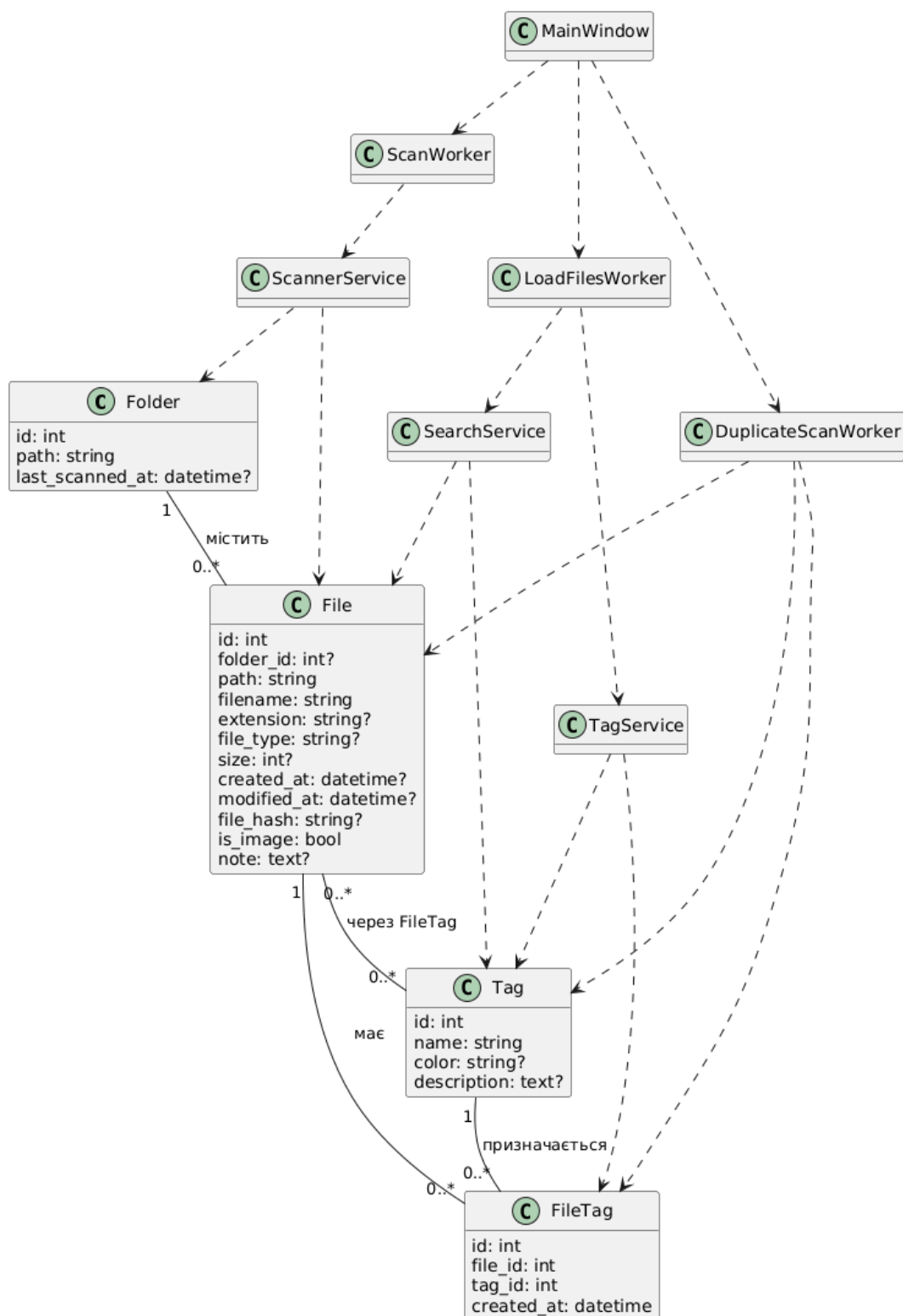


Рисунок 2.3 – UML-діаграма класів системи організації файлів і фотографій

Сценарій сканування папки починається з вибору каталогу користувачем. Після цього MainWindow запускає ScanWorker у фоновому потоці. ScanWorker створює сесію бази даних і викликає ScannerService. Сервіс перевіряє існування папки, створює або знаходить запис Folder, обходить усі файли через rglob, визначає розширення, тип, розмір і часові мітки, після чого додає нові записи або оновлює наявні. Після завершення сканування система фіксує дату останнього сканування папки, зберігає зміни та оновлює список файлів в інтерфейсі. Послідовність сканування папки наведено на рисунку 2.4.

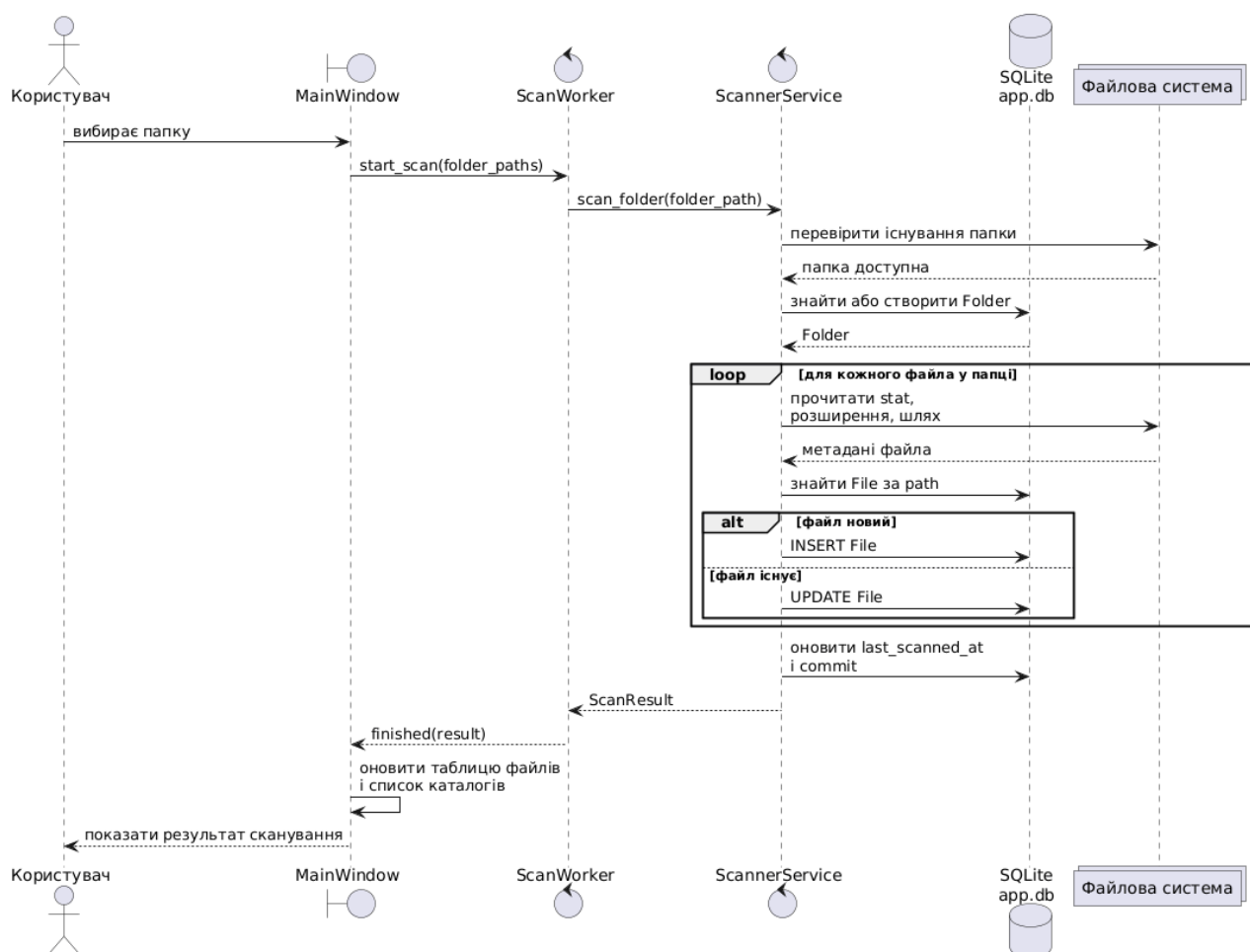


Рисунок 2.4 – UML-діаграма послідовності сканування папки

Сценарій пошуку та фільтрації виконується після введення тексту пошуку, вибору тегу або типу файла. MainWindow запускає LoadFilesWorker, який використовує SearchService і TagService. SearchService поступово формує

SQLAlchemy-запит: враховує частину назви файла, активні теги, ознаку зображення або відсутність тегів, а також сортує результати за назвою. Після отримання даних worker повертає рядки файлів, тегів і папок у головне вікно, а інтерфейс оновлює таблицю файлів, список тегів і список каталогів. Послідовність пошуку та фільтрації наведено на рисунку 2.5.

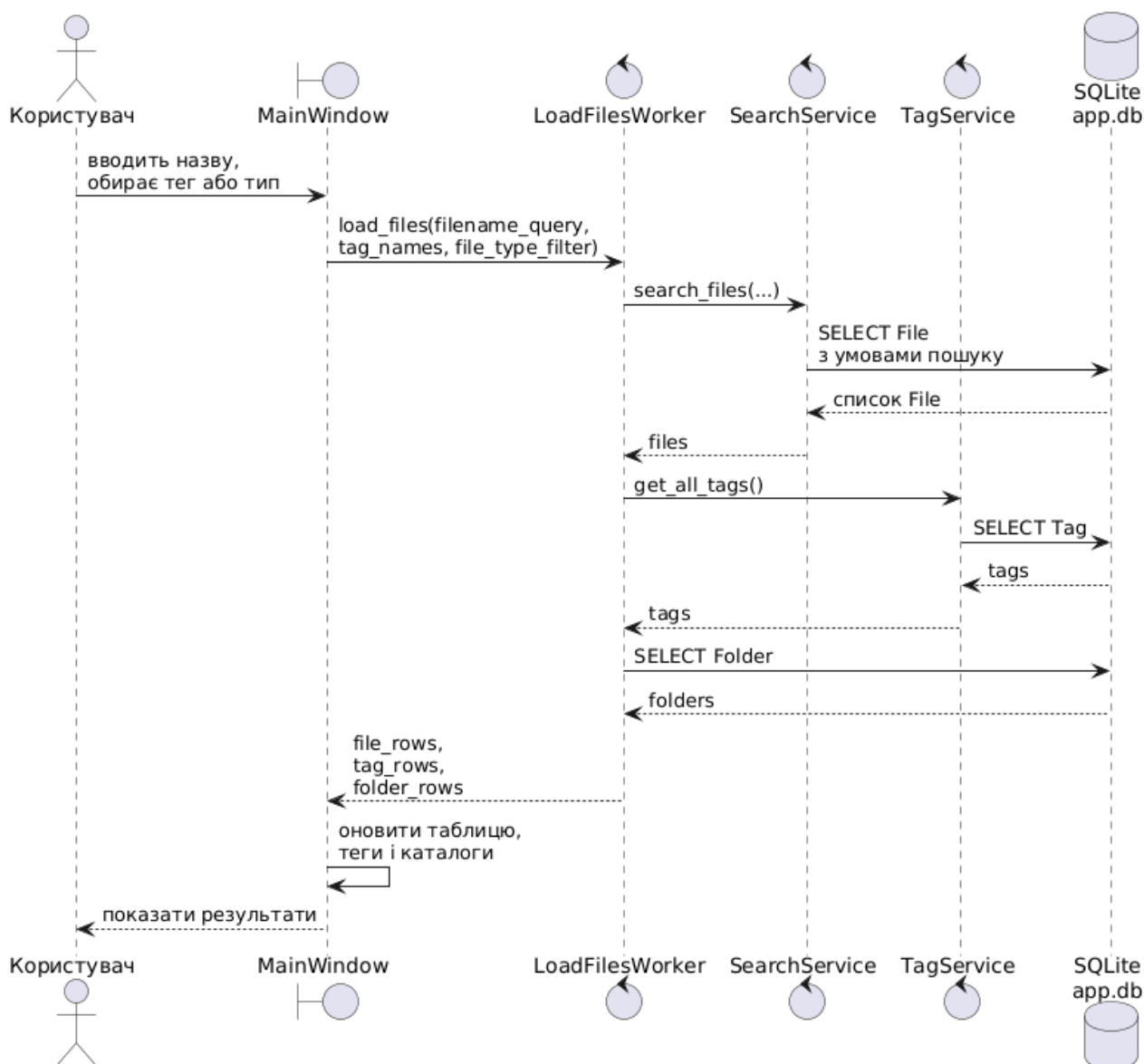


Рисунок 2.5 – UML-діаграма послідовності пошуку та фільтрації файлів

Сценарій пошуку дублікатів винесено в окремий фоновий процес DuplicateScanWorker. Спочатку система завантажує всі проіндексовані файли, відкидає відсутні або порожні файли, групує кандидати за розміром, а потім для

файлів з однаковим розміром обчислює SHA256-хеш. Якщо в одній групі знайдено кілька файлів з однаковим хешем, вони вважаються дублікатами. Для таких файлів створюється або використовується наявний тег «Дублікат», після чого система додає зв'язки у таблицю FileTag. Користувач може зупинити тривалий пошук, оскільки worker перевіряє запит на скасування під час оброблення файлів.

Отже, у підрозділі визначено користувацькі сценарії, функціональні й нефункціональні вимоги, компонентну структуру, модель даних і ключові алгоритми системи організації файлів і фотографій. Запропонована структура відповідає фактичній реалізації проєкту: PyQt6 використовується для інтерфейсу, SQLAlchemy і SQLite – для локального індексу, ScannerService – для сканування папок, SearchService – для пошуку, TagService – для тегування, а фонові worker-класи – для виконання тривалих операцій без блокування інтерфейсу.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Вибір засобів реалізації системи організації файлів і фотографій здійснювався з урахуванням локального характеру роботи, потреби в індексації великих каталогів, збереженні метаданих, підтримці тегової моделі, попередньому перегляді зображень і пошуку дублікатів за вмістом. Система не потребує серверного розгортання, тому основними критеріями вибору стали простота встановлення, швидкість роботи з файловою системою, зручність створення настільного інтерфейсу і надійне локальне зберігання даних.

Для реалізації інтерфейсу обрано PyQt6. PyQt надає Python прив'язки до фреймворку Qt і дозволяє створювати повноцінні настільні застосунки з вікнами, таблицями, списками, діалогами та сигналами подій [1]. У розробленій системі PyQt6 використано для головного вікна, панелі каталогів, таблиці файлів,

списків тегів, області деталей, кнопок дій і діалогів вибору папки або введення назви тегу.

Під час вибору технології інтерфейсу розглядалися PyQt6, Tkinter і вебінтерфейс. PyQt6 підтримує сучасні віджети, таблиці, списки, розділювачі панелей, сигнали, потоки виконання і гнучке компонування вікна. Його обмеженням є потреба встановлення окремої бібліотеки та уважна організація взаємодії між фоновими задачами і головним потоком інтерфейсу. Для цієї роботи такий вибір є виправданим, оскільки файловий органайзер має складний настільний інтерфейс із таблицею файлів, тегами, фільтрами і прев'ю. Tkinter входить до стандартного набору Python і підходить для невеликих віконних програм, але має простіший набір віджетів і менш зручний для побудови насиченого інтерфейсу з кількома панелями. Вебінтерфейс може бути доступним у браузері та легше масштабується до мережевого сценарію, проте потребує локального або віддаленого вебсервера, маршрутизації, додаткового захисту і складнішого доступу до локальних файлів. Тому для поточної реалізації обрано PyQt6.

Логіку доступу до даних побудовано на SQLAlchemy. У документації SQLAlchemy описано єдиний підхід до роботи з SQL базами даних через engine, session, декларативні моделі та запити [2]. У проєкті створено окремий модуль database.py, де визначено підключення до бази, фабрику сесій SessionLocal і базовий клас моделей. Такий підхід відокремлює інтерфейс від роботи з таблицями та дозволяє сервісам виконувати операції через сесії.

Для локального зберігання даних обрано SQLite. За описом SQLite, це самодостатній, серверонезалежний і такий, що не потребує окремого налаштування, SQL рушій [8]. Для файлового органайзера це важливо, оскільки база даних зберігається в одному файлі app.db і може містити шляхи до каталогів, метадані файлів, теги та зв'язки між файлами й тегами без розгортання серверної СКБД.

Під час вибору способу зберігання даних порівнювалися SQLite, JSON файли і PostgreSQL. SQLite зберігає дані у локальному файлі, не потребує

сервера, підтримує реляційні таблиці, індекси та зв'язки. Її обмеженням є те, що вона не призначена для великої кількості одночасних мережових користувачів, однак для локального файлового органайзера це не є критичним. JSON файли прості для читання і резервного копіювання, але незручні для складного пошуку, фільтрації, унікальних обмежень і зв'язків файлів з тегами. PostgreSQL є надійною серверною СКБД, придатною для великих обсягів даних і багатокористувацької роботи, проте потребує встановлення та адміністрування сервера. Тому для поточної версії системи обрано SQLite, а PostgreSQL може розглядатися лише для майбутньої мережової версії.

Структура бази даних відповідає теговій моделі організації файлів. Таблиця бази даних `folders` зберігає каталоги індексації та дату останнього сканування, таблиця `files` містить шлях, назву, розширення, тип, розмір, дати створення й змінення та ознаку зображення, таблиця `tags` описує назви й кольори тегів, а проміжна таблиця `file_tags` реалізує зв'язок файлів з тегами. Унікальність шляху до файла дає змогу оновлювати вже відомі записи під час повторного сканування, не створюючи дублікати в базі.

Для роботи із зображеннями використано Pillow. Модуль Image у Pillow призначений для відкриття, обробки та отримання характеристик графічних файлів [3]. У системі ця бібліотека застосовується для попереднього перегляду фотографій, визначення розмірів зображення і зчитування частини метаданих, потрібних для панелі деталей. Це дозволяє відрізнити фотографії від інших файлів і показувати користувачу корисну інформацію без відкриття сторонніх програм.

Щоб тривалі операції не блокували вікно, сканування каталогів, завантаження списку файлів і пошук дублікатів винесено у фонові задачі на основі QThread. У документації Qt зазначено, що QThread використовується для керування окремими потоками виконання [9-10]. У застосунку класи `ScanWorker`, `LoadFilesWorker` і `DuplicateScanWorker` виконують роботу з базою та файлами у фоновому режимі, а результат повертають у головне вікно через сигнали `finished`, `error`, `progress` і `cancelled`.

Алгоритм пошуку дублікатів використовує попереднє групування за розміром файла, а потім обчислення хешу лише для кандидатів з однаковим розміром. Для цього застосовано модуль `hashlib`, який підтримує конструктори стандартних алгоритмів хешування, зокрема `sha256` [11-12]. Такий підхід зменшує кількість повних читань файлів, оскільки файли різного розміру не можуть бути однаковими за вмістом.

Обрані засоби реалізації визначають подальшу логіку побудови системи. Python використовується для прикладної логіки, роботи з файловою системою, сервісів сканування, пошуку, тегування і виявлення дублікатів. PyQt6 забезпечує настільний інтерфейс з панелями каталогів, таблицею файлів, списками тегів, діалогами й фоновими задачами. SQLite відповідає за локальне зберігання каталогів, файлів, тегів і зв'язків між ними у файлі бази даних. SQLAlchemy використовується для опису моделей, створення сесій, виконання запитів, фільтрації, додавання й оновлення записів. Pillow застосовується для попереднього перегляду зображень і отримання характеристик графічних файлів, а `hashlib` – для обчислення хешів вмісту файлів під час пошуку дублікатів.

Алгоритм сканування каталогу починається з перевірки існування вибраної папки. Після цього система створює або знаходить запис каталогу в таблиці `folders` і рекурсивно обходить його вміст за допомогою `pathlib`. Для кожного файла визначається розширення, тип, розмір, час створення, час змінення та ознака зображення. Якщо файл уже є в базі за шляхом, його метадані оновлюються; якщо запису немає, створюється новий рядок у таблиці `files`.

Алгоритм пошуку і фільтрації реалізовано у `SearchService`. Спочатку формується базовий запит до таблиці `files`, після чого до нього послідовно додаються умови пошуку за частиною назви, фільтрації за зображеннями, відбору файлів без тегів або перевірки наявності вибраних тегів. Результати сортуються за назвою і передаються в головне вікно для відображення у таблиці.

Алгоритм тегування побудовано через сервіс `TagService`. Під час створення тегу назва очищується від зайвих пробілів і перевіряється на порожнє

значення. Якщо тег уже існує, використовується наявний запис, інакше створюється новий. Призначення тегу файлу відбувається через таблицю `file_tags`, де перед додаванням перевіряється наявність такого самого зв'язку. Це захищає базу від повторного призначення одного тегу одному файлу.

Алгоритм роботи з дублікатами складається з декількох етапів. Спочатку всі файли групуються за розміром, після чого для груп з двома і більше елементами обчислюється `sha256` хеш. Файли з однаковим хешем збираються у групи дублікатів, а система створює або використовує тег Дублікат і автоматично призначає його всім знайденим файлам. Користувач може перервати перевірку, оскільки фоновий процес регулярно перевіряє прапорець скасування.

Отже, для реалізації системи організації файлів і фотографій обрано Python, PyQt6, SQLite, SQLAlchemy, Pillow і hashlib. Такий набір засобів відповідає локальному настільному характеру розробки, підтримує реляційну тегову модель, забезпечує зручний інтерфейс для роботи з каталогами і дозволяє автоматизувати пошук, перегляд, тегування та виявлення однакових файлів.

РОЗДІЛ 3

РОЗРОБКА СИСТЕМИ ОРГАНІЗАЦІЇ ФАЙЛІВ І ФОТО З ТЕГОВОЮ МОДЕЛЛЮ

3.1 Практична реалізація об'єкта проєктування

Практична реалізація системи виконана як настільний застосунок Python із графічним інтерфейсом PyQt6. Код поділено на модулі `core`, `models`, `services`, `repositories` та `ui`. У `core` зосереджено конфігурацію і підключення до бази даних, у `models` описано SQLAlchemy сутності, у `services` реалізовано сканування, пошук і тегування, а `MainWindow` координує дії користувача в інтерфейсі.

Структуру проєкту у середовищі розробки показано на рисунку 3.1.

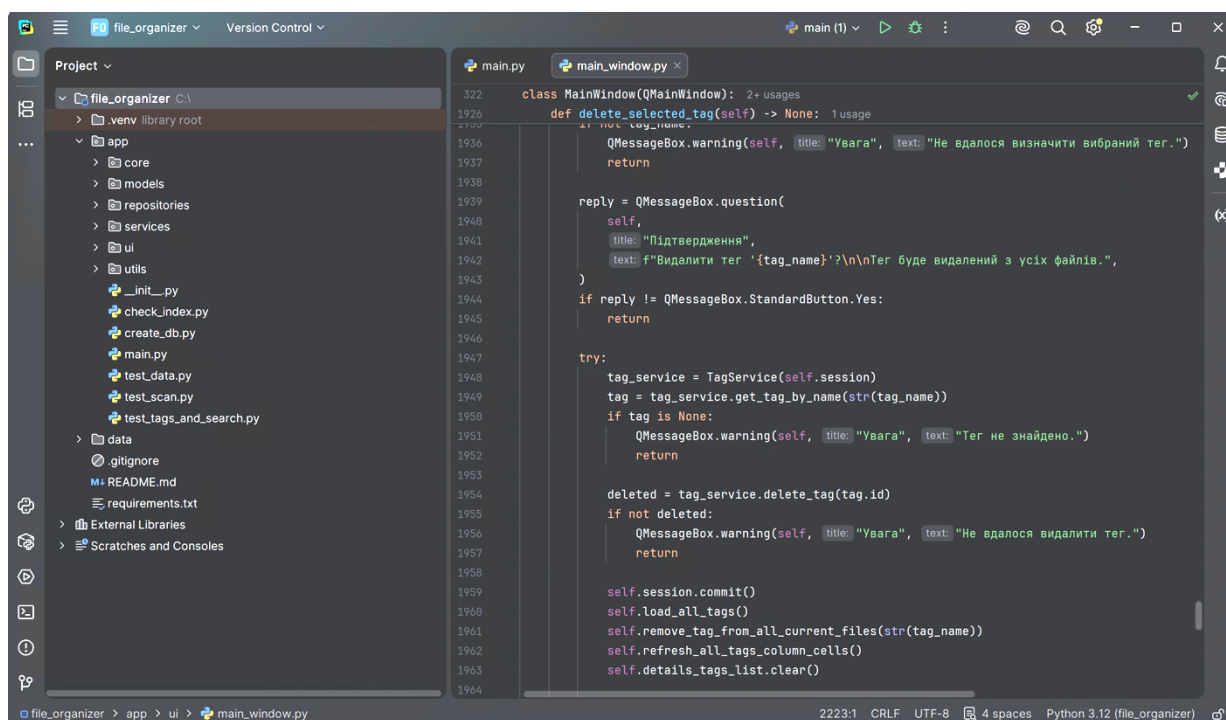


Рисунок 3.1 – Структура проєкту файлового органайзера

Головне вікно складається з трьох основних зон: лівої панелі каталогів, пошуку й тегів, центральної таблиці файлів та правої панелі деталей. Така побудова дає змогу одночасно бачити індексовані папки, результати фільтрації,

теги і властивості вибраного файлу. Головний інтерфейс системи показано на рисунку 3.2.

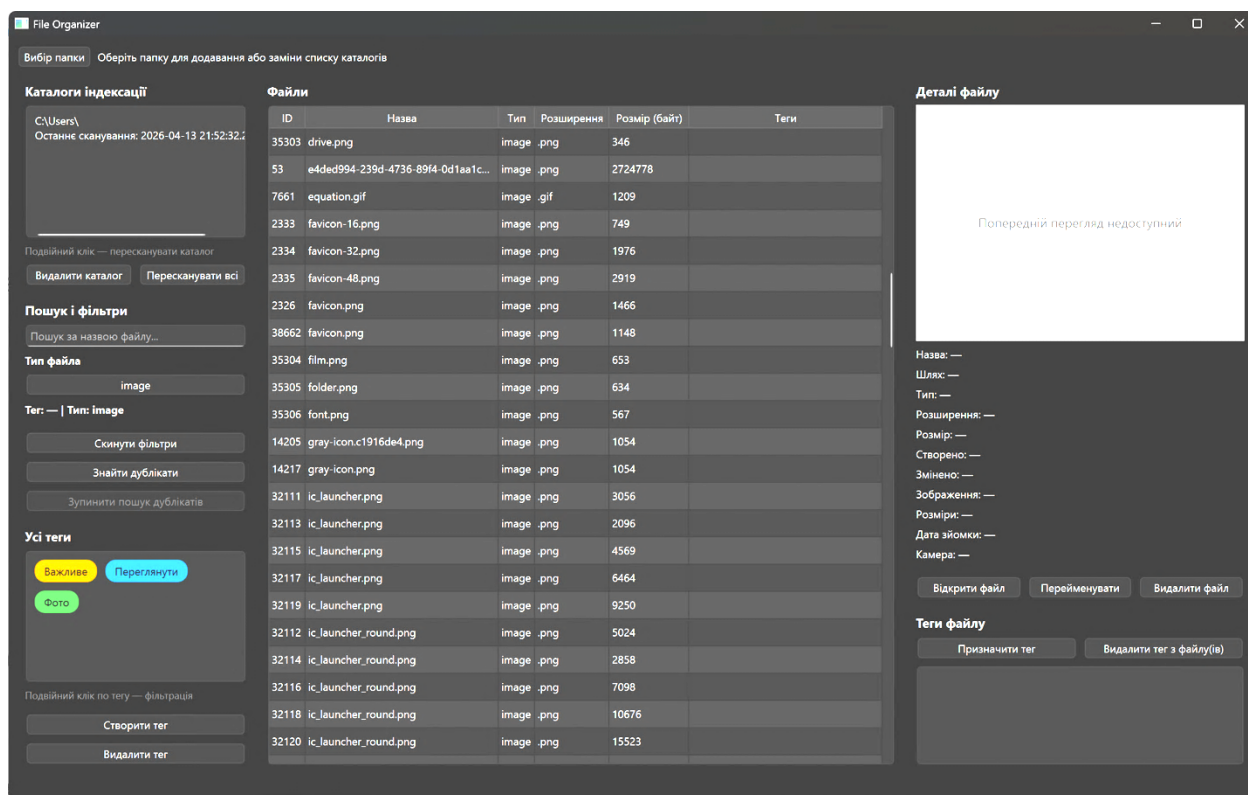


Рисунок 3.2 – Головне вікно системи організації файлів

Індексація папок реалізована в ScannerService. Сервіс перевіряє існування папки, створює або знаходить запис Folder, обходить усі файли через rglob, визначає тип файла за розширенням і створює або оновлює запис File. Ключовий фрагмент наведено у лістингу 3.1.

Лістинг 3.1 – Сканування папки та оновлення індексу файлів

```
def scan_folder(self, folder_path: str | Path) -> ScanResult:
    root_path = Path(folder_path).expanduser().resolve()
    result = ScanResult(folder_path=str(root_path))

    if not root_path.exists():
        raise FileNotFoundError(f"Папку не знайдено: {root_path}")
    if not root_path.is_dir():
        raise NotADirectoryError(f"Це не папка: {root_path}")

    folder = self._get_or_create_folder(root_path)
```

```

for file_path in self._iter_files(root_path):
    result.files_found += 1
    try:
        was_created = self._upsert_file(folder=folder,
file_path=file_path)
        if was_created:
            result.files_added += 1
        else:
            result.files_updated += 1
    except Exception:
        result.errors_count += 1
        result.files_skipped += 1

folder.last_scanned_at = datetime.now()
self.session.commit()
return result

```

Кінець лістингу 3.1

Тривалі операції сканування виконуються у фоновому потоці ScanWorker, щоб інтерфейс не зависав під час обходу великого каталогу. Результат сканування з кількістю знайдених, доданих і оновлених файлів показано на рисунку 3.3.

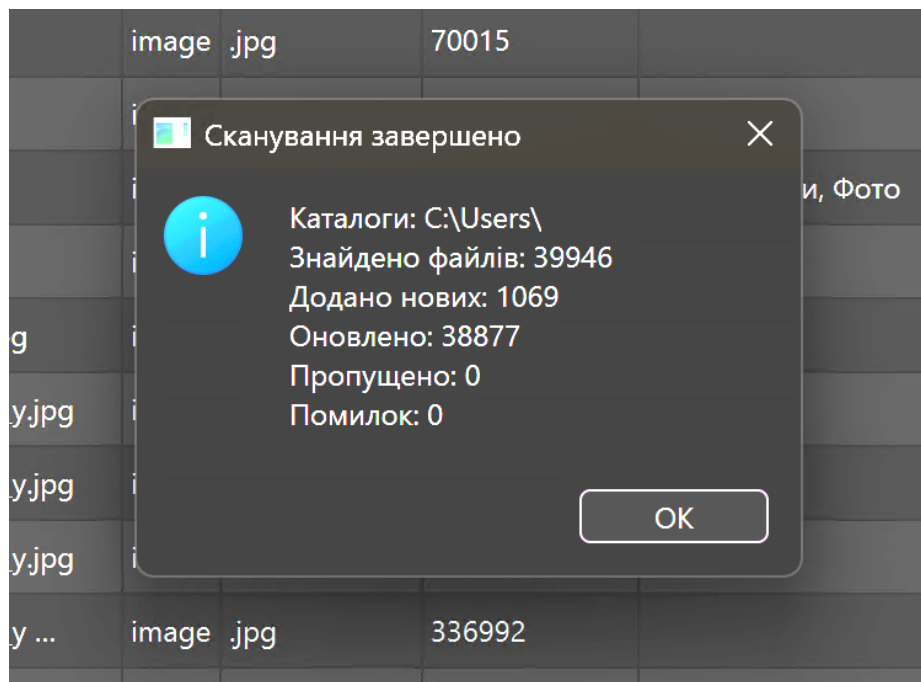


Рисунок 3.3 – Повідомлення про завершення сканування каталогу

Після вибору файлу у таблиці система показує його назву, шлях, тип, розширення, розмір, дати створення та змінення. Для зображень формується графічне прев'ю, а також відображаються розміри, дата зйомки та інформація про камеру, якщо такі дані доступні. Приклад перегляду фото наведено на рисунку 3.4.

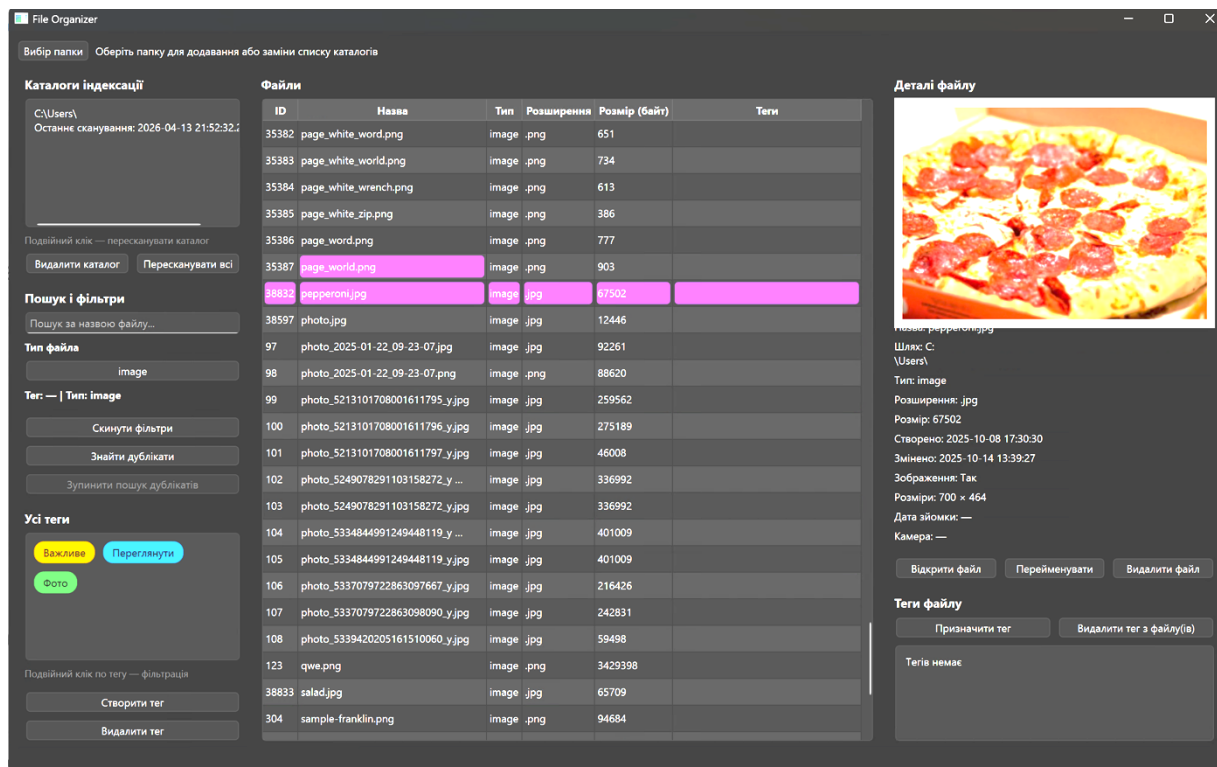


Рисунок 3.4 – Перегляд фото та метаданих вибраного файлу

Пошук за назвою, фільтрація зображень, пошук файлів без тегів і фільтрація за одним або кількома тегами реалізовані у SearchService. Фрагмент наведено у лістингу 3.2, а розширений фрагмент пошуку подано в додатку А.

Лістинг 3.2 – Пошук і фільтрація файлів за назвою та тегами

```
def search_files(self, filename_query=None, tag_names=None,
                  only_images=False, only_untagged=False) ->
list[File]:
    query = self.session.query(File)

    if filename_query:
```

```

        query =
query.filter(File.filename.ilike(f"%{filename_query.strip()}%"))

        if only_images:
            query = query.filter(File.is_image.is_(True))

        if only_untagged:
            query = query.filter(~File.tag_links.any())

        if tag_names:
            normalized_tags = [tag.strip() for tag in tag_names if
tag.strip()]
            for tag_name in normalized_tags:
                query = query.filter(File.tags.any(Tag.name ==
tag_name))

    return query.order_by(File.filename.asc()).all()

```

Кінець лістингу 3.2

Додатково в інтерфейсі реалізовано окреме заповнення таблиці файлів після завершення фонового завантаження. Метод блокує сигнали таблиці на час оновлення, очищує попередні рядки, створює елементи для назви, типу, розширення, розміру та тегів, а ідентифікатор файла зберігає як службові дані рядка. Такий підхід зменшує кількість зайвих подій вибору під час масового оновлення таблиці та забезпечує стабільне відображення результатів пошуку. Фрагмент коду наведено у лістингу 3.3.

Лістинг 3.3 – Заповнення таблиці файлів у головному вікні

```

def populate_files_table(self, files: list[File]) -> None:
    self.files_table.blockSignals(True)
    try:
        self.files_table.clearContents()
        self.files_table.setRowCount(0)
        self.files_table.setRowCount(len(files))

        for row, file in enumerate(files):
            id_item = QTableWidgetItem(str(file.id))
            name_item = QTableWidgetItem(file.filename or "")
            type_item = QTableWidgetItem(file.file_type or "")
            ext_item = QTableWidgetItem(file.extension or "")
            size_item = QTableWidgetItem(
                "" if file.size is None else str(file.size)
            )

```

```

        tags_item = QTableWidgetItem(getattr(file,
"tags_display", ""))
        tags_item.setToolTip(getattr(file, "tags_display",
""))

        id_item.setData(Qt.ItemDataRole.UserRole, file.id)
        self.files_table.setItem(row, 0, id_item)
        self.files_table.setItem(row, 1, name_item)
        self.files_table.setItem(row, 2, type_item)
        self.files_table.setItem(row, 3, ext_item)
        self.files_table.setItem(row, 4, size_item)
        self.files_table.setItem(row, 5, tags_item)

        self.files_table.clearSelection()
    finally:
        self.files_table.blockSignals(False)

```

Кінець лістингу 3.3

Окрему увагу приділено відображенню властивостей зображень. Після вибору файла система не лише показує базові атрибути з бази даних, а й намагається прочитати додаткові EXIF-метадані безпосередньо з файла. Якщо метадані відсутні або файл недоступний, інтерфейс повертає нейтральні значення, не перериваючи роботу користувача. Реалізацію цього механізму наведено у лістингу 3.4.

Лістинг 3.4 – Отримання розмірів зображення та EXIF-метаданих

```

def get_image_metadata(self, file: File) -> dict[str, str]:
    default_result = {
        "dimensions": "-",
        "date_taken": "-",
        "camera": "-",
    }

    if not file.is_image or not file.path:
        return default_result

    image_path = Path(file.path)
    if not image_path.exists():
        return default_result

    try:
        with Image.open(image_path) as img:
            width, height = img.size
            default_result["dimensions"] = f"{width} × {height}"

```

```

raw_exif = img.getexif()
if not raw_exif:
    return default_result

exif_data = {}
for tag_id, value in raw_exif.items():
    tag_name = ExifTags.TAGS.get(tag_id, str(tag_id))
    exif_data[tag_name] = value

date_taken = exif_data.get("DateTimeOriginal") or
exif_data.get("DateTime")
if date_taken:
    default_result["date_taken"] = str(date_taken)

make = exif_data.get("Make")
model = exif_data.get("Model")
camera_parts = [str(part).strip() for part in (make,
model) if part]
if camera_parts:
    default_result["camera"] = " ".join(camera_parts)
except Exception:
    return default_result

return default_result

```

Кінець лістингу 3.4

Для підвищення зручності роботи з різними типами файлів у правій панелі реалізовано попередній перегляд. Для фотографій створюється масштабоване прев'ю через QPіхтap, для текстових файлів формується короткий текстовий фрагмент із підбором кодування, а для інших форматів показується узагальнена позначка типу файла. Така логіка дозволяє швидко оцінити вміст без відкриття зовнішньої програми. Фрагмент наведено у лістингу 3.5.

Лістинг 3.5 – Формування попереднього перегляду файла

```

def show_file_preview(self, file: File) -> None:
    self.preview_label.clear()
    self.preview_label.show()
    self.preview_text.clear()
    self.preview_text.hide()

    if file.is_image:
        if not file.path or not Path(file.path).exists():

```

```

        self.preview_label.setText("Файл зображення не
знайдено")
        return

        pixmap = QPixmap(file.path)
        if pixmap.isNull():
            self.preview_label.setText("Не вдалося завантажити
зображення")
            return

        scaled = pixmap.scaled(
            self.preview_label.size(),
            Qt.AspectRatioMode.KeepAspectRatio,
            Qt.TransformationMode.SmoothTransformation,
        )
        self.preview_label.setPixmap(scaled)
        return

        normalized_extension = (file.extension or
        "").lower().lstrip(".")
        if normalized_extension in {"txt", "csv", "json", "md", "py",
        "html", "css"}:
            raw_bytes = Path(file.path).read_bytes()[:12000]
            for encoding in ("utf-8", "utf-8-sig", "cp1251", "latin-
1"):
                try:
                    preview_text =
raw_bytes.decode(encoding).replace("\x00", "").strip()
                    break
                except Exception:
                    preview_text = None

            if preview_text:
                self.preview_label.hide()
                self.preview_text.show()
                self.preview_text.setPlainText(preview_text)
                return

        extension_text = normalized_extension.upper() if
normalized_extension else "FILE"
        self.preview_label.setText(
            f"<div style='text-align:center; font-
size:28px;'>{extension_text}</div>"
        )

```

Кінець лістингу 3.5

Отже, практична реалізація підрозділу 3.1 охоплює не лише сканування і пошук, а й повний цикл роботи користувача з індексованими файлами: асинхронне завантаження списку, табличне представлення результатів, перегляд

властивостей, читання метаданих зображень і попередній перегляд вмісту. Це робить застосунок придатним для щоденної організації великих локальних каталогів, у яких користувачеві важливо швидко знаходити, переглядати та класифікувати файли без переходу між багатьма окремими програмами.

3.2 Розробка бази даних

База даних системи реалізована у SQLite через SQLAlchemy ORM. Підключення описано в модулі `database.py`: створюється `engine`, фабрика `SessionLocal` і базовий клас `Base` для декларативних моделей. Такий підхід дозволяє сервісам працювати з таблицями як з Python-класами, а не формувати SQL запити вручну.

Схему бази даних системи організації файлів і фото наведено на рисунку 3.5.

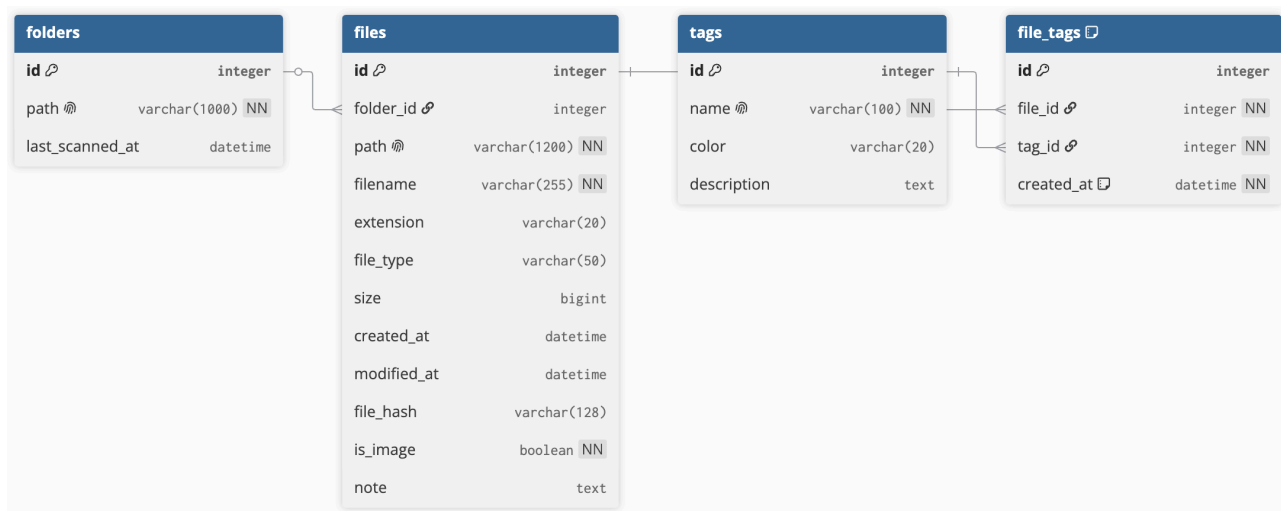


Рисунок 3.5 – Схема бази даних системи організації файлів і фото

Таблиця `folders` зберігає шлях до просканованої папки та дату останнього сканування. Таблиця `files` містить шлях до файла, назву, розширення, тип, розмір, дати створення й змінення, SHA-хеш і ознаку зображення. Таблиця `tags` містить назву, колір і опис тегу. Таблиця `file_tags` реалізує зв'язок багато до

багатьох між файлами й тегами та має унікальне обмеження, яке запобігає повторному призначенню одного тегу тому самому файлу.

Фрагмент моделі File наведено у лістингу 3.6.

Лістинг 3.6 – Опис моделі файла засобами SQLAlchemy

```
class File(Base):
    __tablename__ = "files"

    id: Mapped[int] = mapped_column(Integer, primary_key=True,
autoincrement=True)

    folder_id: Mapped[int | None] = mapped_column(
        ForeignKey("folders.id", ondelete="SET NULL"),
        nullable=True, index=True)

    path: Mapped[str] = mapped_column(String(1200), unique=True,
        nullable=False, index=True)

    filename: Mapped[str] = mapped_column(String(255),
        nullable=False, index=True)

    extension: Mapped[str | None] = mapped_column(String(20),
        nullable=True, index=True)

    file_type: Mapped[str | None] = mapped_column(String(50),
        nullable=True, index=True)

    size: Mapped[int | None] = mapped_column(BigInteger,
        nullable=True)

    file_hash: Mapped[str | None] = mapped_column(String(128),
        nullable=True, index=True)

    is_image: Mapped[bool] = mapped_column(Boolean, default=False,
        nullable=False)
    folder: Mapped["Folder | None"] = relationship("Folder",
        back_populates="files")

    tag_links: Mapped[list["FileTag"]] = relationship(
        "FileTag", back_populates="file", cascade="all, delete-
        orphan")
```

Кінець лістингу 3.6

Реальний стан таблиці tags у SQLite після створення декількох тегів показано на рисунку 3.6.

id	name	color
1	Важливе	#FF9800
2	Переглянути	#2196F3
3	Фото	#4CAF50
4	Новий тег	#F44336

Рисунок 3.6 – Таблиця тегів у базі даних SQLite

Тегова модель винесена в окремий сервіс TagService. Він створює нові теги, повертає наявні, призначає тег файлу, перевіряє існування зв'язку та видаляє тег із файлу. Саме через цей сервіс інтерфейс працює з класифікацією, не звертаючись напряму до таблиці file_tags.

3.3 Захист інформаційно-комп'ютерної системи

Захист інформаційно-комп'ютерної системи в локальному файловому органайзері насамперед пов'язаний із цілісністю індексу, обережною роботою з файловою системою та виконанням довгих операцій у фонових потоках. Система не передає файли на зовнішній сервер і не потребує облікових записів, тому основний ризик полягає у випадковому пошкодженні локальних файлів або блокуванні інтерфейсу під час оброблення великої кількості даних.

Для операцій з тегами використовується перевірка наявності файлу, тегу та вже існуючого зв'язку. Це запобігає дублюванню записів у проміжній таблиці і зберігає цілісність тегової моделі. Фрагмент призначення тегу наведено у лістингу 3.7.

Лістинг 3.7 – Призначення тегу файлу з перевіркою дублювання зв'язку

```
def assign_tag_to_file(self, file_id: int, tag_id: int) ->
FileTag:
```

```

file = self.session.query(File).filter(File.id ==
file_id).first()
if file is None:
    raise ValueError(f"Файл з id={file_id} не знайдено.")

tag = self.session.query(Tag).filter(Tag.id == tag_id).first()
if tag is None:
    raise ValueError(f"Тег з id={tag_id} не знайдено.")

existing_link = (
    self.session.query(FileTag)
    .filter(FileTag.file_id == file_id, FileTag.tag_id ==
tag_id)
    .first()
)
if existing_link:
    return existing_link

link = FileTag(file_id=file_id, tag_id=tag_id)
self.session.add(link)
self.session.flush()
return link

```

Кінець лістингу 3.7

Результат призначення тегу вибраному файлу показано на рисунку 3.7.

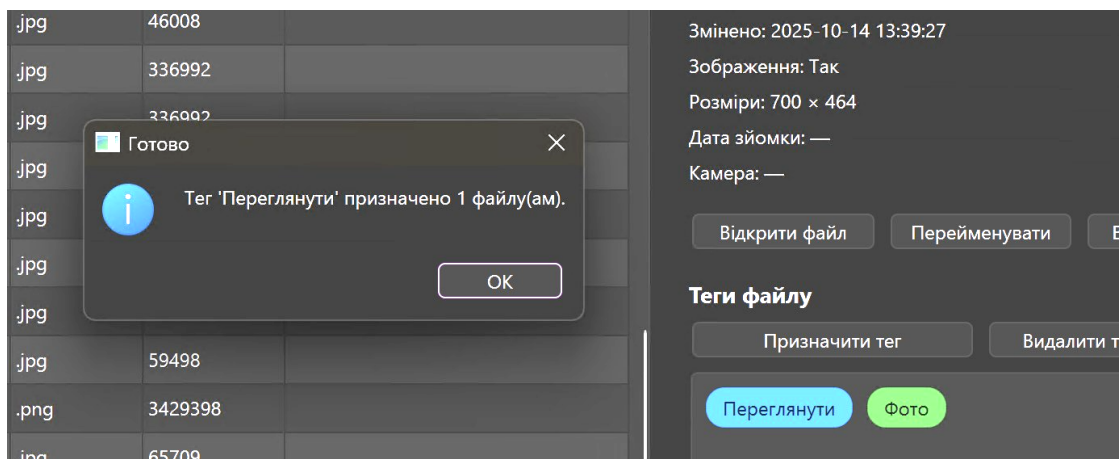


Рисунок 3.7 – Повідомлення про призначення тегу файлу

Потенційно небезпечні дії з файлами, зокрема видалення або перейменування, виконуються тільки після вибору конкретного файлу в таблиці та підтвердження користувацької дії. При повторному скануванні система не створює дублікати записів, а оновлює вже відомі файли за унікальним шляхом.

Фонове виконання реалізовано для сканування каталогів, завантаження списку файлів і пошуку дублікатів. Це захищає інтерфейс від зависання та дозволяє користувачу зупинити тривалий пошук однакових файлів. У разі помилки потік надсилає сигнал error, а сесія бази даних виконує rollback.

3.4 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування та налагодження системи виконувалося за сценаріями, що відповідають основним функціям файлового органайзера: вибір папки, додавання або заміна списку каталогів, повторне сканування, пошук за назвою, фільтрація за типом файла, створення тегів, призначення тегів, перегляд деталей файла, відкриття, перейменування, видалення та пошук дублікатів.

Сценарій створення нового тегу перевірявся через діалог введення назви. Після підтвердження тег додавався до бази даних і відображався у списку всіх тегів. Діалог створення тегу показано на рисунку 3.8, а результат додавання – на рисунку 3.9.

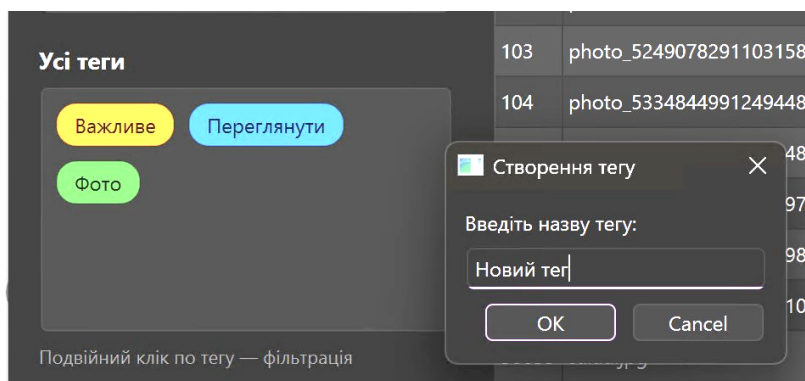


Рисунок 3.8 – Діалог створення нового тегу



Рисунок 3.9 – Відображення створеного тегу в інтерфейсі

Окремо тестувався пошук дублікатів. Алгоритм спочатку групує файли за розміром, щоб не обчислювати хеш для очевидно різних файлів. Для груп з однаковим розміром обчислюється SHA256 за вмістом, після чого файли з однаковим хешем отримують тег Дублікат. Фрагмент алгоритму наведено у лістингу 3.8, а розширений фрагмент подано в додатку Б.

Лістинг 3.8 – Пошук дублікатів за розміром і SHA256-хешем

```
def _calculate_file_hash(self, file_path: Path) -> str:
    hasher = hashlib.sha256()
    with file_path.open("rb") as f:
        while True:
            self._check_cancel()
            chunk = f.read(1024 * 1024)
            if not chunk:
                break
            hasher.update(chunk)
    return hasher.hexdigest()

# Спочатку групуємо файли за розміром, потім перевіряємо хеші.
size_groups: dict[int, list[File]] = {}
for file in files:
    if file.path and file.size and Path(file.path).exists():
        size_groups.setdefault(int(file.size), []).append(file)

candidate_groups = [group for group in size_groups.values() if
len(group) > 1]
```

Кінець лістингу 3.8

Під час тестування перевірялося, що система не блокує інтерфейс під час сканування та пошуку дублікатів, коректно показує прогрес і дозволяє скасувати

тривалу операцію. Також перевірялися ситуації з відсутнім файлом, недоступною папкою, повторним створенням тегу і повторним призначенням одного тегу одному файлу.

Результати тестування підтвердили працездатність основних сценаріїв: система індексує каталоги, зберігає метадані у SQLite, показує прев'ю зображень, дозволяє створювати й призначати теги, фільтрує файли та позначає дублікати за вмістом. Отримана реалізація є придатною для впорядкування локальних колекцій документів і фотографій.

ВИСНОВКИ

У кваліфікаційній роботі розроблено систему організації файлів і фото з використанням Python, PyQt6, SQLite, SQLAlchemy та Pillow. Застосунок забезпечує локальне індексування каталогів, збереження метаданих про файли, роботу з тегами, пошук, фільтрацію, перегляд деталей і виявлення дублікатів.

Здійснено аналіз процесів локального зберігання файлів і фото. Визначено, що основними проблемами користувача є розпорошення матеріалів між папками, складність ручного пошуку, відсутність гнучкого групування за змістом, накопичення дублікатів і потреба швидко бачити властивості вибраного файла.

Сформовано вимоги до настільної системи організації файлів. Передбачено можливість додавання каталогів індексації, пересканування папок, пошуку за назвою, фільтрації за типом файлу, роботи з тегами, перегляду метаданих, відкриття, перейменування та видалення вибраних матеріалів.

Спроектовано структуру бази даних для зберігання каталогів, файлів, тегів і зв'язків між ними. Таблиці folders, files, tags і file_tags забезпечують збереження шляхів, назв, розширень, типів, розмірів, дат створення й змінення, ознаки зображення та багатозначних зв'язків між файлами й тегами.

Обґрунтовано вибір технологій реалізації. Python використано як основну мову розробки, PyQt6 – для створення графічного інтерфейсу, SQLite – для локального зберігання даних, SQLAlchemy – для роботи з моделями та запитами, Pillow – для отримання характеристик фото.

Реалізовано модулі сканування каталогів, пошуку, тегування, перегляду деталей файлів, роботи з фото та пошуку дублікатів. Окремо реалізовано фонове виконання тривалих операцій сканування й перевірки однакових файлів, що покращує зручність роботи з інтерфейсом.

Перевірено працездатність системи на типових сценаріях використання: додаванні папок, повторному скануванні, пошуку за частиною назви, фільтрації за типом, створенні та призначенні тегів, відкритті файлів, перегляді

характеристик фото й позначенні дублікатів. Результати підтверджують придатність системи для впорядкування локальних файлових колекцій.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Introduction. PyQt. Riverbank Computing. URL: <https://riverbankcomputing.com/software/pyqt/intro> (дата звернення: 02.03.2026).
2. SQLAlchemy Unified Tutorial. SQLAlchemy Documentation. URL: <https://docs.sqlalchemy.org/en/20/tutorial/index.html> (дата звернення: 02.03.2026).
3. Image module. Pillow Documentation. URL: <https://pillow.readthedocs.io/en/stable/reference/Image.html> (дата звернення: 06.03.2026).
4. TagSpaces Documentation. TagSpaces. URL: <https://docs.tagspaces.org/> (дата звернення: 12.03.2026).
5. File: TagSpaces 2.5.0 (screenshot). Wikimedia Commons. URL: [https://commons.wikimedia.org/wiki/File:TagSpaces_2.5.0_\(screenshot\).png](https://commons.wikimedia.org/wiki/File:TagSpaces_2.5.0_(screenshot).png) (дата звернення: 24.03.2026).
6. Overview. digiKam. URL: <https://www.digikam.org/about/overview/> (дата звернення: 11.04.2026).
7. Adobe Bridge. Adobe. URL: <https://www.adobe.com/products/bridge.html> (дата звернення: 11.04.2026).
8. About SQLite. SQLite. URL: <https://www.sqlite.org/about.html> (дата звернення: 25.05.2026).
9. QThread Class. Qt 6 Documentation. URL: <https://doc.qt.io/qt-6/qthread.html> (дата звернення: 17.04.2026).
10. PySide6.QtCore.QThread. Qt for Python Documentation. URL: <https://doc.qt.io/qtforpython-6/PySide6/QtCore/QThread.html> (дата звернення: 26.04.2026).
11. hashlib – Secure hashes and message digests. Python Documentation. URL: <https://docs.python.org/3/library/hashlib.html> (дата звернення: 14.05.2026).
12. FIPS PUB 180-4. Secure Hash Standard (SHS). National Institute of Standards and Technology. URL: <https://doi.org/10.6028/NIST.FIPS.180-4> (дата звернення: 16.05.2026).

ДОДАТКИ

ДОДАТОК А

ФРАГМЕНТ ЗАВАНТАЖЕННЯ ФАЙЛІВ У ФОНОВОМУ ПОТОЦІ

```
class LoadFilesWorker(QObject):
    finished = pyqtSignal(object, object, object)
    error = pyqtSignal(str)

    def run(self) -> None:
        session = self.session_factory()
        try:
            search_service = SearchService(session)
            tag_service = TagService(session)
            files = search_service.search_files(
                filename_query=self.filename_query,
                tag_names=self.tag_names,
                only_images=False,
                only_untagged=False,
            )
            if self.file_type_filter:
                files = [file for file in files if (file.file_type
or '') == self.file_type_filter]
            tags = tag_service.get_all_tags()
            folders =
session.query(Folder).order_by(Folder.path.asc()).all()
            self.finished.emit(files, tags, folders)
        except Exception as ex:
            session.rollback()
            self.error.emit(str(ex))
        finally:
            session.close()
```

ДОДАТОК Б

ФРАГМЕНТ ПОЗНАЧЕННЯ ЗНАЙДЕНИХ ДУБЛІКАТІВ ТЕГОМ

```
if duplicate_file_ids:
    duplicate_tag = session.query(Tag).filter(Tag.name ==
'Dублікат').first()
    if duplicate_tag is None:
        duplicate_tag = Tag(name='Дублікат', color='#FF9800')
        session.add(duplicate_tag)
        session.flush()

    existing_links = {
        file_id
        for (file_id,) in session.query(FileTag.file_id)
        .filter(
            FileTag.tag_id == duplicate_tag.id,
            FileTag.file_id.in_(duplicate_file_ids),
        )
        .all()
    }

    new_links = [
        FileTag(file_id=file_id, tag_id=duplicate_tag.id)
        for file_id in duplicate_file_ids
        if file_id not in existing_links
    ]
    if new_links:
        session.add_all(new_links)
    session.commit()
```