ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка Self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

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

_____ Дмитро МИКИТЮК
(підпис)

Виконав: здобувач вищої освіти групи ТЦР-44

Дмитро МИКИТЮК

Керівник: Олексій АНАНЧЕНКО
канд. техн. наук, доц.

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Микитюку Дмитру Сергійовичу

1. Тема кваліфікаційної роботи: «Розробка Self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту» керівник кваліфікаційної роботи канд. техн. наук, доц. Олексій АНАНЧЕНКО затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «18» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про методи організації, індексації та пошуку великих масивів медіа даних, опис технологій штучного інтелекту для формування метаданих зображень, відео та аудіо, технічна документація з мови програмування Go, бібліотек React, TypeScript, SQLite, FTS5, ffmpeg, ffprobe, а також опис архітектури self-hosted програмних систем.

4. Зміст розрахунково-пояснювальної записки:

1. Огляд та аналіз існуючих методів і технологій організації, індексації та пошуку великих масивів медіа даних із використанням штучного інтелекту.
2. Проєктування self-hosted рішення для локального зберігання, індексації, пошуку та перегляду медіа файлів з AI-метаданими.
3. Програмна реалізація та опис функціонування системи megadrive для організації, парсингу та пошуку медіа даних.
4. Впровадження механізмів автентифікації, API-ключів, налаштувань, тем оформлення та роботи з локальними сховищами.

5. Перелік графічного матеріалу:
1. Логічна структура системи.
2. Діаграма варіантів використання системи megadrive.
3. Загальна архітектура self-hosted рішення.
4. Алгоритм індексації та обробки медіа файлів.
5. Структура бази даних системи.
6. Структура серверної частини системи.
7. Структура клієнтської частини системи.
8. Діаграма розгортання системи.
9. Діаграма класів серверних модулів.
10. Екранні форми web-застосунку.
11. Апробація результатів дослідження.

6. Дата видачі завдання «20» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	20.02.2026–28.02.2026	Виконано

2	Аналіз та дослідження існуючих аналогів	01.03.2026–12.03.2026	Виконано
3	Огляд методів організації, індексації та пошуку медіа даних	13.03.2026–24.03.2026	Виконано
4	Проєктування self-hosted системи megadrive	25.03.2026–08.04.2026	Виконано
5	Програмна реалізація системи	09.04.2026–25.04.2026	Виконано
6	Реалізація індексації, AI-метаданих, пошуку та фільтрації	26.04.2026–05.05.2026	Виконано
7	Оформлення роботи: вступ, висновки, реферат	06.05.2026–12.05.2026	Виконано
8	Розробка демонстраційних матеріалів	13.05.2026–17.05.2026	Виконано
9	Попередній захист роботи	18.05.2026–22.05.2026	Виконано

Здобувач вищої освіти _____ Дмитро МИКИТЮК
(підпис)

Керівник
кваліфікаційної роботи _____ Олексій АНАНЧЕНКО
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 62 стор., 2 табл., 21 рис., 25 джерел.

Мета роботи – спрощення організації, індексації, парсингу та пошуку великого масиву медіа даних на основі технологій штучного інтелекту.

Об'єкт дослідження – процеси організації, індексації та пошуку великих масивів медіа даних.

Предмет дослідження – методи та програмні засоби розробки self-hosted системи для організації медіа даних із використанням технологій штучного інтелекту.

У роботі проаналізовано предметну галузь організації медіа даних та визначено основні проблеми, пов'язані зі зберіганням великих локальних архівів, складністю пошуку за змістом файлів, ручним заповненням метаданих і залежністю багатьох рішень від хмарних сервісів. Встановлено, що self-hosted підхід дозволяє зберігати контроль над даними користувача та працювати з медіа архівом у локальному або власному серверному середовищі.

Проаналізовано існуючі програмні засоби для зберігання, перегляду та керування медіа файлами: Google Photos, Immich і PhotoPrism. Визначено їх переваги, недоліки та функціональні обмеження. Особливу увагу приділено підтримці локального розгортання, роботі з медіа файлами, AI-метаданими, пошуком, фільтрацією, OCR-текстом, транскрипціями та можливістю керування власними сховищами.

Додатково обґрунтовано доцільність використання технологій штучного інтелекту для автоматичного формування метаданих медіа файлів. Визначено, що саме AI-метадані дозволяють розширити можливості звичайного файлового пошуку, оскільки система може працювати не лише з назвою файлу, а й з його змістом: описом зображення, розпізнаним текстом, ключовими словами та

транскрипціями аудіо чи відео. Це робить роботу з великими медіа архівами більш зручною та зменшує потребу в ручному сортуванні файлів.

Розроблено структуру self-hosted системи megadrive та визначено її ключові функціональні можливості: додавання локальних сховищ, сканування та індексацію зображень, відео й аудіо, створення мініатюр, формування описів, ключових слів, OCR-тексту та транскрипцій, повнотекстовий пошук, фільтрацію, перегляд галереї, відкриття деталей медіа файлу, керування налаштуваннями, темами оформлення, автентифікацією та API-ключами.

У роботі використано Go для реалізації серверної частини, go-chi/chi для маршрутизації HTTP-запитів, SQLite і FTS5 для зберігання даних та повнотекстового пошуку, React, TypeScript, Vite і TailwindCSS для створення клієнтського інтерфейсу, ffmpeg і ffprobe для обробки відео та аудіо, а також bcrypt, JWT і SHA-256 для захисту доступу та API-ключів.

Сферою використання системи є організація великих локальних медіа архівів, персональні та професійні колекції зображень, відео й аудіо, внутрішні архіви організацій, а також середовища, де потрібен локальний контроль над файлами, автоматизоване формування метаданих і зручний пошук за змістом медіа даних.

КЛЮЧОВІ СЛОВА: SELF-HOSTED, MEGADRIVE, МЕДІА ДАНІ, МЕДІА АРХІВ, ІНДЕКСАЦІЯ, AI-МЕТАДАНІ, OCR, ТРАНСКРИПЦІЯ, ПОВНОТЕКСТОВИЙ ПОШУК, SQLITE, FTS5, GO, REACT, TYPESCRIPT, REST API.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AI – Artificial Intelligence, штучний інтелект.

API – Application Programming Interface, програмний інтерфейс для взаємодії між компонентами системи.

CRUD – Create, Read, Update, Delete, базові операції створення, читання, оновлення та видалення даних.

DB – Database, база даних.

ENV – Environment Variables, змінні середовища для налаштування роботи системи.

FFmpeg – програмний засіб для обробки відео- та аудіофайлів.

ffprobe – інструмент зі складу FFmpeg для отримання технічної інформації про медіа файл.

FTS5 – Full-Text Search 5, модуль SQLite для повнотекстового пошуку.

Go – мова програмування, використана для реалізації серверної частини системи.

React – JavaScript-бібліотека для створення клієнтського інтерфейсу.

REST API – архітектурний підхід до побудови API на основі HTTP-запитів.

SHA-256 – криптографічна хеш-функція, використана для зберігання хешів API-ключів.

SPA – Single Page Application, односторінковий веб-застосунок.

SQLite – локальна реляційна база даних, використана для зберігання даних системи.

SSE – Server-Sent Events, механізм передавання подій від сервера до клієнта.

TOML – формат конфігураційних файлів.

UI – User Interface, інтерфейс користувача.

UX – User Experience, користувацький досвід.

Vite – інструмент для збірки клієнтської частини веб-застосунку.

WAL – Write-Ahead Logging, режим журналювання SQLite для підвищення надійності роботи бази даних.

ВСТУП

Актуальність теми**.** У сучасних умовах обсяг цифрових медіа даних постійно зростає. Користувачі, фахівці, організації та творчі команди накопичують великі колекції зображень, відео, аудіофайлів, скріншотів, графічних матеріалів та іншого мультимедійного контенту. Такі дані можуть зберігатися на локальних дисках, зовнішніх носіях, серверах або в хмарних сервісах, однак зі збільшенням їх кількості виникає проблема зручної організації, швидкого пошуку, індексації та опису файлів.

Розробка системи megadrive спрямована на вирішення цих проблем. Проєкт являє собою локальний медіа-індексатор і галерею з AI-метаданими, що дозволяє додавати локальні сховища, сканувати медіа файли, створювати мініатюри, формувати метадані, виконувати пошук за описом, тегами, назвами файлів, OCR-текстом і транскрипціями, а також переглядати результати у зручному веб-інтерфейсі.

Мета роботи – спрощення організації, індексації, парсингу та пошуку великого масиву медіа даних на основі технологій штучного інтелекту.

Для досягнення поставленої мети у кваліфікаційній роботі необхідно виконати такі завдання:

1. Дослідити особливості організації, зберігання та пошуку великих масивів медіа даних.
2. Проаналізувати існуючі аналоги програмного забезпечення для керування медіа колекціями.
3. Визначити функціональні та нефункціональні вимоги до self-hosted системи.
4. Обґрунтувати вибір технологій для реалізації серверної, клієнтської частини та бази даних.
5. Спроєктувати архітектуру програмного забезпечення, структуру бази даних, арі та інтерфейс користувача.
6. Реалізувати механізми індексації медіа файлів, формування аі-метаданих, пошуку та фільтрації.

7. Реалізувати веб-інтерфейс для перегляду галереї, керування сховищами, налаштуваннями, темами оформлення та арі-ключами.

Об’єкт дослідження – процеси організації, індексації та пошуку великих масивів медіа даних.

Предмет дослідження – методи та програмні засоби розробки self-hosted системи для організації медіа даних із використанням технологій штучного інтелекту.

Методи дослідження**.** Під час виконання кваліфікаційної роботи використано методи теоретичного аналізу предметної галузі, порівняльного аналізу існуючих програмних рішень, методи проєктування архітектури програмного забезпечення, моделювання структури бази даних, методи повнотекстового пошуку, індексації медіа файлів, а також методи програмної реалізації веб-застосунків.

Наукова новизна одержаних результатів полягає у поєднанні self-hosted підходу до зберігання та обробки медіа даних із механізмами автоматичного формування AI-метаданих, повнотекстового пошуку та локального керування медіа архівом без обов’язкової залежності від хмарних сервісів.

Практична значущість одержаних результатів полягає в тому, що розроблене програмне забезпечення може бути використане для організації великих локальних медіа колекцій, швидкого пошуку файлів, перегляду зображень, відео та аудіо, формування описів і ключових слів.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Микитюк Д.С., Ананченко О.Є. Розробка self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту. Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація», 1-3 травня 2026 року, Київ, ДУІКТ. Збірник тез. *(подано до друку).*

2. Микитюк Д.С., Ананченко О.Є. Розробка self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту. Матеріали VI Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15 травня 2026 року, Київ, ДУІКТ. Збірник тез. *(подано до друку).*

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Характеристика предметної галузі організації медіа даних

Сучасне цифрове середовище відображається зростанням обсягів мультимедійної інформації, що збирають люди, компанії, творчесі команди та спеціалісти з різних галузей. Усі дані відносини можна використовувати для архівації, виконання роботи, створення вмісту, навчання або досліджень. Існує висока потреба в програмних продуктах для зберігання, організації, пошуку та перегляду такої інформації [1; 2].

Організація мультимедійних файлів – галузь, яка включає в себе все про процеси зберігання, індексації, пошуку та перегляду файлів. Зокрема, на відміну від текстових файлів, основна інформація міститься не в назві, а саме в мультимедійному файлі. Використання лише назв файлів для пошуку обмежує можливості користувача. Щоб вирішити цей випадок, у мультимедійних файлах вказуються додаткові метадані, які містять опис, ключові слова, визнані тексти та інші дані [3; 4].

Однією із найважливіших рішень у процесі організації мультимедійних файлів є процес індексації. У процесі індексації встановлюється структура представлення текстової інформації про файли, яка використовується для швидкого пошуку і перегляду. Для текстових описів, ключових слів, тексти OCR, транскрипції та назв встановлюється можливість виконання повнотекстового пошуку. Серед можливостей відокремлений SQLite FTS5, який включає можливості встановлення повнотекстового пошуку в великих масивах текстової інформації та використовується у локальних програмах [5].

Однією з технологій, яка використовується для поліпшення процесу організації мультимедійних файлів, є штучний інтелект. Штучний інтелект автоматично створює метадані про мультимедійні файли, описуючи зображення, визначаючи об’єкти, визначаючи ключові слова, розпізнаючи тексти на

зображеннях або аналізу думки з аудіо та відеофайлів. Користувач зможе знайти файли за їх змістом. У відомих зараз системах, створених для ведення архівів зображення та відеофайлів, відокремлені самостійні саме-hosted рішення Immich і PhotoPrism для організації та перегляду мультимедійних файлів [1; 2; 6].

Self-hosted рішення представляють собою особисті дані, які можуть зберігатись на локальний комп'ютер або сервер власного володіння. Хмарні сервери надають широкі можливості для синхронізації, пошуку аудіо/відеофайлів та аналізу їх вмісту. В той час як самостійні сервери зберігаються на локальних комп'ютерах користувача [1; 2].

Однією із актуальних задач стає розробка самостійних рішень для організації масивів мультимедійних файлів, що використовують технології штучного інтелекту. Розробка самостійної системи megadrive, яка спроектована для організації і парсингу великих масивів мультимедійних файлів на основі технологій штучного інтелекту. System здійснює індексацію мультимедійних файлів, створення мініатюр від встановленого вмісту, створення AI-метаданих, встановлення можливості виконання повнотекстового пошуку, роботу зі сховищами та перегляд веб-інтерфейсом. Такий самостійний підхід системи дозволяє зберігати дані на локальних комп'ютерах та автоматично виконувати аналіз мультимедійних файлів.

1.2 Аналіз існуючих аналогів програмного забезпечення

Відомо, що на ринку присутня велика кількість рішень, що призначена для зберігання, перегляду, організації та пошуку медіа даних. Ці системи можуть бути реалізовані у вигляді хмарних сервісів, локальних або мобільних програм, а також в self-hosted реалізаціях. Вони забезпечують спрощення роботи з великою кількістю медіа файлів.

Одні з основних функцій таких систем включають автоматичне резервне копіювання, створення альбомів, сортування файла за датою або місцем зйомки, а також пошук медіа файлів за об'єктами, людьми, локаціями, тегами та іншими

ознаками. Вони використовують алгоритми штучного інтелекту для автоматичного визначення вмісту фотографії, автоматичного сортування медіа файлів, автоматичного створення тегів та підказок до пошуку [1; 2; 3].

Відомими системами є Google Photos, Apple Photos, Immich та PhotoPrism. Google Photos – це хмарний сервіс для зберігання, редагування, організації та пошуку фотографій та відео, а також відокремлене використання можливостей Google AI для виконання цих функцій [1]. Apple Photos та iCloud Photos забезпечують перегляд, організацію, пошук та синхронізацію фотоархівів між пристроями від Apple [4; 5]. Immich – це рішення self-hosted типу, що дозволяє створити свій власний сервер для зберігання та резервного копіювання медіа файлів [2]. PhotoPrism – одне з рішень self-hosted типу, але з використанням технологій штучного інтелекту в його створенні та роботі [3].

Важливим етапом створення програмного забезпечення є його вибір за аналогіями, які вже існують. Хмарні сервіси забезпечують спрощений інтерфейс, інтеграцію з мобільними пристроями, синхронізацію медіа файлів між пристроями, а також величезні можливості для пошуку файлів. Однак вони потребують залежності від хмарного сервісу, мають обмежене сховище, певні тарифні умови та вимоги до передавання файлів на сторонні сервери.

Відновлення медіа файлів у self-hosted системах забезпечує користувачам більше контролю над цими файлами. Вони можуть працювати непосередно на власному сервері або на власному комп’ютері, що дозволяє їм працювати в локальному режимі. Однак вони потребують більшої відлагодженості при налаштуванні програмного забезпечення, документації в області технічного напряму та адміністрування сервісу [2; 3].

До найбільш відомих систем, які відповідають вимогам і рішенням, які вони містять, відносяться ті, які здійснюють організацію медіа файлів, роботу з пошуком файлів за об’єктами та іншими параметрами, використовуючи технології штучного інтелекту та забезпечуючи локальний контроль над медіа файлами. Вони є важливими характеристиками для створення системи megadrive.

Відмінність від існуючих сервісів вказує на те, що система розробляється як з можливістю організації медіа файлів у локальних сховищах, автоматичного створення метаданих на основі штучного інтелекту, а також з можливістю доступу до системи через веб-інтерфейс.

В результаті аналізу існуючих аналогів встановлено основні вимоги до майбутнього програмного забезпечення. Вони включають можливості зберігання великої кількості медіа файлів, перегляду файлів, пошуку за об’єктами, створення метаданих на основі штучного інтелекту, веб-інтерфейс та можливість розгортання в self-hosted режимі.

1.2.1 Google Photos

Google Photos є одним із найпоширеніших хмарних сервісів, спрямованих на зберігання та організацію фотографій та відео. Кількість користувачів цього сервісу є вражаючо високою завдяки величезному колу функцій, доступних через веб-сайт або мобільні пристрої. Сервіс Google Photos надає можливості автоматичного резервного копіювання файлів, перегляду фото- і відеотреків, створення альбомів, редагування зображень, пошуку та впорядкування зберіганих медіаматеріалів [1; 2].

Однією з особливостей Google Photos є використання хмарної інфраструктури компанії Google. Користувачам потрібно завантажувати фотографії та відео до свого облікового запису компанії, а далі вони стануть доступними на різних пристроях. Кожному обліковому запису у компанії належить 15 ГБ сховища, яке може бути використане для Google Photos, Gmail або Google Drive. Якщо користувач хоче отримати більше свободного сховища, йому потрібні платні тарифні плани Google One [1].

Google Photos може автоматично створювати резервні копії фотографій, що зберігаються на мобільних пристроях. В налаштуваннях сервісу до можливості вибору облікового запису, що використовується для резервного копіювання, також належать можливості вибору папок на пристрою, що повинні автоматично

синхронізуватися, а також вибору використання інтернету для завантаження файлів [2]. Це полегшить зберігання файлів та знизить втрату медіа даних.

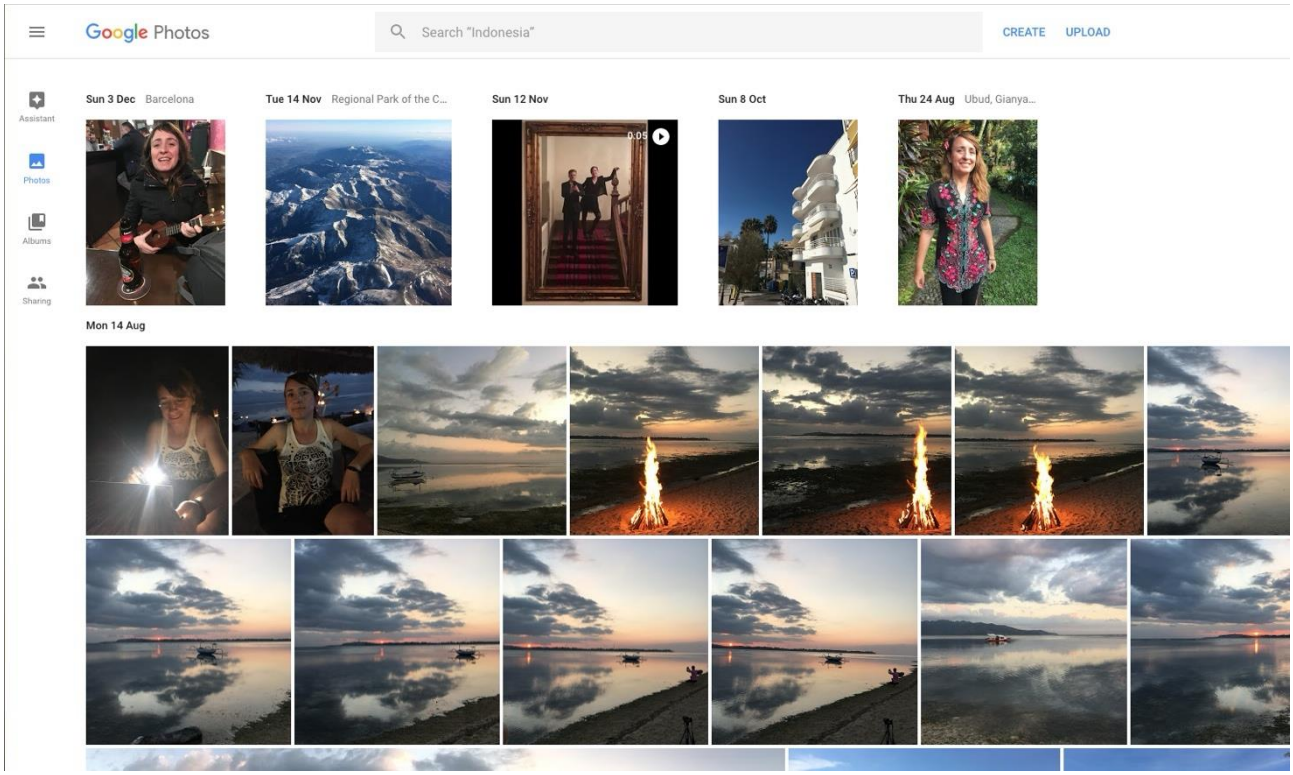


Рис. 1.1 Головний інтерфейс Google Photos

Одна з найважливіших переваг Google Photos – це можливість пошуку медіа файлів. Користувачам дозволено знаходження медіа файлів за датою, місцем, об’єктами, людьми, текстовими запитами тощо. Використовуються можливості штучного інтелекту Google для пошуку файлів, що дозволяє шукати не тільки файли зі зазначеними іменами, а й файли зі зазначеним змістом [1; 3]. Також у Google Photos розвиваються AI-функції, одна з яких зумовлює використання можливості Ask Photos, яка використовує платформу Gemini для розуміння запитів користувача [3].

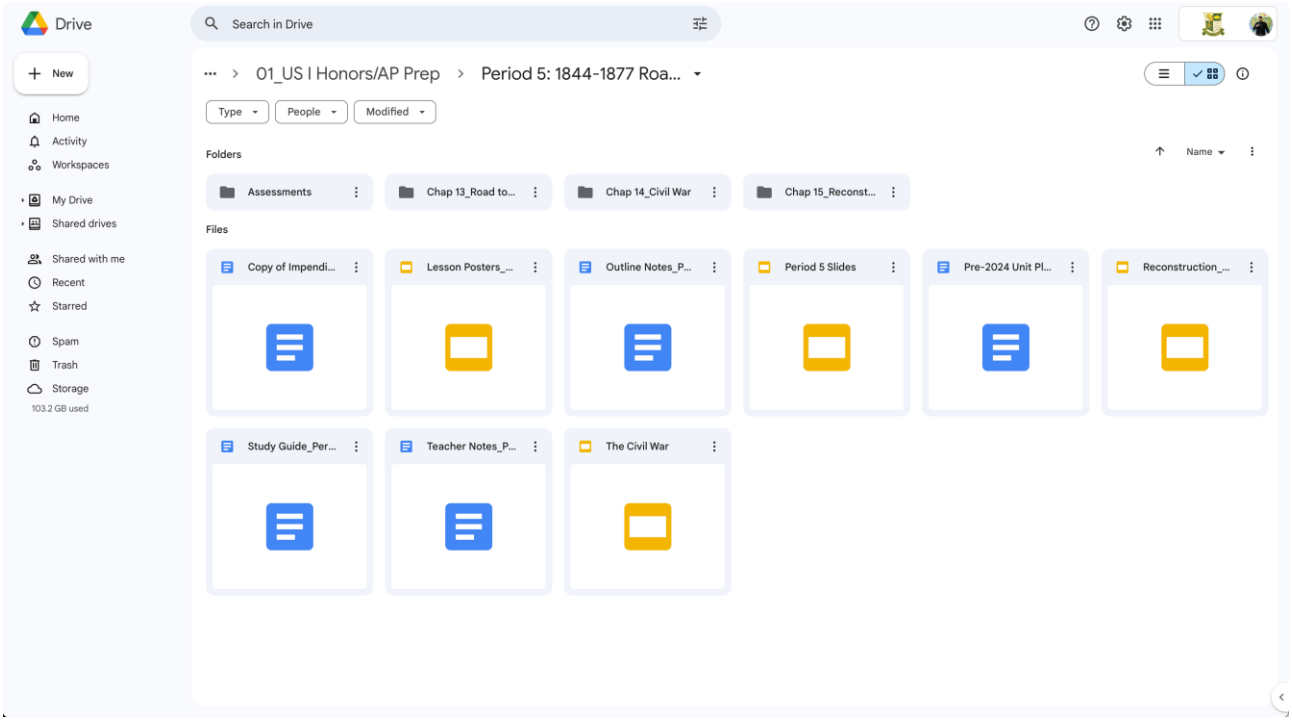


Рис. 1.2 Приклад пошуку медіа файлів у Google Photos

Особливості Google Photos включають в себе величезний інтерфейс, можливість автоматичного резервного копіювання, синхронізації з мобільними пристроями, можливість пошуку медіа файлів за змістом, можливість ведення розмов з файлами та інтеграція з іншими цифровими сервісами. Google Photos є дуже зручним для використання в повсякденному житті.

Google Photos також має ряд обмежень. Перше, що варто вважати обмеженням цього сервісу є те, що він є хмарним сервісом та файли передаються в сервери постачальника. Другим обмеженням є обсяг безкоштовного сховища, який вичерпнуто використовується для інших Google сервісів [1]. Третій фактор – користувачі не мають абсолютної можливості впливу на внутрішні компоненти Google Photos. Останнє, що варто відзначити про цей сервіс – це те, що він не є self-hosted рішенням. Вони можуть бути розгорнуті тільки на локальному пристрою користувача.

Google Photos є однією із найкращих замін для ведення хмарного розпорядження медіа файлів. Однак, власна архітектура сервісу не відповідає вимогам локального контролю над файлами. Для системи megadrive важливо вважати, якими є переваги сервісу Google Photos – відчуття зручності в перегляді

галереї, можливості пошуку, роботи з метаданими файлів до спрощеної взаємодії з системою, але до можливості їх реалізації в рамках самовиконуваного сервісу.

1.2.2 Immich

Immich є сучасним рішенням для резервного копіювання, а також відправлення фотографій і відео. Відмінне від хмарних рішень, таких як Google Photos, Immich виробляється для того, щоб відійти до власного серверу. Це означає, що користувачі можуть зберігати свої файли без втрати даних, а також визначати, як вони будуть зберігатися й адмініструватися [4].

Основна ідея створення Immich – це створення приватного серверу для відправлення та перегляду фотографій і відео. Зокрема, сервер підтримує можливість автоматичного резервного копіювання фото й відео з мобільного пристрою на сервер [4; 5].

Однією з можливостей Immich є робота з мобільним резервним копіюванням. Користувач може вибрати альбоми на мобільному пристрої, що відійдуть до сервера, а також виключити вибір окремих альбомів. У разі вибрання альбомів система використовує контрольні суми, щоб визначити, що вже є на сервері, і не додати їх до сервера [5].

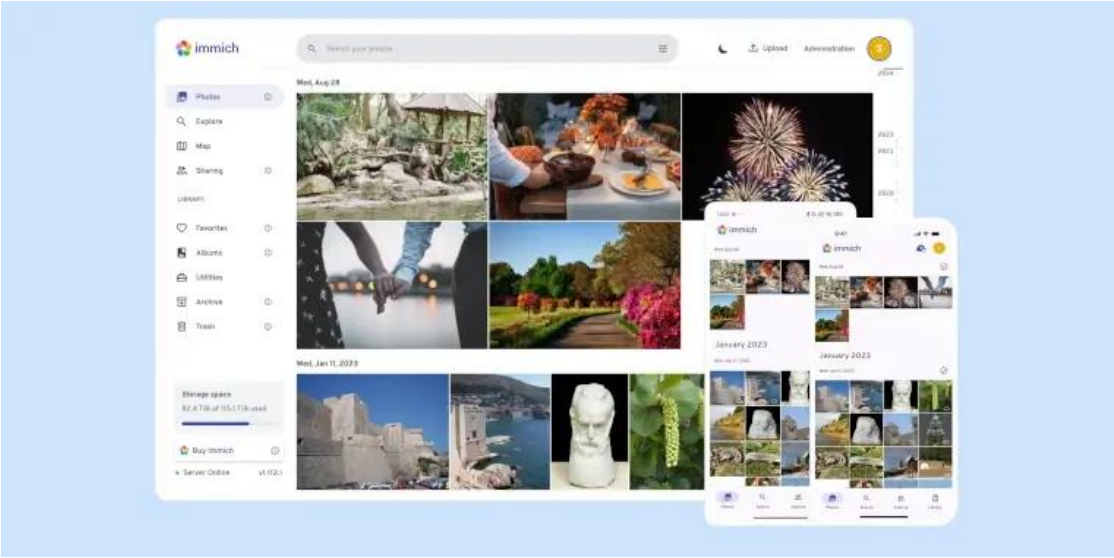


Рис. 1.3 Головний інтерфейс self-hosted системи Immich

Immich підтримує багато різних форматів зображень і відео. Сервер підтримує формати JPEG, PNG, WEBP, GIF, HEIC, HEIF, TIFF, RAW і інші формати [6].

Однією з найважливіших можливостей Immich є система пошуку. Відомо, що сервер використовує PostgreSQL, як рішень для пошуку CLIP метаданих. Це означає, що користувачі можуть виконувати пошук зображень, навіть без використання ключових слів для опису зображень. Це дуже зручно для великих кількостей зображень [7].

1.2.3 PhotoPrism

PhotoPrism – це self-hosted програмне рішення, що забезпечує організацію, перегляд, пошук і автоматичне впорядкування фотографій і відео. Це AI-powered десертилізоване рішення, яке може бути розгорнуте і використане як вдома, так і на сервері чи у хмарному середовищі [9]. Вони виступають важливим аналогом системи megadrive.

Основна функція PhotoPrism – створення медіа бібліотеки для перегляду, пошуку, роботи з альбомами, мітками, метаданими і геолокацією. Інтерфейс дозволяє користувачам працювати з медіа файлами через веб-браузер [9; 10].

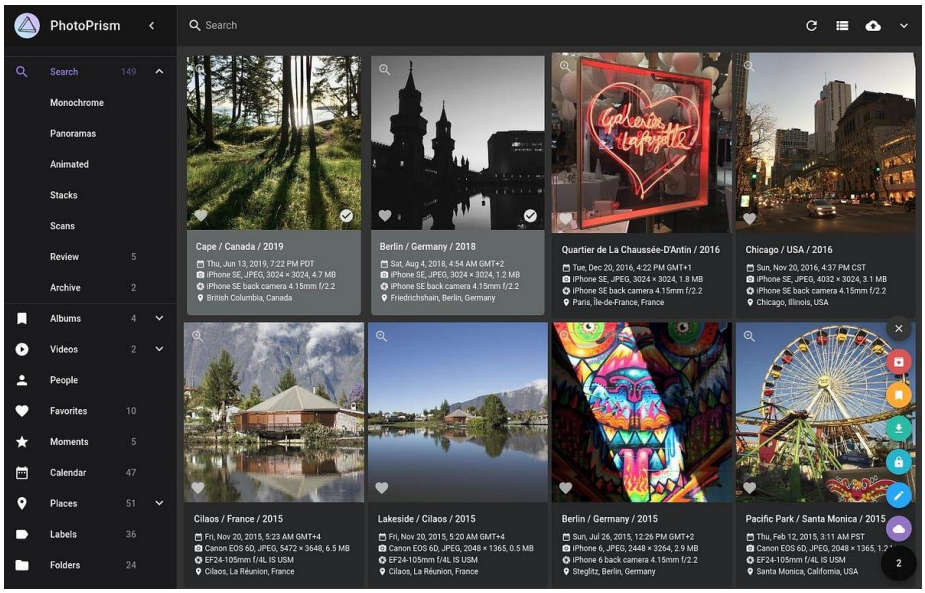


Рис. 1.4 Головний інтерфейс системи PhotoPrism

Одним із найважливіших елементів цієї системи – автоматична класифікація зображень. У PhotoPrism використовуються мітки для класифікації фотографій. Деякі з цих міток створюються автоматично при додаванні нових фотографій у бібліотеку. Вони можуть також бути вручну відредаговані чи видалені [11].

Для пошуку зображень і відео у системі можна використовувати пошукові фільтри, які дозволяють шукати за мітками, місцем, параметрами зображення, якістю, кольором і іншими параметрами [10].

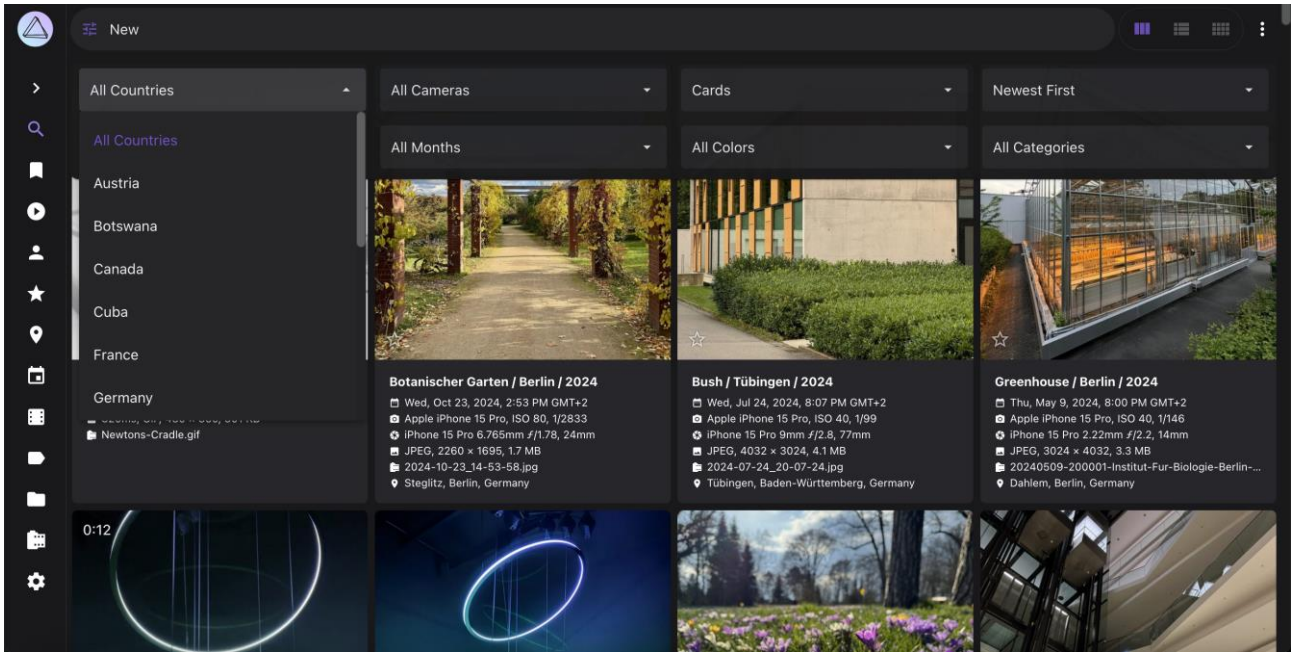


Рис. 1.6 Приклад пошуку та фільтрації медіа файлів у PhotoPrism

Система може автоматично сканувати медіа файли та sidecar файли на їх метадані в форматі Exif, XMP, IPTC, Google Photos JSON і інші формати [12]. PhotoPrism підтримує індексацію, перегляд і конвертацію багатьох форматів зображень, відео і RAW файлів, включаючи JPEG, PNG, GIF, BMP, HEIF, HEIC, MP4, MOV, WebP, WebM і інші формати [10; 13].

PhotoPrism володіє рядом переваг, що відзначають це програмне рішення як високоякісну систему для керування медіа файлами. До них належать відкриті файли, підтримка великої кількості форматів, автоматичне створення міток, робота з метаданими, пошукові фільтри, виявлення дублікатів, перегляд медіа файлів

через веб-інтерфейс. Вони відзначають систему близькою до теми кваліфікаційної роботи.

Відмінністю від розроблюваного рішення є те, що megadrive є не тільки фотогалереєю, але й локальним медіа-індексатором, який передбачає роботу з великою кількістю зображень, відео і аудіо файлів з використанням AI-метадани, OCR-тексту, транскрипцій і зовнішніх API. Архітектура megadrive базується на використанні монолітного сервера на джава мові Go з вбудованим інтерфейсом на React, база використовує SQLite базу відомостей з підтримкою FTS5, SSE-подіями та можливістю відкрити файли у локальному файловому менеджері у режимі локального режиму.

Отже, PhotoPrism виступає важливим аналогом системи megadrive через її можливості. Вони можуть бути враховані в розробці системи megadrive. Водночас систему можна відрізняти від розроблюваного рішення її іншими можливостями.

1.3 Порівняння аналогів

Для визначення вимог до розроблюваної системи megadrive доцільно виконати порівняння існуючих аналогів. Вже в попередніх підрозділах було розглянуто кілька програмних рішень, що відповідають відомим функціям організації, перегляду та пошуку медіа даних – Google Photos, Immich та PhotoPrism.

Google Photos є хмарним сервісом, що забезпечує зберігання, резервне копіювання, пошук і організацію фотографій та відео [1; 2]. Основною перевагою рішення вважається їх простота використання, автоматична синхронізація між пристроями, веб- та мобільний інтерфейс, а також використання AI-функцій для редагування й пошуку медіа даних [1; 3]. Google Photos не дає можливості встановлення серверної частини, місця зберігання відео файлів та виконання своєї обробки. Сервіс не є self-hosted, що відокремлює його від інших рішень.

Immich є рішенням, що спрямоване на розгортання власного сервера для умов зберігання, резервного копіювання, організації та перегляду фото- і відеофайлів [4].

Перевагами системи є можливість мобільного резервного копіювання, праця зі зовнішніми бібліотеками, пошук по метаданим, об’єктам, обличчям та CLIP, підтримка форматів медіа файлів [5; 6; 7; 8]. Необхідність розгортання сервера для системи Immich викликає складніший процес щодо її розгортання та адміністрування. Immich не є хмарною системою для розгортання відео і аудіофайлів.

PhotoPrism також є рішенням self-hosted типу, але працює у режимі децентралізації зображення і відео. У рішенні передбачено автоматичне тегування, праця з метаданами, пошукові фільтри для підвищення зручності перегляду медіа файлів через веб-інтерфейс [9; 10; 11]. Рішення дозволяє робити метадані відео файлів у форматі Exif, XMP, IPTC, Google Photos JSON [12].

Порівняльну характеристику розглянутих аналогів наведено в таблиці 1.1.

Таблиця 1.1

Порівняння аналогів програмного забезпечення для організації медіа даних

Критерій порівняння	Google Photos	Immich	PhotoPrism	megadrive
Тип рішення	Хмарний сервіс	Self-hosted система	Self-hosted система	Self-hosted система
Основне призначення	Зберігання, пошук і резервне копіювання фото та відео	Резервне копіювання, організація і пошук фото та відео	Організація, індексація і пошук фотоколекцій	Локальна індексація, парсинг і пошук медіа даних
Розгортання на власному сервері	Ні	Так	Так	Так
Локальний контроль над даними	Обмежений	Так	Так	Так
Підтримка зображень	Так	Так	Так	Так
Підтримка відео	Так	Так	Так	Так

Продовження таблиці 1.1

Критерій порівняння	Google Photos	Immich	PhotoPrism	megadrive
Підтримка аудіо	Обмежено	Ні / не основна функція	Ні / не основна функція	Так
AI-метадані	Так	Так	Так	Так
OCR-текст	Частково залежить від сервісів Google	Обмежено	Частково	Так
Транскрипція аудіо/відео	Не є основною функцією	Не є основною функцією	Не є основною функцією	Так
Повнотекстовий пошук	Так	Так	Так	Так
Робота з локальними папками	Ні	Через зовнішні бібліотеки	Через індексацію бібліотеки	Так
Користувацькі поля	Обмежено	Обмежено	Обмежено	Так
API-ключі	Не для локального керування системою	Обмежено	Обмежено	Так
Зовнішній стабільний API	Обмежено	Частково	Частково	Так
Вбудований веб-інтерфейс	Так	Так	Так	Так
Гнучка зміна тем оформлення	Обмежено	Обмежено	Обмежено	Так
Залежність від стороннього постачальника	Висока	Низька	Низька	Низька

1.4 Поняття проблем області

Після аналізу існуючих рішень можна визначити основні проблеми, що виникають у галузі організації, індексації та пошуку великих масивів медіа даних. Також відомо, що існують хмарні та самостійні платформи, які можуть вирішити ці проблеми. Проте досі користувачі стикаються зі складностями при роботі з великими локальними архівами зображень, відео та аудіофайлів.

Першою є проблема складної організації великої кількості медіа файлів. Більшість медіа файлів зберігаються у папках без спроможності створити для них єдну структуру. Назви файлів, які автоматично створюються операційними системами, такими як IMG_1024.jpg, не дозволяють швидко зрозуміти вміст файлів. Змінити їх і вказати інші назви вручну – неефективний процес при роботі із великою кількістю медіа файлів.

Другою є проблема обмеженого доступу до пошуку файлів. Операційні системи дозволяють шукати файл в основних параметрах, які не відображають зміст медіа файлів. Вони не дозволяють шукати за описом, ключовими словами, текстом, який визначений за допомогою AI. Саме тому вони використовують системи з можливостями повнотекстового пошуку, індексації медіа файлів [7; 10; 14].

Третьою є трудомісткість ручного заповнення метаданих. Для пошуку медіа файлів до необхідні описи до файлів. Відкрити і написати такий опис до сотень і тисяч файлів потребує від користувача значних витрат часу. Тому, основні локальні архіви не відкриваються до доступу до метаданих, а доступ до них обмежується лише базовими параметрами файлів. Для автоматизації цього процесу використовують штучний інтелект [3; 7; 11].

Четвертою є залежність від хмарних сервісів. Хмарні сервіси, такі як Google Photos дозволяють роботу з медіа файлами. Проте медіа файли відкриваються тільки в хмарі, тобто вони передаються в сторонній сервер постачальника хмарних послуг [1; 2]. Це може бути небажаним в разі зберігання приватних або професійних медіа файлів, внутрішніх архівів тощо.

Self-hosted рішення можуть частково вирішити проблему, пов’язану із доступом до хмарних серверів. Програми, такі як Immich або PhotoPrism дозволяють розгорнути програми у власному середовищі. Проте вони переважно спрямовані на роботу з фото- або відеофайлами [4; 9]. Можливості інтеграції з іншими типами медіа даних, такими як аудіофайли, транскрипції, OCR-текст, або інші поля відкриваються тільки частково або потребують додаткової адаптації.

Одною із відомих проблем є відсутність зручного веб-інтерфейсу, яким можна було б працювати з локальними медіа файлами. Крім зберігання також мають бути можливість перегляду, відкривання, зміни налаштувань для медіа файлів, їх управління та інше.

Одна із основних проблем відкриття медіа файлів відокремлюється саме через різні типи даних, які можуть бути зберігані у медіа архівах. Наприклад, зображення можуть мати опис, відео можуть мати тривалість, відео та аудіо може мати транскрипції, зображення – мініатюри, відео – кадри, аудіо – тривалість, формат тощо. Система організації медіа даних має бути спроєктована із можливостями відокремлення і індексації цих типів даних.

У межах розробки системи megadrive визначеними проблемами для вирішень стануть самостійна архітектура, індексація медіа файлів притамінено до SQLite FTS5, використання AI для створення метаданих, OCR-текстів, транскрипцій, інтеграцій із користувацькими полями, інтеграції із відкритими API, використанням протоколу SSE для виведення подій і веб-інтерфейсу. Вони дозволятимуть створити систему, яка вирішить вказані проблеми.

1.5 Постановка завдання на розробку системи

Внаслідок проведеного аналізу встановлюється необхідність розробки рішення, що буде дозволяти організацію, індексацію, парсинг та пошук великих масивів медіа даних. Основною ідеєю було вирішити задачу праці з власними сховищами файлів, не передавати їх до хмарних служб.

Розроблюване програмне забезпечення отримало назву megadrive. Відповідно до ставлення до запропонованого завдання, необхідно буде вирішити такі завдання: додавання локальних сховищ, їх сканування та індексація, створення мініатюр, додавання метаданих та впереміщення до веб-інтерфейсу. Відмінні від файлових менеджерів, система буде дозволяти працювати з описами, тегами, OCR-текстом та транскрипціями.

Загальне завдання – створення програмної системи, яка вирішує задачу роботи з великими масивами медіа даних. Вона має виконувати задачі, що відповідають за зберігання медіа даних, індексацію, використання AI для аналізу даних та веб-інтерфейс. Такий підхід дозволяє ефективно вирішувати поставлене завдання.

Вирішення проблеми системи буде виконуватися досить складним способом. Серверна частина розробляється на мові програмування Go. Для цього вирішуватимуться задачі, що мають відповідність запитам, працювати з базою даних, сховищами, створення мініатюр, обробки запитів до AI-провайдерів, а також автентифікацію та API для веб-підходу. Для маршрутизації буде використано бібліотеку go-chi/chi для створення HTTP-сервісів [16].

Для зберігання даних відповідно до вимог до системи використовується SQLite. Ця система баз даних вже вирішує багато запропонованих вимог до self-hosted систем: вона працює з локальними базами, не вимагає окремого процесу сервісу та дозволяє зберігати дані в файлі. В базі зберігатимуться дані про сховища, медіа файли, метадані, тему, налаштування та API-ключі. Для пошуку метаданих використовується механізм повнотекстового пошуку FTS5 [14].

Веб-застосунок для системи вирішується на базі React, TypeScript, Vite та TailwindCSS. Система зможе відкрити медіа файли в галереї, виконати пошук за метаданими та фільтрацію відповідно до параметрів, відкрити бічну панель для вибору сховищ, відкрити файли в модальному вікні та впереміщуватися до відео, аудіо, зображення, змінювати налаштування, тему відображення та пароль [17; 18; 19].

Одне з основних завдань для розробки системи – це створення механізму індексації медіа файлів. Відповідно до задачі відбуватимуться дії від сканування локальних папок, визначення типу файлів, зберігання метаданих, створення мініатюр, визначення технічних характеристик файлів та передачу їх до системи відокремлення на базі AI. Для роботи із відео та аудіо файлами вказуватиметься інтеграція з ffmpeg та ffprobe [15].

Другим важливим завданням є створення механізму пошуку та фільтрації. Користувачі зможуть виконувати пошук за метаданими за текстовими полями та використовувати фільтр відповідно до вибраних тегів, сховищ та MIME-типу файлів. Поради до пошуку матимуть такі ж вимоги до відображення метаданих: пошук у назві файлу, описі, OCR-тексті, транскрипції та відображення тегів, назв сховищ та типів файлів.

Одне з важливих завдань в розробці системи – забезпечити безпеку доступу до системи. Для вирішення цієї задачі буде вирішено питання автентифікації за паролем, JWT сесіями, вихідом і вхідом з системи, зміною та виблиском паролю. Також система зможе мати механізм API-ключів для доступу до відокремленого стабільного API. Ключі зберігатимуться у базі даних у вигляді хешу для забезпечення безпеки системи.

Крім встановленого, система має також вирішувати задачі щодо налаштувань, відображення за допомогою тем, SSE події та дії над масивами файлів. Наприклад, можливо вирішити задачу індексації файлів, видалення їх від індексу файлів або видалення файлів з диска у режимі локального режиму. Такий варіант працює не лише як спосіб перегляду медіа, а також як інструмент для керування медіа архівами.

1.6 Визначення вимог до програмного забезпечення

На основі аналізу предметної галузі, існуючих аналогів та постановки завдання встановлюється співвідношення між вимогами та функціями програмного забезпечення. Вимоги до системи можна розділити на дві групи: функціональні

вимоги та нефункціональні. До функціональних вимог відносяться саме те, що вимагається від програмного продукту, а до нефункціональних – вимоги щодо якості системи, її продуктивності, зручності використання, безпеки та інших умов виконання функцій.

Розроблювана система повинна виконувати функції локальної організації великої кількості медіа даних (зображеннями, відео, аудіофайлами), а також забезпечувати їм доступний веб-інтерфейс для перегляду. Робота системи має відбуватися у self-hosted режимі, тобто на локальних комп'ютерах користувача чи на власному сервері власної мережі.

Функціональні вимоги до програмного забезпечення включають такі пункти:

- відображення медіа файлів у галереї;
- попередній перегляд медіа файлів у веб-інтерфейсі;
- попередження про невідповідність формату файлу відносно режиму відображення;
- навігація у галерії (напрямок прокручування);
- відкриття модального вікна для детального виведення відокремленого медіа файлу;
- відображення метаданих файлів;
- відображення текстового опису медіа файлу;
- забезпечення пошуку за описом або іншими полем;
- відображення результатів пошуку;
- мультиселекція медіа файлів;
- пакетні дії (запуск нової індексації, видалення із індексу, видалення файлів з диска у режимі self-hosted);
- відображення статусу роботи системи;
- автентифікація користувачів за паролем;
- загальна робота із JWT-токенами (створення, перевірка, завершення);
- зміна паролю;
- створення, перегляд та відкликання API-ключів;
- відкриття зовнішнього API відповідно до специфікації /api/v1;

- наладка програмного забезпечення (налаштування, зміна у вигляді TOML-файлу чи змінних середовища);
- встановлення вимог до конкретних параметрів, що визначаються через змінні середовища (env override);
- встановлення системи тем;
- завантаження власної теми в форматі JSON.

Оскільки сама система megadrive передбачає самовиконується режим в локальних мережах (self-hosted) вона має бути до можливості виконання системи без складної інфраструктури. Використання бази даних SQLite дозволяє зберігати всі дані локально, без вимоги створення окремого сервера для бази даних. Також база даних має використовувати модуль FTS5 для виконання повнотекстового пошуку [14]. Всі внутрішні системні компоненти виконуються на мові Go: серверна частина виконується у вигляді єдиного застосунку на Go, а інтерфейс відображається через використання упаковування за допомогою Go embed [16].

Вимогами до системи є:

- виконання роботи в режимі self-hosted на власному комп’ютері та сервері користувача;
- реалізація серверної частини в єдиному застосунку на Go;
- використання бази даних SQLite;
- використання режиму WAL бази даних SQLite;
- використання модулів повнотекстового пошуку FTS5 в базі даних SQLite;
- реалізація інтерфейсу на веб-сторінці на візуальних бібліотеках React;
- завантаження галереї посторінково, з величиною сторінки 50 елементів;
- завантаження мініатюр медіа файлів в лінійному режимі, без перевантаження інтерфейсу;
- виконання системного зберігання та відновлення інформації про відео, аудіо та зображення;
- інтеграція із програмами ffmpeg та ffprobe для роботи з медіа файлами (отримання визначеної інформації, відтворення відео та аудіо) [15];

- відображення стану системи відкритим доступом використовуючи JWT-токени та API-ключі;
- вимога для API-ключів відкритого доступу виключена;
- використання файлу налаштування config.toml для передачі інформації у систему;
- використання змінних середовища з пріоритетом відносно бази даних;
- зміна окремих параметрів системи вимагає перезавантаження програми;
- відображення статусу роботи системи без перезавантаження сторінки;
- відображення власної теми інтерфейсу;
- забезпеченими можливостями розширення системи новими видами AI-провайдерів або провайдерами для зберігання медіа файлів.

Одною з основних вимог до системи є забезпечення безпеки. Відкритий доступ до системи може бути забезпечено шифруванням даних.

Система може вимагати від користувачів встановлення паролів, JWT-токенів, API-ключів. Підходом до паролів є використання більшості бікриптографічних хешувань, а самі API-ключи мають зберігатися у вигляді хешу у базі даних. Вимога щодо більшості відкритих даних може перекрити вимогу до зручності виконання основних робіт у системі.

Користувачам треба мати можливість виконувати основні дії в рамках роботи системи: додавання медіа файлів, виведення інформації про них, попередження про відхилення від форматів, дії з галереєю, здійснення пошуку, зміни налаштувань системи чи саме інструментів, що відносяться до API-ключів.

2 АНАЛІЗ ЗАСОБІВ РЕАЛІЗАЦІЇ ПРОГРАМНОЇ СИСТЕМИ

2.1 Огляд технологій розробки

Для реалізації системи megadrive було створено технологічне рішення, що дозволяє виконувати локальний доступ до медіа файлів, веб-інтерфейсів, встановленого пошуку, здатність працювати з відео та аудіофайлами, а також інтеграції з системами автентифікації та API. Загальний вибір технологій був здійснений для того, щоб технологія була продуктивною, зручною для розгортання та можливості розширення функцій системи.

Основна кількість технологій серверної частини виконалася на мові Go, оскільки ця мова відома тим, що може виконувати роботу з мережею, командною лінією, роботи з файлами та інтеграції статичних файлів. Така мова використовується для створення HTTP-запитів, взаємодії з базою даних, запуску процесу індексації, роботи з файлами на комп'ютері, створення мініатюр та автентифікації, а також для передачі API клієнтської частини [16; 20].

Для розподілення запитів використовується бібліотека go-chi/chi. Вона дозволяє створити компактні маршрути, групувати API-ендпоінти, а також використовувати middleware для журналів та відновлення системи в разі помилок. В системі megadrive відзначається використання маршрутизатора для створення внутрішнього та зовнішнього API, маршрути для автентифікації, роботи із зображеннями, мініатюрами, сховищами, налаштуваннями та ключами API [16].

Система зберігання вибрана на базі технології SQLite. Відома тим, що база даних встановлена без відокремленої інфраструктури та зберігає дані у файлі. Такий вибір дозволяє системі легко відділити себе в локальній системі або сервері. База даних використовується для зберігання інформації про медіа файли, сховища, інформації від системи відео та аудіо, налаштувань, тем та міграцій схеми бази даних. Для встановлення пошуку використовується розширення SQLite FTS5, яке дозволяє встановити повнотекстовий пошук в текстових даних бази даних [14; 21].

Відокремлено встановлено систему клієнтської частини на технології React SPA. Відомо, що React дозволяє створювати складні інтерфейси на основі окремих компонентів, що важливо для створення системи інтерфейсу. В React використовується TypeScript для типізації коду, що дозволяє підвищити надійність коду та зпростити процес відновлення й оновлення великих компонентів інтерфейсу [17; 18]. Встановлено використання Vite, яка дозволяє швидко створювати веб-застосунки, що забезпечують створення оптимізованих статичних файлів для системи в режимі роботи [19].

Для стилізації використовується TailwindCSS, який дозволяє створювати веб-інтерфейс, використовуючи готові CSS-класи, не створюючи велике число окремих правил [22]. В системі megadrive TailwindCSS використовується для створення галереї, бокової панелі, модальних вікон, кнопок, елементів стану, бейджів та встановлення тем і адаптивності вікна.

В окремих випадках для виконання обробки медіа файлів встановлюється бібліотека imaging для роботи зі зображеннями, а також встановлюється ffmpeg та ffprobe для інтеграції з відео та аудіофайлами. Ці технології встановлено в основній системі для вивчення технічної інформації про медіа файли, визначення тривалості відео та аудіофайлів та вивчення їх вмісту [15].

Важливою системою є автентифікація. В системі використовується bcrypt для паролів, а також HMAC-SHA256 для JWT-токенів та системи доступу до системи через API. Це дозволяє зберігати дані безпечно та забезпечувати можливість контролюваного доступу.

Для системи встановлено використання системи SSE. Відомо, що система дозволяє автоматично отримати оновлення від сервера до клієнта. Відображення стану індексації, оновлення сховищ та конфігурації, зміна теми, службові повідомлення без постійного оновлення веб-сторінки. Така система дозволяє виділити стан індексації великого масиву медіа файлів, які встановлені у системі.

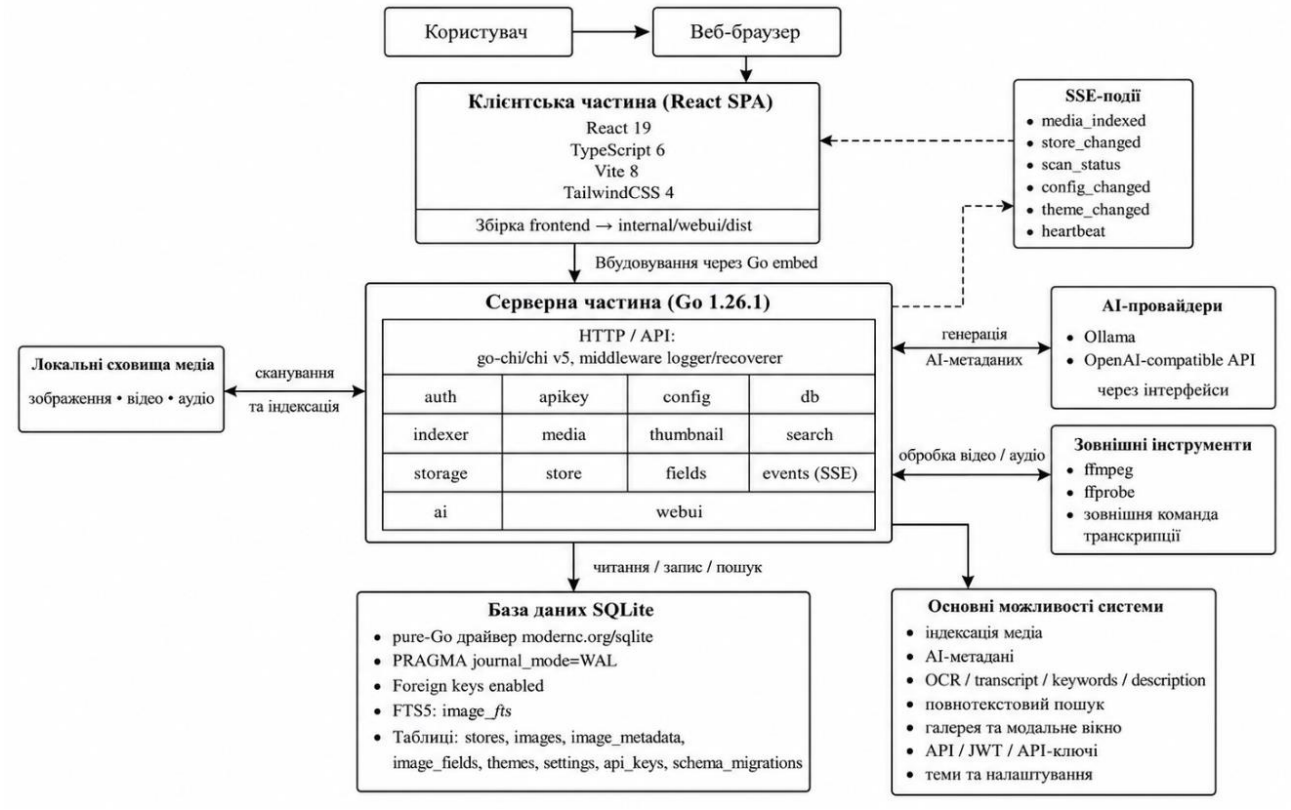


Рис. 2.1 Загальний технологічний стек розроблюваної системи megadrive

Таким чином, вибраний технологічний стек відповідає технологіям, що мають бути використані для вирішення завдання кваліфікаційної роботи. Мова Go відіграє роль у виробництві серверної частини системи, а також виробництві локальної взаємодії з системою. База даних SQLite та розширення FTS5 забезпечують можливість зберігання та пошуку медіа файлів. Веб-інтерфейс виробляється на базі React, TypeScript, Vite та TailwindCSS, а також встановлено можливість виконання інтеграції з відео та аудіофайлами для їх розкриття [15]. Дані технології дозволяють розробити самоузагальнену систему, яка дає можливість виконання всіх функцій та створити зручний інтерфейс для користувачів.

2.2 Серверна частина системи

Центральна частина програмного забезпечення системи megadrive – це серверна частина системи. Саме вона відповідає за обробку запитів від клієнтської

частини, взаємодію з медіа сховищами, базою даних, виконання AI-аналізу файлів, автентифікацію та створення API для клієнтської частини. Дану частину реалізовано на мові Go. Мова Go обрана для цього процесу завдяки своїй можливості створення продуктивних серверних систем, мережевих сервісів або self-hosted дій [20].

Для організації HTTP запитів використовується бібліотека go-chi/chi/v5. Вона дозволяє створити компактний маршрутизатор для запитів від веб-інтерфейсу [16]. В системі використовується відокремлення внутрішнього та зовнішнього API, розміщення маршрутів авторизації, сховищ, медіа файлів, мініатюр, налаштування та тем.

Основними API запитами для обробки є запити до роботи зі сховищами, отримання інформації про storage-провайдерів, роботи з медіа файлами, перегляду детальної інформації про медіа файли, повторного встановлення медіа файлів до черги AI-індексації, відкриття медіа файлів у відділеному файловому менеджері у локальному режимі, отримання мініатюр медіа файлів та доступ до файлу.

До окремих запитів на сервері відносяться операції, що відкривають можливість виконання пакетних операцій: індексації вибраних медіа файлів, видалення медіа файлів з індексу та видалення медіа файлів з диска в випадку ввімкнення локального режиму.

Одним із важливих елементів серверної частини системи відокремлюється як індексатор. Він відповідає за сканування медіа файлів на диску, визначення їх типів, відображення їх в базі даних, встановлення мініатюр до медіа файлів та передачу медіа файлів до AI для виконання аналізу. Індексатор сканує папки з медіа файлами за такими параметрами, які визначаються типом файлів: ім’я файлу, розмір, MIME тип, тип медіа, технічні характеристики файлу. Випадки використання зовнішніх інструментів ffmpeg та ffprobe дозволяють виконати аналіз відео та аудіо файлів [15].

Для розширення можливостей системи відокремлено окремий модуль, що відповідає за виконання аналізу медіа файлів за допомогою штучного інтелекту, або AI. Запрограмованими можливостями є підключення різних провайдерів

штучного інтелекту, використання локальних моделей через Ollama або OpenAI-compatible API. Відокремлені поля для збереження результатів аналізу AI: опис, слова, OCR-текст, транскрипція, сигналізация або інші вказані поля.

Зберігання всіх даних відбувається завдяки взаємодії серверної частини з базою даних SQLite. База даних використовується для зберігання даних про сховища, медіа файли, метадані, поля для користувачів, теми, налаштування, API-ключі та міграції схеми. Для виконання пошуку в базі даних використовується таблиця FTS5, яка дозволяє виконувати пошук медіа файлів за ім'ям, описом, OCR-текстом, транскрипцією та словами [14; 21].

Автентифікація користувача та їх захист від небажаного доступу здійснюється серверною частиною системи. До запитів відносяться автентифікація за паролем, створення сесій (JWT), перевірка входу, вихід з системи, зміна та очистка паролю. Використання API-ключів створює можливість доступу до системи за зовнішніми серверами, так як у базі даних зберігається хеш ключа, а сам ключ не відображається.

Одним із окремих механізмів, які відділені від системи, є SSE. Цей механізм дозволяє відділити події від серверної частини до веб-інтерфейсу. До подій відносяться події відбування індексації медіа файлів, зміни в сховищах, виконання медіа файлів, відновлення медіа файлів, налаштування системи, зміна теми відображення веб-інтерфейсу.

Доцільність виконання цього підрозділу – вдосконалення блок-схеми проекту. Блок-схема відображає всі основні компоненти системи. Основним компонентом відокремлюється Go server, який відділений від іншої частини системи. Він відділений від HTTP API, SQLite, медіа сховищ, системи AI, зовнішніх інструментів ffmpeg/ffprobe та веб-інтерфейсу, розміщеного на основі React. Блоки, що відділені всередині Go server відокремлюються таким чином: API, автентифікація, API-ключі, налаштування, база даних, індексатор, медіа файли, мініатюрні файли, пошук медіа файлів, сховища, сховищі, поля для користувачів, події, AI та веб-інтерфейс.

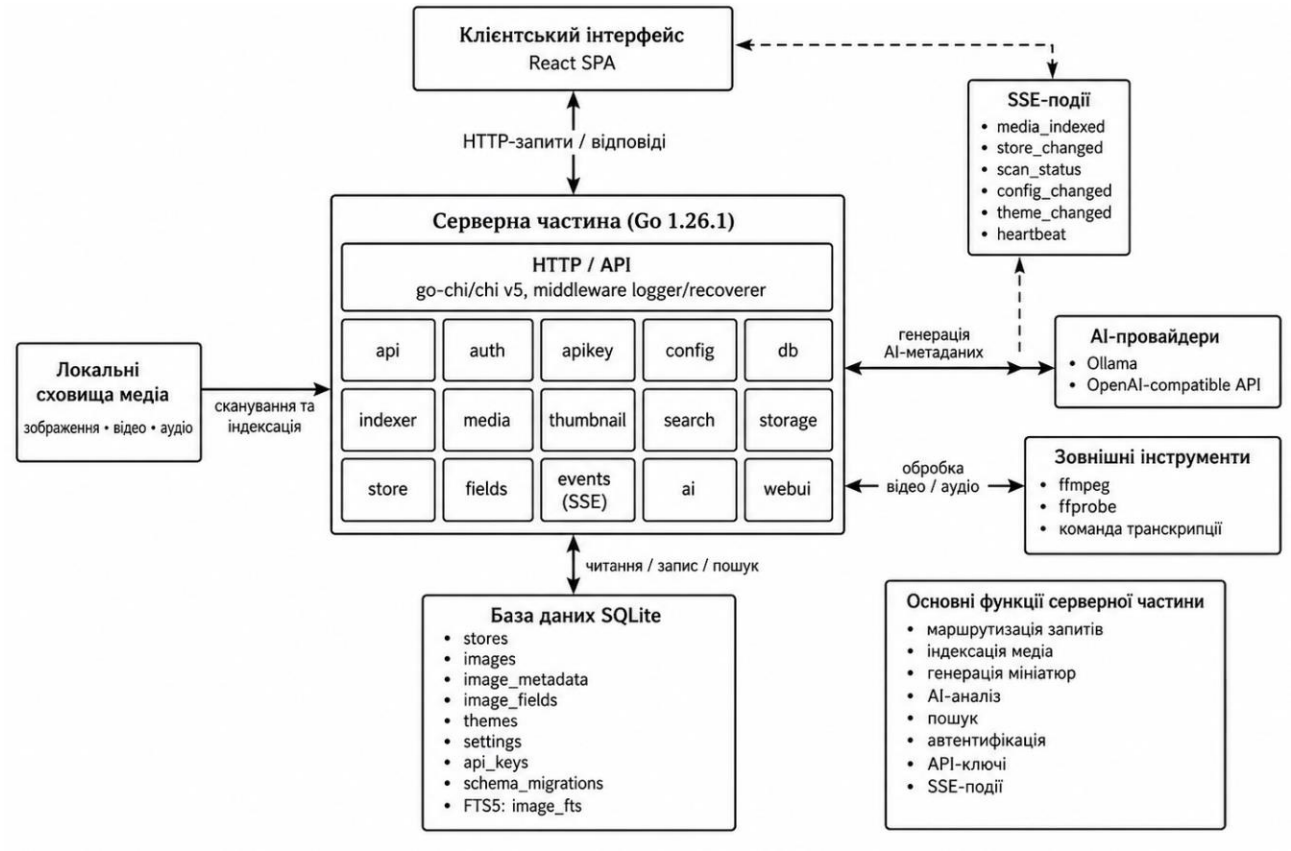


Рис. 2.2 Структура серверної частини системи megadrive

Відокремлення серверної частини системи megadrive виконане для відділення основного розділу системи, що відповідає за взаємодію між всіма іншими компонентами системи. Вони включають саме відділення медіа сховищ, бази даних, системи AI, зовнішніх інструментів, веб-інтерфейсу та інших зовнішніх API. Використання мови програмування Go, бібліотеки go-chi/chi/v5, зберігання даних в базі даних SQLite, таблиці FTS5 та створення веб-інтерфейсу на React дозволяє створити self-hosted систему, що мінімально складна для розгортання, але одночасно відображає можливості роботи з великими масивами медіа даних.

2.3 Клієнтська частина системи

Клієнтська частина системи megadrive виконується у вигляді односторінкового веб-застосунку (SPA), що забезпечує взаємодію користувача з основними функціями системи. У веб-інтерфейсі можна переглядати файли медіа,

виконувати пошук, керувати сховищами, переглядати інформацію про вибраний файл, виконувати повторну індексацію файлів, робити налаштування системи, вибирати тему відображення, встановлювати пароль та працювати з API-ключами.

Для реалізації клієнтської частини використовується популярний фреймворк React 19. React – бібліотека, що дозволяє створювати користувацькі інтерфейси. Він дозволяє розділити інтерфейс на окремі елементи. Для системи megadrive такими можна вважати наступні елементи: header, search bar, store manager sidebar, gallery, image modal, indexer status bar, settings modal, theme picker, login page [17]. Такий спосіб розподілу дозволяє спрощувати процес розробки та наступного розширення функціоналу системи.

Клієнтську частину системи відіграє роль React SPA. Такий варіант дозволяє клієнтській частині взаємодіяти з користувачем без перезавантаження сторінки. Всі дані, які вимагає користувач або отримує системи від серверної частини за допомогою API. Такий спосіб взаємодії з користувачем оптималізує процес роботи з великою кількістю медіа файлів.

Код клієнтської частини написано на TypeScript 6. TypeScript дозволяє додавати до властивості JavaScript систему типів. Це дозволяє описувати структуру даних, що виводяться з серверних API [18]. Кількість різних об’єктів у системі велика: медіа файли, сховища, метадані, користувацькі поля, теми оформлення, налаштування та API-ключі.

Для збірки клієнтської частини використовується Vite 8. Vite дозволяє створити середовище розробки для веб-застосунку та створити виробничий бандл веб-інтерфейсу [19]. В результаті збірки в директорії internal/webui/dist отримується веб-інтерфейс, що вбудовується у Go-сервер.

Для стилізації використовується TailwindCSS 4. TailwindCSS дозволяє створювати інтерфейсні елементи, використовуючи готові CSS класи [22]. Це дозволяє швидко реалізувати всі елементи інтерфейсу системи megadrive.

Однією з основних функцій є відображення медіа файлів у веб-інтерфейсі у вигляді галереї. Галерея дозволяє переглядати файли в масоні стилі [22]. Такий спосіб розташування файлів дозволяє зручно відображати файли різних розмірів.

Оскільки кількість елементів може бути великою, використовується нескінченне прокручування елементів у галереї. Розмір сторінки для галереї становить 50 елементів. Таке визначення дозволяє не завантажувати усі файли одночасно. Це дозволяє відображати великі колекції медіа файлів.

У галереї використовуються ліниві мініатюри. Такий варіант дозволяє медіа файлам завантажуватися поступово, коли вони необхідні для відображення в галереї. Такий спосіб дозволяє краще використовувати ресурси веб-бровера.

Заголовок рядків відображається у вигляді header, що містить пошуковий рядок, перемикач тем, налаштування та панель сховищ. У пошуковому рядку можна виконувати пошук тексту, що вказаний у медіа файлах. Також він може бути відділений на наступні режими: діапазонів пошуків за назвами файлів (tag:keyword), назвами сховищ (store:name) та типами файлів (type:image/png).

Бічна панель відображається як Store Manager. У Store Manager відображається можливість додати локальну папку, переглянути всі сховища, побачити кількість медіа файлів, видалити та сканувати сховище. Такий інтерфейс дозволяє самостійно додати будь-які локальні сховища файлів.

При виборі деякого з медіа файлів відкривається модальне вікно із відображенням детальної інформації про вибраний файл. Така інформація відображається у вигляді попередньо кодованого перегляду картинки, відеоплеєр чи аудіоплеєр, відображення медіа файлу, назва, розміри, тип, тривалість, розмір файлу, MIME-тип, опис, OCR-текст, транскрипція, ключові слова, попередження, нотатки, користувацькі поля.. Таке вікно дозволяє відкрити файл у файловому менеджері чи відновити його індексації.

Можливість вибору декількох файлів дозволяє вибирати файли через клавіші Ctrl або Cmd та вибору діапазону вибраних файлів. Вибраними файлами можна індексувати, видаляти з індексу чи видаляти файл від вибору з диска в режимі локального режиму.

Статус може бути відображеним у вікні indexer status bar. У статус барі відображається відсоток створення мініатюр, стан черги з AI-процесами, кількість відібраних файлів, кількість помилок, що відбулися та кнопка для постановки всіх

неіндексованих файлів у чергу індексації. Оновлення відбувається через відправлення SSE подій.

Для налаштування системи також відкривається вікно. У вікні є можливість редагування конфігурації системи, встановлення та видалення паролів, встановлення API-ключа та виходу з системи. Налаштування, що встановлено через змінні середовища, можна зробити невидимими для користувачів. Деякі параметри можна встановити як ті, що вимагають перезапуску системи.

Тема клієнтського вікна може бути визначена відокремленням теми у вікні Theme picker. У системі визначено дві теми: dark-purple та light-purple. Відокремлено можливість завантаження власної теми, що відображається у вигляді файлу у форматі JSON. Теми встановлюються через CSS змінні.

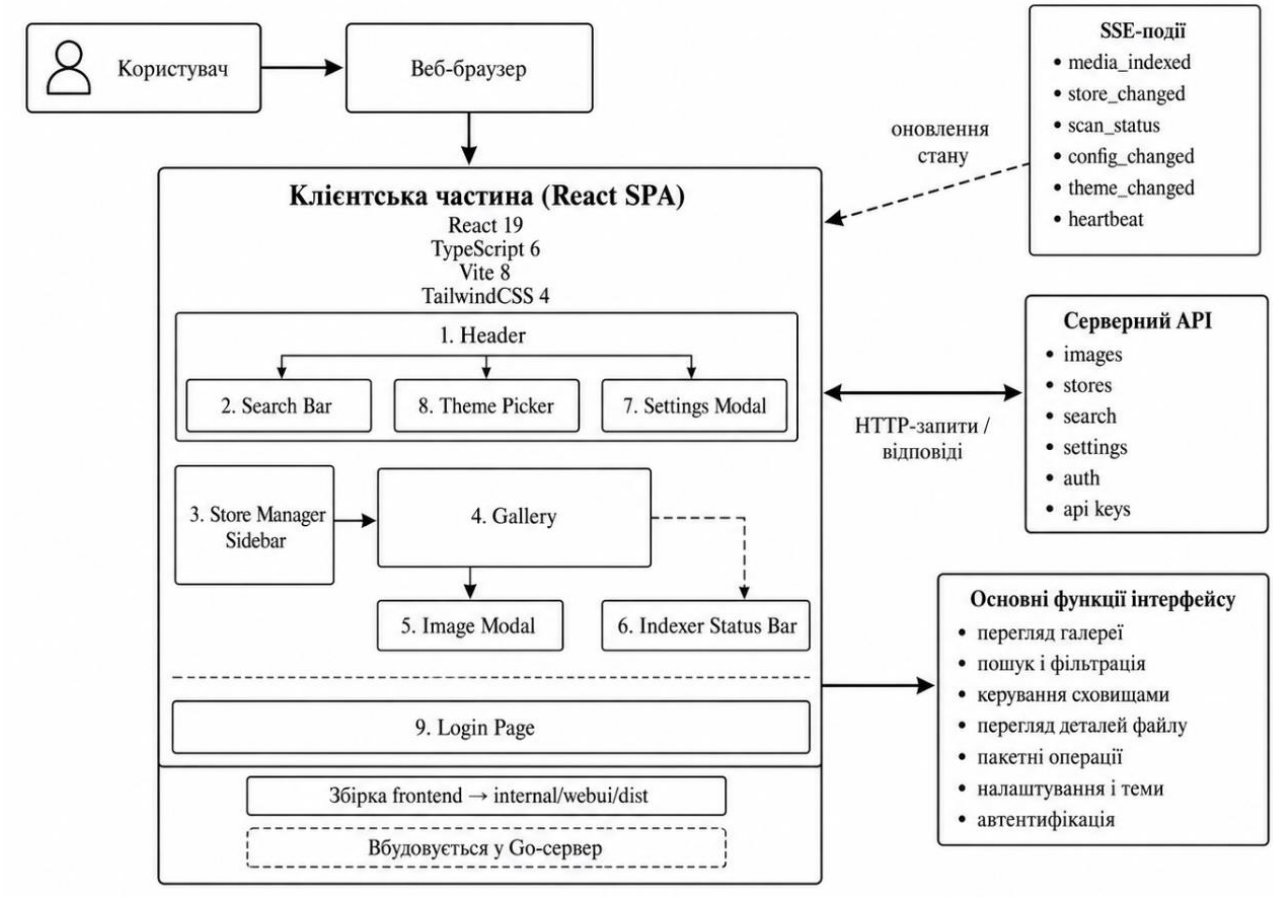


Рис. 2.3 Структура клієнтської частини системи megadrive

Отже, клієнтська частина системи виконує основну роботу взаємодії користувача та системи. Інтерфейс відображає всі основні елементи та функції

системи: галерею зі списком медіа файлів, можливість виконувати пошук, керувати сховищами, переглядати вибрані файли, статус індексації, встановлення параметрів режимів, встановлення тем відображення та відправку запитів на авторизацію.

Використання React, TypeScript, Vite та TailwindCSS забезпечує створення зручного і розширюваних веб-інтерфейсів для роботи з великими кількостями медіа даних.

2.4 База даних та організація пошуку

Одієї з ключових частин системи megadrive є база даних, у яку зберігається багато важливих для системи інформації про локальні сховища, медіа файли, їх метадані, поля користувачів, налаштування і ключі API. Для реалізації данної бази даних було використано SQLite, що є одним із найкращих рішень для самостійних локальних додатків або, краще кажучи, для додатків, які мають вимогу бути самостійними і не вимагати окремого сервера [21].

Використання бази даних SQLite дозволяє системі автоматично запускатися локально, або на власному сервері користувача. Важливо зазначити, що для SQLite не потрібно встановлювати службу бази даних. Відмінно від серверних БД, таких як PostgreSQL або MySQL, база даних SQLite автоматично створюється і запускається при виконанні додатку. Це значить, що він є дуже зручним для використання в самостійних (self-hosted) системах.

Для взаємодії з SQLite використовується чисто Go драйвер modernc.org/sqlite. Він дозволяє збирати застосунок без вимог використання C компілятора (CGO_ENABLED=0). Це значить, що можливо спростити процес компіляції та розповсюдження додатка. Бінарник програми, що було збирається, може бути легко виконаний на різних серверах.

Для збільшення стабільності бази даних між різними комп’ютерами встановлено режим WAL. WAL (Write-Ahead Logging) дозволяє базі даних SQLite ефективніше працювати в часи одночасного читання і запису даних. Зміни

записуються спочатку у журналі, а потім в основну базу даних [21]. Такий режим дозволяє системі megadrive виконувати одночасно відображення медіа файлів, а також індексувати медіа файли.

Оскільки база даних є відкритою для з'єднання з іншими комп'ютерами, встановлено підтримку зовнішніх ключів. Така можливість дозволяє підтримувати цілісність даних у базі. Наприклад, медіа файли пов'язані із сховищем, метадані пов'язані із самим зображеннем, а поля для користувачів належать самому медіа файлу. Використання зовнішніх ключів дозволяє уникнути встановлення записів, які вказують на видалені файли.

Особою роллю в базі даних виконує пошук. У SQLite встановлений модуль FTS5 для повнотекстового пошуку. Він дозволяє створювати віртуальні таблиці, які виконують пошукові запити до бази даних [14]. Використовується віртуальна таблиця image_fts, яка виконує повнотекстовий пошук у полях description, ocr_text, keywords, filename. Також враховуються поля transcript, що створені після відео транскрипції.

У базі даних встановлено можливість виконання повнотекстового пошуку за допомогою кількох алгоритмів. Основним є виконання FTS5 prefix query. Він дозволяє виконувати повнотекстовий пошук за початком слова. Відсутність результатів виконує LIKE fallback. Він дозволяє виконувати простий текстовий пошук. FTS5 trigram fallback також встановлено для можливості неточного пошуку по частинам слова.

Відомо, що користувачі можуть виконувати пошукові запити в локальній базі даних, а також виконувати фільтрацію результатів. Основними фільтрами є фільтри виду tag:keyword, store:name, type:image/png. Такі фільтри дозволяють відфільтрувати знайдені медіа файли за такими параметрами, які вказані користувачем.

2.5 Засоби обробки медіа даних та AI-метаданих

Однією із основних функцій системи є обробка медіа даних. Відсутність можливості аналізу вмісту медіа файлів – одного із основних аспектів системи megadrive – сприймається користувачами як недолік продукту. Медіа даними є дані, що відносяться до зображень, відео та аудіофайлів. Кожен із цих видів вимагає окремої обробки, відповідно до кожного із видів відносяться відповідні засоби, що забезпечують отримання відповідної технічної інформації, створення мініатюр, визначення описів, визначення тексту (через OCR) або транскрипції, та збереження отриманих AI-метаданих.

Першим етапом, що стає можливим завдяки використанню системи файлових шляхів, є визначення типу файлів. Кожен файл може мати окремі параметри: назву, шлях, розмір, MIME-тип, тип медіа, а також для зображень і відео може бути встановлено їх ширину та висоту. Для відео-та аудіофайлів визначається також їх тривалість. Всі дані відображаються у базі даних та можна використовувати для перегляду медіа файлів.

Для визначення зображення використовується бібліотека github.com/disintegration/imaging. Вона дозволяє виконати такі елементи з обробкою зображень: змінити їх розмір, виконати обрізання, масштабування або створити мініатюри. Мініатюри дозволяють переглядати величезний масив зображень, відображених у вигляді галереї, без потенційного завантаження самих великих файлів. Зображення можна зчитати через стандартні декодери Go, а також за допомоги пакета x/image/webp для розширення можливостей розпізнавання форматів, що використовуються у відомих локальних медіа колекціях. Згенероване мініатюрне зображення та дані про зображення зберігаються в таблиці бази даних з іменами images.

Для розповсюдження можливостей відео та аудіофайлів відокремлені зовнішні інструменти – ffmpeg та ffprobe. ffprobe дозволяє зчитати технічну інформацію про медіа файли, а ffmpeg – виконувати різні команди в рамках обробки відео та аудіофайлів [15]. Обидва зазначених інструменти

використовуються для аналізу медіа файлів, створення прев'ю, визначення тривалості медіа файлів та їх подальшої обробки в системі.

Окремим напрямком роботи медіа файлів є транскрипція аудіо- та відеофайлів. Відокремлено зовнішню команду для транскрипції доданий у систему. Цей елемент дозволяє вказати команду програмного забезпечення, що виконує транскрипцію аудіо- та відеофайлів, що дозволяє зберігати отриманий текст як поле transcript у базі даних. Такий підхід дає можливості для користувача виконувати пошук відео- та аудіофайлів відокремлено від їх назв, а також шукати їх відокремлено від інших даних за словами, що містяться в самому аудіофайлі.

Одним із основних елементів системи є створення AI-метаданих, що відокремлені від технічних метаданих. AI-метадані відокремлені від технічних метаданих та містять такі дані, як description, keywords, OCR-текст, transcript, warnings та інші. Field description містить опис відео, зображення, аудіофайлу; keywords містять окремі слова, що описують вміст відео, зображення; transcript містить текст, що розшифрований від аудіофайлів; warnings вказує на наявність попереджувальних вказівок щодо вмісту відео.

AI-метадані відокремлені через використання інтерфейсів провайдерів штучного інтелекту. Це дозволяє підключати до системи різні провайдери штучного інтелекту, які можуть використовувати локальні моделі, встановлені через Ollama, а також провайдери, що від OpenAI. Використання інтерфейсів дозволяє створити гнучку систему, яка може бути легко вдосконалена в майбутньому введенням нових провайдерів штучного інтелекту.

Аналіз медіа файлів відбуватиметься окремо від кожного із провайдерів штучного інтелекту. Чим більше інформації віддається до моделі, тим більшою є її можливість вимірювати такі показники метаданих, як description, OCR-текст, транскрипція аудіо, визначення ключових слів, вивчення вмісту відео та встановлення відокремлено відокремлених відказів відносно вмісту файлів. Величезна кількість медіа файлів, що зберігається в системі, можлива для аналізу через передачу кожного файлу до окремого AI-модуля.

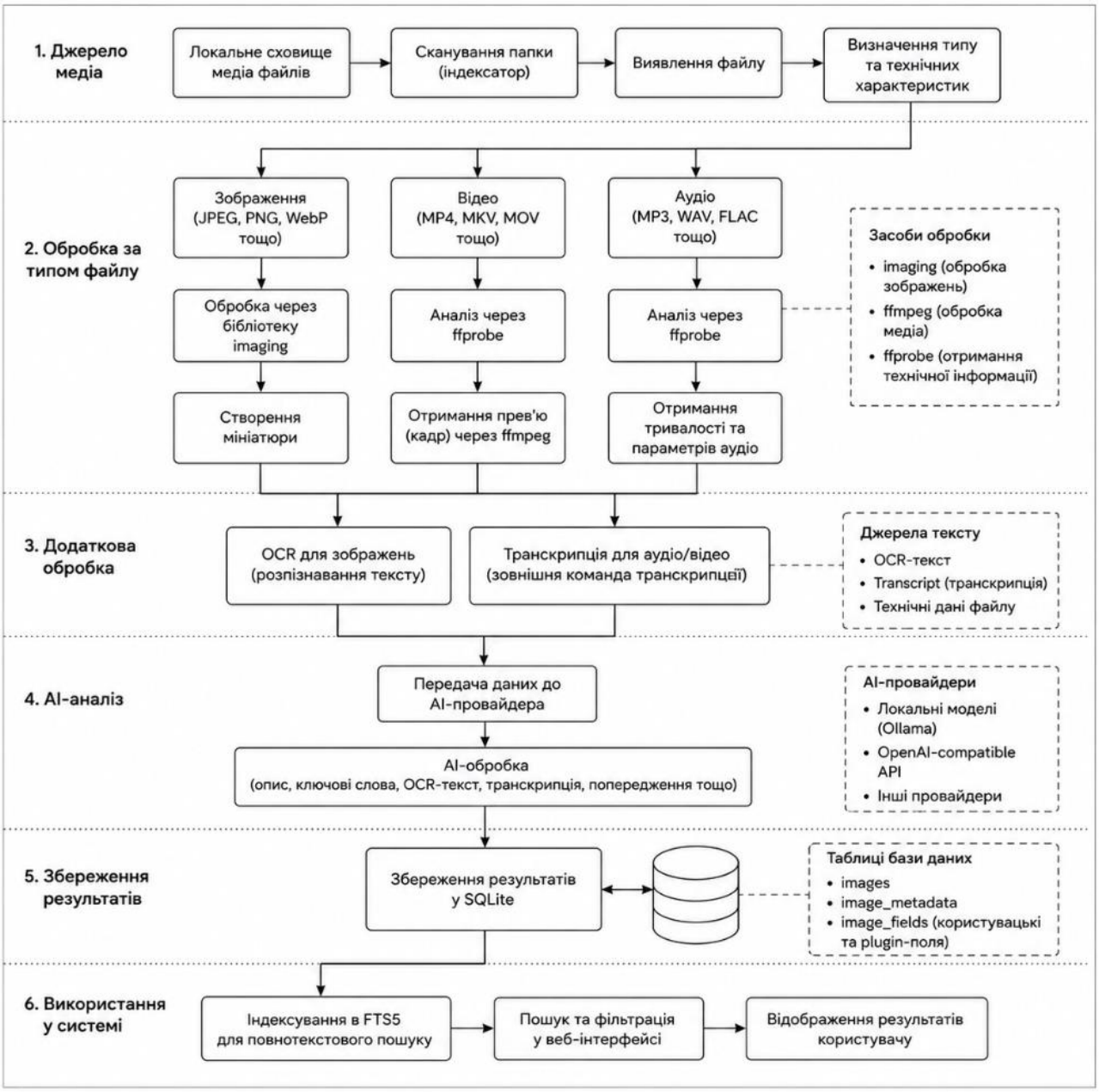


Рис. 2.5 Процес обробки медіа даних та формування AI-метаданих у системі megadrive

Таким чином, засоби обробки медіа даних дозволяють системі створити з простих файлів структурований і змістовий архів. Використання бібліотеки imaging, ffmpeg, ffprobe, зовнішніх команд транскрипції та встановленні AI-провайдерів дозволяє створювати систему, яка здатна вимірювати всі характеристики відео, аудіо, зображень, формувати їх описи, ключові слова, OCR-текст, транскрипції. Такий засіб забезпечує високу якість пошуку та виробляє комфортну роботу з великими колекціями медіа файлів.

2.6 Засоби автентифікації та захисту доступу

Ключі API використовуються у форматі mgd_.... Відкриття ключа відбувається лише один раз і в базі даних зберігається не сам ключ, а його SHA-256 хеш. При кожному вході до API система хешує введений ключ і порівнює його із запам’ятованим хешем. Такий спосіб дозволяє зменшити можливості випадку отримання інформації про API ключі випадках з отриманням доступу до бази даних.

Для API-ключів в системі встановлені службові параметри. В базі даних можна зберігати назву ключа, дату створення ключа, дату останнього використання, строк дії ключа, статус відкликання ключа. Збереження інформації дозволяє користувачу керувати самими ключами системи: створенням, переглядом, відкликанням. Параметр last_used дозволяє знати про використання ключа, а параметр expires_at створює обмеження до його дії.

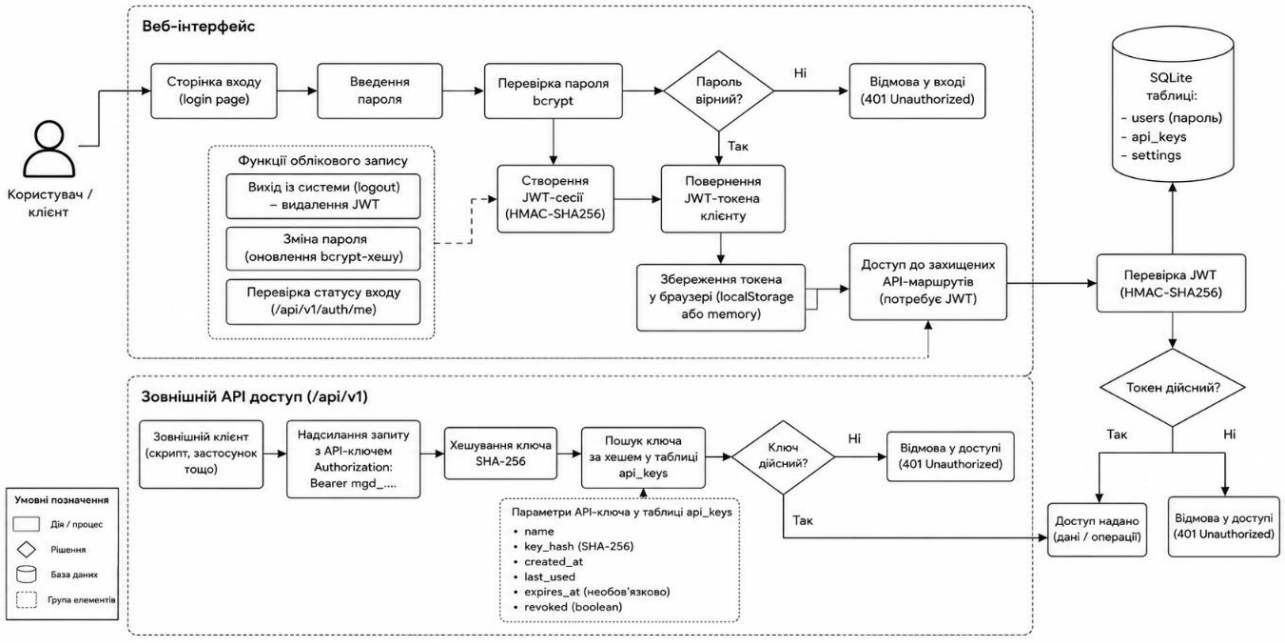


Рис. 2.6 Механізм автентифікації та захисту доступу в системі megadrive

3 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Загальна архітектура системи

Загальна архітектура системи megadrive побудована за принципом self-hosted веб-застосунку. Серверна частина, клієнтський інтерфейс, база даних, модулі індексації а також засоби обробки медіа даних – все це працює як єдина програмна система. Основною ідеєю при виконанні системи є можливість розгорнути її локально або на власному сервері, підключити власні медіа сховища та отримати до них доступ в веб-інтерфейсі без використання сторонніх хмарних сервісів [4; 9].

Система megadrive складається з кількох основних компонентів – клієнтської та серверної частини, локальної бази даних, модуля індексації, модуля пошуку, модуля AI-аналізу, модуля автентифікації, системи подій SSE та зовнішніх інструментів для роботи з медіа файлами. Кожен із цих компонентів виконує окрему функцію, а все система взаємодіє між собою через серверну частину.

Клієнтська частина реалізована у вигляді React SPA. У веб-браузері відображається галерея, рядок для пошуку, бічна панель сховищ, модальне вікно з деталями файлу, панель статусу індексації, налаштувань та тем оформлення. Взаємодія користувача з системою проходить через веб-браузер, а всі дії відправляються до серверної частини у вигляді HTTP-запитів. Такий підхід дозволяє відокремити логіку відображення від серверної логіки обробки даних [17].

Серверна частина є центральним елементом системи. Вона здійснюється на мові Go, вона відповідає за маршрутизацію запитів, роботу з API, доступ до бази даних, сканування локальних сховищ, індексацією файлів, створення мініатюр, формування метаданих, автентифікацію користувача та передачу подій до клієнтської частини. Для організації HTTP-маршрутів використовується go-chi/chi/v5 підсистема, що дозволяє структурувати API-ендпоінти за окремими функціональними групами [16; 20].

Однією з важливих особливостей архітектури є відділення клієнтського інтерфейсу після збірки – він зберігається у директорії internal/webui/dist а вбудовується в серверний застосунок через механізм Go embed. Звідси megadrive може вироблятися як єдиний виконуваний файл або єдиний серверний застосунок, який одночасно містить і backend, і frontend. Це спрощує запуск системи та відповідає ідеї self-hosted рішення.

Для зберігання даних використовується SQLite. База даних зберігає інформацію про підключені сховища, самі медіа файли та їх технічні характеристики відповідно також як та інші відомості – AI-метадані, користувацькі поля, теми оформлення, налаштування та API-ключі (з міграціями схеми). Така база добре підходить для локального застосунку, оскільки не потребує окремого серверного процесу та зберігає всі дані у файлі [21]. Для ведення пошуку за текстовими даними використовується SQLite FTS5, що дозволяє індексувати такі відомості, як назви файлів, описи, OCR-текст та транскрипції та ключові слова [14].

Модуль індексації відповідатиме за скануванням локальних сховищ. Користувач додає папку у веб-інтерфейсі після чого сервер запускає процес сканування. Індексатор відображається по файлам, визначає їх тип та технічні характеристики, zapisyє дані в базу даних та передає файли на подальшу обробку. Так само для зображень створюється мініатюра, для відео та аудіо – ffmpeg та ffprobe, а для змістового аналізу – AI-провайдери або зовнішня команда транскрипції [15].

Модуль AI-аналізу призначений для створення змістовних метаданих. Він здійснює створення опису файлу, ключових слів, OCR тексту, транскрипції та службових попередження. Архітектура AI-модуля побудована через інтерфейси – система не прив’язана до одного випадкового провайдера. Це дозволяє використовувати локальні моделі через Ollama або OpenAI-compatible API. Такий підхід робить систему гнучкою і підготовленою до подальшого розширення.

Модуль пошуку базується на даних основних таблиць бази даних та FTS5 індексу. Користувач виконує вільний текстовий пошук, а також використовує фільтри за тегами, сховищами та MIME типами. Пошук охоплює таку інформацію

як назви файлів, описи, OCR-текст та транскрипції та ключові слова. Це дозволяє знаходити медіа файли не лише за технічними параметрами, а й за змістом.

Система автентифікації забезпечує захист системи та інтерфейсу. Для входу використовується пароль, що зберігається у вигляді bcrypt-хешу. Після успішного входу створюється JWT-сесія, що підписана за допомогою HMAC-SHA256 [23; 24]. Для зовнішнього API використовуються API-ключі формату mgd_..., які також зберігаються в базі даних у вигляді SHA-256 хешу. Це дозволяє використовувати систему вже не лише через веб-інтерфейс, а й через зовнішні інтеграції.

Окремим елементом архітектури є система SSE-подій. Ідея такої системи – передача оновлень стану від сервера до клієнта. Наприклад, під час сканування сховища або індексації медіа відбувається надсилання подій типу media_indexed, store_changed, scan_status, config_changed, theme_changed і heartbeat. Звідси відображений веб-інтерфейс може оновлювати стан галереї, індексації або налаштувань без перезавантаження сторінки.

Загальна логіка роботи системи передбачає наступну послідовність. Користувач проходить автентифікацію у веб-інтерфейсі. Після чого додає до системи локальне сховище медіа файлів. Серверна частина індексує файли з метадані, створює мініатюри, засвоює технічні характеристики файлів і їх метадані в базу SQLite. Далі відбувається перегляд файлів у галереї, виконується пошук, можлива редагування користувацьких полів або запуск AI-індексації в разі подальшого розширення можливостей.

3.2 Структура програмних модулів

Структура програмних модулів системи megadrive побудована таким чином, щоб відділити окремі модулі від окремих функцій. Хоча програмне забезпечення розроблене як монолітний сервер на інструменті Go, внутрішній код системи є досить структурованим. Кожен модуль системи відповідає за окремий елемент роботи системи: API, автентифікація, API-ключі, налаштування, база даних, події,

користувацькі поля, індексація, медіа, галерея, AI-аналіз і вбудований веб-інтерфейс.

Така структура дозволяє зберегти простоту розгортання приладу, але і знову дозволяє розширити окремі компоненти без змін всієї системи. Наприклад, можна додати новий провайдер AI-технологій, змінити тип сховищ чи додати новий фільтр під час виконання пошуку.

Центральним модулем системи є модуль API. Він забезпечує приймання запитів від інтерфейсу, виконання необхідних дій і відправку результатів. Через API можна виконувати дії зі сховищами, медіа файлами, виконувати пошук, змінювати налаштування, тему, виконувати операції з автентифікацією та API-ключами. Router для API використовується бібліотека go-chi/chi/v5 [16].

Модуль автентифікації відповідає за авторизацію користувачів. Він забезпечує перевірку паролів, створення сесій, вихід із системи, зміну паролів. Для захисту паролів використовується бібліотека bcrypt, а для створення JWT-токенів використовується HMAC-SHA256 [23; 24].

Модуль API-ключів відповідає за створення API-ключів для зовнішніх програм і автоматизацій. Ключі формату mgd_... використовуються для здійснення запитів до внутрішнього API системи. В модулі також відбувається створення хеша від ключа SHA-256, збереження данних у базі даних, перевірка ключа та його відкликання [11].

Модуль налаштувань відповідає за зчитування налаштувань системи. Для megadrive такі налаштування виглядають через файли формату TOML, базу даних та змінні середовища. Основний файл налаштувань знаходиться за шляхом ~/.config/megadrive/config.toml, а директорія даних за замовчуванням міститься за шляхом ~/.local/share/megadrive. Змінні середовища мають вищеї пріоритет від бази даних, тому їх не можна редагувати у веб-інтерфейсі.

Модуль бази даних відповідає за взаємодію з базою даних SQLite. Він відкриває базу даних, виконує міграції, виконує запити до таблиці, активує режим WAL, встановлює зовнішні ключі. База даних SQLite була вибрана через її автономність та зручність для застосунків. Для пошуку медіа файлів в базі

використовується індекс FTS5, що дозволяє виконувати повнотекстовий пошук за описами, OCR-текстом, транскрипціями, ключовими словами та назвами файлів [14; 21].

Модуль зберігання відповідає за взаємодію зі сховищами, що встановлені на комп'ютері користувача. У модулі зберігається інформація про сховища, що додаються до системи, відновлення списку сховищ, їх видалення чи запуск повторного сканування сховищ. Також модуль може відновити кількість медіа файлів в кожному сховищі.

Модуль відновлення відповідає за відновлення медіа файлів і додання їх до бази даних. Він пройде по кожному файлу в сховищі встановленого користувачем, визначить, що тип файлу прийнятий до відновлення системою, зчитає його технічні дані та додає до бази даних.

Модуль медіа відповідає за зчитування технічних параметрів медіа файлів. Він визначає, що тип файлу (медіа, документ, архів тощо), читує його MIME типи, технічні параметри, тривалість, розміри відео чи зображень. Для роботи зі відео і аудіофайлами використовується програми ffmpeg та ffprobe [15].

Модуль мініатюр відповідає за створення мініатюр від медіа файлів. Для зображень використовується бібліотека imaging, а для відео – кадр, створений за допомогою ffmpeg. Мініатюри створюються для відображення медіа файлів в галереї, оскільки завантаження великих файлів може викликати відкликання інтерфейсу.

Модуль AI відповідає за обробку медіа файлів через технології AI. Він може використовувати різні провайдери, включаючи локальні моделі, встановлені через Ollama або OpenAI-compatible API. Опис файлів, визначення ключових слів, відтворення тексту OCR, відтворення транскрипції аудіо або відео вказуються як результати AI. Дані зберігаються в базі даних.

Модуль пошуку відповідає за виконання пошуку медіа файлів. Він аналізує запит від інтерфейсу, виконує фільтрацію за вказаними тегами, сховищами та видами медіа, виконує запит в основній база даних та індекс FTS5. У системі використовується три стратегії пошуку FTS5 prefix query, LIKE fallback, FTS5

trigram fallback, що дозволяє знаходити медіа файл із неповним або неточними запитами до системи [14].

Модуль полів призначений для роботи з користувацькими полями та полями, що встановлені плагінами. Користувацькі поля можуть використовуватися для збереження додаткової інформації про медіа файли, а плагінові поля можуть створюватись автоматично, але відображатись для читання.

Модуль подій відповідає за передачу подій від серверної частини в клієнтську частину. Завдяки використанню технології Server-Sent Events сервер може відправляти повідомлення про виконання операцій в системі, які не вимагають відновлення сторінки в інтерфейсі.

Модуль веб-інтерфейсу відповідає за розміщення веб-інтерфейсу, що відновлюється бібліотекою React. Відновлення відбувається після компіляції і розміщення у директорії internal/webui/dist в Go сервера. Це дозволяє запускати систему як єдиний сервер.

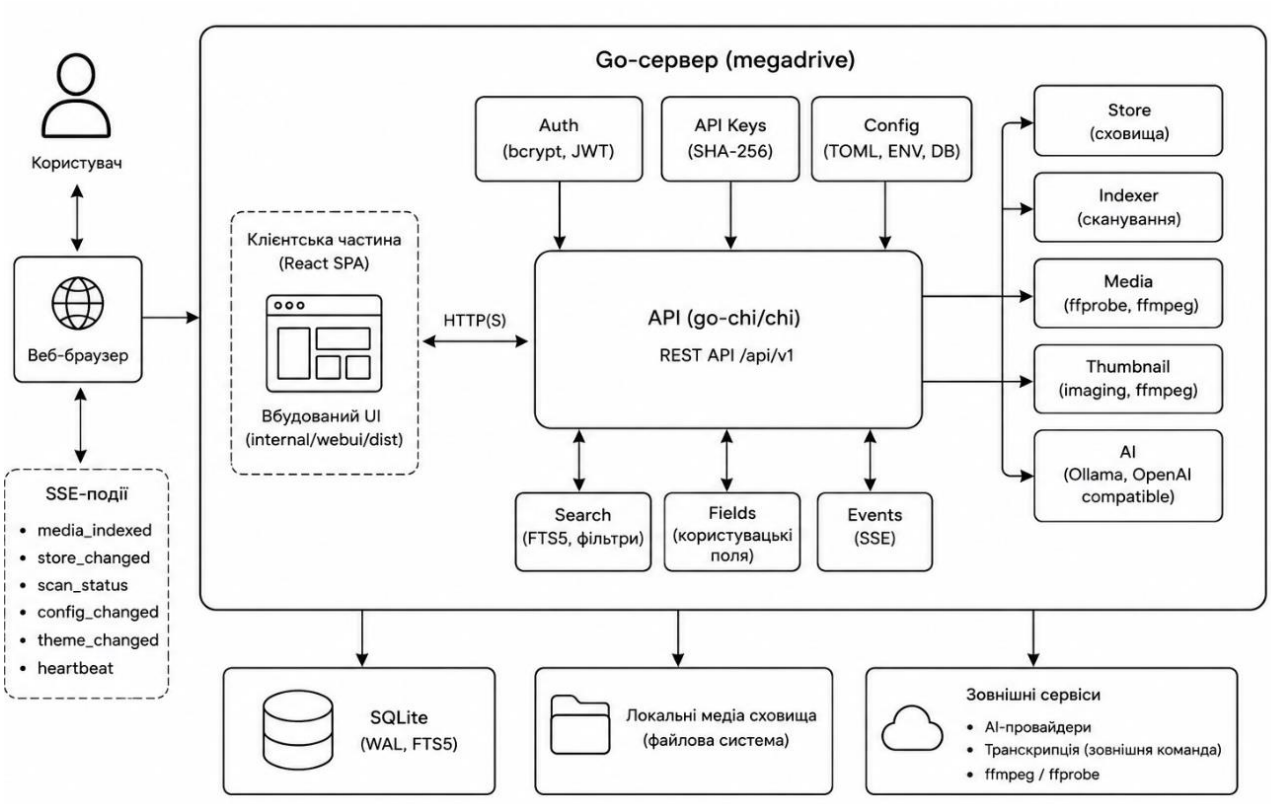


Рис. 3.2 Структура програмних модулів системи megadrive

3.3 Проєктування бази даних

Проєктування бази даних для системи megadrive вироблене з урахуванням її основних задач. До основних задач програмного забезпечення належить зберігання інформації про локальні сховища, власні медіа файли, метадані до них, поля юзерів, налаштування системи, теми оформлення і API-ключа. Оскільки система self-hosted, база даних має бути автономною, простішою у розгортанні і гнучкою у випадку роботи з великими медіа архівами.

Для зберігання даних у системі використовується SQLite. База даних не вимагає окремого сервера, а її дані зберігаються у файлі. Відомо, що дана технологія відмінно підходить для локальних розширення [21]. В SQLite використовується також режим WAL, зовнішній ключ і повнотекстовий пошук FTS5. Такий вибір технології дозволяє базі даних одночасно отримувати і записувати дані для веб-інтерфейсу.

Таблиця stores використовується для зберігання інформації про локальні сховища медіа файлів. Кожне сховище відповідає одній папці медіа файлів, які додаються до системи через веб-інтерфейс. До інформації про кожне сховище можна зберігати такі поля, як назва, шлях до директорії, тип провайдера, статус сканування, дата створення і оновлення. Таблиця stores зв’язана з таблицею images відношенням один-до-багатьох.

Таблиця images використовується для зберігання основних даних про медіа файли. До полів таблиці відносяться такі дані, як ідентифікатор файлу, ідентифікатор сховища, назва файлу, шлях до файлу, хеш, MIME-тип, тип медіа, розмір, ширина, висота, тривалість, статус індексації, створення файлу і його оновлення. Значення таблиці використовуються не лише для зображень, а й для відео і аудіо файли.

Таблиця image_metadata використовується для зберігання змістових метаданих про файли, що визначаються шляхом виконання алгоритмів AI. У таблиці можна зберігати такі інформаційні поля, як опис про медіа файл, текст, отриманий шляхом виконання OCR алгоритмів, транскрипція відео файлу, ключові

слова, та можливі попередження. Таке поле зв’язане з таблицею images відношенням один-до-одного.

Таблиця image_fields використовується для зберігання додаткових полів щодо медіа файлу. Вони додаються до основних полів метаданих. Наприклад, можна додати нотатку до медіа файлу. До таблиці можна також додавати поля, що створені зовнішніми додатками (plugin-sourced fields), які створюються зовнішнім джерелом і доступні лише для читання.

Таблиця themes використовується для зберігання тем відображення веб-інтерфейсу. Відомо, що існують вбудовані теми dark-purple, light-purple і можливість додавання власних тем, які зберігаються у форматі JSON. Кожна тема є об’єктом, у якому можна зберігати такі поля, як назва теми, вигляд теми у вигляді JSON, вказівка на активність теми, вказівка на вбудовану тему та створення теми.

Таблиця settings використовується для зберігання налаштувань системи. Вони визначають такі параметри, як використання провайдерів AI, кількість worker-процесів, що виконують індексацію медіа файлів, режим роботи системи у локальному режимі, параметри визначення транскрипції та інші параметри. Один із параметрів вказує на те, що налаштування було встановлене через змінну середовища.

Таблиця api_keys використовується для зберігання інформації про API-ключі. У базі даних зберігається унікальний хеш відкритого ключа до API. До інших полів належить таке визначення, як назва ключа, час створення та оновлення ключа, строк дії ключа, вказівка на відкликання ключа. Відкритий ключ у базі даних не зберігається, щоб забезпечити його безпечне використання [14].

Таблиця schema_migrations використовується для відстеження версій бази даних. У таблиці записуються дані про вже встановлені міграції схеми бази даних. Така таблиця дозволяє додавати нові відокремлення до бази даних, що добрістіється пізнішими версіями програмного забезпечення.

Структура бази даних системи megadrive визначена в таблиці 3.1.

Таблиця 3.1

Основні таблиці бази даних системи megadrive

Таблиця	Призначення
stores	Зберігання інформації про локальні сховища медіа файлів
images	Зберігання основної інформації про медіа файли
image_metadata	Зберігання опису, OCR-тексту, транскрипції, ключових слів і попереджень
image_fields	Зберігання користувацьких і plugin-sourced полів
themes	Зберігання тем оформлення інтерфейсу
settings	Зберігання параметрів конфігурації системи
api_keys	Зберігання хешів API-ключів і службової інформації про них
schema_migrations	Контроль застосованих міграцій бази даних
image_fts	Повнотекстовий пошук за медіа метаданими

Таблиці бази даних встановлені зв’язки. Table stores і таблиця images зв’язані через поле сховища. Table images зв’язана із таблицею image_metadata та таблицею image_fields через ідентифікатор медіа файлу. Останні таблиці (themes, settings, api_keys, schema_migrations) вважаються службовими та не мають взаємодії відносно медіа файлів.

Відокремлення таблиць можливе використовуючи зовнішні ключі. У випадку видалення сховища з бази даних, відповідні файли повинні видалятися. Відповідно і у випадку видалення медіа файлу воно відноситься до видалення метаданих.

Структура бази даних має бути здатною до збереження медіа файлів, а також до пошуку медіа файлів. У разі внесення змін у метадані медіа файлів, їх потрібно додати до таблиці image_fts. У разі внесення опису до медіа файлу, він повинен бути доступний для пошуку. Це дозволяє швидко знаходиться медіа файли за метаданими.

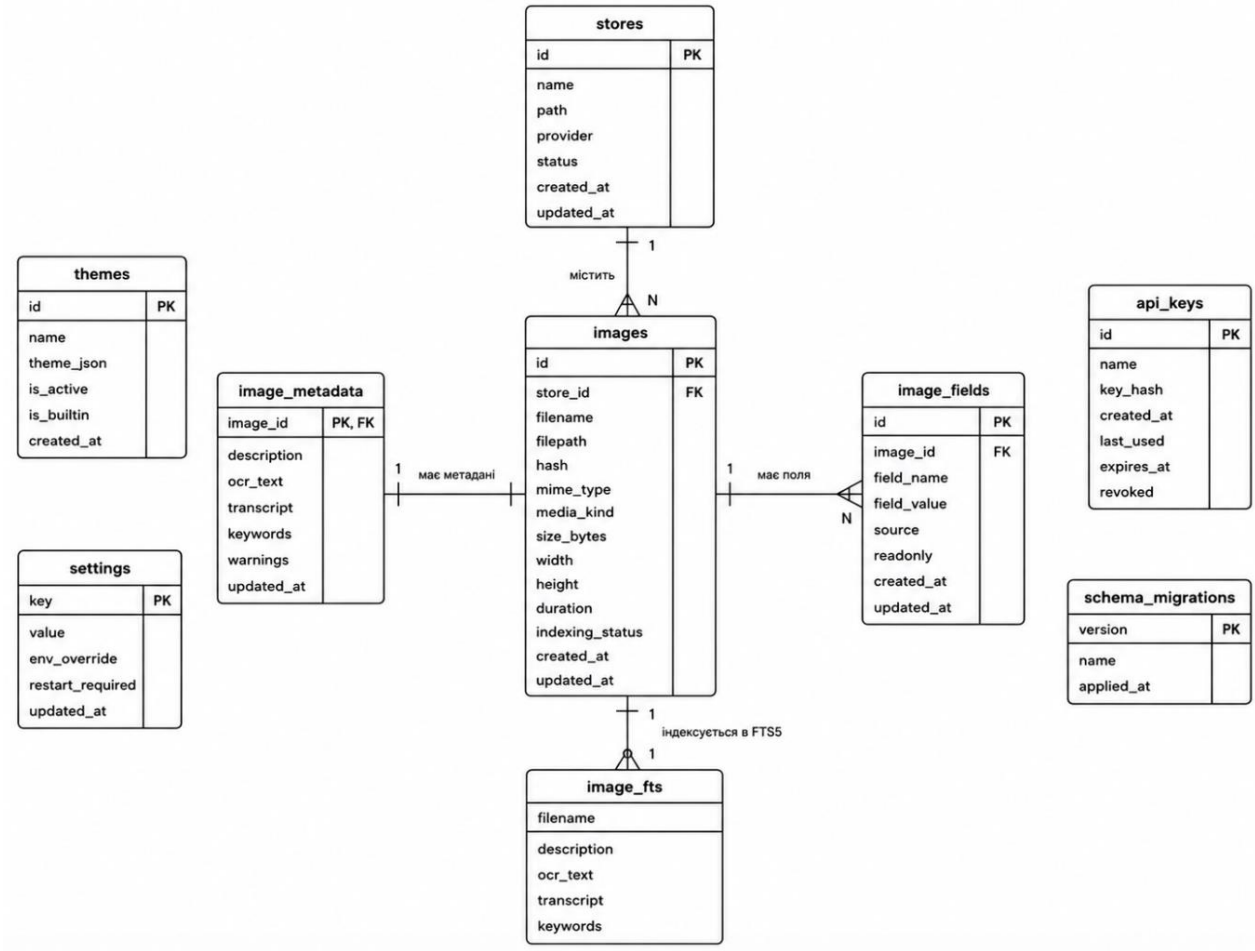


Рис. 3.3 Схема бази даних системи megadrive

3.4 Проєктування механізму індексації медіа файлів

Механізм індексації медіафайлів є одним із ключових процесів самої системи megadrive – саме він відповідає за перетворення локального набору файлів у медіаархів. Збереження файлів просто у папках не забезпечить системі ефективної роботи з такими функціями, як пошук, фільтрація, перегляд галереї або AI-аналіз. Тож під час проектування системи необхідно визначити послідовність дій у рамках того, як сама система знаходить файли, визначає їхні властивості, створює записи в базі даних і готує їх до подальшого аналізу.

Процес індексації в самій системі megadrive починається з додавання локального сховища. Користувач визначає папку для медіафайлів. Така інформація зберігається в таблиці stores. Для кожного сховища фіксуються такі параметри, як назва, шлях до директорії, тип провайдера та статус сканування/службові дати.

Після додавання сховища користувач може його сканувати або виконувати повторне сканування у разі появи нових файлів у сховищі.

Основним компонентом, що відповідає за цей процес, є модуль indexer. Завдання цього модуля – проходження по файловій структурі сховища, пошук медіафайлів, перевірка їхнього типу та передача даних далі в модулі системи. Індексатор в обов'язковому порядку повинен відокремлювати потрібні файли від непотрібних і службових у базі знань. До основних типів медіа належать зображення, відео та аудіофайли.

На першому етапі індексатор здійснює сканування директорії. У процесі проходження по всіх файлах виділеного сховища він визначає, до якого з підтримуваних типів належить файл. Така перевірка може проводитися за розширенням файлу, MIME-типом або вмістом. При виявленні файлу, який не є медіа-об'єктом, він проходить далі. У разі включення підтримки такого файлу створюється базовий набір даних для нього: шлях, ім'я, розмір, хеш і дата модифікації, пов'язані зі сховищем.

Наступним етапом є визначення технічних характеристик медіаоб'єкта. У випадку зображень важливі ширина, висота, формат і MIME-тип. Для відео додатково візуалізується тривалість відео, параметри відеопотоку та формування превью. Для аудіо важливі тривалість, формат і технічні характеристики аудіопотоку. Для аналізу відео та аудіо можна вбудувати зовнішні засоби ffmpeg та ffprobe, призначені для роботи з багатьма медіафайлами [15].

Після визначення основних характеристик відбувається перевірка існування такого файлу вже в базі даних. Така перевірка може здійснюватися за шляхом до файлу, хешем або комбінованими параметрами. У разі, якщо файл раніше вже був проіндексований і не був змінений, процес можна пропустити. У разі, якщо файл є новим або був змінений, запис у таблиці images створюється або оновлюється.

Важливою частиною індексації є створення мініатюр. Мініатюри дозволяють швидко відображати файли в галереї без необхідності завантаження повноцінних оригіналів. Мініатюри зображень можуть створюватися за допомогою бібліотеки imaging – спеціально для зміни розміру та підготовки до відображення в інтерфейсі.

Для відео мініатюри можуть формуватися на основі окремого кадру, доступного через ffmpeg [15]. Аудіофайли в інтерфейсі можна відображати спеціальною позначкою або стандартним типом превью.

Після базової індексації файли можуть бути поміщені в чергу обробки AI-технологіями. Така обробка необхідна для формування змістовних метаданих – опису, ключових слів, візуального тексту (OCR) або транскрипції. Така обробка може виконуватися не відразу, а у фоновому режимі через тривалий аналіз великої кількості медіафайлів. Для цього в системі передбачені статуси індексації та черг обробки, які відображаються у веб-інтерфейсі.

Загалом послідовність індексації може бути представлена так: користувач додає склад і запускає сканування, індексатор знаходить медіафайли та визначає їх тип і технічні характеристики, створює або оновлює записи в базі даних SQLite, формує мініатюри, ставить файли в чергу на обробку AI-технологіями, оновлює індекс FTS5 і передає події процесу клієнтському інтерфейсу. Пізніше файли стають доступними для пошуку, перегляду та фільтрації.

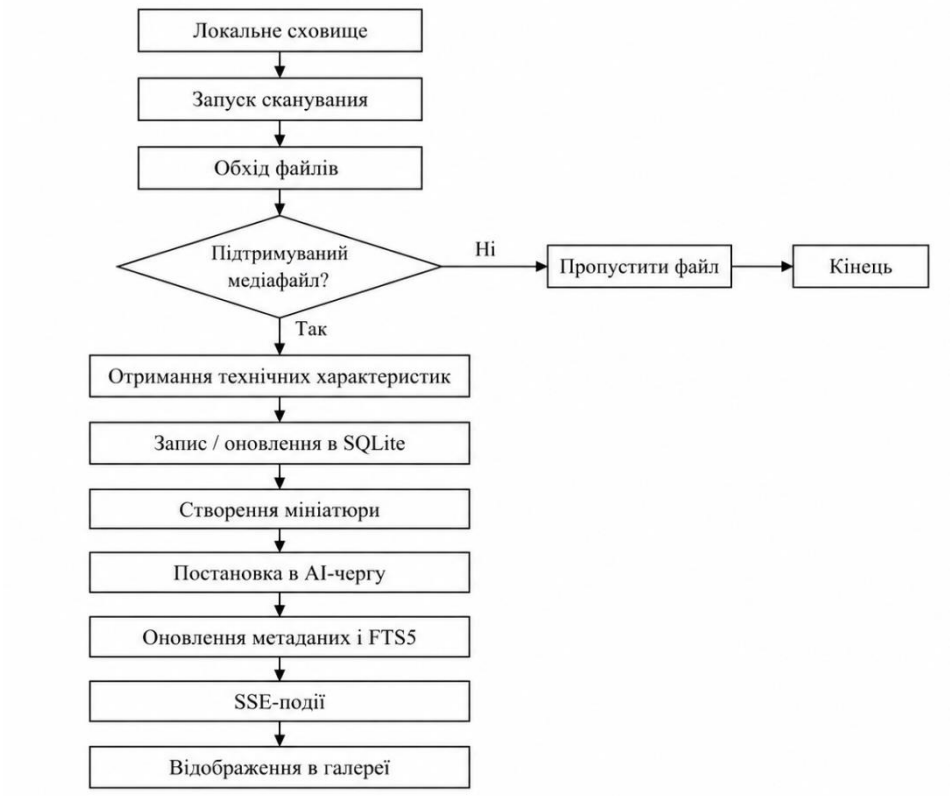


Рис. 3.4 Механізм індексації медіафайлів у системі megadrive

4 ПРОГРАМНА РЕАЛІЗАЦІЯ SELF-HOSTED РІШЕННЯ MEGADRIVE

4.1 Загальна структура реалізації програмного забезпечення

Програмна реалізація системи megadrive виконана у вигляді self-hosted веб-застосунку, що включає у себе серверну частину, клієнтський інтерфейс, базу даних, механізм власного індексації медіа файлів, вбудований пошук, AI-обробку, автентифікацію та випуск відкритого API. Основною метою створення такого застосунку є його можливість працювати локально чи на власному сервері користувача, без залежності від хмарної інфраструктури від провайдера програмного забезпечення.

Система megadrive в реалізації побудована за монолітною архітектурою. Всі компоненти – сервер, API, індексація, пошук, автентифікація, бід бази даних і клієнтський інтерфейс в одному запуску від відокремленого Go-застосунку. Це спрощує розгортання системи, оскільки для її запуску не потрібно запускати всі компоненти окремо.

Серверна частина в реалізації виконана мовою Go 1.26.1. У складі серверної частини вперше створено маршрутизацію відповідно до запитів клієнтського інтерфейсу, інформацію з локальної файлової системи, індексацію медіа файлів, створення мініатюр відповідно до кожного з файлів, роботу з базою даних, створення AI-метаданих, автентифікацію користувача, перевірку API-ключів і передачу даних в клієнтський інтерфейс. Маршрутизація відбувалася використовуючи бібліотеку go-chi/chi/v5, а також middleware для логування та відновлення запитів [16; 20].

Клієнтський інтерфейс в реалізації створено на React 19, TypeScript 6, Vite 8 та TailwindCSS 4. Він як односторінковий інтерфейс, що надає можливості користувачу переглядати всі файли, відновлювати за вказаними відомостями, здійснювати пошук, відновлювати, переглядати, змінювати налаштування, тему відображення та встановлювати API-ключі [17; 18; 19; 22]. Відео встановлено в

директорію internal/webui/dist, що дозволяє його вбудовувати в серверний запуск через Go embed. Таким чином саме виконання серверного застосунку встановить відеоінтерфейс.

В системі використовується база даних SQLite через pure-Go драйвер modernc.org/sqlite. Такий спосіб дозволяє складати програму без використання CGO, а отже без залежності від системних бібліотек. У базі даних зберігатимуться дані про медіа файли, сховища, метадані, поля користувача, тему відображення, налаштування та API-ключі. Для збереження даних в базі використовується режим WAL з базою даних, що підтримує зовнішні ключі [21].

В системі використовується поле в пошуку на основі SQLite FTS5. У базі є поле індексації, що вказується для медіа файлів, опису, OCR-тексту, транскрипції та іншого. Провідний пошук виконується за допомогою вільних запитів, фільтрації за вибраними тегами, сховищами та типами файлів, використовуючи FTS5 prefix query, LIKE fallback і FTS5 trigram fallback [14].

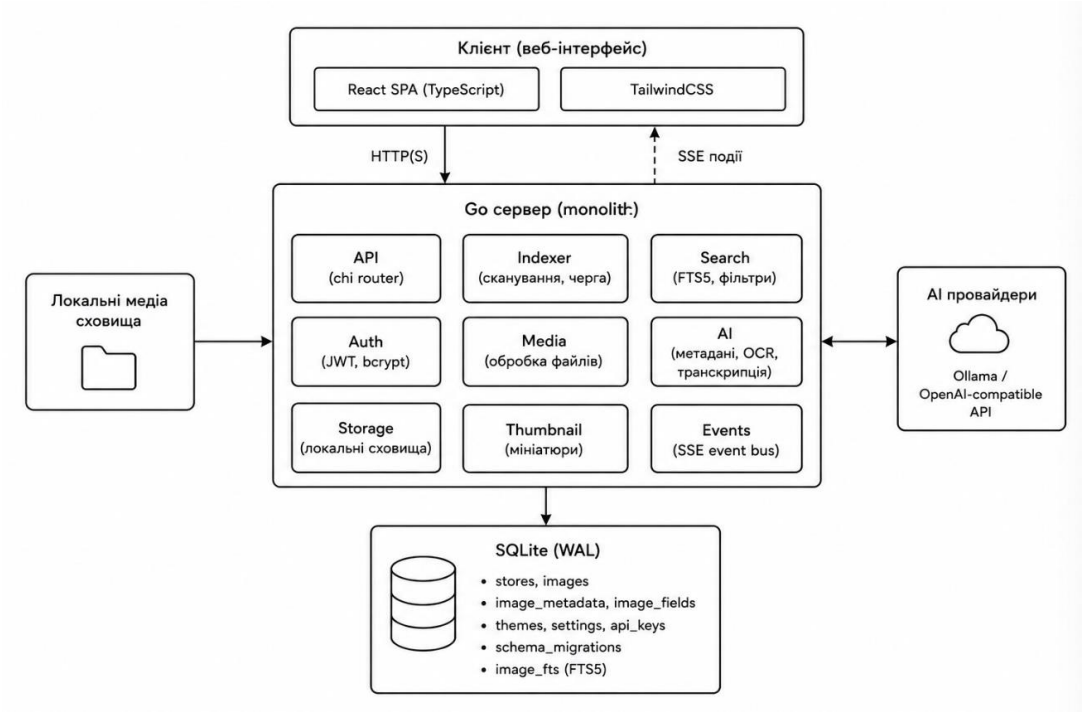


Рис. 4.1 Структура реалізації програмного забезпечення megadrive

Для роботи з медіа файлами в системі є окремі модулі. Зображення в бібліотеці github.com/disintegration/imaging, а також використовуються вбудовані

бібліотеки Go для декодера та x/image/webp. Відео та аудіо-файли передбачає інтеграцію з ffmpeg та ffprobe для дослідження медіа файлів. Вони використовуються для вивчення технічної інформації про файли відео, аудіо, визначення тривалості, відео та аудіопараметрів, а також створення прев'ю для відеофайлів [15]. Для транскрипції відео та аудіофайлів в системі передбачена зовнішня команда для їх виконання, що задається в конфігураційному файлі.

4.2 Реалізація серверної частини системи

Серверна частина системи Megadrive реалізована на мові програмування Go 1.26.1. У рамках серверної частини системи реалізовано механізм обробки запитів від клієнтського інтерфейсу, маршрутизації до API, взаємодії з базою даних SQLite, індексації локальних медіафайлів, формування мініатюр, вставки метаданих AI, аутентифікації користувача, перевірки API-ключів та передачі даних клієнтському інтерфейсу.

Пакетів у серверній частині системи дозволили надалі вести роздільну розробку функціональних компонентів системи. Основними пакетами додатка є: api, auth, apikey, config, db, events, fields, indexer, media, search, storage, store, thumbnail, ai та webui.

Пакет api відповідає за маршрутизацію запитів від клієнтського інтерфейсу та саму роботу з клієнтським інтерфейсом. У пакеті реалізовано маршрутизацію до сховищ, до медіафайлів, до пошуку, налаштувань системи, тем, аутентифікації, ключів API та зовнішнього API. Для маршрутизації використовується бібліотека go-chi/chi/v5 [16].

Пакет auth відповідає за аутентифікацію користувача. Для роботи з паролем використовується бібліотека bcrypt, а для роботи сесій використовується JWT-токен, підпис якого ведеться за допомогою HMAC-SHA256 [23; 24].

Пакет apikey відповідає за роботу з ключами API. Ключі формуються у форматі mgd_..., однак у базі даних зберігається лише хеш ключа від SHA-256. Цей ключ дозволяє зовнішнім клієнтам отримувати доступ до API /api/v1. Також ключі

можна фіксувати за датою створення, датою останнього використання ключа, терміном дії ключа та статусом скасування ключа.

Пакет config відповідає за налаштування системи. Налаштування завантажуються з TOML-файлу, з бази даних та зі змінних середовища. Змінні середовища мають пріоритет над іншими параметрами, тому зміни, внесені через env overrides, не можна буде змінювати в інтерфейсі.

Пакет db відповідає за роботу з базою даних SQLite. У системі використовується бібліотека modernc.org/sqlite для доступу до бази даних SQLite. Режим роботи бази даних – WAL, а також включені зовнішні ключі. У базі зберігається інформація про сховища, медіафайли, метадані, користувацькі поля, теми та налаштування. Для повноти текстового пошуку використовується таблиця FTS5 – image_fts [14; 21].

Пакет store відповідає за логіку роботи зі сховищами. Робота зі сховищами включає додавання локальних папок, отримання списку сховищ, запуск сканування сховища, видалення сховища та підрахунок кількості медіафайлів. Пакет storage в рамках пакета store відповідає за абстракцію сховищ медіафайлів. На даний момент єдиним сховищем є локальна файлова система, однак абстракція дозволяє в майбутньому включити інші сховища медіафайлів.

Пакет indexer відповідає за індексацію медіафайлів. Пакет проходить по сховищу та сканує його на наявність медіафайлів, отримує інформацію про медіафайли та вставляє її в базу даних, створює мініатюри та відправляє файли в чергу на обробку AI.

Пакет ai в системі відповідає за роботу з провайдерами штучного інтелекту. Пакет реалізований через інтерфейси, що забезпечує автономність системи від одного провайдера ШІ. У системі можна використовувати локальні моделі з використанням Ollama або відкрити доступ до зовнішніх API, сумісних з OpenAI. Результат роботи інтелектуального забезпечення системи – опис медіафайлу, ключові слова, текст зображення (OCR), транскрипції аудіо- та відеомедіафайлів.

Пакет search відповідає за реалізацію системи пошуку медіафайлів. Можливий пошук за введеним текстом, а також фільтрація медіафайлів за тегами

(tag), сховищами (store) та типом медіа (type). Під час пошуку використовується FTS5 та таблиці бази даних. Реалізовано три стратегії пошуку: FTS5 prefix query, LIKE fallback та FTS5 trigram fallback. Це забезпечує знаходження медіафайлів навіть при неповних і неточних запитах.

Пакет fields відповідає за роботу з користувацькими полями та полями, що ведуться плагінами. Користувацькі поля можуть змінюватися через інтерфейс, поля, що ведуться плагінами, відображаються тільки для читання.

Пакет events відповідає за реалізацію подій через SSE event bus. Пакет забезпечує передачу таких подій клієнтському інтерфейсу: media_indexed, store_changed, scan_status, config_changed, theme_changed та heartbeat. Можливість здійснювати такий обмін подіями надає користувачеві можливість відстежувати зміни в медіа-індексі, сховищах та налаштуваннях у реальному часі.

Діаграма класів (UML) представляє структуру серверного компонента. Оскільки мова програмування Go не підтримує класи, діаграма буде представлена як діаграма структур. Діаграма міститиме компоненти класів (структур): Server, Router, AuthService, APIKeyService, ConfigService, DB, StoreService, IndexerService, MediaService, ThumbnailService, AIService, SearchService, FieldService, EventBus та WebUI. Також доцільно відобразити інтерфейси: StorageProvider та AIProvider, які дозволять у майбутньому включати нових провайдерів медіа та ШІ.

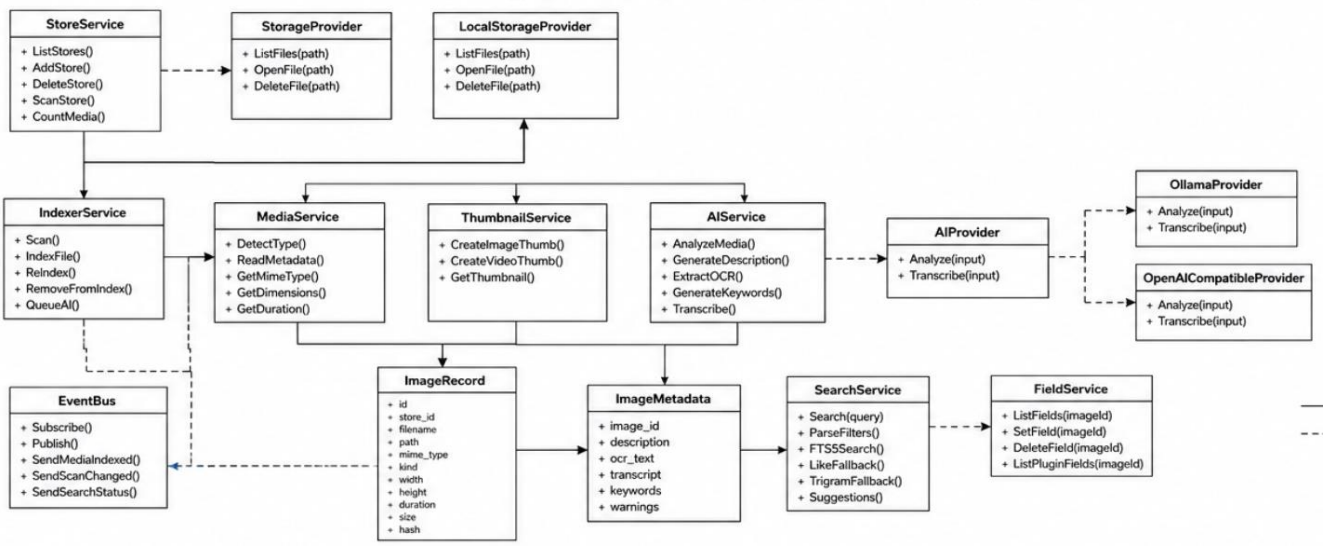


Рис. 4.2 Діаграма класів серверної частини системи megadrive

4.3 Реалізація механізму індексації та обробки медіа файлів

Механізм індексації та обробки медіа файлів є одним із базових процесів, що реалізується у серверній частині системи megadrive. Він забезпечує сканування локальних сховищ, виявлення медіа файлів, визначення їх технічних характеристик, створення мініатюр, збереження інформації у базі даних та приготування файлів до їх AI-аналізу. Саме задіяння цього механізму дозволяє перетворити будь-які локальні файли на медіа-індекси, що можуть бути переглянуті через веб-інтерфейс.

Після додавання сховища користувачем через інтерфейс Store Manager, користувач вказує шлях до сховища. Відділ серверної частини зберігає таку інформацію та передає можливість запуску процесу сканування сховища. Сканування може бути виконане після додавання сховища або відновлено автоматично у разі додавання до сховища нових файлів або в разі зміни файлів.

Такий процес відділено від пакету indexer, який відповідає за обхід сховища, виявлення медіа файлів та передання їх для наступних виконання.

До пакету ongumented відносяться такі механізми як визначення основних характеристик медіа файлів (назва, шлях, розмір, MIME тип, тип медіа, хеш, зв’язок зі сховищчям). Також визначення статусу файлів. Якщо файл виявлений вперше або відновлено після зміни його відомостей, інформація про файл створюється або оновлюється у таблиці images бази даних.

Після встановлення базової індексації медіа файлів він також встановлюється у чергу для виконання AI-обробки. Виконання AI-аналізу не є моментальним, оскільки обробка великої кількості медіа файлів може витрачати більше часу.

Результати AI-аналізу зберігаються у вигляді метаданих. Метадані включають такі поля як description, ocr_text, transcript, keywords та warnings. Порівняння змісту медіа файлів та інформації про медіа файлів з допомогою AI-алгоритмів. Такий підхід зберігається для подальшого відображення у модальному вікні відображення медіа файлів [14].

Основна логіка механізму реалізована з наступними діями: додавання сховища, запуск сканування, обхід файлів, визначення типу медіа файлу, ознайомлення із технічними вимогами файлу, встановлення запису про медіа файл у базу даних, створення мініатюри, встановлення у чергу для AI-аналізу, встановлення метаданих, оновлення FTS5-індексу та відділення SSE-подій до відділеного клієнтського інтерфейсу.

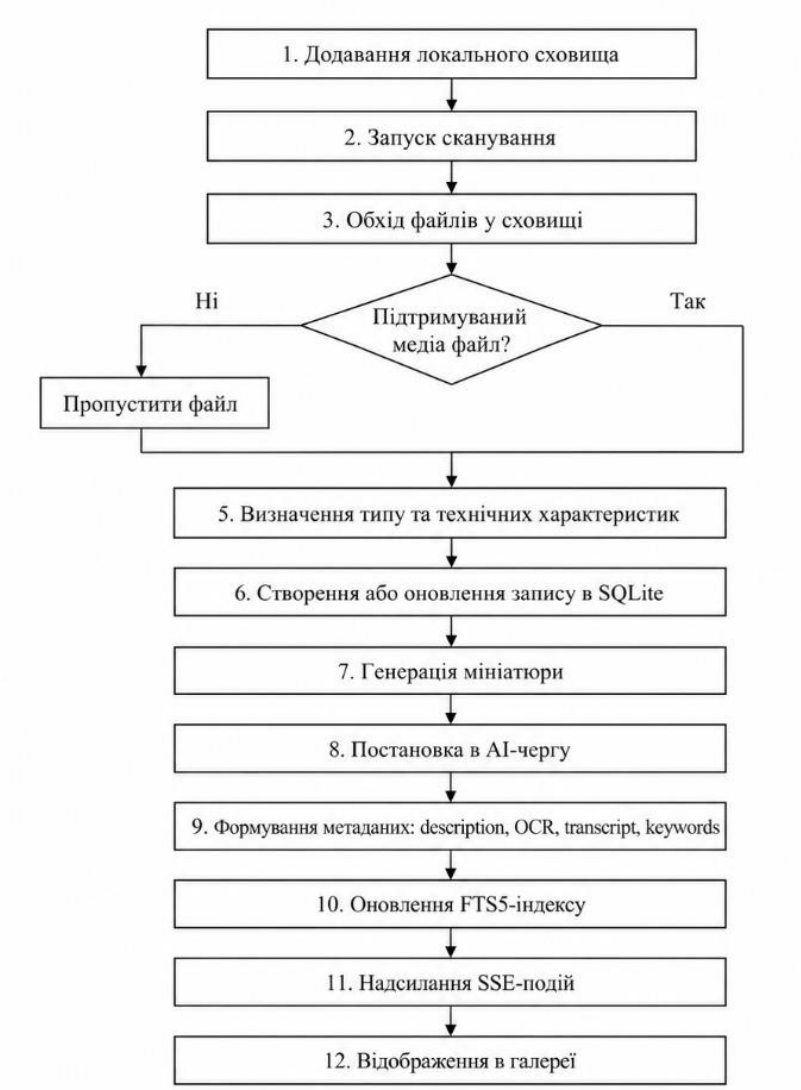


Рис. 4.3 Алгоритм індексації та обробки медіа файлів у системі megadrive

4.4 Реалізація клієнтського інтерфейсу системи

Клієнтський інтерфейс системи megadrive реалізований як односторінковий web-застосунок. Основна задача інтерфейсу – дати користувачу зручний доступ до

локальних медіа сховищ, галереї файлів, пошуку, налаштувань, AI-метаданих і службових дій, пов’язаних з індексацією.

Інтерфейс побудований на React, TypeScript, Vite та TailwindCSS. Такий підхід дозволив зробити сторінку динамічною: користувач може відкривати модальні вікна, перемикати сховища, запускати повторну індексацію, переглядати деталі файлів і змінювати налаштування без повного перезавантаження сторінки.

Основна сторінка застосунку складається з верхньої панелі, області галереї та бічної панелі керування сховищами. У верхній частині розміщено назву системи, пошуковий рядок, кнопки Theme, Settings і Stores. Через пошуковий рядок користувач може виконувати пошук за назвою файлу, описом, ключовими словами, OCR-текстом або транскрипцією.

На сторінці Stores користувач може додати локальну папку, задати назву сховища, вибрати локальний тип сховища та запустити його сканування. Після додавання сховища система показує його у списку, відображає кількість знайдених медіа файлів і надає кнопки Rescan та Remove. Це дозволяє не тільки один раз додати директорію, а й повторно перевіряти її після появи нових файлів.

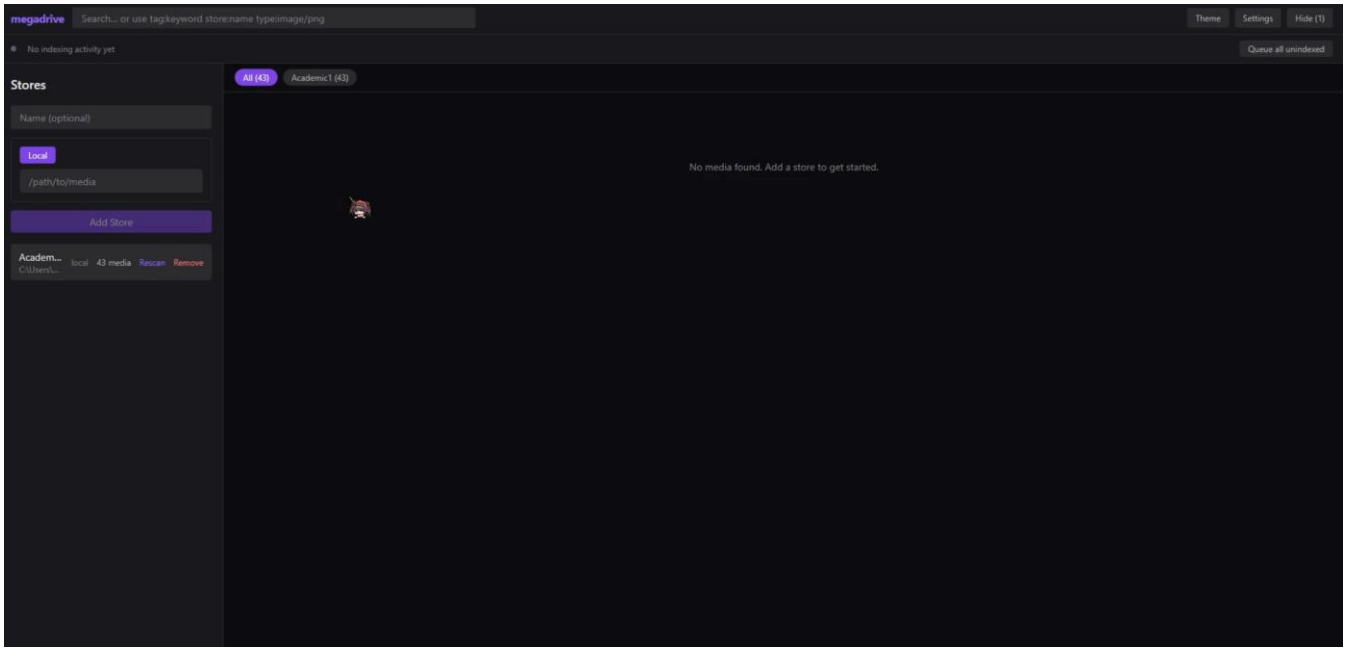


Рис. 4.4 Сторінка Stores для додавання та сканування локального сховища

Після сканування файли відображаються у вигляді галереї. Галерея побудована за принципом сітки, де медіа файли мають різні розміри та розміщуються компактно. На скріншоті видно, що система працює з різними типами візуальних матеріалів: фотографіями, зображеннями, мемами, скріншотами та іншими файлами. Для елементів, які ще перебувають у процесі обробки, відображається статус indexing. Це зручно, оскільки користувач бачить, що файл уже доданий до галереї, але AI-метадані для нього ще можуть формуватися.

У верхній частині галереї також показується статус обробки: кількість створених мініатюр, кількість завершених AI-операцій і кількість файлів у черзі. Кнопка Queue all unindexed дозволяє поставити всі ще не оброблені файли в чергу на AI-індексацію.

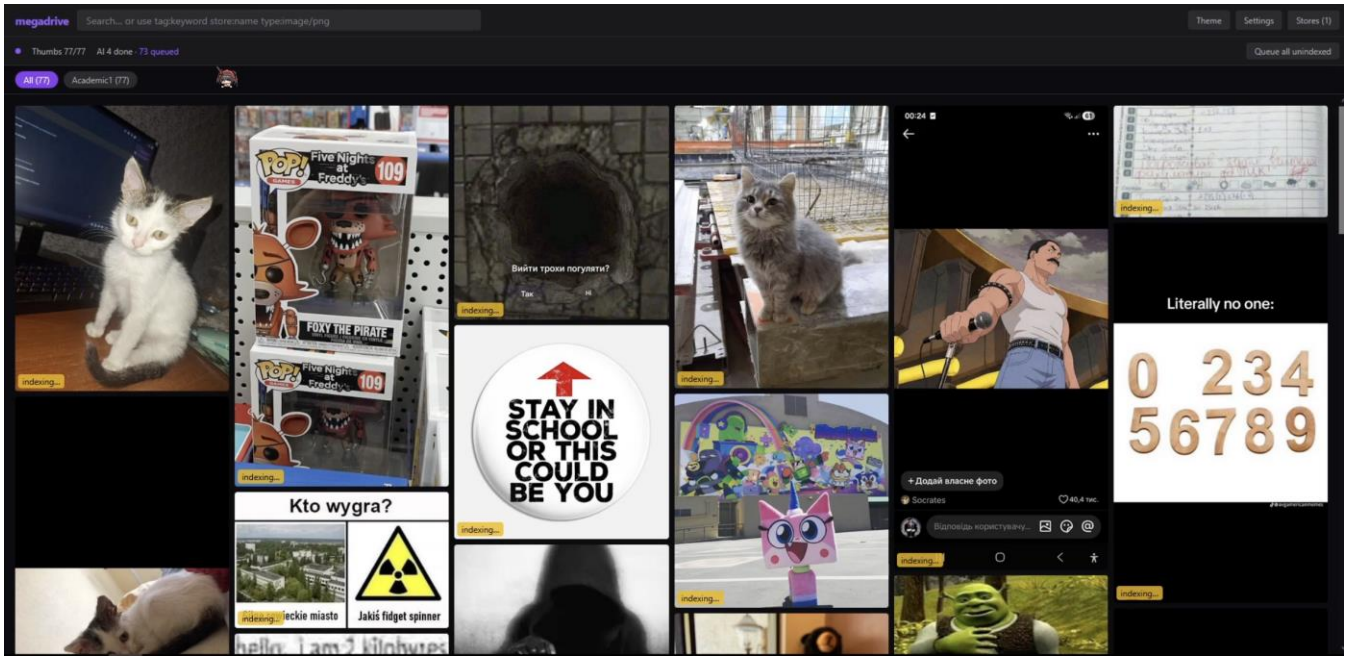


Рис. 4.5 Відображення проіндексованих медіа файлів у галереї

Окремо реалізовано вікно налаштувань. Воно відкривається поверх основної галереї, при цьому фон затемнюється, щоб користувач розумів, що зараз працює саме з модальним вікном. У вікні налаштувань доступні вкладки Config та API Keys. У вкладці Config можна переглянути основні параметри роботи системи: порт, директорію даних, кількість AI workers, дозвіл на локальні файлові дії, AI provider, адресу Ollama, модель Ollama, OpenAI-compatible URL, модель і ключ.

Частина параметрів має позначку restart. Це означає, що після їх зміни для коректного застосування може знадобитися перезапуск системи. Такий підхід є зрозумілим для користувача, оскільки він одразу бачить, які налаштування впливають на роботу серверної частини.

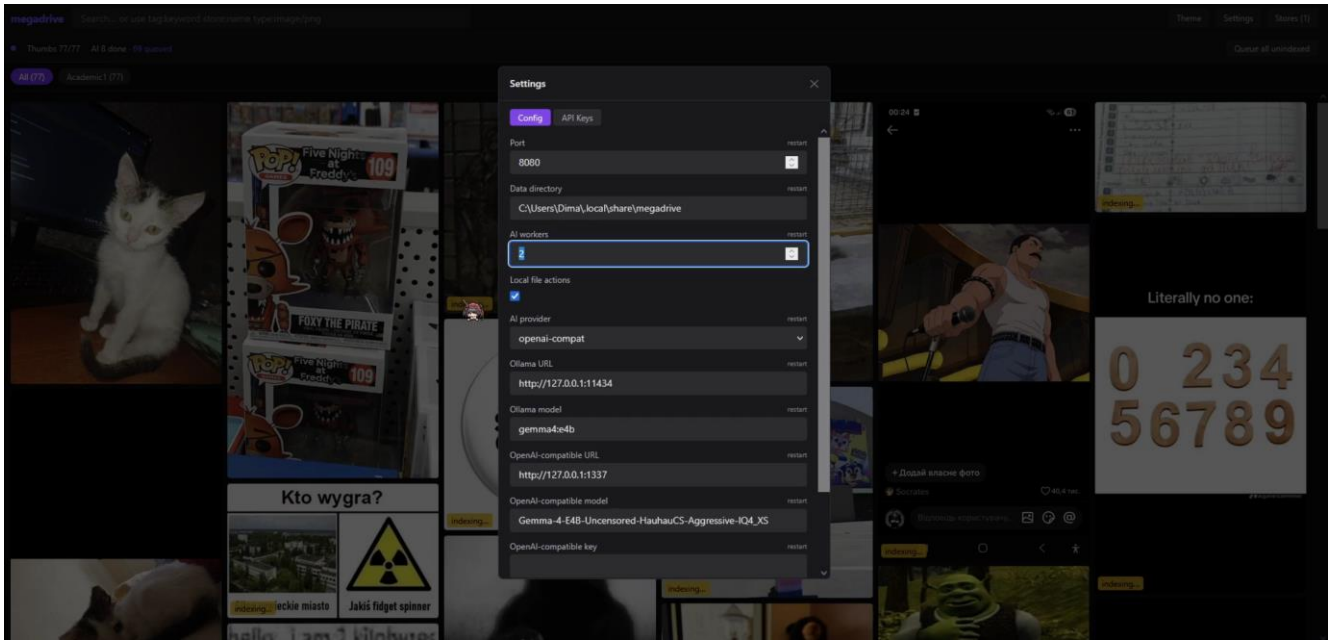


Рис. 4.6 Вікно налаштувань системи megadrive

Для перегляду конкретного медіа файлу реалізовано модальне вікно з деталями. Після натискання на файл у галереї відкривається вікно, де зліва показується саме зображення або медіа preview, а справа – технічна та змістова інформація. На прикладі зі скріншоту відображено назву файлу, роздільну здатність, розмір, MIME-тип, опис, текст із зображення, ключові слова та поле для нотаток.

Важливо, що опис і ключові слова формуються не вручну, а в результаті AI-обробки. Наприклад, для зображення з котом система сформувала англомовний опис, визначила об’єкти на фото та додала набір ключових слів. Такі дані надалі використовуються для пошуку. Тобто користувач може знайти цей файл не тільки за назвою, а й за змістом зображення.

У цьому ж модальному вікні доступні кнопки Re-index with AI, Open original та Open in file manager. Перша кнопка запускає повторну AI-обробку файлу, друга

відкриває оригінальний файл, а третя дозволяє перейти до файлу у файловій системі, якщо локальний режим увімкнений.

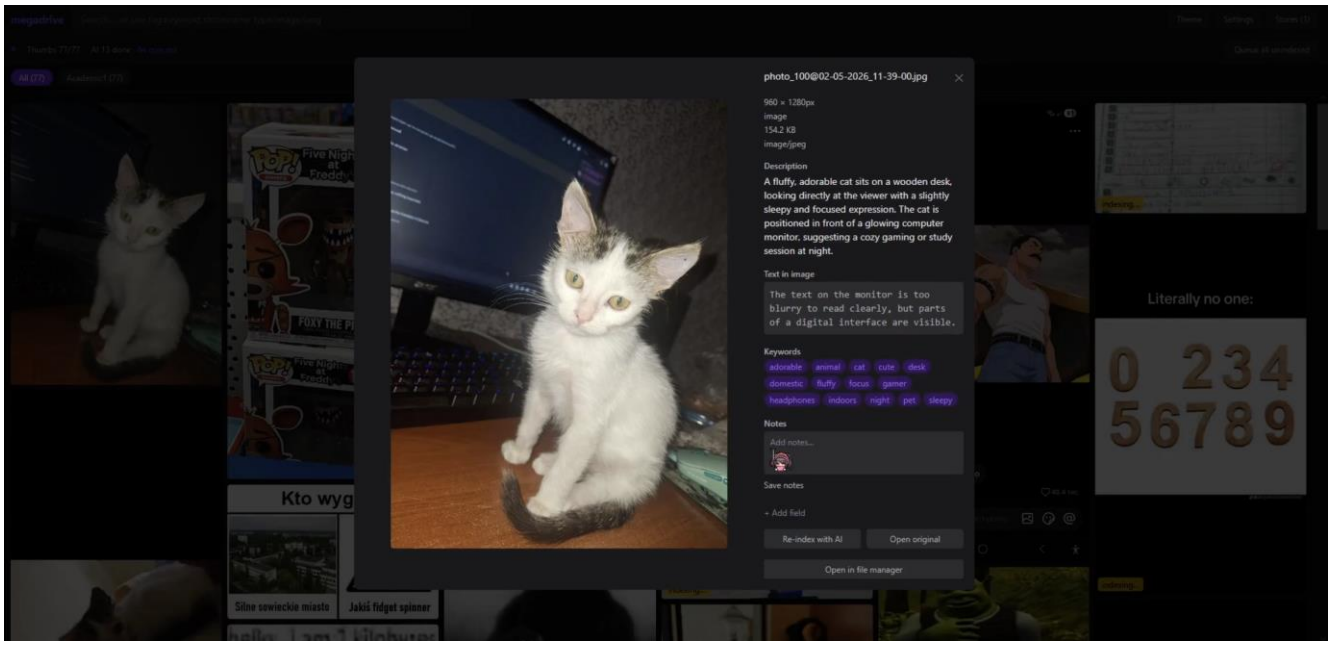


Рис. 4.7 Модальне вікно перегляду медіа файлу з AI-метаданими

Загалом інтерфейс побудований так, щоб користувач міг виконати основний сценарій без додаткових інструментів: додати папку, запустити сканування, переглянути галерею, знайти потрібний файл, відкрити його деталі й за потреби повторно запустити AI-індексацію. Під час демонстрації роботи програми видно, що основні елементи інтерфейсу вже пов’язані між собою та використовуються як єдиний робочий процес.

Отже, клієнтський інтерфейс megadrive реалізує основні функції роботи з медіа архівом: керування сховищами, перегляд галереї, пошук, відображення статусу індексації, налаштування системи та перегляд AI-метаданих. Інтерфейс є практичною частиною системи, через яку користувач взаємодіє з усіма основними можливостями застосунку.

4.5 Реалізація конфігурації, збірки та запуску системи

Реалізація конфігурації, збірки та запуску системи megadrive виконана з урахуванням того, що застосунок повинен працювати у self-hosted режимі. Тобто користувач має можливість запускати систему локально або на власному сервері без складного розгортання великої кількості окремих сервісів.

Основні налаштування системи можуть задаватися через конфігураційний файл, базу даних або змінні середовища. Конфігураційний файл має формат TOML і розміщується за шляхом ~/.config/megadrive/config.toml. У ньому можуть зберігатися параметри запуску, налаштування AI-провайдера, директорія даних, кількість worker-процесів для індексації та інші значення.

Директорія даних за замовчуванням розміщується за шляхом ~/.local/share/megadrive. У цій директорії можуть зберігатися службові файли системи, база даних SQLite, мініатюри, кешовані дані та інші матеріали, необхідні для роботи застосунку. Такий підхід дозволяє відокремити конфігурацію від робочих даних.

У системі також передбачено перевизначення налаштувань через змінні середовища. До основних змінних належать PORT, DATA_DIR, INDEXER_WORKERS, MEGADRIVE_LOCAL, MEGADRIVE_PASSWORD, AI_PROVIDER, OLLAMA_URL, OLLAMA_MODEL, OPENAI_URL, OPENAI_MODEL, OPENAI_KEY та TRANSCRIPTION_COMMAND. Якщо параметр задано через змінну середовища, він має пріоритет над значенням із бази даних або конфігураційного файлу.

Це важливо для серверного розгортання, оскільки адміністратор може керувати критичними параметрами без зміни внутрішньої бази даних. Наприклад, через змінні середовища можна задати порт запуску, директорію даних, режим локальних файлових дій, AI-провайдера або ключ доступу до зовнішнього AI-сервісу.

Частина параметрів доступна для перегляду та зміни через вікно Settings у web-інтерфейсі. При цьому налаштування, які були задані через env override, не

повинні редагуватися напряму з інтерфейсу. Це зроблено для того, щоб користувач випадково не змінив параметри, які контролюються на рівні запуску системи.

Серверна частина системи збирається як Go binary. Оскільки для роботи з SQLite використовується pure-Go драйвер modernc.org/sqlite, застосунок може збиратися з параметром CGO_ENABLED=0. Це спрощує запуск на різних середовищах, оскільки не потрібно окремо налаштовувати C-компілятор або зовнішні бібліотеки SQLite.

Клієнтська частина збирається окремо за допомогою pnpm і Vite. Після збірки файли frontend-частини розміщуються у директорії internal/webui/dist. Далі вони вбудовуються у Go-сервер через механізм Go embed. У результаті серверний застосунок може самостійно віддавати готовий web-інтерфейс без окремого frontend-сервера.

Для розробки також може використовуватися Nix flake/devShell. Це дозволяє підготувати стабільне середовище з потрібними версіями інструментів і залежностей. Такий підхід корисний для повторюваної збірки проєкту на різних комп’ютерах.

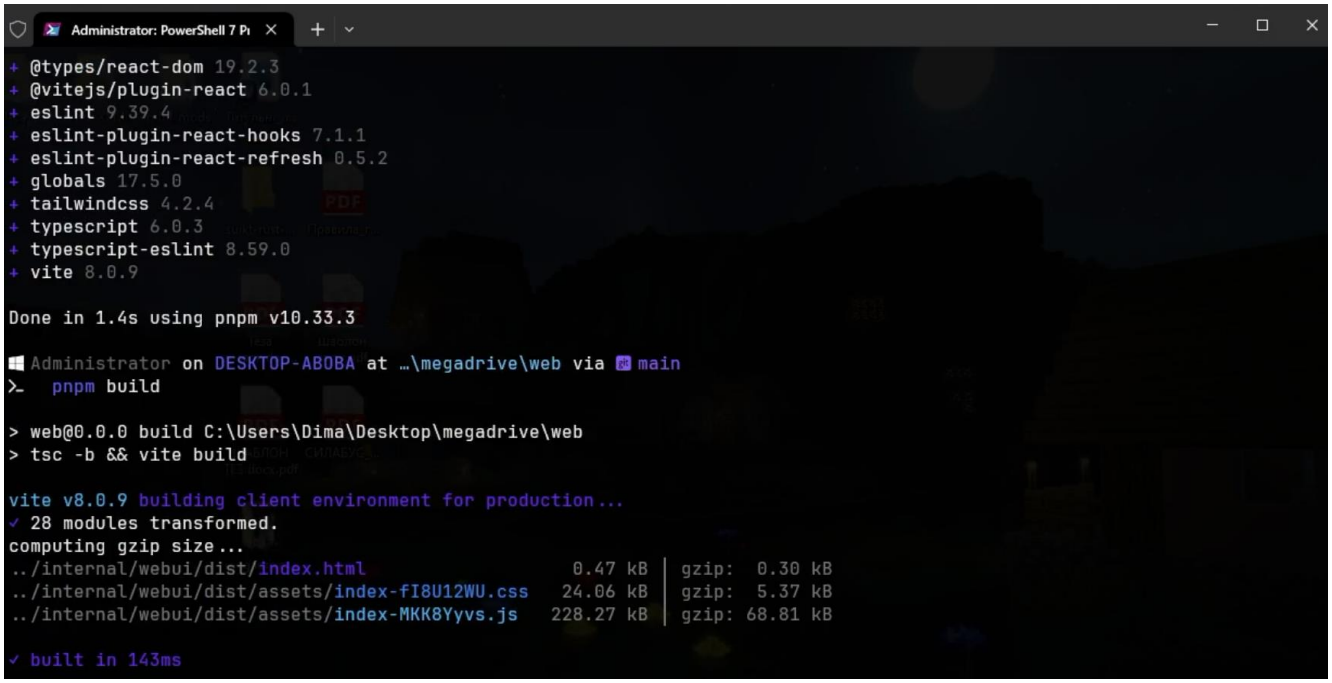


Рис. 4.8 Схема конфігурації, збірки та запуску системи megadrive

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було розроблено self-hosted рішення megadrive для організації, індексації, парсингу та пошуку великого масиву медіа даних із використанням технологій штучного інтелекту.

Проведено аналіз предметної галузі та визначено основні проблеми роботи з великими медіа архівами. Було встановлено, що звичайне зберігання файлів у папках не забезпечує достатньої зручності, оскільки пошук за назвою файлу не дозволяє швидко знаходити потрібні зображення, відео або аудіо. Також було визначено проблему залежності багатьох сучасних рішень від хмарних сервісів, що не завжди підходить для приватних або локальних архівів.

Проаналізовано існуючі програмні рішення для роботи з медіа файлами, зокрема Google Photos, Immich та PhotoPrism. У результаті аналізу було визначено їх переваги та недоліки. Google Photos забезпечує зручну роботу з медіа, але є хмарним сервісом. Immich та PhotoPrism підтримують self-hosted підхід, однак переважно орієнтовані на фото- та відеоколекції. Це підтвердило доцільність створення системи, яка поєднує локальний контроль над даними, підтримку різних типів медіа та формування AI-метаданих.

Сформовано функціональні та нефункціональні вимоги до системи. До основних функцій було віднесено додавання локальних сховищ, індексацію медіа файлів, створення мініатюр, формування описів, ключових слів, OCR-тексту та транскрипцій, пошук і фільтрацію, перегляд деталей файлів, роботу з налаштуваннями, автентифікацію та API-ключі.

Спроєктовано архітектуру системи megadrive. Було визначено основні компоненти: клієнтську частину, серверну частину, локальну базу даних, модуль індексації, модуль пошуку, AI-модуль, систему подій SSE та зовнішні інструменти для обробки відео й аудіо. Також було розроблено структуру бази даних, у якій центральною сутністю є медіа файл, пов’язаний зі сховищем, AI-метаданими, користувацькими полями та повнотекстовим індексом.

Реалізовано серверну частину системи мовою Go. Вона відповідає за API, роботу з базою даних SQLite, індексацію файлів, обробку медіа, пошук, автентифікацію, API-ключі та взаємодію з клієнтським інтерфейсом. Для маршрутизації HTTP-запитів використано go-chi/chi, для зберігання даних – SQLite, а для повнотекстового пошуку – FTS5.

Реалізовано механізм індексації та обробки медіа файлів. Система сканує локальні папки, визначає тип файлу, отримує технічні характеристики, створює мініатюри та передає файли на AI-обробку. Для зображень, відео та аудіо передбачено різні способи обробки, зокрема використання ffmpeg, ffprobe та зовнішньої команди транскрипції.

Реалізовано механізм AI-метаданих, пошуку та фільтрації. Система формує опис файлу, ключові слова, OCR-текст, транскрипцію та попередження. Ці дані зберігаються в базі даних і використовуються для повнотекстового пошуку. Завдяки цьому користувач може знаходити медіа файли не лише за назвою, а й за змістом.

Розроблено клієнтський інтерфейс системи на основі React, TypeScript, Vite та TailwindCSS. Інтерфейс дозволяє керувати локальними сховищами, переглядати галерею, виконувати пошук, відкривати деталі файлів, переглядати AI-метадані, змінювати налаштування та керувати доступом. Екранні форми підтверджують працездатність основних сценаріїв роботи системи.

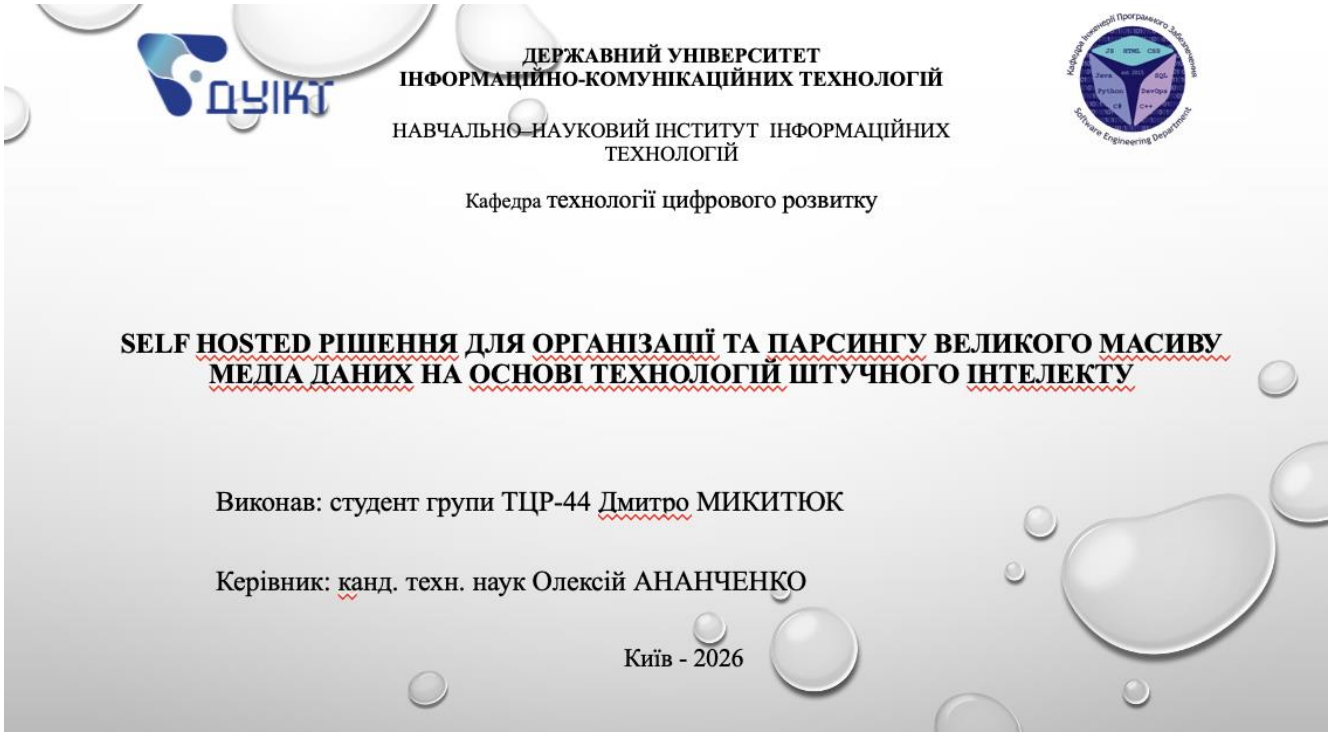
Отже, у результаті роботи було створено self-hosted систему **megadrive**, яка забезпечує локальну організацію великих масивів медіа даних, автоматизовану індексацію, формування AI-метаданих, повнотекстовий пошук і зручний web-інтерфейс для користувача. Розроблене рішення може бути використане для персональних, навчальних або професійних медіа архівів, де важливі локальний контроль над даними та швидкий пошук за змістом файлів.

ПЕРЕЛІК ПОСИЛАНЬ

1. Дідух Л., Міхалко В. Властивості локальних серверів та систем зберігання даних для створення відмовостійкого сховища архівних документів в електронній формі. Інформація та соціум. 2025. С. 67–71. URL: https://jias.donnu.edu.ua/article/view/18837 (дата звернення: 28.02.2026).
2. FFmpeg Documentation. URL: https://ffmpeg.org/documentation.html (дата звернення: 24.03.2026).
3. go-chi/chi. Lightweight, idiomatic and composable router for building Go HTTP services. URL: https://github.com/go-chi/chi (дата звернення: 25.04.2026).
4. Google Photos. Edit, Organize, Search, and Backup Your Photos. URL: https://www.google.com/intl/en/photos/about/ (дата звернення: 12.03.2026).
5. Google Photos Help. Back up photos & videos. URL: https://support.google.com/photos/answer/6193313 (дата звернення: 12.03.2026).
6. Google Blog. Ask Photos: A new way to search with Gemini. URL: https://blog.google/products/photos/google-photos-ask-photos-google-io-2024/ (дата звернення: 12.03.2026).
7. Go Documentation. URL: https://go.dev/doc/ (дата звернення: 25.04.2026).
8. Immich Docs. External Libraries. URL: https://docs.immich.app/features/libraries/ (дата звернення: 12.03.2026).
9. Immich Docs. Backup. URL: https://docs.immich.app/features/backup/ (дата звернення: 12.03.2026).
10. Immich Docs. Supported Media Formats. URL: https://docs.immich.app/features/supported-formats/ (дата звернення: 12.03.2026).
11. Immich Docs. Search. URL: https://docs.immich.app/features/search/ (дата звернення: 12.03.2026).

12. Immich API Documentation. URL: https://immich.app/docs/api (дата звернення: 12.03.2026).
13. Jones M., Bradley J., Sakimura N. RFC 7519: JSON Web Token (JWT). URL: https://datatracker.ietf.org/doc/html/rfc7519 (дата звернення: 25.04.2026).
14. PhotoPrism. Features. URL: https://www.photoprism.app/features (дата звернення: 12.03.2026).
15. PhotoPrism Docs. Search Filters. URL: https://docs.photoprism.app/user-guide/search/filters/ (дата звернення: 12.03.2026).
16. PhotoPrism Docs. Labels. URL: https://docs.photoprism.app/user-guide/organize/labels/ (дата звернення: 12.03.2026).
17. PhotoPrism Docs. Metadata. URL: https://docs.photoprism.app/user-guide/organize/metadata/ (дата звернення: 12.03.2026).
18. PhotoPrism Docs. File Formats. URL: https://docs.photoprism.app/user-guide/library/file-formats/ (дата звернення: 12.03.2026).
19. Provos N., Mazières D. A Future-Adaptable Password Scheme. URL: https://www.usenix.org/legacy/event/usenix99/provos/provos.pdf (дата звернення: 24.03.2026).
20. React Documentation. Quick Start. URL: https://react.dev/learn (дата звернення: 25.04.2026).
21. SQLite Documentation. FTS5 Extension. URL: https://www.sqlite.org/fts5.html (дата звернення: 24.03.2026).
22. SQLite Documentation. URL: https://sqlite.org/docs.html (дата звернення: 25.04.2026).
23. Tailwind CSS Documentation. URL: https://tailwindcss.com/docs (дата звернення: 25.04.2026).
24. TypeScript Documentation. The TypeScript Handbook. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 25.04.2026).
25. Vite Documentation. Getting Started. URL: https://vite.dev/guide/ (дата звернення: 25.04.2026).

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ



МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: спрощення організації, індексації, парсингу та пошуку великого масиву медіа даних на основі технологій штучного інтелекту.

Об'єкт дослідження: процеси організації, індексації та пошуку великих масивів медіа даних.

Предмет дослідження: self-hosted система для організації медіа даних із використанням технологій штучного інтелекту.

2

ЗАДАЧІ ДИПЛОМНОЇ РОБОТИ

1. Провести аналіз літературних джерел, технічної документації та сучасних підходів до організації, індексації, пошуку й автоматизованої обробки великих масивів медіа даних.
2. Здійснити огляд та аналіз існуючих програмних рішень для зберігання, перегляду, пошуку й керування медіа файлами, визначити їх переваги та недоліки.
3. Провести огляд засобів для розробки self-hosted програмного забезпечення для організації медіа архіву з підтримкою AI-метаданих.
4. Сформувати вимоги до розробки self-hosted рішення для індексації, парсингу, пошуку та перегляду медіа даних.
5. Спроєктувати архітектуру програмної системи, структуру основних модулів, базу даних, механізм індексації медіа файлів, пошуку та взаємодії з API.
6. Розробити self-hosted систему megadrive для організації великого масиву медіа.
7. Реалізувати клієнтський інтерфейс системи для перегляду галереї, пошуку, фільтрації, керування сховищами, перегляду деталей медіа файлів, налаштувань, тем оформлення та автентифікації користувача.

3

АНАЛІЗ АНАЛОГІВ

Критерій порівняння	Google Photos	Immich	PhotoPrism	megadrive
Тип рішення	Хмарний сервіс	Self-hosted система	Self-hosted система	Self-hosted система
Основне призначення	Зберігання, пошук і резервне копіювання фото та відео	Резервне копіювання, організація і пошук фото та відео	Організація, індексація і пошук фотоколекцій	Локальна індексація, парсинг і пошук медіа даних
Розгортання на власному сервері	Ні	Так	Так	Так
Локальний контроль над даними	Обмежений	Так	Так	Так
Підтримка зображень	Так	Так	Так	Так
Підтримка відео	Так	Так	Так	Так
Підтримка аудіо	Обмежено	Ні / не основна функція	Ні / не основна функція	Так
AI-метадані	Так	Так	Так	Так
OCR-текст	Частково залежить від сервісів Google	Обмежено	Частково	Так
Транскрипція аудіо/відео	Не є основною функцією	Не є основною функцією	Не є основною функцією	Так
Повнотекстовий пошук	Так	Так	Так	Так
Робота з локальними папками	Ні	Через зовнішні бібліотеки	Через індексацію бібліотеки	Так
Користувацькі поля	Обмежено	Обмежено	Обмежено	Так
API-ключі	Не для локального керування системою	Обмежено	Обмежено	Так
Зовнішній стабільний API	Обмежено	Частково	Частково	Так
Вбудований веб-інтерфейс	Так	Так	Так	Так
Гнучка зміна тем оформлення	Обмежено	Обмежено	Обмежено	Так
Залежність від стороннього постачальника	Висока	Низька	Низька	Низька

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Користувач може додавати до системи локальні папки з фото, відео та аудіо, переглядати їх і запускати повторне сканування.
2. Система знаходить медіа файли у вибраних папках, визначає їх тип і додає до загального каталогу.
3. Для файлів створюються мініатюри, щоб користувач міг швидко переглядати велику кількість медіа без відкриття кожного файлу окремо.
4. Система автоматично формує опис файлу, ключові слова, розпізнаний текст із зображень і текстову розшифровку аудіо або відео.
5. Користувач може шукати медіа не лише за назвою, а й за описом, текстом на зображенні, словами з відео або аудіозапису.
6. Система дозволяє звужувати пошук за тегами, папками та типом файлу, наприклад окремо показати тільки зображення або відео.
7. У вікні деталей відображається сам файл, його розмір, тип, опис, ключові слова, розпізнаний текст та інша інформація.
8. Користувач може вибрати кілька файлів одразу та запустити повторну індексацію, прибрати їх з каталогу або видалити у локальному режимі.
9. Передбачено керування параметрами роботи системи, темами оформлення, API-ключами та паролем користувача.
10. Доступ до системи обмежується паролем, сесіями користувача та API-ключами для зовнішніх підключень.

5

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

1. Self-hosted режим — запуск локально або на власному сервері користувача.
2. Швидкість пошуку — відповідь пошуку до 1–2 с для проіндексованої колекції.
3. Надійність БД — WAL-режим і зовнішні ключі для цілісності даних.
4. Безпека — паролі зберігаються через bcrypt, API-ключі у SHA-256 хеші.
5. Зручність — основні дії виконуються за 2–4 кліки.
6. Кросбраузерність — підтримка Chrome, Edge, Firefox.
7. Інтегрованість — інтеграція великих локальних медіа архівів.
8. Розширюваність — можливість додавання нових AI та storage-провайдерів через інтерфейси.

6

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

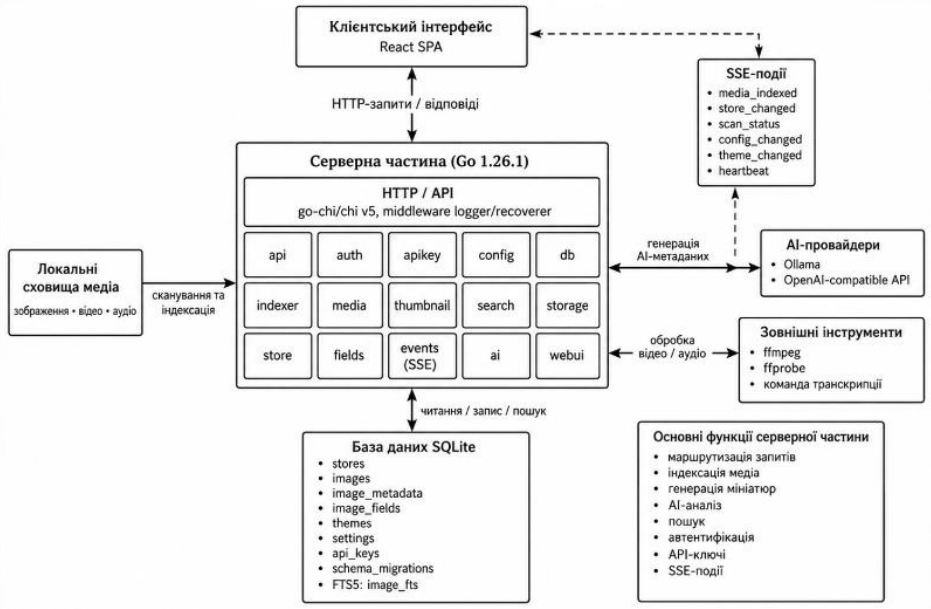
1. **Go** використано для реалізації серверної частини, API, індексації файлів і роботи з локальними сховищами.
2. **go-chi/chi** відповідає за маршрутизацію HTTP-запитів і побудову REST API системи.
3. **SQLite** використано як локальну базу даних для зберігання інформації про файли, сховища, метадані, налаштування та API-ключі.
4. **SQLite FTS5** забезпечує повнотекстовий пошук за назвами файлів, описами, OCR-текстом, транскрипціями та ключовими словами.
5. **React** використано для створення клієнтського веб-інтерфейсу системи.
6. **TypeScript** забезпечує типізацію frontend-коду, компонентів і відповідей від API.
7. **Vite** використано для швидкої збірки клієнтської частини та підготовки файлів для вбудовування у сервер.
8. **TailwindCSS** відповідає за стилізацію інтерфейсу, галереї, модальних вікон, кнопок і тем оформлення.
9. **ffmpeg та ffprobe** застосовуються для обробки відео й аудіо, отримання технічних характеристик і створення прев'ю.
10. **bcrypt, JWT, SHA-256** використано для захисту паролів, сесій користувача та API-ключів.

7

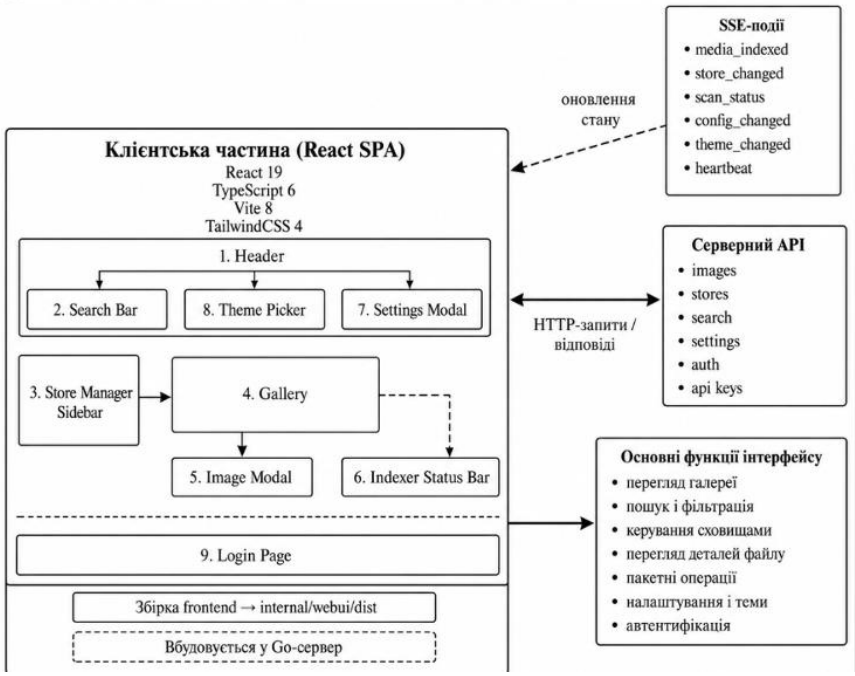
ЗАГАЛЬНИЙ ТЕХНОЛОГІЧНИЙ СТЕК

8

СТРУКТУРА СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ



СТРУКТУРА КЛІЄНТСЬКОЇ ЧАСТИНИ СИСТЕМИ



ДІАГРАМА РОЗГОРТАННЯ

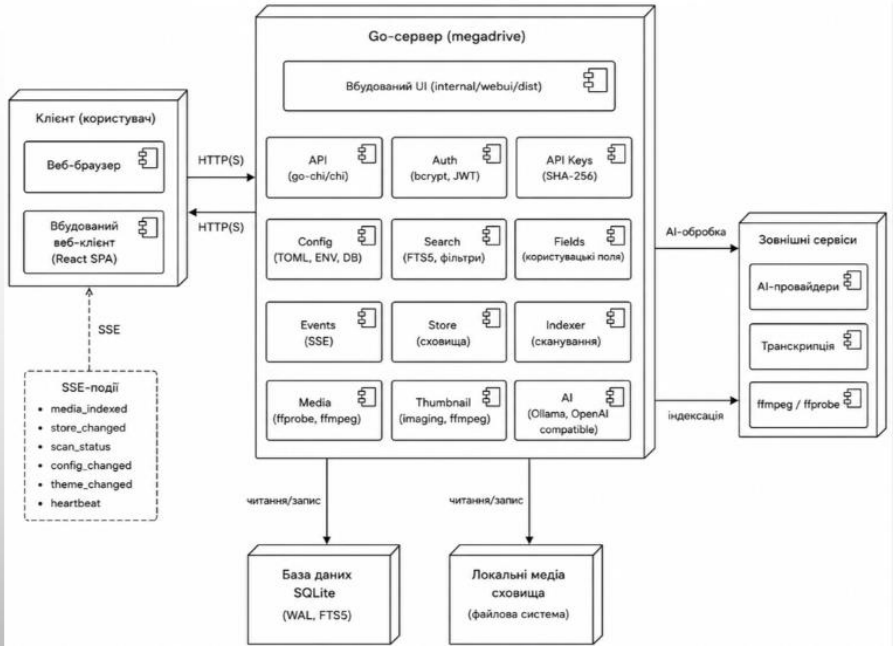
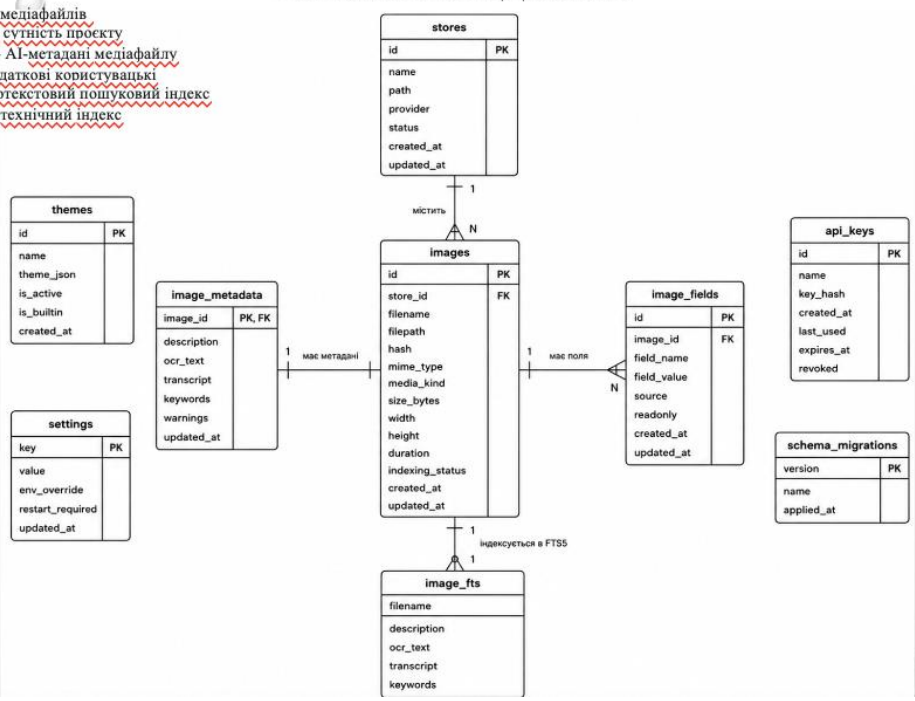
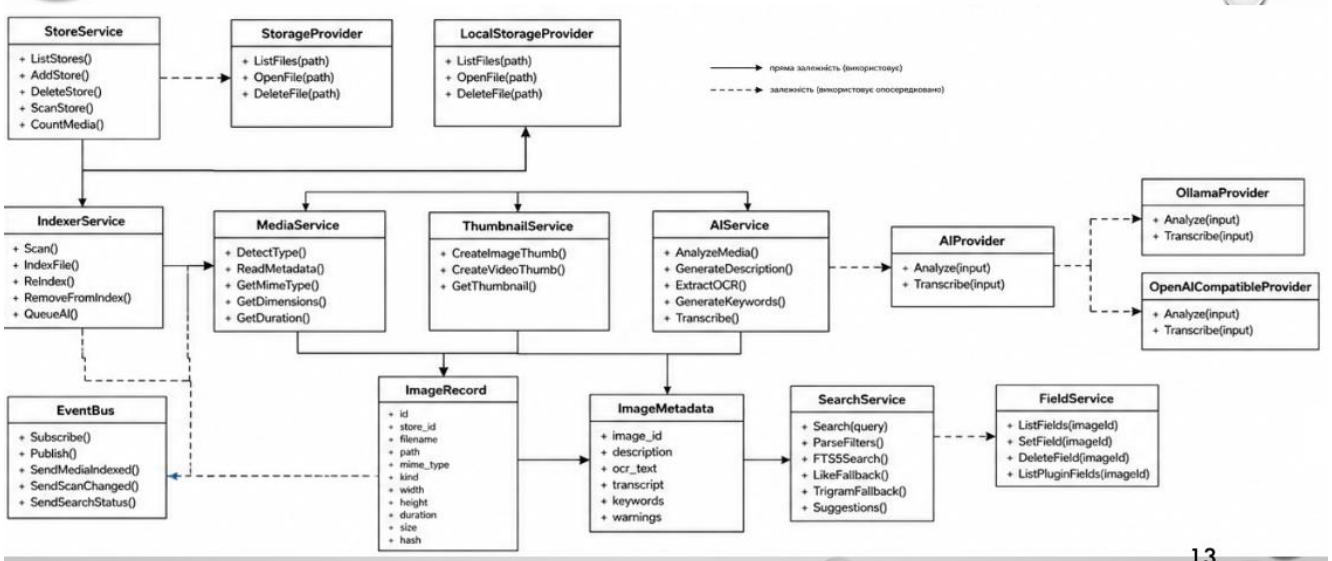


СХЕМА БАЗИ ДАНИХ

stores - сховища медіафайлів
images - головна сутність проєкту
image_metadata - AI-метадані медіафайлу
image_fields - додаткові користувацькі
image_fts - повнотекстовий пошуковий індекс
image_trigrams - технічний індекс

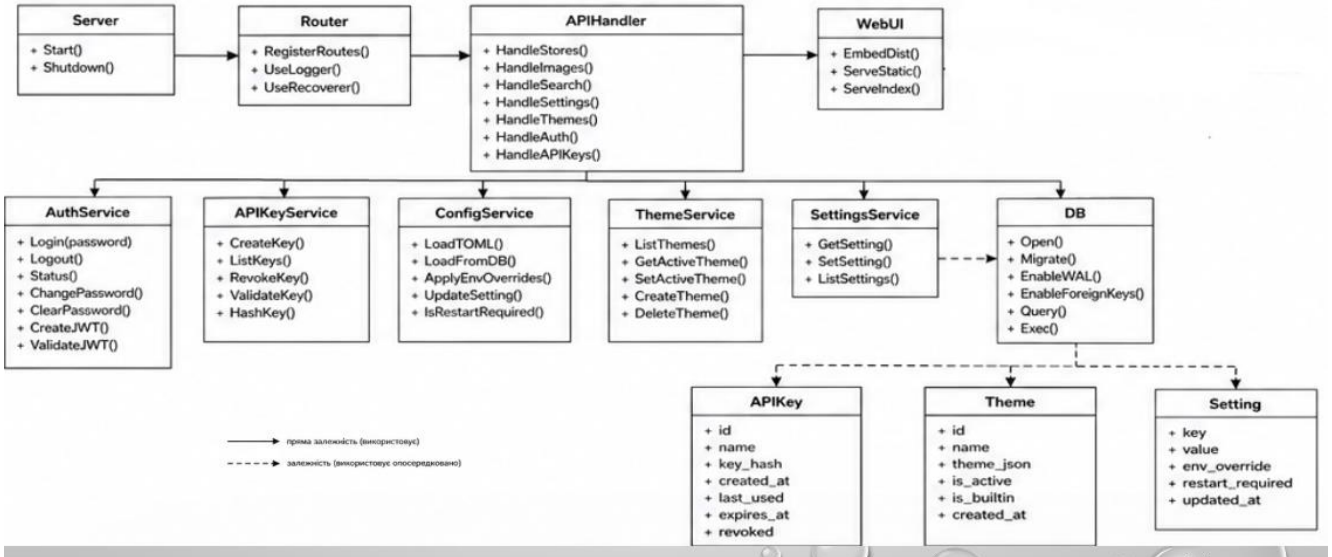


ДІАГРАМА КЛАСІВ МЕДІАКАТАЛОГ, СКАНУВАННЯ ТА AI-ІНДЕКСАЦІЯ



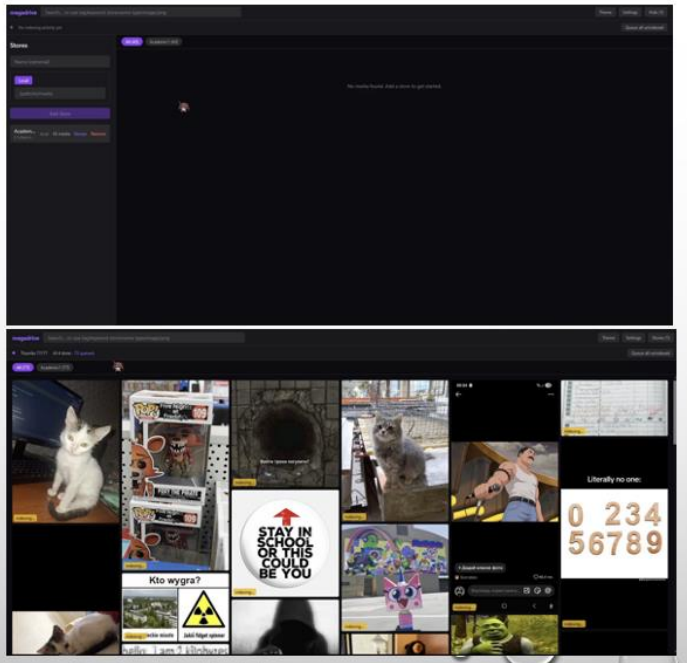
13

ДІАГРАМА КЛАСІВ API, АВТОРИЗАЦІЯ, НАЛАШТУВАННЯ ТА ІНТЕРФЕЙС



14

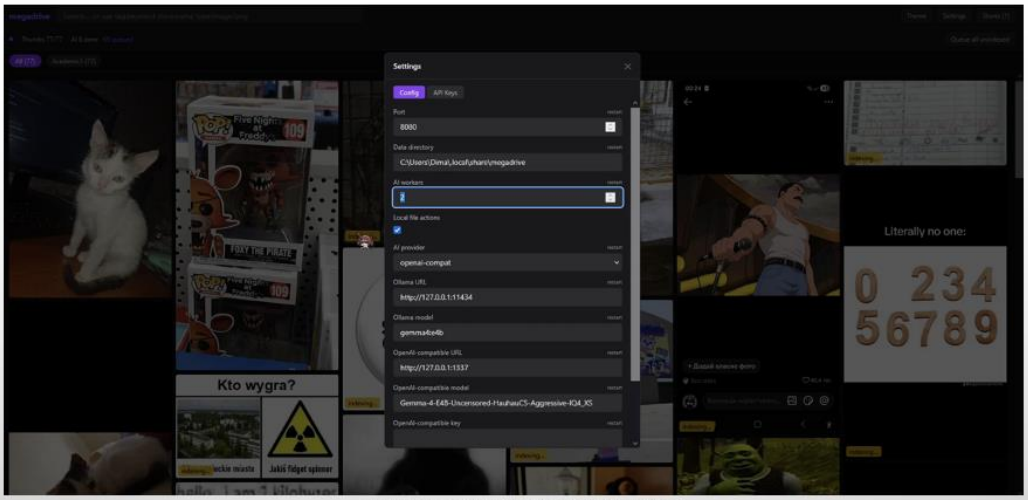
ЕКРАННІ ФОРМИ



Сторінка Stores для додавання та сканування локального сховища

15

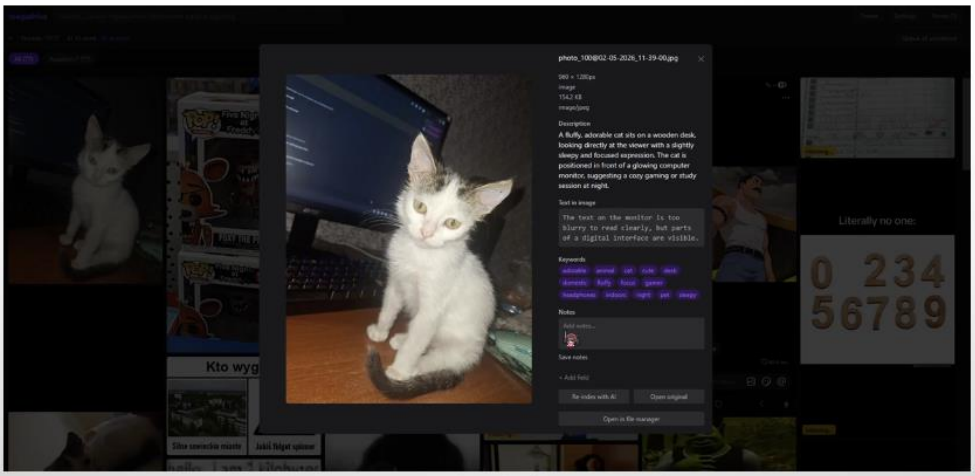
ЕКРАННІ ФОРМИ



Вікно налаштувань системи megadrive

16

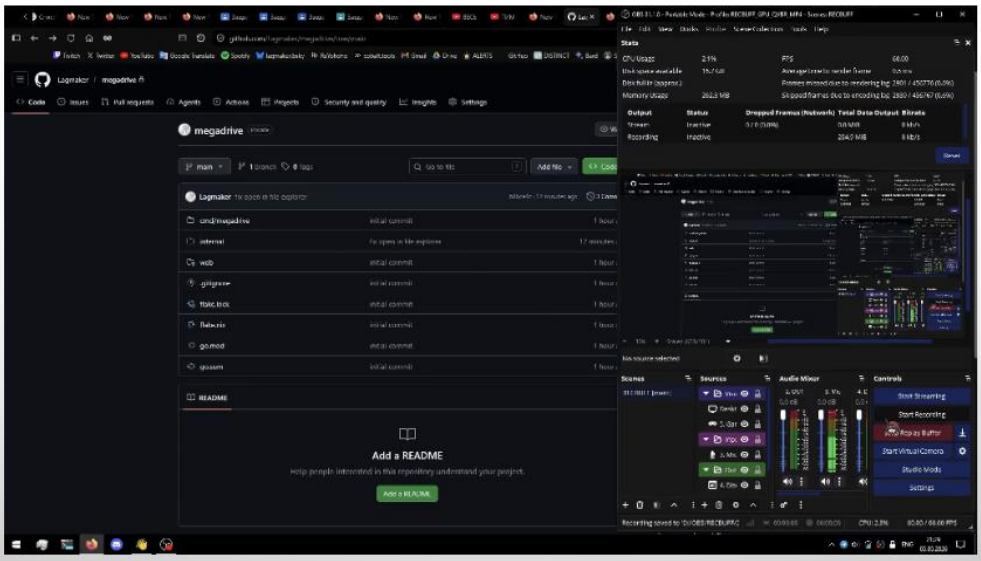
ЕКРАННІ ФОРМИ



Модальне вікно перегляду медіа файлу з AI-метаданими

17

ДЕМОНСТРАЦІЯ РОБОТИ ПРОГРАМИ



18

АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

Тези доповідей:

1. Микитюк Д.С., Ананченко О.Є. Розробка self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту. Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація», 1-3 травня 2026 року, Київ, ДУІКТ. Збірник тез. *(подано до друку).*

2. Микитюк Д.С., Ананченко О.Є. Розробка self hosted рішення для організації та парсингу великого масиву медіа даних на основі технологій штучного інтелекту. Матеріали VI Всеукраїнської науково-практичної конференції «Сучасні інтелектуальні інформаційні технології в науці та освіті», 15 травня 2026 року, Київ, ДУІКТ. Збірник тез. *(подано до друку).*

19

ВИСНОВКИ

1. Проведено аналіз літературних джерел, технічної документації та підходів до організації, індексації, пошуку й автоматизованої обробки медіа даних. Визначено основні проблеми: неструктурованість архівів, складність пошуку за змістом і потребу в AI-метаданих.
2. Проаналізовано існуючі рішення для роботи з медіа файлами: Google Photos, Immich та PhotoPrism. Визначено їх переваги й недоліки, зокрема залежність від хмарних сервісів.
3. Розглянуто засоби реалізації системи.
4. Сформовано вимоги до self-hosted рішення для індексації, парсингу, пошуку та перегляду медіа даних. Визначено основні функції.
5. Спроєктовано архітектуру системи, структуру модулів, базу даних, механізм індексації, пошуку та взаємодії з API.
6. Розроблено систему megadrive для організації великого масиву медіа даних.
7. Реалізовано клієнтський інтерфейс для перегляду галереї, пошуку, фільтрації, керування сховищами, перегляду деталей файлів, налаштувань, тем оформлення та автентифікації.

20

ДОДАТОК Б. ЛІСТИНГИ ОСНОВНИХ МОДУЛІВ

cmd/megadrive/main.go

```
package main

import (
	"fmt"
	"io/fs"
	"log"
	"net/http"
	"os"
	"path/filepath"

	"lagmaker/megadrive/internal/ai"
	"lagmaker/megadrive/internal/api"
	"lagmaker/megadrive/internal/auth"
	"lagmaker/megadrive/internal/config"
	"lagmaker/megadrive/internal/db"
	"lagmaker/megadrive/internal/events"
	"lagmaker/megadrive/internal/indexer"
	"lagmaker/megadrive/internal/webui"
	)

	func main() {
	cfg, err := config.Load()
	if err != nil {
	log.Fatalf("config: %v", err)
	}

	if err := os.MkdirAll(filepath.Join(cfg.DataDir,
"thumbs"), 0755); err != nil {
	log.Fatalf("data dir: %v", err)
	}

	dbPath := filepath.Join(cfg.DataDir,
"megadrive.db")
	database, err := db.Open(dbPath)
	if err != nil {
	log.Fatalf("db: %v", err)
	}
	defer database.Close()
	cfg = config.LoadFromSettings(database, cfg)

	// Set up authenticator.
	authenticator, err := auth.New(database)
	if err != nil {
	log.Fatalf("auth: %v", err)
	}
	// If a password is provided in config, set it
(only takes effect if not already set).
	if cfg.Password != "" &&
!authenticator.IsEnabled() {
	if err :=
authenticator.SetPassword(cfg.Password); err != nil
{
	log.Printf("warning: set password: %v", err)
	} else {
	log.Printf("auth: password configured")
	}
	}

	var aiProvider ai.Provider
	switch cfg.AIProvider {
	case "openai-compat":
	aiProvider =
ai.NewOpenAICompat(cfg.OpenAIURL,
cfg.OpenAIModel, cfg.OpenAIKey)
	log.Printf("AI provider: openai-compat @ %s
model=%s", cfg.OpenAIURL, cfg.OpenAIModel)
default:
	aiProvider = ai.NewOllama(cfg.OllamaURL,
cfg.OllamaModel)
	log.Printf("AI provider: ollama @ %s
model=%s", cfg.OllamaURL, cfg.OllamaModel)
	}
	if err := db.PopulateTrigrams(database); err !=
nil {
	log.Printf("warning: trigram index populate:
%v", err)
	}

	bus := events.NewBus()
	idx := indexer.New(database, cfg.DataDir,
aiProvider, cfg.IndexerWorkers, bus,
cfg.TranscriptionCommand)

	var static http.Handler
	if devMode() {
static = http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
	http.Error(w, "frontend served by Vite on
:5173", http.StatusServiceUnavailable)
	})
	} else {
	sub, err := webui.FS()
	if err != nil {
	log.Fatalf("webui fs: %v", err)
	}
	fileServer := http.FileServer(http.FS(sub))
static = http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
	if _, err := fs.Stat(sub, r.URL.Path[1:]);
os.IsNotExist(err) {
	data, _ := fs.ReadFile(sub, "index.html")
	w.Header().Set("Content-Type", "text/html")
	w.Write(data)
	return
	}
	fileServer.ServeHTTP(w, r)
	})
```

```
	}

	srv := api.New(database, idx, bus, cfg.DataDir,
cfg, authenticator, static)
	addr := fmt.Sprintf(":%d", cfg.Port)
	log.Printf("megadrive listening on
http://localhost%s", addr)
	if err := http.ListenAndServe(addr,
srv.Router()); err != nil {
	log.Fatal(err)
	}
	}

	func devMode() bool {
	_, err := os.Stat("web/src")
	return err == nil && os.Getenv("PROD") ==
""
	}
	internal/api/api.go
	package api

	import (
	"database/sql"
	"encoding/json"
	"fmt"
	"io"
	"net/http"
	"path/filepath"
	"regexp"
	"strconv"
	"strings"
	"time"
	"unicode"

	"github.com/go-chi/chi/v5"
	"github.com/go-chi/chi/v5/middleware"

	"lagmaker/megadrive/internal/apikey"
	"lagmaker/megadrive/internal/auth"
	"lagmaker/megadrive/internal/config"
	"lagmaker/megadrive/internal/events"
	"lagmaker/megadrive/internal/fields"
	"lagmaker/megadrive/internal/indexer"
	"lagmaker/megadrive/internal/search"
	"lagmaker/megadrive/internal/store"
	)

	type Server struct {
	db     *sql.DB
	idx    *indexer.Indexer
	bus    *events.Bus
	dataDir string
	cfg    *config.Config
	auth   *auth.Authenticator
static  http.Handler
	}

	func New(db *sql.DB, idx *indexer.Indexer,
bus *events.Bus, dataDir string, cfg *config.Config,
authenticator *auth.Authenticator, static
http.Handler) *Server {
	return &Server{
	db:     db,
	idx:    idx,
	bus:    bus,
	dataDir: dataDir,
	cfg:    cfg,
	auth:   authenticator,
static:  static,
	}
	}

	func (s *Server) Router() http.Handler {
	r := chi.NewRouter()
	r.Use(middleware.Logger)
	r.Use(middleware.Recoverer)

	// Always-public endpoints.
	r.Post("/api/auth/login", s.login)
	r.Get("/api/auth/status", s.authStatus)
	r.Get("/api/capabilities", s.capabilities)

	// Protected group – wrapped in auth
middleware when auth is enabled.
	r.Group(func(r chi.Router) {
	if s.auth != nil && s.auth.IsEnabled() {
	r.Use(s.auth.Middleware())
	}

	r.Route("/api", func(r chi.Router) {
	// Stores
	r.Get("/stores", s.listStores)
	r.Post("/stores", s.addStore)
	r.Get("/storage/providers", s.storageProviders)
	r.Delete("/stores/{id}", s.deleteStore)
	r.Post("/stores/{id}/scan", s.scanStore)

	// Batch image operations (must be before
/{id} routes in chi trie)
	r.Post("/images/batch/reindex",
s.batchReindex)
	r.Post("/images/batch/remove",
s.batchRemove)
	r.Post("/images/batch/delete", s.batchDelete)

	// Single image
	r.Get("/images", s.listImages)
	r.Get("/images/{id}", s.getImage)
	r.Post("/images/{id}/requeue",
s.requeueImage)
	r.Post("/images/{id}/open", s.openImage)

	// Image custom fields
	r.Get("/images/{id}/fields", s.listFields)
```

```
        r.Put("/images/{id}/fields/{key}", s.setField)
        r.Delete("/images/{id}/fields/{key}",
s.deleteField)

        // Search suggestions
        r.Get("/search/suggestions",
s.searchSuggestions)

        // Indexer
        r.Get("/indexer/status", s.indexerStatus)
        r.Post("/indexer/queue-unindexed",
s.queueUnindexed)

        r.Get("/config", s.getConfig)
        r.Put("/config", s.updateConfig)

        // Themes
        r.Get("/themes", s.listThemes)
        r.Get("/themes/active", s.getActiveTheme)
        r.Put("/themes/active", s.setActiveTheme)
        r.Post("/themes", s.createTheme)
        r.Delete("/themes/{name}", s.deleteTheme)

        // Auth management (requires being
authenticated)
        r.Post("/auth/logout", s.logout)
        r.Post("/auth/password", s.changePassword)
        r.Get("/auth/keys", s.listAPIKeys)
        r.Post("/auth/keys", s.createAPIKey)
        r.Delete("/auth/keys/{id}", s.revokeAPIKey)

        // SSE
        r.Get("/events", s.sseEvents)
        })

        // /api/v1/ – stable external API, accepts API
key or session JWT.
        r.Route("/api/v1", func(r chi.Router) {
        r.Use(s.apiKeyOrSession)
        r.Get("/images", s.listImages)
        r.Get("/images/{id}", s.getImage)
        r.Get("/stores", s.listStores)
        })

        r.Get("/thumbs/{hash}", s.serveThumbnail)
        r.Get("/images/raw/{id}", s.serveOriginal)
        })

        r.Handle("/*", s.static)
        return r
        }

// apiKeyOrSession middleware accepts either a
valid API key header or a session JWT.
        func (s *Server) apiKeyOrSession(next
http.Handler) http.Handler {
        return http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
        // Check JWT first.
        if s.auth != nil &&
s.auth.TokenFromRequest(r) != "" {
        next.ServeHTTP(w, r)
        return
        }
        // Check API key.
        key := r.Header.Get("X-API-Key")
        if key == "" {
        if hdr := r.Header.Get("Authorization");
strings.HasPrefix(hdr, "Bearer mgd_") {
        key = strings.TrimPrefix(hdr, "Bearer ")
        }
        }
        if key != "" {
        k, err := apikey.Verify(s.db, key)
        if err == nil && k != nil {
        next.ServeHTTP(w, r)
        return
        }
        }
        w.Header().Set("Content-Type",
"application/json")
        w.WriteHeader(http.StatusUnauthorized)
        w.Write([]byte(`{"error":"unauthorized"}`))
        })
        }

// --- stores ---

        func (s *Server) listStores(w
http.ResponseWriter, r *http.Request) {
        stores, err := store.List(s.db)
        if err != nil {
        jsonError(w, err, 500)
        return
        }
        jsonOK(w, stores)
        }

        func (s *Server) addStore(w
http.ResponseWriter, r *http.Request) {
        var body struct {
        Name        string        `json:"name"`
        Path        string        `json:"path"`
        ProviderType   string
`json:"provider_type"`
        ProviderConfig json.RawMessage
`json:"provider_config"`
        }
        if err :=
json.NewDecoder(r.Body).Decode(&body); err !=
nil {
        jsonError(w, err, 400)
        return
        }
```

```
	if body.Path != "" &&
len(body.ProviderConfig) == 0 {
	body.ProviderType = "local"
	body.ProviderConfig =
json.RawMessage(fmt.Sprintf(`{"path":%q}`,
body.Path))
	}
	if body.ProviderType == "" {
	body.ProviderType = "local"
	}
	if body.Name == "" {
	if body.Path != "" {
	body.Name = filepath.Base(body.Path)
	} else {
	body.Name = body.ProviderType
	}
	}

	st, err := store.AddWithProvider(s.db,
body.Name, body.ProviderType,
body.ProviderConfig)
	if err != nil {
	jsonError(w, err, 400)
	return
	}
	go s.idx.ScanAndEnqueue(st.ID)
	s.bus.Publish(events.Event{Type:
"store_changed", StoreID: st.ID})
	jsonOK(w, st)
	}

	func (s *Server) storageProviders(w
http.ResponseWriter, r *http.Request) {
	jsonOK(w, []map[string]any{
	{
	"type": "local",
	"name": "Local folder",
	"fields": []map[string]string{
	{"name": "path", "type": "text", "label":
"Folder path"},
	},
	},
	})
	}

	func (s *Server) deleteStore(w
http.ResponseWriter, r *http.Request) {
	id, err := strconv.ParseInt(chi.URLParam(r,
"id"), 10, 64)
	if err != nil {
	jsonError(w, err, 400)
	return
	}
	if err := store.Delete(s.db, id); err != nil {
	jsonError(w, err, 500)
	return
	}
	s.bus.Publish(events.Event{Type:
"store_changed", StoreID: id})
	w.WriteHeader(http.StatusNoContent)
	}

	func (s *Server) scanStore(w
http.ResponseWriter, r *http.Request) {
	id, err := strconv.ParseInt(chi.URLParam(r,
"id"), 10, 64)
	if err != nil {
	jsonError(w, err, 400)
	return
	}
	go s.idx.ScanAndEnqueue(id)
	s.bus.Publish(events.Event{Type:
"scan_status", StoreID: id})
	jsonOK(w, map[string]string{"status":
"scanning"})
	}

// --- images ---

	type imageRow struct {
	ID          int64   `json:"id"`
	StoreID     int64   `json:"store_id"`
	Path        string  `json:"path"`
	Filename    string  `json:"filename"`
	Hash        string  `json:"hash"`
	Width       int     `json:"width"`
	Height      int     `json:"height"`
	Kind        string  `json:"kind"`
	DurationMS  int64   `json:"duration_ms"`
	SizeBytes   int64   `json:"size_bytes"`
	MimeType    string  `json:"mime_type"`
	Discovered  int64   `json:"discovered"`
	IndexedAt   *int64
`json:"indexed_at,omitempty"`
	Description *string
`json:"description,omitempty"`
	OCRText     *string
`json:"ocr_text,omitempty"`
	TranscriptText *string
`json:"transcript_text,omitempty"`
	Warning     *string
`json:"warning,omitempty"`
	Keywords    []string
`json:"keywords,omitempty"`
	ThumbURL    string  `json:"thumb_url"`
	}

	var ftsSpecialChars =
regexp.MustCompile(`[^\p{L}\p{N}_\-]`)

	func buildFTSQuery(raw string) string {
	var terms []string
	for _, word := range strings.FieldsFunc(raw,
func(r rune) bool {
```

```
        return unicode.IsSpace(r)
        }) {
        word =
ftsSpecialChars.ReplaceAllString(word, "")
        if len([]rune(word)) >= 2 {
        terms = append(terms, `"`+word+`"*`)
        }
        }
        return strings.Join(terms, " ")
        }

        const imageSelectCols = `
        i.id, i.store_id, i.path, i.filename, i.hash,
i.width, i.height, i.kind, i.duration_ms, i.size_bytes,
i.mime_type,
        i.discovered, i.indexed_at, m.description,
m.ocr_text, m.transcript_text, m.warning,
m.keywords`

        func (s *Server) listImages(w
http.ResponseWriter, r *http.Request) {
        q := r.URL.Query()
        storeID := q.Get("store_id")
        rawSearch := strings.TrimSpace(q.Get("q"))
        page, _ := strconv.Atoi(q.Get("page"))
        if page < 1 {
        page = 1
        }
        limit, _ := strconv.Atoi(q.Get("limit"))
        if limit < 1 || limit > 200 {
        limit = 50
        }
        offset := (page – 1) * limit

        var (
        rows *sql.Rows
        err  error
        )

        if rawSearch != "" {
        parsed := search.Parse(rawSearch)
        rows, err = s.searchImages(parsed, storeID,
limit, offset)
        } else {
        rows, err = s.browseImages(storeID, limit,
offset)
        }

        if err != nil {
        jsonError(w, err, 500)
        return
        }
        defer rows.Close()

        var images []imageRow
        for rows.Next() {
        var img imageRow
        var kwJSON sql.NullString
        var indexedAt sql.NullInt64
        if err := rows.Scan(
        &img.ID, &img.StoreID, &img.Path,
&img.Filename, &img.Hash,
        &img.Width, &img.Height, &img.Kind,
&img.DurationMS, &img.SizeBytes,
&img.MimeType,
        &img.Discovered, &indexedAt,
&img.Description, &img.OCRText,
&img.TranscriptText, &img.Warning, &kwJSON,
        ); err != nil {
        jsonError(w, err, 500)
        return
        }
        if indexedAt.Valid {
        img.IndexedAt = &indexedAt.Int64
        }
        if kwJSON.Valid && kwJSON.String != "" {
        _ = json.Unmarshal([]byte(kwJSON.String),
&img.Keywords)
        }
        img.ThumbURL = "/thumbs/" + img.Hash
        images = append(images, img)
        }
        if images == nil {
        images = []imageRow{}
        }
        jsonOK(w, images)
        }

        func (s *Server) searchImages(q search.Query,
storeID string, limit, offset int) (*sql.Rows, error) {
        var wheres []string
        var args []any

        if storeID != "" {
        wheres = append(wheres, "i.store_id = ?")
        args = append(args, storeID)
        }
        for _, f := range
q.FiltersOfKind(search.FilterStore) {
        wheres = append(wheres, "s.name = ?")
        args = append(args, f.Value)
        }
        for _, f := range
q.FiltersOfKind(search.FilterType) {
        wheres = append(wheres, "i.mime_type = ?")
        args = append(args, f.Value)
        }
        for _, f := range
q.FiltersOfKind(search.FilterTag) {
        wheres = append(wheres, "EXISTS (SELECT
1 FROM json_each(m.keywords) j WHERE j.value
= ?)")
        args = append(args, f.Value)
        }
```

```
	if q.HasText() {
	return s.searchImagesText(q.TextParts(),
wheres, args, limit, offset)
	}

	// Pure structured query – no free text.
	where := ""
	if len(wheres) > 0 {
	where = "WHERE " + strings.Join(wheres, "
AND ")
	}
	query := `
	SELECT` + imageSelectCols + `
	FROM images i
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	LEFT JOIN stores s ON s.id = i.store_id
	` + where + `
	ORDER BY i.discovered DESC
	LIMIT ? OFFSET ?`
	args = append(args, limit, offset)
	return s.db.Query(query, args...)
	}

	func (s *Server) searchImagesText(text string,
extraWheres []string, extraArgs []any, limit, offset
int) (*sql.Rows, error) {
	extraWhere := ""
	if len(extraWheres) > 0 {
	extraWhere = "AND " +
strings.Join(extraWheres, " AND ")
	}

	// Strategy 1: FTS5 prefix
	ftsQ := buildFTSQuery(text)
	if ftsQ != "" {
	query := `
	SELECT` + imageSelectCols + `
	FROM image_fts f
	JOIN images i ON i.id = f.rowid
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	LEFT JOIN stores s ON s.id = i.store_id
	WHERE image_fts MATCH ? ` + extraWhere
+ `
	ORDER BY rank
	LIMIT ? OFFSET ?`
	args := append([]any{ftsQ},
append(extraArgs, limit, offset)...)
	rows, err := s.db.Query(query, args...)
	if err == nil {
	if rows.Next() {
	rows.Close()
	rows, err = s.db.Query(query, args...)
	if err == nil {
	return rows, nil
	}
	} else {
	rows.Close()
	}
	}
	}

	// Strategy 2: LIKE fallback
	like := "%" + text + "%"
	likeQuery := `
	SELECT` + imageSelectCols + `
	FROM images i
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	LEFT JOIN stores s ON s.id = i.store_id
	WHERE (
	i.filename    LIKE ? ESCAPE '\'
	OR m.description LIKE ? ESCAPE '\'
	OR m.ocr_text    LIKE ? ESCAPE '\'
	OR m.transcript_text LIKE ? ESCAPE '\'
	OR m.keywords    LIKE ? ESCAPE '\'
	) ` + extraWhere + `
	ORDER BY i.discovered DESC
	LIMIT ? OFFSET ?`
	likeArgs := append([]any{like, like, like, like,
like}, append(extraArgs, limit, offset)...)
	likeRows, likeErr := s.db.Query(likeQuery,
likeArgs...)
	if likeErr == nil {
	if likeRows.Next() {
	likeRows.Close()
	likeRows, likeErr = s.db.Query(likeQuery,
likeArgs...)
	if likeErr == nil {
	return likeRows, nil
	}
	} else {
	likeRows.Close()
	}
	}

	// Strategy 3: trigram fuzzy
	safeText := strings.ReplaceAll(text, `"`, `""`)
	trigramQuery := `
	SELECT` + imageSelectCols + `
	FROM image_trigrams t
	JOIN images i ON i.id = t.rowid
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	LEFT JOIN stores s ON s.id = i.store_id
	WHERE image_trigrams MATCH ? ` +
extraWhere + `
	ORDER BY i.discovered DESC
	LIMIT ? OFFSET ?`
	trigramArgs :=
append([]any{fmt.Sprintf(`"%s"`, safeText)},
append(extraArgs, limit, offset)...)
```

```
	return s.db.Query(trigramQuery,
trigramArgs...)
	}

	func (s *Server) browseImages(storeID string,
limit, offset int) (*sql.Rows, error) {
	query := `
	SELECT` + imageSelectCols + `
	FROM images i
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	LEFT JOIN stores s ON s.id = i.store_id`
	args := []any{}
	if storeID != "" {
	query += " WHERE i.store_id = ?"
	args = append(args, storeID)
	}
	query += " ORDER BY i.discovered DESC
LIMIT ? OFFSET ?"
	args = append(args, limit, offset)
	return s.db.Query(query, args...)
	}

	func (s *Server) getImage(w
http.ResponseWriter, r *http.Request) {
	id, err := strconv.ParseInt(chi.URLParam(r,
"id"), 10, 64)
	if err != nil {
	jsonError(w, err, 400)
	return
	}

	var img imageRow
	var kwJSON sql.NullString
	var indexedAt sql.NullInt64
	err = s.db.QueryRow(`
	SELECT i.id, i.store_id, i.path, i.filename,
i.hash, i.width, i.height, i.kind, i.duration_ms,
i.size_bytes, i.mime_type,
	i.discovered, i.indexed_at, m.description,
m.ocr_text, m.transcript_text, m.warning,
m.keywords
	FROM images i
	LEFT JOIN image_metadata m ON
m.image_id = i.id
	WHERE i.id = ?
	`, id).Scan(
	&img.ID, &img.StoreID, &img.Path,
&img.Filename, &img.Hash,
	&img.Width, &img.Height, &img.Kind,
&img.DurationMS, &img.SizeBytes,
&img.MimeType,
	&img.Discovered, &indexedAt,
&img.Description, &img.OCRText,
&img.TranscriptText, &img.Warning, &kwJSON,
	)
	if err == sql.ErrNoRows {
	http.NotFound(w, r)
	return
	}
	if err != nil {
	jsonError(w, err, 500)
	return
	}
	if indexedAt.Valid {
	img.IndexedAt = &indexedAt.Int64
	}
	if kwJSON.Valid && kwJSON.String != "" {
	_ = json.Unmarshal([]byte(kwJSON.String),
&img.Keywords)
	}
	img.ThumbURL = "/thumbs/" + img.Hash
	jsonOK(w, img)
	}

// --- capabilities ---

	func (s *Server) capabilities(w
http.ResponseWriter, r *http.Request) {
	jsonOK(w, map[string]bool{"local":
s.cfg.Local})
	}

// --- open in file manager ---

	func (s *Server) openImage(w
http.ResponseWriter, r *http.Request) {
	if !s.cfg.Local {
	jsonError(w, fmt.Errorf("local access not
enabled"), http.StatusForbidden)
	return
	}
	id, err := strconv.ParseInt(chi.URLParam(r,
"id"), 10, 64)
	if err != nil {
	jsonError(w, err, 400)
	return
	}

	var imgPath string
	var storeID int64
	if err := s.db.QueryRow(`SELECT path,
store_id FROM images WHERE id = ?`,
id).Scan(&imgPath, &storeID); err ==
sql.ErrNoRows {
	http.NotFound(w, r)
	return
	} else if err != nil {
	jsonError(w, err, 500)
	return
	}

	prov, err := store.ProviderFor(s.db, storeID)
	if err != nil {
```

```
	jsonError(w, err, 500)
	return
	}
	if err := prov.OpenInFileManager(imgPath);
err != nil {
	jsonError(w, err, 500)
	return
	}
	jsonOK(w, map[string]interface{}{})
	}

// --- batch ---

	func (s *Server) batchReindex(w
http.ResponseWriter, r *http.Request) {
	var body struct {
	IDs []int64 `json:"ids"`
	}
	if err :=
json.NewDecoder(r.Body).Decode(&body); err !=
nil || len(body.IDs) == 0 {
	jsonError(w, fmt.Errorf("ids required"), 400)
	return
	}
	s.idx.EnqueueForAI(body.IDs)
	jsonOK(w, map[string]int{"queued":
len(body.IDs)})
	}

	func (s *Server) batchRemove(w
http.ResponseWriter, r *http.Request) {
	var body struct {
	IDs []int64 `json:"ids"`
	}
	if err :=
json.NewDecoder(r.Body).Decode(&body); err !=
nil || len(body.IDs) == 0 {
	jsonError(w, fmt.Errorf("ids required"), 400)
	return
	}
	removed := 0
	for _, id := range body.IDs {
	res, err := s.db.Exec(`DELETE FROM images
WHERE id = ?`, id)
	if err == nil {
	n, _ := res.RowsAffected()
	removed += int(n)
	}
	}
	jsonOK(w, map[string]int{"removed":
removed})
	}

	func (s *Server) batchDelete(w
http.ResponseWriter, r *http.Request) {
	if !s.cfg.Local {
	jsonError(w, fmt.Errorf("local access not
enabled"), http.StatusForbidden)
	return
	}
	var body struct {
	IDs []int64 `json:"ids"`
	}
	if err :=
json.NewDecoder(r.Body).Decode(&body); err !=
nil || len(body.IDs) == 0 {
	jsonError(w, fmt.Errorf("ids required"), 400)
	return
	}

	deleted := 0
	var errs []string
	for _, id := range body.IDs {
	var imgPath string
	var storeID int64
	if err := s.db.QueryRow(`SELECT path,
store_id FROM images WHERE id = ?`,
id).Scan(&imgPath, &storeID); err != nil {
	errs = append(errs, fmt.Sprintf("id %d: %v",
id, err))
	continue
	}
	prov, err := store.ProviderFor(s.db, storeID)
	if err != nil {
	errs = append(errs, fmt.Sprintf("id %d: %v",
id, err))
	continue
	}
	if err := prov.Delete(imgPath); err != nil {
	errs = append(errs, fmt.Sprintf("id %d: %v",
id, err))
	continue
	}
	s.db.Exec(`DELETE FROM images WHERE
id = ?`, id)
	deleted++
	}
	jsonOK(w, map[string]any{"deleted": deleted,
"errors": errs})
	}
```