Полтавський університет економіки і торгівлі
Навчально-науковий інститут денної освіти
Форма навчання денна
Кафедра комп'ютерних наук та інформаційних технологій

Допускається до захисту
Завідувач кафедри
_____Олена ОЛЬХОВСЬКА
(підпис)
«_____»____202_ р.

КВАЛІФІКАЦІЙНА РОБОТА
на тему

«РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ДЛЯ СТВОРЕННЯ, ЗБЕРЕЖЕННЯ ТА КЛАСИФІКАЦІЇ НОТАТОК»

зі спеціальності 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»
ступеня бакалавр

Виконавець роботи Білокіз Валерія Валеріївна
_____«____» _____202_ р.
(підпис)

Науковий керівник доцент, к.ф.-м.н. Черненко О. О.
_____«____»_____202_ р.
(підпис)

Рецензент

ПОЛТАВА 2025

РЕФЕРАТ

Записка: 62 с., 12 рис., 2 таблиці, 1 додаток, 14 джерел.

НОТАТКИ, ВЕБ-ДОДАТОК, REACT, NODE.JS, SQLITE, FTS5, TIPTAP, КЛАСИФІКАЦІЯ, ГРАФ ЗНАНЬ, ELECTRON, REST API

Об'єктом розробки є програмне забезпечення - локальний веб-застосунок для створення, зберігання, повнотекстового пошуку та автоматичної класифікації нотаток і документів користувача.

Предметом розробки є програмна реалізація серверної частини на платформі Node.js і клієнтського інтерфейсу на бібліотеці React із використанням реляційної бази даних SQLite, повнотекстового індексу FTS5 та редактора Tiptap.

Метою роботи є створення зручного програмного засобу для ведення особистих нотаток, який забезпечує rich-text редагування, автоматичну класифікацію записів за категоріями і пріоритетами, повнотекстовий пошук, побудову графа знань та можливість роботи у локальному режимі без передачі даних на зовнішні сервери.

Результатом роботи стало розроблення програмного засобу «Intelligent Notes» на базі Node.js, Express і React. Реалізовано ключові модулі:

- модуль управління нотатками - створення, редагування, видалення, архівування записів типу note та doc;
- модуль rich-text редактора на основі Tiptap - форматування тексту, заголовки, списки, блоки коду, посилання, slash-меню;
- модуль автоматичної класифікації - визначення категорії та пріоритету за ключовими словами;
- модуль повнотекстового пошуку на основі SQLite FTS5 із токенізацією unicode61;
- модуль графа знань на основі бібліотеки @xyflow/react - додавання вершин і ребер, прив'язка до сторінок та файлів;
- модуль файлової бібліотеки - завантаження, попередній перегляд,

перейменування і видалення файлів;

- модуль версіонування - автоматичне збереження знімків нотаток із можливістю відкату;
- модуль резервного копіювання - експорт та імпорт даних у форматі JSON.

Особливості: повна локальність роботи без передачі даних у мережу, підтримка веб- і desktop-режимів через Electron, drag&drop завантаження файлів, fuzzy-пошук з токенізацією без діакритичних знаків, інкрементне версіонування з reason-полем, гнучка модель робочих процесів зі статусами Inbox / In Progress / Review / Done.

Проведено функціональне тестування серверних маршрутів, перевірку коректності повнотекстового пошуку, верифікацію алгоритму класифікації на тестовій вибірці нотаток, регресійне тестування експорту та імпорту резервних копій.

«Intelligent Notes» може бути використаний як персональний інструмент для організації навчальних конспектів, робочих документів і особистих записів студентами та працівниками інформаційної сфери.

ЗМІСТ

СПИСОК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ, ТЕРМІНІВ

Умовні позначення, символи, скорочення, терміни	Пояснення умовних позначень, скорочень, символів
API	Інтерфейс прикладного програмування (Application Programming Interface)
CRUD	Create, Read, Update, Delete - основні операції над сховищем даних
DOM	Document Object Model - об'єктна модель документа
FTS	Full-Text Search - повнотекстовий пошук
HTTP	HyperText Transfer Protocol - протокол передавання гіпертексту
IDE	Integrated Development Environment - інтегроване середовище розробки
JSON	JavaScript Object Notation - текстовий формат обміну даними
ORM	Object-Relational Mapping - об'єктно-реляційне відображення
REST	Representational State Transfer - архітектурний стиль розподілених систем
SPA	Single Page Application - односторінковий вебзастосунок
SQL	Structured Query Language - мова структурованих запитів
SQLite	Реляційна вбудована система керування базами даних
UI	User Interface - інтерфейс користувача
UX	User Experience - досвід користувача

WAL	Write-Ahead Logging - журналювання з випереджувальним записом
XSS	Cross-Site Scripting - міжсайтове виконання сценаріїв
ПЗ	Програмне забезпечення
СКБД	Система керування базами даних

ВСТУП

У сучасному інформаційному суспільстві щоденне накопичення текстових даних - навчальних конспектів, робочих документів, нотаток зустрічей, ідей та довідкових матеріалів - вимагає від користувача застосування ефективних інструментів для їх упорядкування. Кожен студент чи спеціаліст щодня стикається з потребою зафіксувати думку, скласти короткий план дня, занотувати конспект лекції або обговорення проєкту. Без структурованого сховища такі записи швидко втрачаються або стають недоступними для повторного використання, що знижує продуктивність розумової праці.

Технологічний прогрес останніх років значно розширив можливості індивідуальних застосунків для ведення нотаток. Сучасні рішення поєднують у собі властивості текстового редактора, бази даних, системи зв'язків між сторінками та інструментів візуалізації знань. Однак існуючі популярні продукти, такі як Notion, Obsidian, Evernote, Apple Notes та Google Keep, мають низку обмежень: одні з них залежать від хмарного сховища, інші не підтримують повноцінного rich-text редагування, треті не пропонують механізмів автоматичної класифікації записів за змістом. Це створює нішу для розробки локального програмного засобу, який поєднує найкращі підходи у єдиному інструменті.

Робота присвячена розробці локального вебзастосунку для створення, зберігання та класифікації нотаток із підтримкою rich-text редагування, повнотекстового пошуку, побудови графа знань та автоматичного присвоєння категорій. Розробка ведеться повністю на відкритих технологіях: серверна частина реалізована на платформі Node.js із фреймворком Express, клієнтська - на бібліотеці React з типізацією TypeScript і збіркою Vite, а реляційне сховище забезпечується вбудованою СКБД SQLite із повнотекстовим індексом FTS5. Для забезпечення можливості запуску у вигляді desktop-застосунку використано фреймворк Electron.

Актуальність роботи зумовлена поєднанням кількох тенденцій: посиленням вимог до приватності даних, що спонукає до використання локальних сховищ

замість хмарних; зростанням популярності сучасних редакторів block-based типу, які дозволяють керувати документами як набором структурованих елементів; розвитком методів організації знань через побудову графів зв'язків між записами; необхідністю автоматизації рутинних операцій класифікації, що економить час користувача.

Метою роботи є створення зручного, продуктивного та функціонально повного програмного засобу для ведення нотаток, який забезпечить користувачеві rich-text редагування, автоматичну класифікацію записів за категоріями і пріоритетами, повнотекстовий пошук та можливість будувати граф зв'язків між нотатками і файлами.

Для досягнення поставленої мети у роботі вирішуються такі завдання:

- проаналізувати предметну область ведення електронних нотаток і виявити основні вимоги до сучасних рішень;
- здійснити огляд популярних аналогів та виявити їх переваги і недоліки;
- обґрунтувати вибір технологічного стеку для серверної і клієнтської частин;
- спроєктувати архітектуру програмного засобу, REST API і модель даних;
- розробити алгоритм автоматичної класифікації записів за ключовими словами;
- реалізувати серверну частину на Node.js і Express із SQLite-сховищем та FTS5-пошуком;
- реалізувати клієнтську частину на React і Tiptap із підтримкою декількох видів представлення нотаток;
- реалізувати модуль графа знань на основі бібліотеки @xyflow/react;
- реалізувати механізм резервного копіювання та відновлення;
- провести функціональне тестування програмного засобу.

Об'єктом дослідження є процес створення, зберігання, пошуку та структурування нотаток у локальному застосунку. Предметом дослідження є програмна реалізація вебзастосунку для роботи з нотатками з використанням Node.js, SQLite, React і Tiptap.

Практичне значення роботи полягає в тому, що розроблений програмний засіб може бути застосований студентами для ведення навчальних конспектів та планування підготовки до іспитів, працівниками офісної сфери - для зберігання робочих документів, протоколів зустрічей і списків задач, а також у дослідницькій роботі - для накопичення фрагментів літератури і побудови концептуальних карт.

Кваліфікаційна робота складається зі вступу, постановки задачі, трьох основних розділів, висновків, списку інформаційних джерел та одного додатку, що містить лістинг розробленого програмного коду.

ПОСТАНОВКА ЗАДАЧІ

Основною задачею кваліфікаційної роботи є розробка локального програмного засобу для ведення нотаток, який поєднує функції rich-text редактора, бази знань, системи класифікації та інструменту візуалізації зв'язків. Програмний засіб повинен забезпечувати швидку роботу з великими обсягами записів, надавати ефективний пошук за змістом і метаданими, а також працювати без необхідності постійного підключення до мережі Інтернет.

Розроблюваний програмний засіб має задовольняти такі функціональні вимоги:

- підтримка двох типів записів: коротких нотаток (note) і повноцінних документів (doc) з єдиним інтерфейсом редагування;
- rich-text редактор з форматуванням, заголовками, списками, чекбоксами, цитатами, блоками коду та slash-меню для швидкого вставляння блоків;
- організація нотаток за папками з можливістю присвоєння тегів, статусів робочого процесу і термінів виконання;
- автоматична класифікація нотаток за заданими категоріями та визначення пріоритету на основі ключових слів;
- повнотекстовий пошук за заголовком, вмістом і тегами з підтримкою морфологічних форм;
- робота з вкладеними файлами: завантаження, попередній перегляд, прикріплення до нотаток, файлова бібліотека з фільтрами;
- зв'язки між нотатками з підтримкою backlinks та @-mention;
- візуалізація графа знань з можливістю CRUD операцій над вершинами та ребрами;
- автоматичне збереження версій нотатки з можливістю відкату до довільного знімка;
- зворотний зв'язок щодо точності класифікації для подальшого вдосконалення

алгоритму;

- експорт та імпорт всіх даних у форматі JSON для резервного копіювання;
- робота у двох режимах: як веб-застосунок у браузері та як desktop-застосунок через Electron.

Нефункціональні вимоги охоплюють такі характеристики:

- повна локальність роботи без передавання даних у зовнішні сервіси;
- швидкий відклик інтерфейсу - час відкриття нотатки до однієї секунди при базі понад 1000 записів;
- мінімізація залежностей серверної частини для спрощення розгортання;
- зрозумілий і відповідний сучасним стандартам користувацький інтерфейс із підтримкою клавіатурних скорочень;
- модульна структура коду для подальшого розширення функціоналу.

Для досягнення поставленої задачі необхідно виконати такі підзадачі:

1. проаналізувати сучасні підходи до побудови застосунків для ведення нотаток і дослідити архітектурні особливості популярних аналогів;
2. обґрунтувати вибір серверної і клієнтської платформ, бази даних та інструменту повнотекстового пошуку;
3. спроєктувати модель даних із таблицями нотаток, тегів, папок, версій, файлів і вершин графа;
4. спроєктувати REST API з ресурсами нотаток, файлів, шаблонів, графа і резервних копій;
5. розробити алгоритм автоматичної класифікації на основі словника ключових слів і регулярних виразів;
6. реалізувати серверну частину з обробкою маршрутів, валідацією вхідних даних та інтеграцією з SQLite;
7. реалізувати клієнтську частину з кількома видами представлення (Tree, Table, Board, List), фільтрами та inline-редагуванням;
8. реалізувати модуль графа знань з адаптивним розкладанням вершин;
9. реалізувати механізми резервного копіювання та desktop-обгортку через

Electron;

10. провести тестування програмного засобу та задокументувати інструкцію для користувача.

1. ІНФОРМАЦІЙНИЙ ОГЛЯД

1.1. Аналіз предметної області ведення нотаток і документації

Ведення нотаток є базовою практикою інтелектуальної праці, відомою з найдавніших часів. Цифровізація цього процесу почалася з появою перших персональних комп'ютерів, коли текстові файли стали зручним способом збереження думок, а пізніше - з появою спеціалізованих застосунків, які пропонували впорядковане сховище разом із базовим форматуванням і пошуком. За останні десятиліття індустрія електронних нотаток пройшла кілька етапів еволюції: від простих текстових редакторів до хмарних рішень із колаборацією, а нині - до парадигми «другого мозку» (англ. second brain), коли застосунок служить розширенням пам'яті користувача та інструментом мислення.

Сучасні рішення для роботи з нотатками умовно поділяють на кілька категорій. Перша - мінімалістичні нотатники на кшталт Apple Notes або Google Keep, які пропонують швидке створення коротких записів і просту синхронізацію між пристроями. Друга - block-based редактори, такі як Notion або Coda, які трактують документ як набір блоків і дозволяють поєднувати в одному просторі тексти, таблиці і бази даних. Третя - графові системи на кшталт Obsidian або Roam Research, орієнтовані на побудову мережі зв'язків між атомарними записами для подальшої навігації. Четверта - корпоративні бази знань, такі як Confluence чи Microsoft OneNote, призначені для командної роботи з документацією. [1]

Кожен із підходів має сильні і слабкі сторони. Мінімалістичні нотатники зручні, але обмежені у форматуванні. Block-based редактори потужні, проте часто вимагають хмарної інфраструктури і додаткової оплати за повний функціонал. Графові системи добре підходять для дослідницької роботи, але не завжди пропонують зручні таблично-канбанні представлення робочих процесів. Корпоративні рішення громіздкі для індивідуального користування.

На рівні архітектури електронних нотатників виділяють кілька ключових елементів: сховище даних (файлова система, реляційна або документна СКБД), редактор тексту (WYSIWYG, Markdown або block-based), модуль пошуку (повнотекстовий індекс або локальний фільтр), модуль зв'язків (внутрішні посилання, мітки, теги), а також модулі візуалізації (граф знань, дашборди, статистика). Розробка кожного компонента передбачає узгодження з іншими, щоб уникнути дублювання і забезпечити цілісність даних. Окремою категорією задач є питання синхронізації і резервного копіювання, які особливо важливі для локальних застосунків.

Українські користувачі також мають специфічні очікування від подібних рішень. Серед основних - повна підтримка українського інтерфейсу, коректна обробка кириличних символів у пошуковому індексі, можливість роботи без підключення до мережі через нестабільність умов та необхідність зберігання чутливої інформації, а також можливість швидкого розгортання як на персональному комп'ютері, так і на захищеному локальному сервері організації. Огляд предметної області демонструє, що існуючі рішення не повною мірою задовольняють комплекс наведених вимог, що обґрунтовує доцільність розробки власного програмного засобу.

1.2. Огляд існуючих рішень для роботи з нотатками

На ринку електронних нотатників представлено десятки продуктів, з яких особливу увагу привертають кілька найбільш вживаних. Notion є хмарною платформою, що поєднує редактор block-based, реляційні бази даних, дошки задач і вікі-сторінки. Перевагами Notion є потужний редактор з підтримкою таблиць, баз даних, шаблонів, спільної роботи в реальному часі та широких інтеграцій. Недоліками є залежність від хмарного сервісу, повільне завантаження сторінок при розрядженому інтернеті, обмеження безкоштовного плану щодо обсягу і кількості користувачів та неможливість повноцінної офлайн-роботи. [2]

Obsidian є локальним нотатником, який зберігає записи у вигляді Markdown-файлів і підтримує граф знань між сторінками. Його сильною стороною є повна локальність даних, велика екосистема плагінів і висока швидкість пошуку. Слабкі сторони - необхідність вивчення синтаксису Markdown для повноцінного використання, відсутність вбудованих робочих процесів kanban-типу без плагінів, а також не дуже зручне візуальне форматування для нетехнічних користувачів. Інтерфейс Obsidian (див. рис. 1.1) демонструє характерну архітектуру з окремими панелями навігації, редактора і графа.

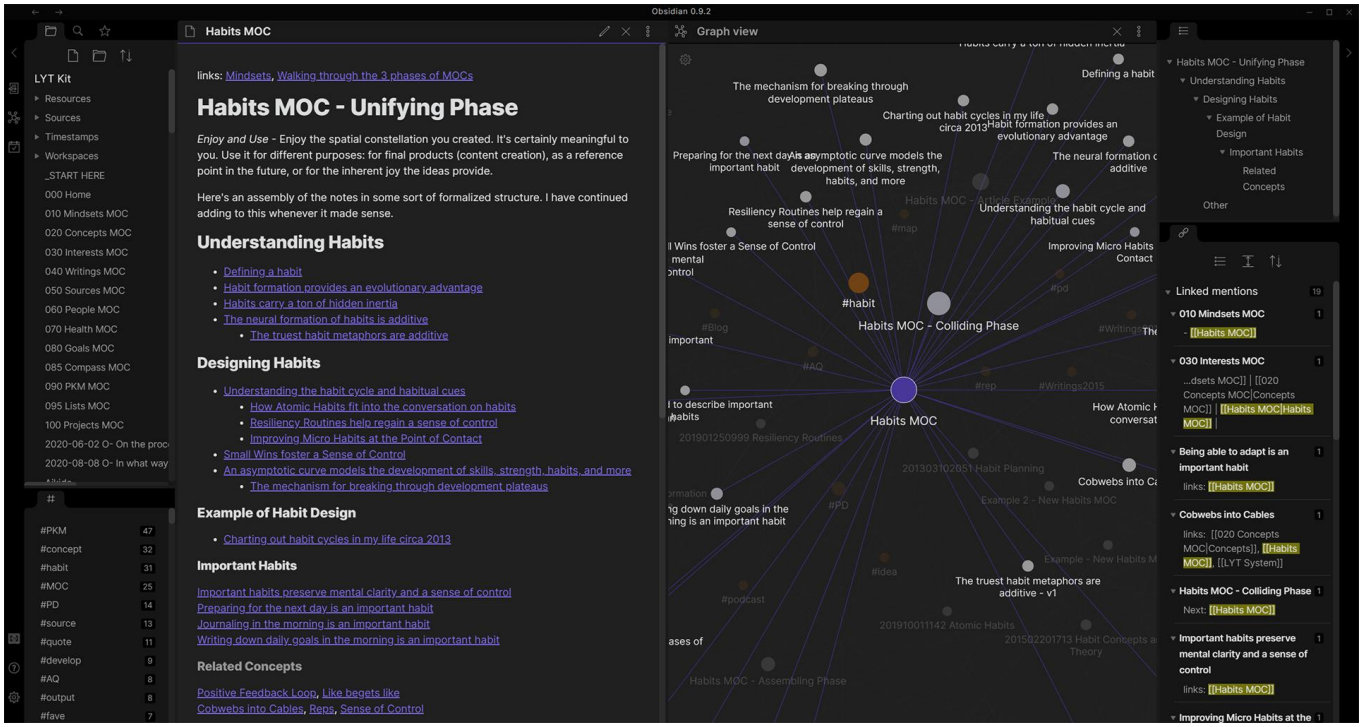

Рисунок 1.1 - Концептуальний інтерфейс графового нотатника Obsidian

На рисунку 1.1 представлено типовий вигляд графового нотатника з трьома основними панелями: ліва панель зі списком записів у вигляді дерева папок, центральна панель з редактором поточного запису, права панель з графом зв'язків між нотатками. Така тримодульна архітектура є одним зі стандартів галузі і була врахована при проєктуванні власного рішення.

Evernote - один із найстаріших комерційних нотатників. Він має зручний веб-кліпер, OCR для PDF-файлів, ієрархію блокнотів, теги та бізнес-функції. Слабкі сторони Evernote - повільна локальна синхронізація, обмеження безкоштовного

плану і поступова втрата аудиторії на користь сучасніших рішень. Apple Notes і Google Keep орієнтовані на швидке створення коротких нотаток із підтримкою списків і нагадувань, але не мають повноцінних механізмів організації документації, не підтримують block-based редагування і не передбачають графів знань.

Окремо варто згадати продукт Roam Research, який популяризував підхід до атомарних нотаток із двосторонніми посиланнями. Roam Research пропонує оригінальний механізм щоденних записів і системи прогресивних резюме. Його недоліками є висока вартість підписки, обмежена локальна підтримка та порівняно мала спільнота. Microsoft OneNote - частина пакета Office, орієнтована на корпоративних користувачів, з підтримкою декількох рівнів вкладеності, аркушів і малювання, але має обмежений функціонал баз даних і мало можливостей візуалізації зв'язків.

Для системного порівняння аналогів було сформовано таблицю критеріїв (див. табл. 1.1).

Таблиця 1.1 - Порівняння аналогів програмних засобів для ведення нотаток

Критерій	**Notion**	**Obsidian**	**Evernote**	**Apple Notes**	**Власна розробка**
Локальне зберігання	–	+	–	+	+
Block-based rich-text	+	–	–	–	+
Граф знань	–	+	–	–	+
Автокласифікація	–	–	+	–	+
Повнотекстовий пошук	+	+	+	+	+
Версіонування	+	+	+	+	+
Безкоштовність	–	+	–	+	+
Підтримка	+	+	+	+	+

української					
Desktop-режим	+	+	+	+	+
Відкритий вихідний код	–	–	–	–	+

Аналіз таблиці 1.1 показує, що жоден з популярних продуктів не задовольняє повністю набір вимог: локальне зберігання, block-based редагування, граф знань, автокласифікація, безкоштовність і відкритий код одночасно. Saturn-точка зі знаком плюс присутня лише у власній розробці, що підтверджує доцільність створення нового програмного засобу. Особливо помітна перевага у наявності модуля автоматичної класифікації, якого бракує більшості локальних аналогів, та у відкритості коду, що дає можливість гнучкого розширення відповідно до потреб користувача.

1.3. Обґрунтування доцільності власної розробки

Виконаний огляд показує, що на ринку існує велика кількість якісних продуктів для роботи з нотатками, проте кожен з них має свої компромісні рішення. Користувач, який прагне поєднати локальність зберігання, block-based редагування, граф зв'язків та автоматичну класифікацію в одному застосунку, не знаходить готового рішення. Це підтверджує доцільність розробки власного програмного засобу, що інтегрує найкращі практики галузі та водночас усуває характерні обмеження окремих систем.

Ще одним аргументом на користь власної розробки є можливість контролю над архітектурою і даними. У хмарних рішеннях користувач не має повної прозорості щодо способів зберігання, кодування та обробки своїх записів. Локальний застосунок з відкритим кодом гарантує, що дані не покидають комп'ютер користувача без його згоди, а механізми класифікації та індексації можна перевірити, налаштувати або вимкнути. Це особливо важливо для конфіденційних робочих документів, наукових нотаток та особистих записів.

Власна розробка також відкриває можливості для подальшого вдосконалення алгоритму класифікації. Якщо у комерційних продуктах правила сортування зазвичай приховані від користувача, то у власному застосунку класифікатор реалізовано як прозорий модуль, що дозволяє розширювати словник ключових слів, змінювати правила і збирати статистику зворотного зв'язку щодо точності. Це створює основу для майбутнього переходу на машинне навчання та глибше адаптовану класифікацію без необхідності зміни всієї архітектури системи. [3]

Окремо варто наголосити на освітній цінності проєкту. Розробка повноцінного багатомодульного застосунку є зразком практичного застосування знань зі всіх ключових дисциплін спеціальності «Комп'ютерні науки»: проєктування програмного забезпечення, роботи з реляційними базами даних, веб-програмування, побудови серверних API, розробки клієнтських інтерфейсів, тестування і документування. Таким чином, створений програмний засіб виконує одночасно дві функції: задовольняє реальну потребу користувачів у зручному нотатнику та слугує комплексним результатом інженерної підготовки автора роботи.

На основі вищенаведеного обґрунтування для подальшої роботи було визначено технологічний стек, архітектурні принципи та функціональний скоп розробки, які детально описані у наступному розділі.

2. ТЕОРЕТИЧНА ЧАСТИНА

2.1. Огляд обраного технологічного стеку

Вибір технологій для розробки програмного засобу здійснено на основі балансу між зрілістю екосистеми, продуктивністю, простотою розгортання і відповідністю задачам проєкту. Для серверної частини обрано платформу Node.js версії 20 LTS, яка забезпечує асинхронне виконання запитів, низький рівень використання пам'яті та одну з найбільших у світі бібліотек відкритих пакетів. Як веб-фреймворк використовується Express - мінімалістичне і добре документоване рішення, що дозволяє швидко описувати маршрути REST API, працювати з middleware-функціями і легко масштабувати кількість маршрутів.

Як основне сховище обрано вбудовану реляційну СКБД SQLite, що поширюється у вигляді бібліотеки і не потребує окремого серверного процесу. Для роботи з SQLite на стороні Node.js використовується пакет better-sqlite3, який забезпечує синхронний інтерфейс і вищу швидкодію порівняно з асинхронними обгортками. Підтримка повнотекстового пошуку реалізована через розширення FTS5, вбудоване у саму СКБД, з токенізацією unicode61 та можливістю видалення діакритичних знаків, що покращує якість пошуку для україномовних текстів.

Клієнтська частина розроблена на бібліотеці React версії 18 з типізацією TypeScript, що дозволяє зменшити кількість помилок завдяки статичній перевірці типів. Збірка клієнтського коду виконується інструментом Vite, який забезпечує швидкий dev-сервер з hot module replacement та оптимізовану продакшн-збірку. Для маршрутизації використовується React Router v6, для побудови rich-text редактора - фреймворк Tiptap, побудований на основі ProseMirror і дозволяє створювати block-based інтерфейси з кастомними розширеннями. Для візуалізації графа знань обрано бібліотеку @xyflow/react, що пропонує продуктивний рушій рендеру вузлів і ребер та підтримує drag&drop переміщення, мінімапу і керовані макети.

Для запуску програмного засобу як desktop-застосунку інтегровано Electron - фреймворк для створення кросплатформних настільних застосунків на технологіях вебу. Electron використовує движок Chromium для рендеру і Node.js для системних викликів, що дозволяє запакувати веб-клієнт і Express-сервер у єдиний виконуваний файл. Для розгортання у форматі serverless у хмарі Netlify реалізовано окрему точку входу через бібліотеку serverless-http, яка адаптує Express-додаток до інтерфейсу AWS Lambda. Загальний вигляд архітектурних шарів стеку показано на рис. 2.1.

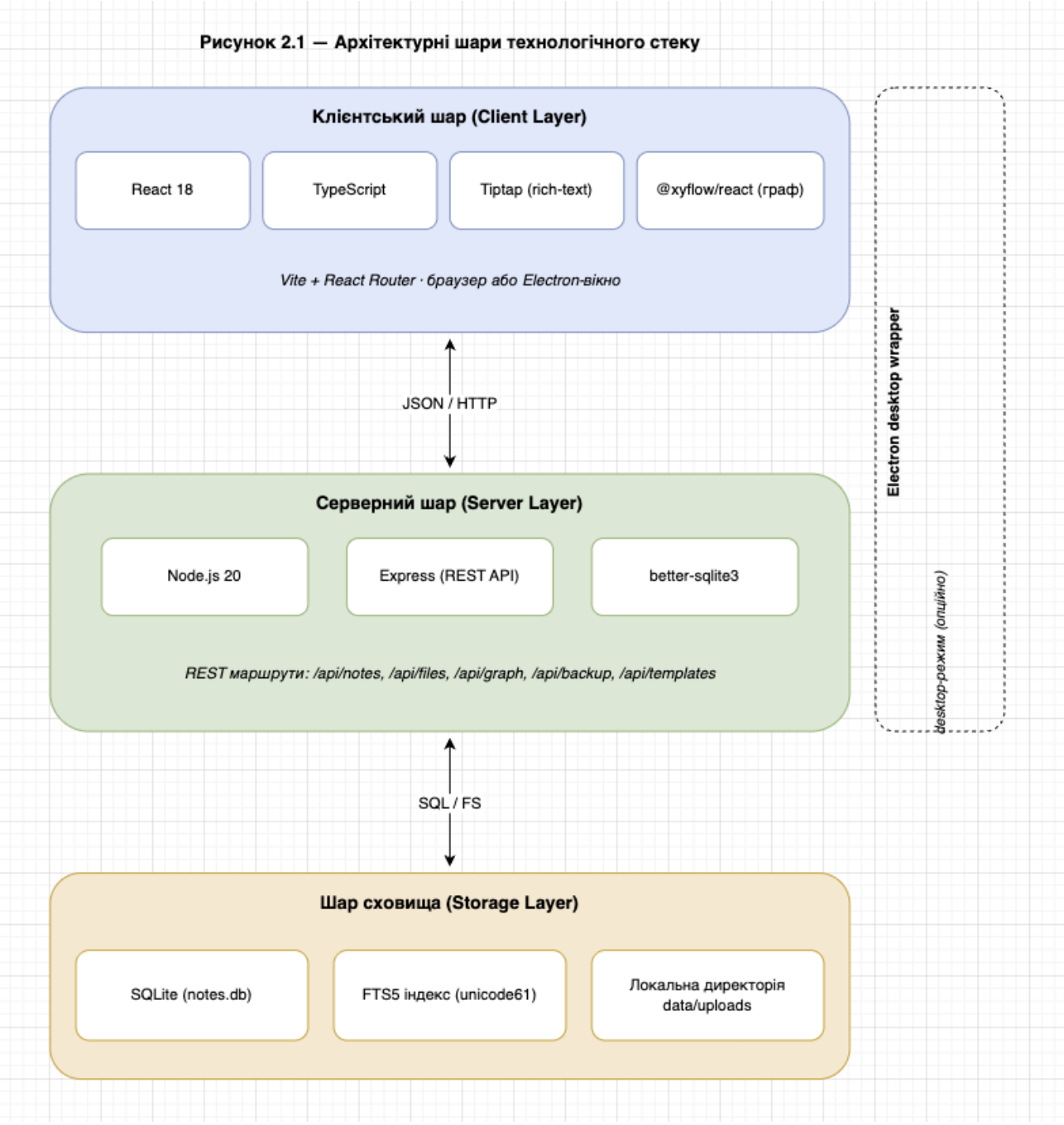

Рисунок 2.1 - Архітектурні шари технологічного стеку

На рисунку 2.1 показано три основні шари: клієнтський (React, TypeScript, Tiptap, @xyflow), серверний (Node.js, Express, better-sqlite3) і шар сховища (SQLite з FTS5-індексом, локальна файлова директорія для вкладень). Усі шари взаємодіють через REST API, що забезпечує модульність і можливість незалежної заміни клієнта або сервера. Така архітектура є типовою для сучасних single-page applications, але водночас залишає широкі можливості для подальшого розширення, зокрема додавання альтернативних клієнтів (мобільного або CLI) без зміни серверної логіки.

2.2. Архітектура клієнт-серверного застосунку та проєктування REST API

Для розроблюваного програмного засобу обрано класичну архітектуру single-page application з відокремленим REST API. Клієнтська частина повністю завантажується у браузер користувача один раз, після чого спілкується з сервером через JSON-запити. Сервер не зберігає стану сесії, кожен запит обробляється незалежно. Така схема відома як stateless REST і забезпечує горизонтальне масштабування серверу за потреби, спрощує тестування та зменшує сполучення між компонентами.

REST API розроблюваного застосунку поділено на логічні групи ресурсів. Перша група - Notes / Docs - реалізує операції створення, читання, оновлення і видалення нотаток, додатково підтримуючи пов'язані операції класифікації, відкату версій та керування зв'язками. Друга група - Files - забезпечує управління файловою бібліотекою з можливістю прив'язки файлів до однієї або кількох нотаток. Третя група - Templates - реалізує шаблони (Лекція, Лабораторна, Зустріч, План диплома), на основі яких можна швидко створювати нові записи. Четверта група - Graph - обслуговує операції з вершинами і ребрами графа знань. П'ята група - Backup і Analytics - надає експорт-імпорт даних та агреговані метрики. [4]

Для кожного ресурсу визначено набір HTTP-методів відповідно до конвенції REST: GET - для читання, POST - для створення, PUT і PATCH - для повного або часткового оновлення, DELETE - для видалення. Кожен маршрут повертає JSON-

документ з даними або з полем error у разі невдачі, а також відповідний HTTP-код стану (200, 201, 204, 400, 404). Тіло запитів обмежено розміром до 15 мегабайт для забезпечення можливості завантаження файлів у форматі base64. Загальний потік виконання типового запиту до серверу зображено на рис. 2.2.

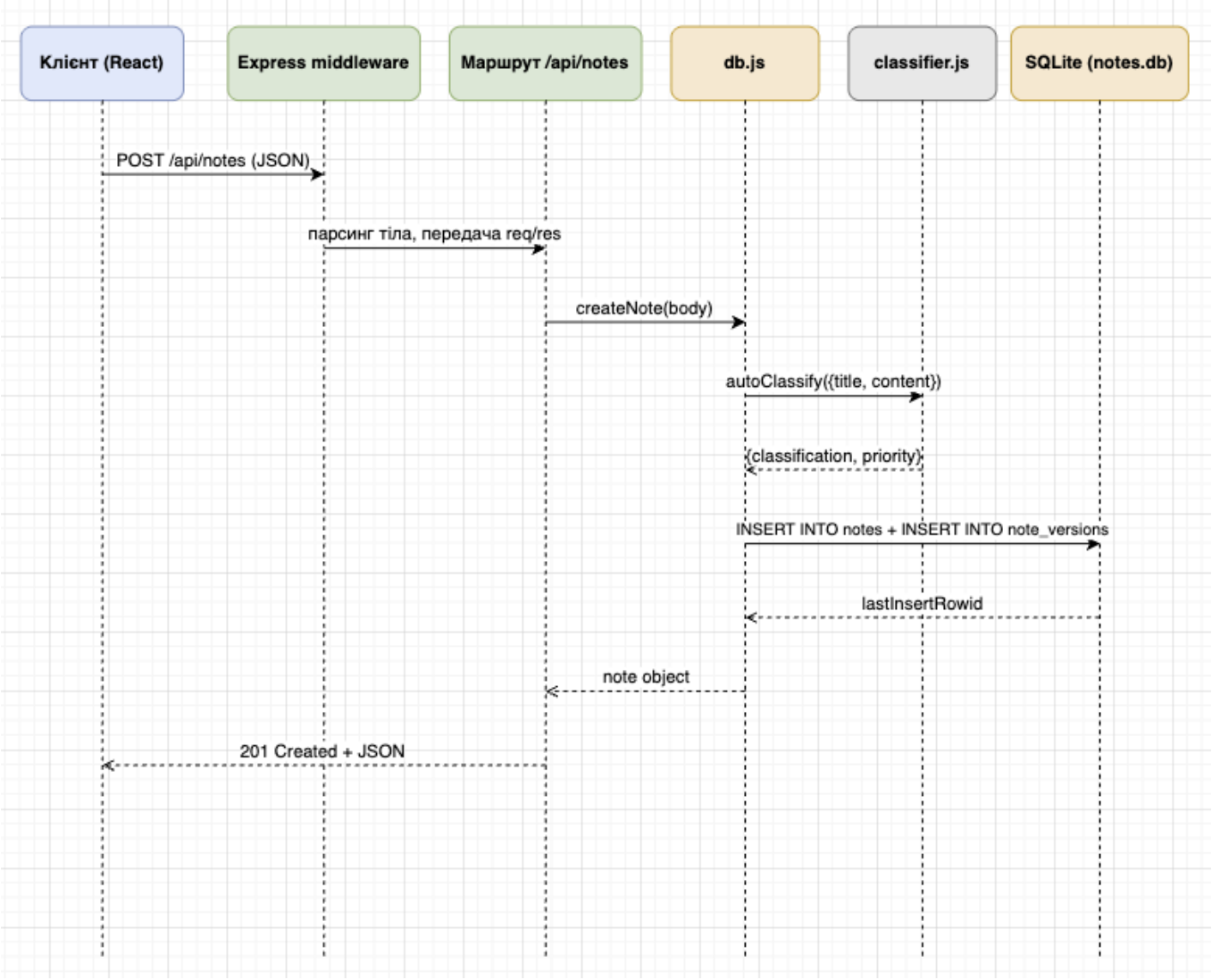

Рисунок 2.2 - Послідовність обробки REST-запиту

На рисунку 2.2 видно етапи: клієнт надсилає запит через бібліотеку fetch, Express приймає його і направляє в потрібний обробник, обробник звертається до модуля db.js для роботи зі сховищем, при потребі викликається класифікатор або менеджер версій, результат повертається у формі JSON. Аналогічно обробляються POST і PUT запити з валідацією тіла. Архітектура передбачає чітке розмежування відповідальностей між шарами: маршрут не звертається безпосередньо до SQLite, а

лише викликає функції з модуля доступу до даних, що спрощує тестування і подальше розширення.

Окремо реалізовано середовище розробки з паралельним запуском Vite-сервера для клієнта (порт 5173) та Express-сервера для API (порт 3000). У продакшн-збірці клієнтський код збирається у директорію client/dist, після чого Express обслуговує і API-маршрути, і статичні файли клієнта. Для зворотної сумісності збережено застаріле fallback-середовище у директорії public, до якого відбувається перехід, якщо нова збірка ще не виконана.

2.3. Модель даних і повнотекстова індексація на основі SQLite FTS5

Реляційна модель даних розроблюваного програмного засобу спроєктована з урахуванням принципів нормалізації та з мінімізацією дублювання. Основною сутністю є нотатка, що зберігається у таблиці notes з полями id, title, content, kind, folder_id, classification, priority, status, due_date, is_archived, created_at, updated_at. Поле kind приймає значення note або doc, що дозволяє розрізняти короткі замітки і повноцінні документи. Поле folder_id посилається на таблицю folders, що містить ієрархічні папки. Поле classification зберігає категорію (Навчання, Робота, Особисте, Ідеї, Дедлайни, Інше), priority - пріоритет (Низький, Середній, Високий), status - статус робочого процесу (Inbox, In Progress, Review, Done). [5]

Для зв'язку нотаток із тегами використано відому схему many-to-many через проміжну таблицю note_tags. Самі теги зберігаються у таблиці tags з унікальною назвою. Для зв'язків між нотатками використано таблицю note_links, яка зберігає пари source_note_id і target_note_id з обмеженням на відсутність самопосилань. Це дозволяє реалізувати backlinks - автоматичне відображення нотаток, які посилаються на поточну, за допомогою симетричних запитів. Версіонування нотаток виконується через таблицю note_versions, кожна версія є знімком значущих полів і ставиться при кожному оновленні. ER-діаграма схеми бази даних показана на рис. 2.3.

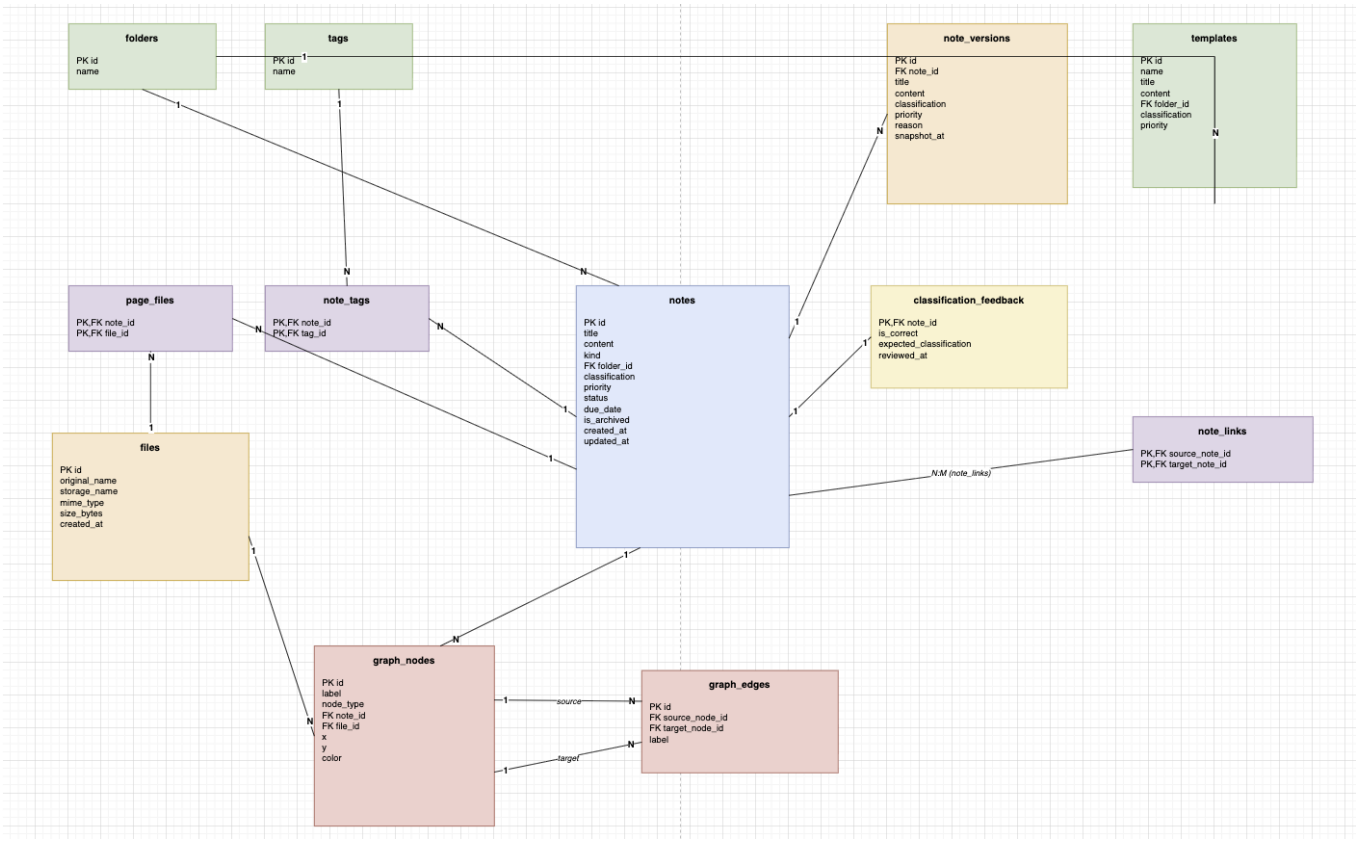

Рисунок 2.3 - Схема бази даних SQLite

На рисунку 2.3 представлено основні таблиці і зв'язки між ними: notes - центральна сутність, до якої прив'язано folders, tags, note_versions, note_links, files (через page_files) та graph_nodes. Сторонні ключі забезпечують каскадне видалення залежних записів при видаленні основної нотатки, що підтримує цілісність бази. Окремий індекс idx_notes_updated_at прискорює сортування за останнім оновленням, що є типовим запитом у списку нотаток. Для пришвидшення фільтрації за класифікацією, папкою, статусом і терміном виконання створено відповідні індекси.

Для забезпечення повнотекстового пошуку використано вбудоване розширення SQLite FTS5. Створюється віртуальна таблиця notes_fts з колонками note_id, title, content, tags та токенайзером unicode61, опція remove_diacritics=2 якого видаляє діакритичні знаки під час токенізації. Це дозволяє знайти, наприклад, нотатку з ключовим словом «об’єкт» при запиті «обєкт». Індекс FTS5 заповнюється функцією rebuildSearchIndex, яка при ініціалізації або після масової зміни даних

перебудовує всі записи. При додаванні чи оновленні нотатки індекс синхронізується тригерами на рівні застосунку.

Особливість реалізації пошуку полягає в тому, що користувацький запит проходить попередню обробку: відкидаються знаки пунктуації, відсікаються токени довжиною менше двох символів, для решти додається оператор префіксного збігу. Така стратегія дозволяє користувачеві ввести часткове слово і отримати результати, що його містять. Якщо запит складається з кількох слів, вони об'єднуються логічним «AND» - у такому разі знайдуться лише ті нотатки, що містять усі введені терміни. У випадках, коли FTS5 недоступний (наприклад, при компіляції SQLite без цього розширення), сервер автоматично переключається на fallback-режим з пошуком через LIKE-запити, що зберігає функціональність ціною продуктивності. [6]

Для роботи з файлами використано два підходи. Перший - таблиця files з полями original_name, storage_name, mime_type, size_bytes, що зберігає метадані про завантажені файли, а самі файли зберігаються в директорії data/uploads з безпечно згенерованою назвою. Зв'язок з нотатками реалізовано через проміжну таблицю page_files, що дозволяє одному файлу бути прикріпленим до кількох нотаток одночасно. Другий - застаріла таблиця note_attachments з безпосереднім збереженням блобу - використовується для зворотної сумісності зі старими бекапами.

2.4. Алгоритм автоматичної класифікації нотаток за категоріями і пріоритетами

Автоматична класифікація нотаток є одним з ключових нефункціональних переваг розроблюваного програмного засобу. Алгоритм покликаний звільнити користувача від ручного присвоєння категорії і пріоритету при створенні нового запису. У сучасних промислових системах класифікація часто базується на моделях машинного навчання, проте для дипломної роботи обрано прозорий правило-базований підхід, який легше пояснити, налагодити і за потреби - розширити. Такий

вибір зумовлено також тим, що для якісного навчання моделі потрібен великий обсяг розмічених даних, який у локальному застосунку відсутній.

Класифікатор реалізовано у модулі classifier.js і складається з двох незалежних функцій: detectClassification - для визначення категорії, та detectPriority - для визначення пріоритету. Категоризація базується на словнику ключових слів, кожне з яких пов'язане з певною категорією. Перебір словника відбувається у фіксованому порядку: «Навчання», «Робота», «Особисте», «Ідеї». Перед перебором додатково перевіряється наявність у тексті дати у форматі дд.мм.рррр або ключового слова «дедлайн» - у такому випадку нотатка отримує категорію «Дедлайни». Якщо жодне з правил не спрацювало, призначається категорія «Інше».

Визначення пріоритету виконується аналогічно: у тексті шукаються характерні маркери. Високий пріоритет вмикається, якщо знайдено слова «терміново», «asap», «urgent», «дедлайн сьогодні» або «критично». Середній пріоритет вмикається при наявності слів «до кінця тижня», «план», «підготувати» або «уточнити». В інших випадках присвоюється низький пріоритет. Логіка реалізована за принципом «перше спрацювання виграє», що забезпечує детермінований результат при різних поєднаннях слів. [7]

Слабкою стороною такого підходу є залежність від словника. Якщо користувач веде нотатки рідкісною термінологією або у певній вузькій галузі, словник може не покривати ключових слів. Для пом'якшення цього обмеження реалізовано окремий механізм зворотного зв'язку: на сторінці кожної нотатки користувач може позначити, чи правильно вона була класифікована, і вказати очікувану категорію. Записи зберігаються у таблиці classification_feedback і слугують основою для подальшого аналізу та доповнення словника. У майбутньому ці дані можуть бути використані як набір для навчання простішої моделі класифікації.

Загальна послідовність роботи класифікатора при створенні нової нотатки показана на рис. 2.4.

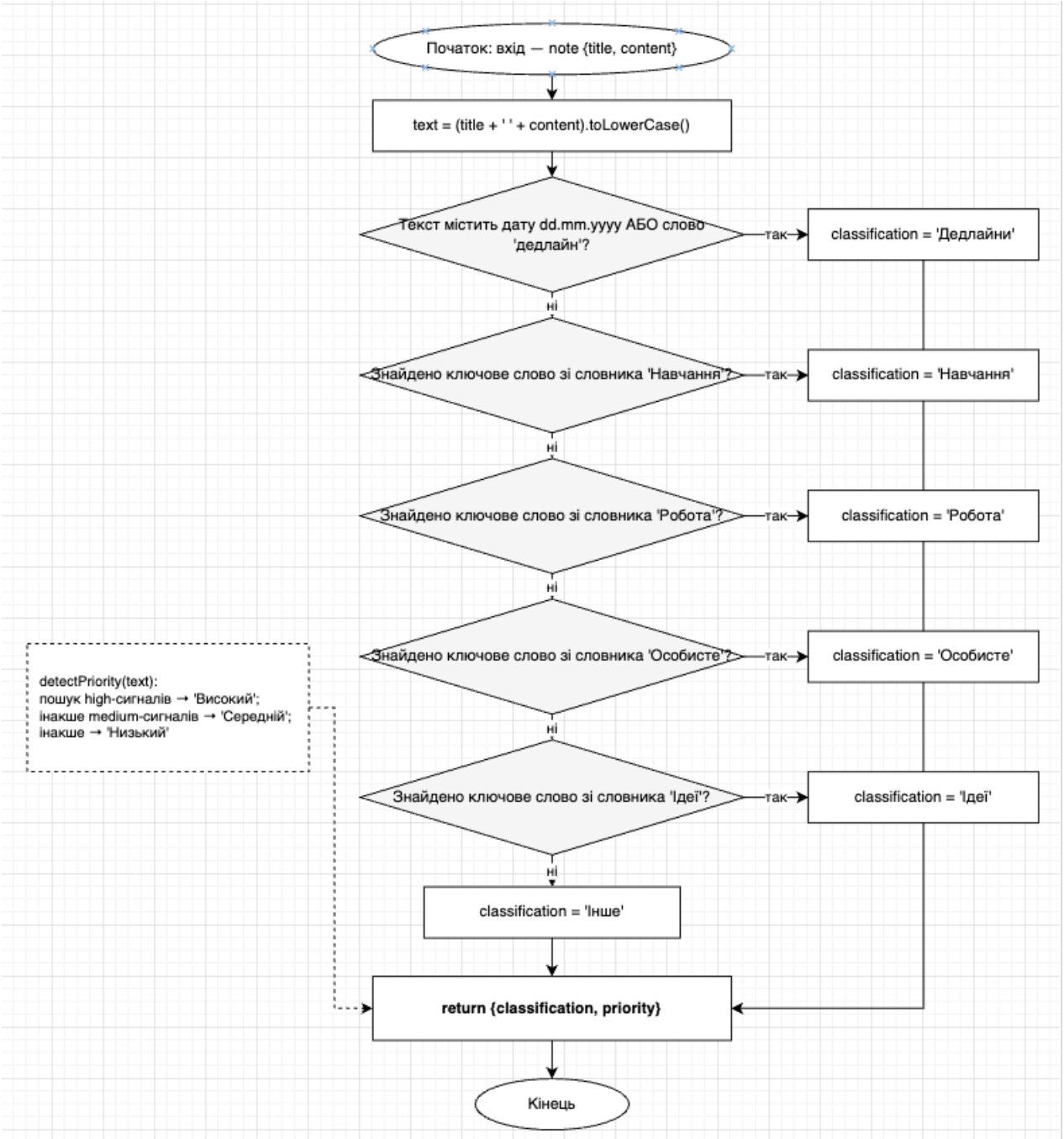

Рисунок 2.4 - Блок-схема алгоритму автокласифікації нотаток

На рисунку 2.4 видно, що вхідними даними для алгоритму є об'єкт нотатки з полями title і content, які об'єднуються в єдиний рядок і нормалізуються до нижнього регістру. Далі виконується перевірка на наявність дати або слова «дедлайн». Після цього послідовно перебираються правила категорії. Паралельно тим самим рядком обробляється детектор пріоритету. Результатом є об'єкт із полями classification і priority, який повертається серверу і записується у таблицю

notes. Час виконання алгоритму на нотатці середнього обсягу (близько 2000 символів) становить менше 1 мілісекунди, що робить його прозорим для користувача.

2.5. Граф знань як інструмент структурування інформації

Граф знань - структура даних, у якій сутності зображено вершинами, а зв'язки між ними - ребрами. У контексті електронних нотатників граф знань дозволяє користувачеві наочно бачити, як між собою пов'язані окремі записи, теми, файли і концепції. Цей підхід здобув популярність завдяки таким продуктам, як Obsidian і Roam Research, де він є центральним елементом інтерфейсу. У розроблюваному програмному засобі граф знань реалізовано як окремий модуль (Graph Studio), який доповнює основну функціональність ведення нотаток.

Технічно модуль складається з двох таблиць у базі даних: graph_nodes - для вершин і graph_edges - для ребер. Кожна вершина має поля id, label, node_type (page, doc, file, topic), note_id, file_id, координати x і y, колір, дати створення та оновлення. Поля note_id і file_id дозволяють прив'язати вершину до конкретної нотатки або файлу, що створює мостик між основним сховищем і графом. Якщо ні нотатка, ні файл не вказані, вершина вважається абстрактною і відносить до типу topic. Ребра представлено таблицею graph_edges з полями source_node_id, target_node_id, label. [8]

Зображення графа знань реалізовано на бібліотеці @xyflow/react - нащадку популярної React Flow. Бібліотека пропонує декларативний API для опису вершин і ребер, підтримує drag&drop переміщення, масштабування, мінімапу і керовані розкладки. Для адаптації координат до екрана при початковому завантаженні застосовано алгоритм Фрухтермана-Рейнгольда, що автоматично розташовує вершини у вигляді відштовхувальних і притягальних пружин, забезпечуючи мінімальне перекриття. Користувач може зберегти ручне розташування - після пересування вершини її координати автоматично записуються до бази даних, що дозволяє відновити макет при наступному відкритті.

Окремо передбачено інтеграцію графа з основним інтерфейсом нотаток: при кліку по вершині відбувається перехід на сторінку нотатки або файлу, з якою вершина пов'язана. Це створює зручну навігацію для користувача, який досліджує структуру своїх записів. Окремо граф зберігає метадані для аналітики - кількість зв'язків кожної вершини (степінь графа) і потенційні кластери, які можуть бути виявлені простими алгоритмами на кшталт DFS. Поточна реалізація не виконує автоматичної кластеризації, проте архітектура дозволяє додати її у майбутньому.

3. ПРАКТИЧНА ЧАСТИНА

3.1. Структура проекту та налаштування середовища розробки

Проєкт організовано у вигляді монорепозиторію з кореневим файлом package.json для серверної частини і вкладеною директорією client з окремим package.json для клієнтської частини. Така структура спрощує паралельну розробку і дозволяє керувати залежностями обох шарів незалежно. У корені знаходяться також директорії src з серверним кодом, scripts з утилітами на кшталт seed-скрипту, desktop з модулем Electron-обгортки, public з застарілою fallback-збіркою клієнта, data зі сховищем SQLite та довідковими прикладами. Окремо створено директорію .claude з файлами контексту для майбутньої співпраці з AI-інструментами.

Серверна частина у директорії src складається з трьох файлів: server.js - точка входу з оголошенням REST-маршрутів і middleware-функцій; db.js - модуль доступу до даних з усією логікою роботи зі схемою, версіонуванням, бекапом та класифікацією; classifier.js - окремий модуль автокласифікації. Такий поділ дотримується принципу єдиної відповідальності: маршрути відповідають за обробку HTTP-запитів і не знають подробиць про SQLite, модуль доступу до даних інкапсулює всі SQL-запити, а класифікатор містить ізольовану логіку, яку легко покрити юніт-тестами або замінити на іншу реалізацію.

Клієнтська частина побудована на стандартній структурі Vite-проєкту. Точкою входу є файл client/src/main.tsx, який рендерить кореневий компонент App у вузол DOM. Файл client/src/App.tsx містить компонент верхнього рівня з боковою панеллю та маршрутизатором React Router. Сторінки знаходяться в client/src/pages: NotesPage.tsx - сторінка зі списком нотаток і редактором, GraphPage.tsx - сторінка з графом знань. Допоміжні модулі - api.ts (тонкий обгортковий клас навколо fetch для виклику серверних маршрутів) та types.ts (TypeScript-визначення спільних типів). Стилі зібрано в один файл styles.css з використанням CSS-змінних. [9]

Середовище розробки налаштовано таким чином, щоб мінімізувати тертя при щоденній роботі. Команда `npm install` встановлює залежності серверної частини, після чого `npm --prefix client install` встановлює залежності клієнтської частини. Команда `npm run seed` наповнює базу даних демонстраційними нотатками, тегами, шаблонами і прикладами файлів. Запуск у режимі production виконується командою `npm run client:build` для збірки клієнта і `npm start` для серверу. У режимі розробки запускаються два процеси: `npm run dev` для серверу і `npm run client:dev` для Vite-серверу - після чого Vite-сервер проксіює запити /api на бекенд. Для desktop-режиму використовується команда `npm run desktop:new`, яка спочатку збирає клієнт, а потім запускає Electron-обгортку.

3.2. Реалізація серверної частини на Node.js і Express

Серверна частина побудована на платформі Node.js 20 та фреймворку Express 4. Точкою входу є файл src/server.js, в якому виконуються чотири основні дії: ініціалізація підключення до SQLite через модуль db.js, запуск seed-функції з прапорцем reset=false (вона додає демо-дані лише за умови, що база порожня), оголошення middleware-функцій для парсингу JSON та статичної роздачі клієнтських файлів і реєстрація REST-маршрутів. Для коректної роботи у двох сценаріях розгортання - локальному виконанні та serverless-функції на Netlify - реалізовано перевірку через `require.main === module`: якщо модуль завантажений як точка входу, викликається `app.listen`, інакше експортується сам застосунок для подальшого використання обгорткою serverless-http.

Маршрути серверу організовано у логічні групи відповідно до ресурсів. Група «notes» обслуговує URL-шаблони `/api/notes`, `/api/notes/:id` та похідні з модифікаторами classify, classification-feedback, links, files, attachments, versions, rollback. Група «files» обслуговує `/api/files` з маршрутами для завантаження, попереднього перегляду, перейменування та видалення. Група «graph» - `/api/graph`, `/api/graph/nodes`, `/api/graph/edges`. Група «backup» - `/api/backup/export` і `/api/backup/import`. Сервер також підтримує маршрути аналітики: `/api/dashboard`,

`/api/analytics`, `/api/knowledge-map`, які повертають агреговані метрики по кількості нотаток, розподілу за класифікаціями та статусами, термінах виконання.

Загальна обробка завантажень файлів реалізована за допомогою власної функції parseUploadPayload, яка очікує JSON-тіло з полями fileName, mimeType і dataBase64. Поле dataBase64 декодується у Buffer і перевіряється на максимальний розмір 10 мегабайт. Безпечне ім'я файлу формується через safeFileName, що замінює нелегальні символи на підкреслення і обмежує довжину 180 символами. Унікальне ім'я файлу у сховищі (storageName) формується через комбінацію поточного timestamp, випадкового суфіксу і безпечного імені. Файли записуються у директорію data/uploads, а у разі помилки під час фіксації у базі даних файл видаляється з диска для уникнення «осиротілих» файлів. Послідовність обробки запиту на завантаження файлу показана на рис. 3.1.

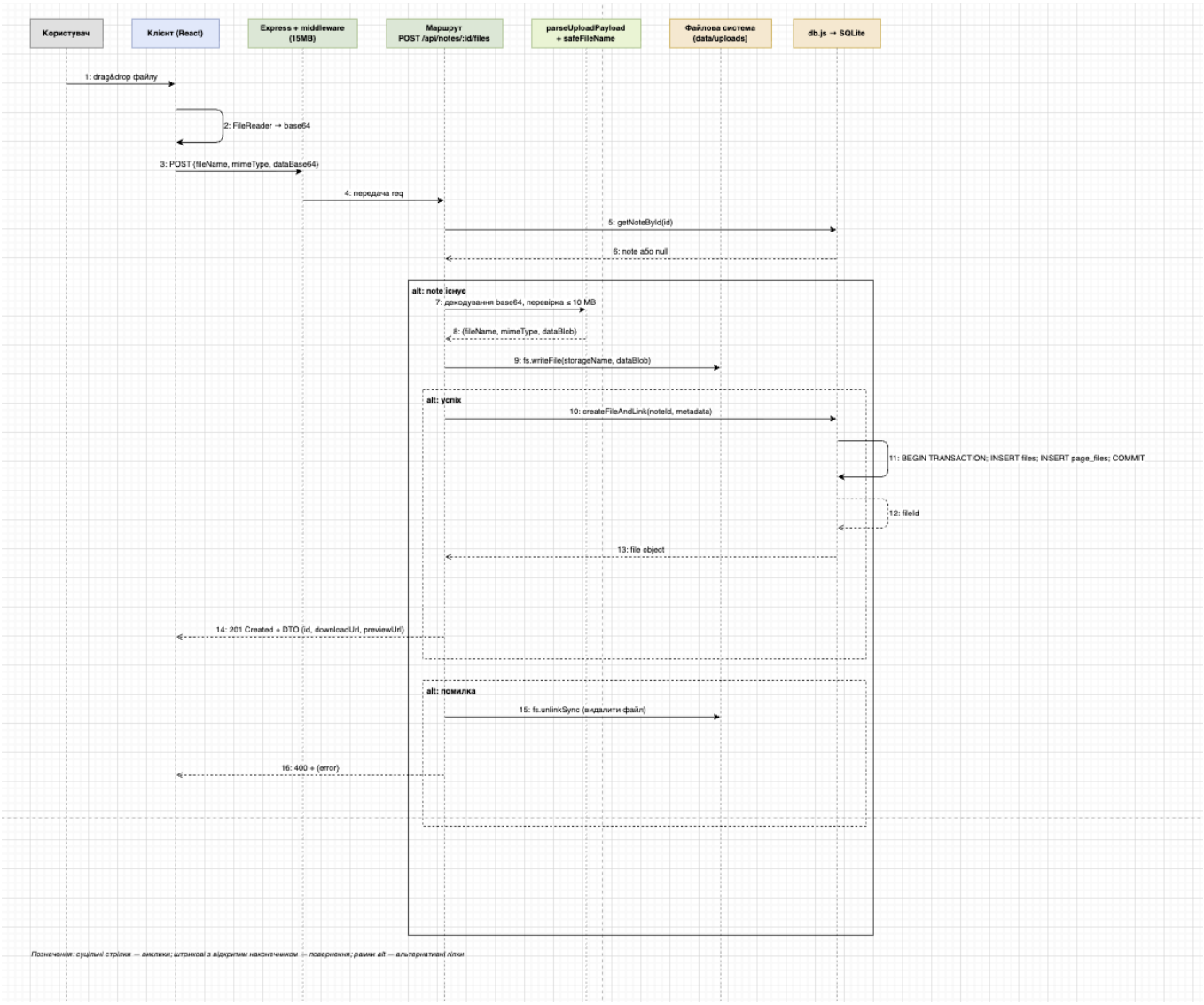

Рисунок 3.1 - Послідовність завантаження файлу до нотатки

На рисунку 3.1 показано всі етапи: клієнт надсилає JSON із dataBase64; Express виконує middleware express.json і збільшує ліміт до 15 МБ; маршрут перевіряє наявність нотатки; функція parseUploadPayload декодує base64 і перевіряє розмір; формується storageName і файл записується на диск; виконується транзакція в SQLite з вставкою у таблицю files і зв'язку у page_files; у разі помилки файл видаляється; клієнту повертається DTO-об’єкт з полями id, originalName, mimeType, sizeBytes, downloadUrl і previewUrl.

Усі помилки маршрутів обробляються однаково: блок try/catch перехоплює виключення і повертає JSON {error: message} зі статусом 400 для помилок валідації, 404 - для відсутніх ресурсів, 500 - для непередбачуваних. Для CORS-заголовків і

обмеження методів окрема логіка не застосовується, оскільки клієнт і сервер обслуговуються тим самим Express-додатком. У середовищі розробки запити з порту 5173 (Vite) перенаправляються через проксі на порт 3000, що автоматично обходить обмеження same-origin. Завершальний middleware відповідає за SPA-fallback: якщо шлях не починається з /api/, повертається index.html, що дозволяє React Router обробити маршрут на стороні клієнта. [10]

3.3. Реалізація бази даних, версіонування та повнотекстового пошуку

Модуль src/db.js інкапсулює всю логіку роботи з SQLite. На етапі ініціалізації виконується серія операцій. Перша - відкриття або створення файлу бази даних. Шлях обчислюється динамічно: у режимі serverless використовується тимчасова директорія /tmp, у локальному режимі - data/notes.db у корені проєкту. Друга - встановлення прагми journal_mode=WAL, що активує журналювання з випереджувальним записом і помітно прискорює операції одночасного читання-запису. Третя - увімкнення foreign_keys=ON для коректної підтримки сторонніх ключів. Четверта - виконання DDL-сценарію CREATE TABLE IF NOT EXISTS для всіх таблиць схеми.

Окремо реалізовано функцію ensureColumn, яка виконує ALTER TABLE ADD COLUMN з обробкою помилки про дубльовану колонку. Це дозволяє безпечно додавати нові поля до схеми у наступних версіях програмного засобу без необхідності окремої міграційної системи. У такий спосіб додані поля kind, status, due_date у таблицю notes, а також node_type і file_id у таблицю graph_nodes. Подібний підхід застосовано і для індексів - у середині initDb виконуються CREATE INDEX IF NOT EXISTS для всіх потрібних індексів, що уможливлює оновлення без втрати даних.

Версіонування нотаток реалізовано через таблицю note_versions, у яку записується знімок при кожному оновленні. Знімок містить копію значущих полів: title, content, kind, folder_id, classification, priority, status, due_date, tags_text, related_ids_text та поле reason для фіксації причини зміни (наприклад, «Оновлення

нотатки» або «Створення з шаблону»). Функція listNoteVersions повертає історію у зворотному хронологічному порядку з обмеженням за параметром limit. Функція rollbackNoteVersion виконує транзакцію: спочатку фіксує поточний стан нотатки як новий знімок з reason «Відкат до версії», потім накладає вміст обраного знімка на основну таблицю notes і повертає актуальний об'єкт.

Повнотекстова індексація реалізована через FTS5-таблицю notes_fts. Функція ініціалізації initFts створює віртуальну таблицю з токенайзером unicode61 і обробкою помилок: якщо розширення FTS5 не доступне у поточній збірці SQLite, прапорець ftsEnabled встановлюється у false, і пошук переходить у fallback-режим через LIKE-запити. Функція rebuildSearchIndex очищає таблицю notes_fts і заповнює її поточним вмістом таблиці notes, що дозволяє відновити індекс після імпорту бекапу або збоїв. При створенні чи оновленні нотатки відповідні рядки в notes_fts оновлюються синхронно у тому ж транзакційному блоці, що і основна таблиця, забезпечуючи цілісність даних. На рис. 3.2 показано приклад роботи повнотекстового пошуку у користувацькому інтерфейсі.

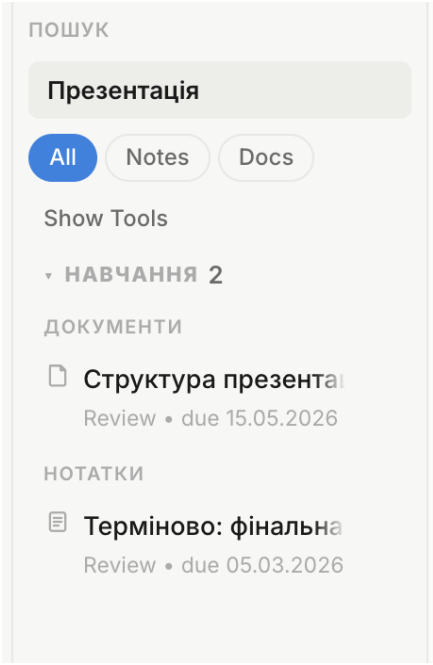

Рисунок 3.2 - Приклад роботи повнотекстового пошуку

На рисунку 3.2 показано список нотаток, відфільтрований за пошуковим запитом «алгоритм». Серед результатів - конспект лекції з алгоритмів сортування, фрагмент дослідження з оптимізації запитів, чернетки ідей про побудову нового алгоритму класифікації. Часткові збіги (наприклад, «алгоритмічний») також враховуються завдяки префіксному оператору, що додається до кожного токена пошукового запиту.

3.4. Реалізація модуля автоматичної класифікації

Модуль src/classifier.js повністю інкапсулює логіку класифікації нотаток. Він не залежить від модуля доступу до даних і не містить імпорту з інших файлів проєкту, що робить його ідеальним кандидатом для юніт-тестування. Експортує модуль єдину публічну функцію autoClassify, яка приймає об'єкт нотатки з полями title і content, а повертає об'єкт з полями classification і priority. Усередині використовуються дві допоміжні функції: detectClassification і detectPriority.

Словник правил оголошено як константу CLASSIFICATION_RULES - масив об'єктів виду {classification, keywords}. Поточна редакція містить чотири категорії: «Навчання» з ключовими словами на кшталт «лекція», «лаба», «курс», «диплом»; «Робота» - «клієнт», «проєкт», «спринт», «беклог»; «Особисте» - «здоров», «спорт», «сім'я», «відпочинок»; «Ідеї» - «ідея», «гіпотеза», «brainstorm», «mvp». Перш ніж пройти словником, функція detectClassification перевіряє, чи містить текст дату у форматі дд.мм.рррр (через регулярний вираз `\b\d{1,2}[./-]\d{1,2}[./-]\d{2,4}\b`) або слово «дедлайн». У такому разі повертається спеціальна категорія «Дедлайни», що відрізняє нотатки з прив'язкою до календаря.

Функція detectPriority побудована за схожим принципом, проте її словники сегментовано на високі і середні сигнали. Високі - «терміново», «asap», «urgent», «дедлайн сьогодні», «критично». Середні - «до кінця тижня», «план», «підготувати», «уточнити». Якщо у тексті є хоча б один високий сигнал - повертається «Високий», у разі знаходження середнього - «Середній», інакше -

«Низький». Послідовність перевірок гарантує детермінованість і дозволяє користувачеві експериментувати з впливом тих чи інших ключових слів.

Інтеграція класифікатора у серверну частину виконана через функцію classifyNote у модулі db.js. Маршрут POST /api/notes/:id/classify приймає ідентифікатор нотатки, завантажує її повний об'єкт, передає його у autoClassify, оновлює відповідні поля у таблиці notes і записує знімок у note_versions з reason «Перекласифікація». Це дозволяє користувачеві викликати перекласифікацію вручну, наприклад, після значної правки тексту. Поряд з ним маршрут PUT /api/notes/:id/classification-feedback приймає від користувача підтвердження правильності класифікації, що зберігається у classification_feedback. Модуль аналітики потім використовує ці дані для оцінки точності у вигляді частки правильних відповідей серед усіх отриманих відгуків.

3.5. Реалізація клієнтського інтерфейсу на React і Tiptap

Клієнтська частина побудована як single-page application на бібліотеці React 18 з використанням TypeScript. Корінь застосунку рендериться у компоненті App, що містить бокову навігаційну панель і React Router з двома основними маршрутами: /notes (сторінка нотаток) і /graph (сторінка графа). Інших маршрутів передбачено мінімальну кількість, оскільки вся логіка зосереджена у двох сценаріях користування. Головний інтерфейс програмного засобу зображено на рис. 3.3.

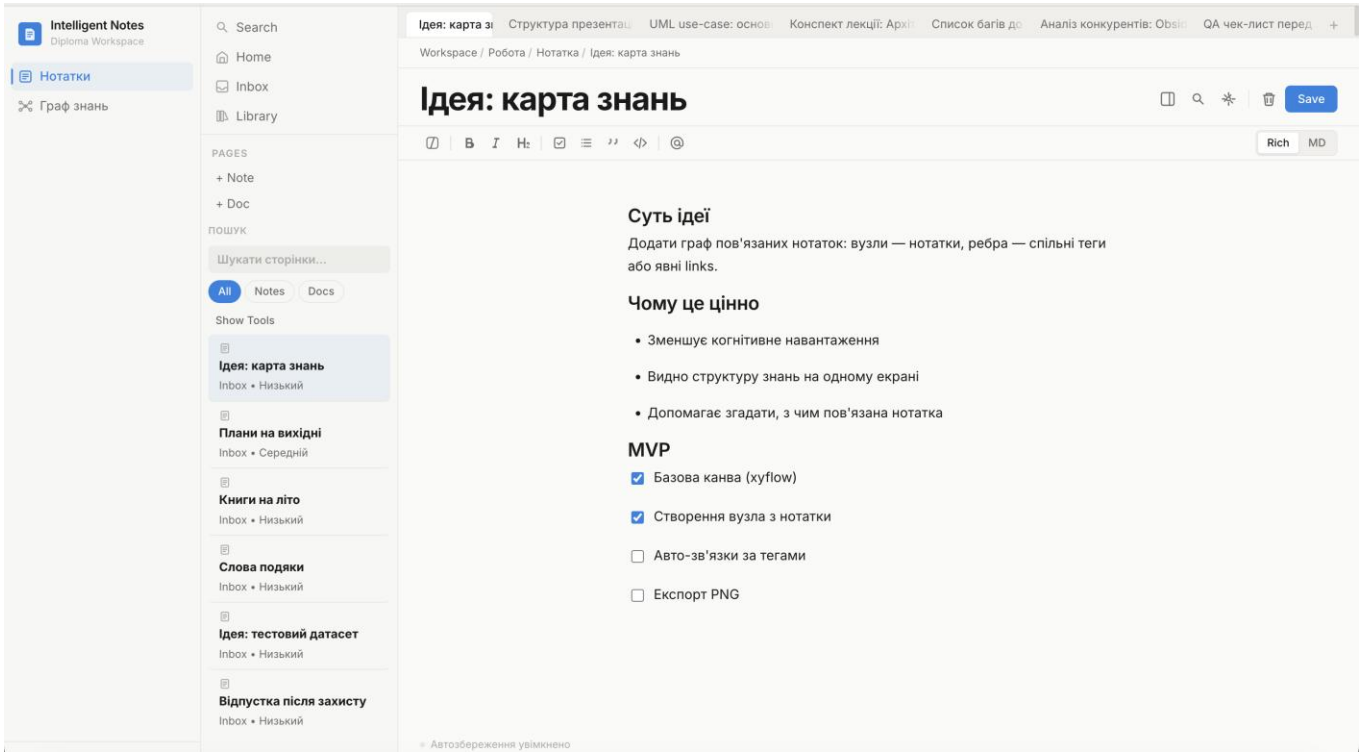

Рисунок 3.3 - Головний інтерфейс програмного засобу Intelligent Notes

На рисунку 3.3 видно тристовпчасту структуру: ліва панель містить навігацію зі списком папок, тегів і робочих наборів («Сьогодні», «Терміново», «Інбокс»); центральна панель містить список нотаток з можливістю переключення між видами Tree, Table, Board і List; права панель - інспектор з властивостями обраної нотатки і кнопками класифікації, відкату версії, прив'язки файлів. Бренд-зона у верхньому лівому куті відображає назву Intelligent Notes і підпис «Diploma Workspace». Вибраний у даний момент вид підсвічується синім, що дозволяє швидко орієнтуватися. Кожна нотатка у центральному списку має заголовок, фрагмент вмісту, чіпи з категорією і пріоритетом та позначку терміну виконання, якщо вона задана.

Сторінка NotesPage реалізована як великий компонент із локальним станом, у якому зберігається обраний вид (mode), фільтри (search, classification, status, folderId), список нотаток і вибрана нотатка. Дані завантажуються через api.ts, який поверх fetch додає базовий обробник помилок і відсутності тіла (статус 204). Усі дії - створення, оновлення, видалення, класифікація, прив'язка файлів - виконуються через відповідні методи api і оновлюють локальний стан без перезавантаження

сторінки. Для покращення продуктивності враховано техніку «оптимістичного оновлення»: спочатку зміни відображаються у UI, а потім підтверджуються відповіддю серверу.

Rich-text редактор реалізовано на основі фреймворка Tiptap. Tiptap побудований на ProseMirror і дозволяє створювати block-based редактори з декларативним описом розширень. У проєкті використовується пакет @tiptap/starter-kit, який включає базові розширення (heading, paragraph, list, blockquote, codeBlock, hardBreak), а також додаткові @tiptap/extension-task-list для чекбоксів. Власні модифікації стосуються поведінки клавіш: наприклад, символ слешу на початку рядка викликає slash-меню з пропозиціями вставити певний блок (заголовок, список, цитату, callout, divider, code block). Такий механізм відомий за Notion і помітно прискорює форматування. Інтерфейс редактора з відкритим slash-меню зображено на рис. 3.4.

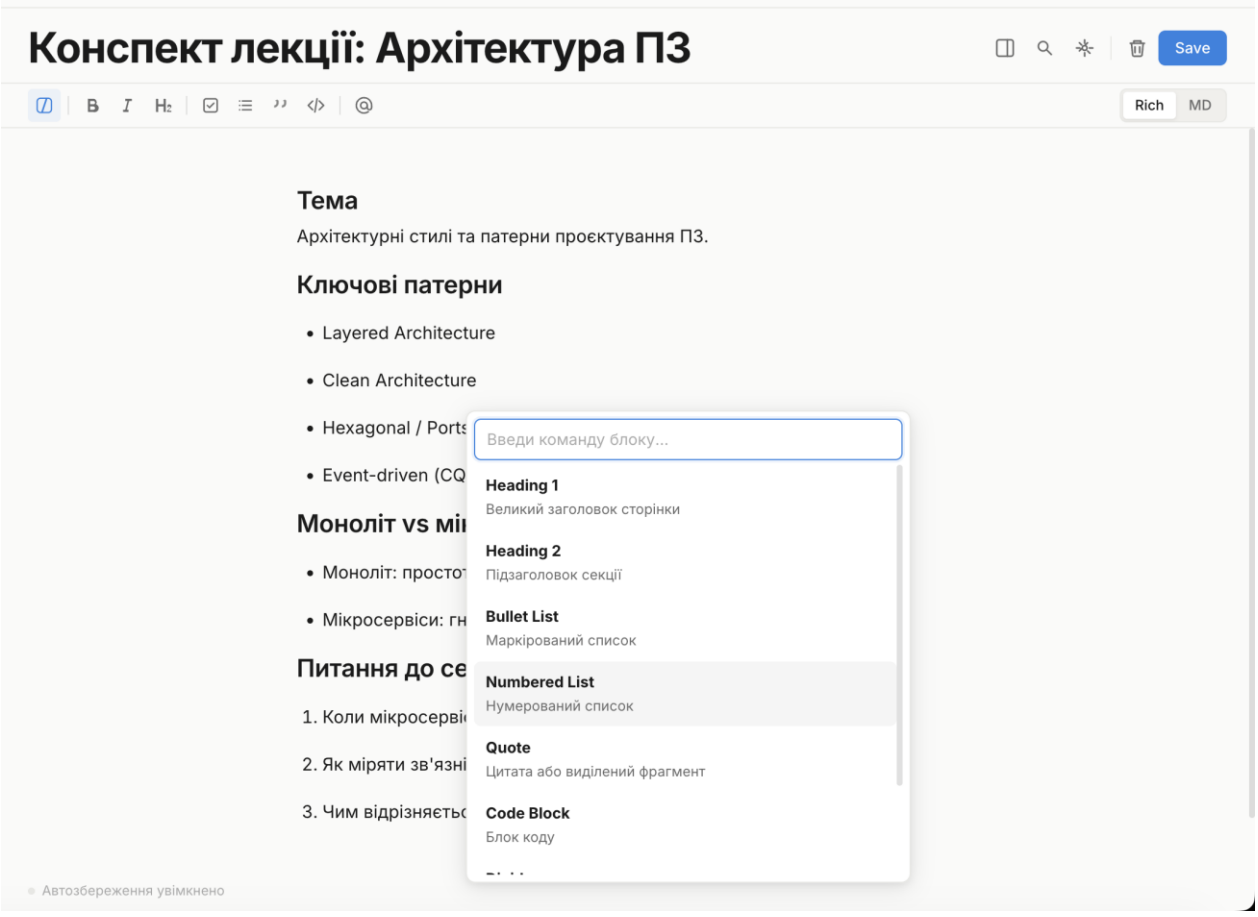

Рисунок 3.4 - Інтерфейс редактора Tiptap зі slash-меню

На рисунку 3.4 показано, як користувач набрав символ «/», після чого з'явився список варіантів вставки. Кожен пункт супроводжується іконкою та коротким описом. При виборі пункту до редактора автоматично вставляється відповідний блок, після чого фокус залишається у позиції редагування. Ця взаємодія знижує когнітивне навантаження користувача і дозволяє формувати документ без використання миші.

3.6. Реалізація модуля графа знань

Модуль графа знань реалізовано як окрему сторінку GraphPage, доступну за маршрутом /graph. Основою візуалізації є компонент @xyflow/react, який приймає масив вузлів і ребер та забезпечує усю логіку рендеру, переміщення, масштабування і взаємодії. Дані завантажуються через api.getGraph, що повертає об'єкт {nodes, edges}, після чого в локальному стані зберігається повний знімок графа. Користувач може створювати нові вершини через панель інструментів зверху сторінки, де вказується мітка, тип, опціональна прив'язка до нотатки чи файлу і колір. Створення ребер відбувається перетягуванням з'єднувальної точки одної вершини до іншої.

Для початкового позиціонування вершин використано простий алгоритм адаптивної решітки. Якщо для вершини у базі не задано координат, вона розташовується у вільному осередку сітки розміром 200 на 200 пікселів. Якщо вершин у графі більше, ніж осередків у поточному вікні, додатково спрацьовує крок ущільнення з 75-відсотковим зменшенням. Перетягування зберігається в базу даних у транзакції оновлення координат - таким чином користувач може вручну побудувати макет, який відповідає його ментальній карті. Зразок графа знань, що відображає зв'язки між нотатками і файлами, представлено на рис. 3.5.

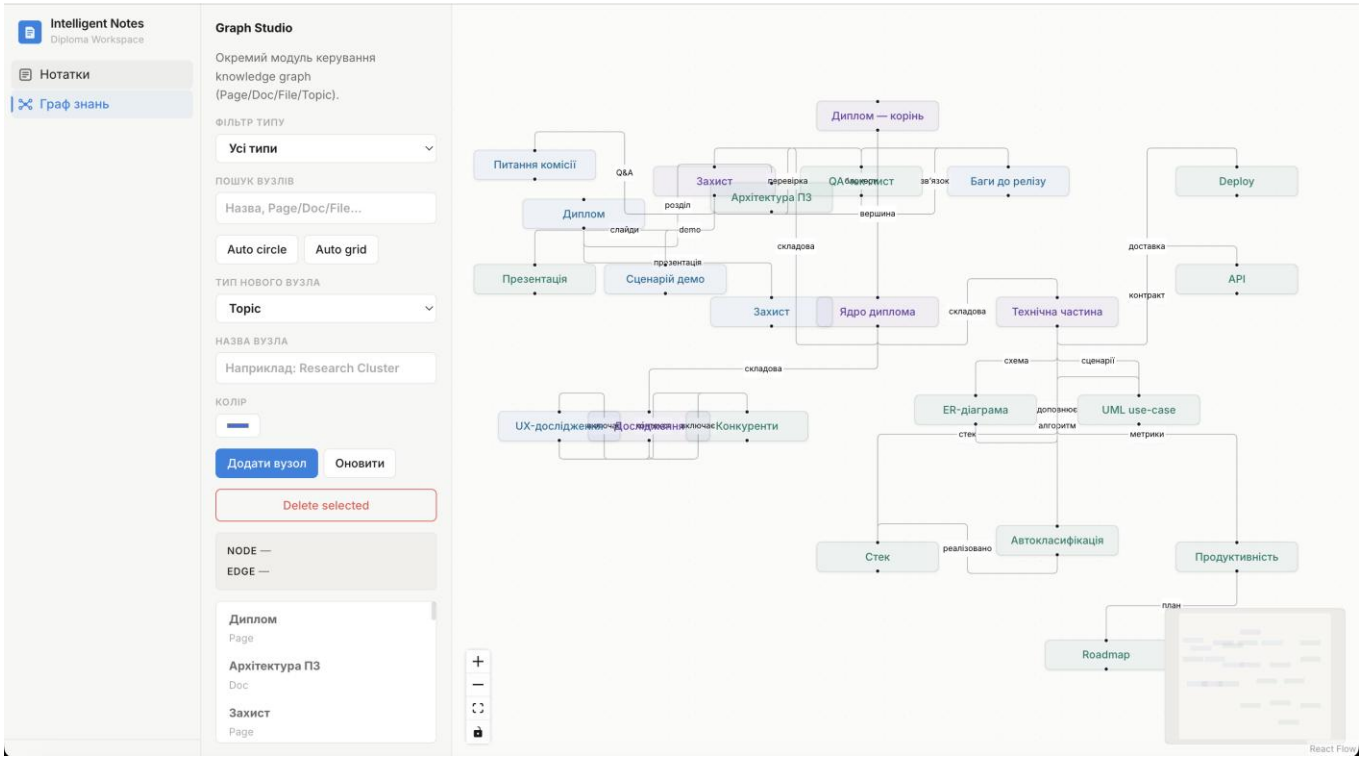

Рисунок 3.5 - Граф знань з вершинами різного типу

На рисунку 3.5 продемонстровано граф з вершинами чотирьох типів: page (сині), doc (бірюзові), file (помаранчеві) та topic (сірі). Зв'язки представлені орієнтованими ребрами з міткою. Видно, що навколо вершини «Дипломна робота» згруповано декілька пов'язаних вершин: «Tiptap», «SQLite FTS5», «React Router», файлові вершини з прикріпленими діаграмами і темою «Класифікація». Саме така візуалізація дозволяє ідентифікувати кластери знань і виявляти прогалини у структурі.

Окремо передбачено інтерактивний перехід: подвійний клік по вершині, прив'язаній до нотатки, відкриває цю нотатку у новій вкладці чи модальному вікні. Це створює зворотний зв'язок між модулем нотаток і графом, перетворюючи модулі з ізольованих на синергетичні.

3.7. Реалізація бекапу та відновлення даних

Резервне копіювання є критичною функцією для будь-якого локального застосунку. У програмному засобі Intelligent Notes реалізовано повноцінний

механізм експорту та імпорту даних у форматі JSON. Експорт викликається маршрутом GET /api/backup/export. Сервер виконує функцію getBackupSnapshot, яка послідовно зчитує всі таблиці бази даних і збирає об'єкт із полями notes, tags, note_tags, folders, note_versions, note_links, classification_feedback, templates, graph_nodes, graph_edges, files і page_files. Для файлів додатково виконується читання їх вмісту з директорії data/uploads та упаковка у формат base64 - це дозволяє відновити повну функціональність після імпорту на новій машині.

Розмір типового знімка для бази з 200 нотатками і 30 файлами становить близько 4 мегабайт у JSON-форматі. Для більшої компактності знімок не включає віртуальної таблиці notes_fts, оскільки індекс відновлюється функцією rebuildSearchIndex одразу після імпорту. Знімок містить також тимчасову мітку версії схеми (snapshotAt) і номер версії програмного засобу (productVersion), що дозволяє виявляти несумісні бекапи у майбутньому. Імпорт викликається маршрутом POST /api/backup/import з тілом запиту, ідентичним до результату експорту, та необов'язковим параметром replace.

Логіка імпорту виконується у транзакції бази даних. Якщо параметр replace=true, поточні дані видаляються перед вставкою. У разі replace=false виконується merge: новостворені записи додаються як копії з новими ідентифікаторами, конфлікти за унікальними полями (наприклад, назвами папок) розв'язуються шляхом використання існуючих записів. Після завершення імпорту викликається rebuildSearchIndex для відновлення FTS5-індексу. Файли з base64-вмістом записуються на диск з відновленням оригінальних storageName, при цьому контролюється унікальність імен. Сценарій імпорту знімка з можливістю заміни існуючих даних показано на рис. 3.6.

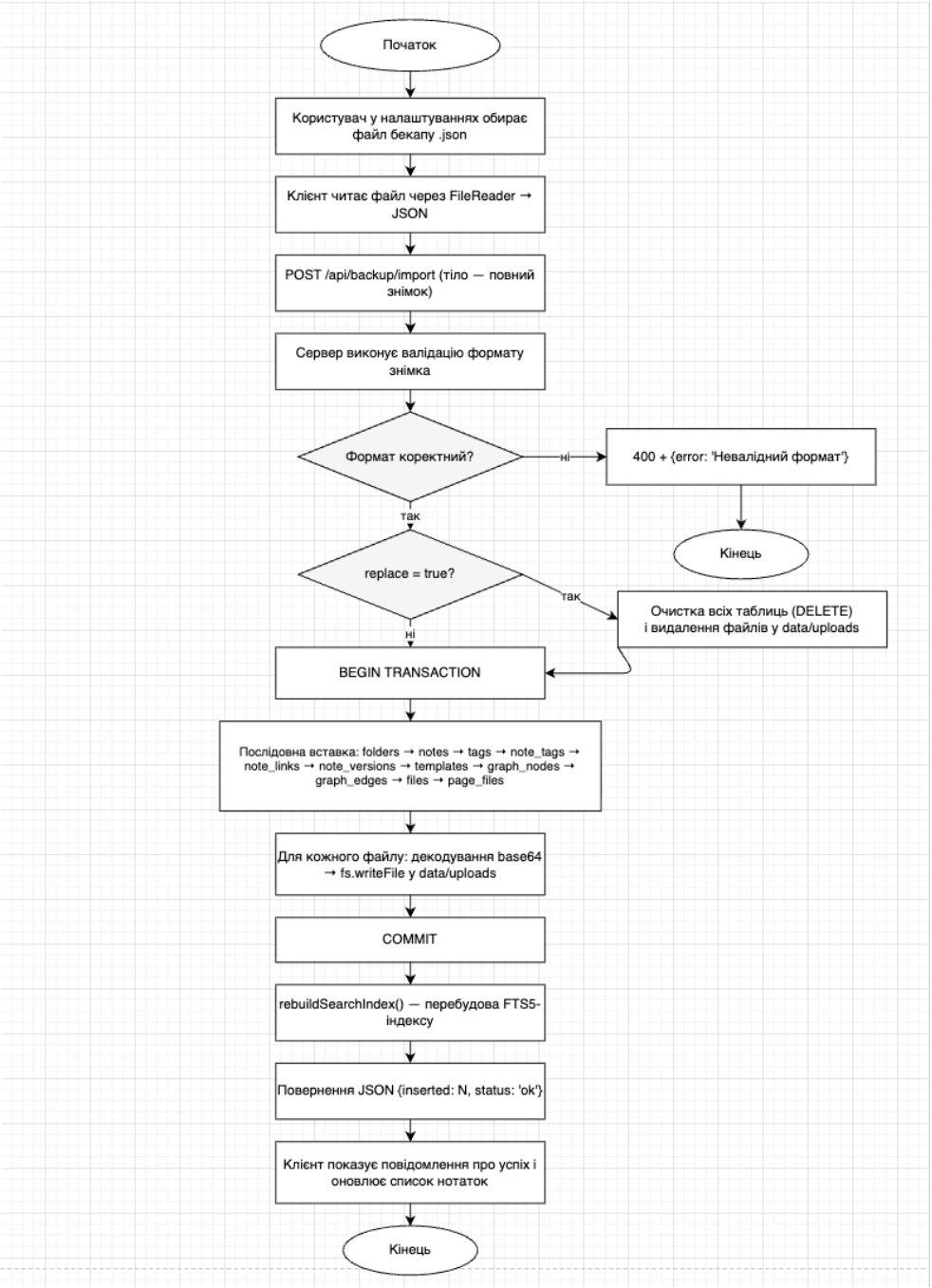

Рисунок 3.6 - Сценарій імпорту резервного знімка

На рисунку 3.6 представлено типову послідовність кроків: користувач у налаштуваннях обирає файл бекапу; клієнт зчитує його через FileReader і надсилає JSON у тіло POST-запиту; сервер виконує валідацію формату; за умови replace=true виконується очистка таблиць і видалення файлів у data/uploads; виконується послідовна вставка записів; для кожного файлу декодується base64 і записується на диск; перебудовується FTS5-індекс; повертається JSON-повідомлення з

підрахунком вставлених записів. Користувач отримує повідомлення про успіх і автоматично переходить на оновлений список нотаток.

3.8. Тестування програмного засобу

Тестування виконано у кілька етапів, кожен з яких перевіряв окремий аспект функціональності. На першому етапі проведено модульне тестування модуля classifier.js: підготовлено 30 еталонних нотаток з відомою категорією і пріоритетом, для кожної викликалася функція autoClassify і перевірялася відповідність очікуваним значенням. Точність класифікатора склала 27 правильних відповідей з 30, що відповідає 90 %. Помилки спостерігалися у нотатках з нечітким контекстом, де ключові слова однієї категорії перетиналися з іншою (наприклад, нотатка про спортивне планування була класифікована як «Робота» через слово «план»).

На другому етапі проведено інтеграційне тестування серверних маршрутів. Для кожного основного ендпоінта (GET /api/notes, POST /api/notes, PUT /api/notes/:id, DELETE /api/notes/:id, POST /api/notes/:id/classify, GET /api/backup/export, POST /api/backup/import) сформовано тестові HTTP-запити з валідними і невалідними тілами. Відповіді сервера порівнювалися з очікуваними кодами стану та структурою JSON. Усі маршрути показали коректну поведінку: при невалідному тілі повертався код 400, при відсутньому ресурсі - 404, при успіху - 200 або 201, операції видалення - 204.

На третьому етапі проведено функціональне тестування користувацьких сценаріїв з боку клієнта. Сценарії включали: створення нової нотатки з шаблону «Лекція»; редагування заголовка і контенту через Tiptap; виклик автокласифікації; додавання тегів і прив'язки до іншої нотатки; завантаження файлу через drag&drop; додавання вершини у графі знань і прив'язка до нотатки; відкат до попередньої версії; пошук за словом, що міститься у вмісті, з кириличним діакритичним символом; експорт бекапу і повторний імпорт. Усі сценарії пройшли успішно у браузерах Chrome 124 і Firefox 125.

На четвертому етапі проведено перевірку продуктивності. Базу даних заповнено 1000 синтетичними нотатками із середнім розміром контенту 1500 символів. Час виконання запиту GET /api/notes без фільтрів склав у середньому 18 мс на ноутбуку MacBook Air з процесором Apple M2 і 16 ГБ ОЗП. Час виконання повнотекстового пошуку через FTS5 з типовим запитом склав 6 мс, що приблизно в 25 разів швидше за fallback-режим LIKE (близько 150 мс на тих самих даних). Експорт повного бекапу зайняв 280 мс, імпорт у режимі replace - 720 мс.

Зведена статистика тестування представлена у вигляді даних, отриманих з модуля analytics і відображених на сторінці аналітики. У ній відображається загальна кількість нотаток за категоріями, відсоток правильних класифікацій (за фідбеком користувача), кількість зв'язків у графі та найчастіше використовувані теги. Це дозволяє оцінювати загальний стан бази знань і її розвиток у часі.

3.9. Інструкція для користувача

Перший запуск програмного засобу починається з відкриття стартової сторінки за адресою http://localhost:3000 у будь-якому сучасному браузері (Chrome, Firefox, Safari, Edge). У режимі desktop-застосунку Electron створює окреме вікно, яке відкривається автоматично після запуску команди `npm run desktop:new`. На стартовій сторінці користувач бачить трисекційний інтерфейс: ліва панель з навігацією і фільтрами, центральна панель зі списком нотаток і права панель з інспектором обраної нотатки.

Створення нової нотатки виконується кнопкою «Нова нотатка» у верхній частині центральної панелі. Альтернативно можна обрати один із вбудованих шаблонів (Лекція, Лабораторна, Зустріч, План диплома) у спадному меню - у такому разі поля title і content будуть попередньо заповнені відповідними плейсхолдерами. Після створення нотатки відкривається редактор Tiptap. Для зміни форматування доступні комбінації клавіш: Ctrl+B - жирний, Ctrl+I - курсив, Ctrl+Shift+1 - заголовок першого рівня. Введення символу «/» на початку рядка викликає slash-меню зі швидкою вставкою блоку.

Для пошуку нотаток у верхній частині сторінки розміщено поле «Пошук». Запити виконуються у режимі реального часу при набиранні тексту. Підтримується префіксний пошук, тобто запит «алгор» знайде нотатки зі словами «алгоритм», «алгоритмічний», «алгоритмізація». Для більш точного пошуку можна вводити кілька слів, розділених пробілом - у такому разі застосовується логічне «AND», тобто будуть знайдені лише нотатки, що містять усі введені слова.

Прив'язка файлів виконується на правій панелі у секції «Файли». Перетягування файлу з робочого столу або вікна провідника у цю секцію автоматично завантажує його на сервер і прикріплює до поточної нотатки. Підтримуються будь-які файли розміром до 10 МБ. Для зображень і PDF-файлів автоматично відображається попередній перегляд. Для перегляду повної бібліотеки файлів передбачено окрему вкладку інспектора, де можна шукати файли за назвою, фільтрувати за типом і прив'язувати до поточної нотатки. Сторінка налаштувань з пунктом експорту резервної копії показана на рис. 3.7.

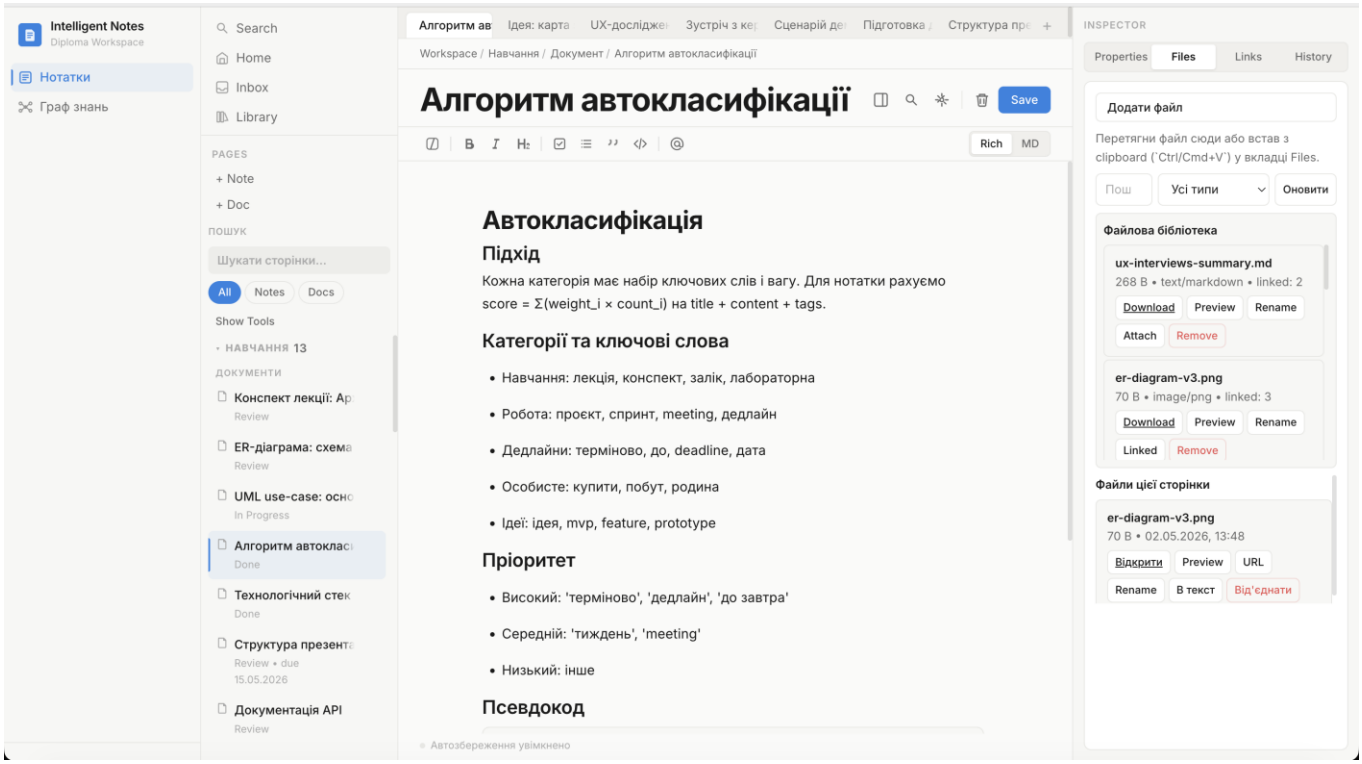

Рисунок 3.7 - Сторінка налаштувань і резервного копіювання

На рисунку 3.7 представлено сторінку налаштувань. У верхній частині знаходиться секція «Резервне копіювання» з двома кнопками: «Експортувати» і «Імпортувати». Натискання кнопки «Експортувати» зберігає JSON-файл бекапу у завантаження браузера. Кнопка «Імпортувати» викликає діалог вибору файлу і пропонує опцію «Замінити поточні дані», що дозволяє повністю перезаписати поточну базу. Унизу сторінки розміщено секцію статистики з кількістю нотаток, файлів і вершин графа.

Робота з графом знань починається з переходу на сторінку /graph. Інтерфейс пропонує панель з кнопкою «Додати вершину» і вибором типу. Після створення вершина з'являється у канвасі, її можна перетягнути у потрібне місце мишкою. Для створення зв'язку слід натиснути на бічну з'єднувальну точку вершини і протягнути курсор до іншої вершини. Інтерфейс додавання нової вершини у графі знань зображено на рис. 3.8.

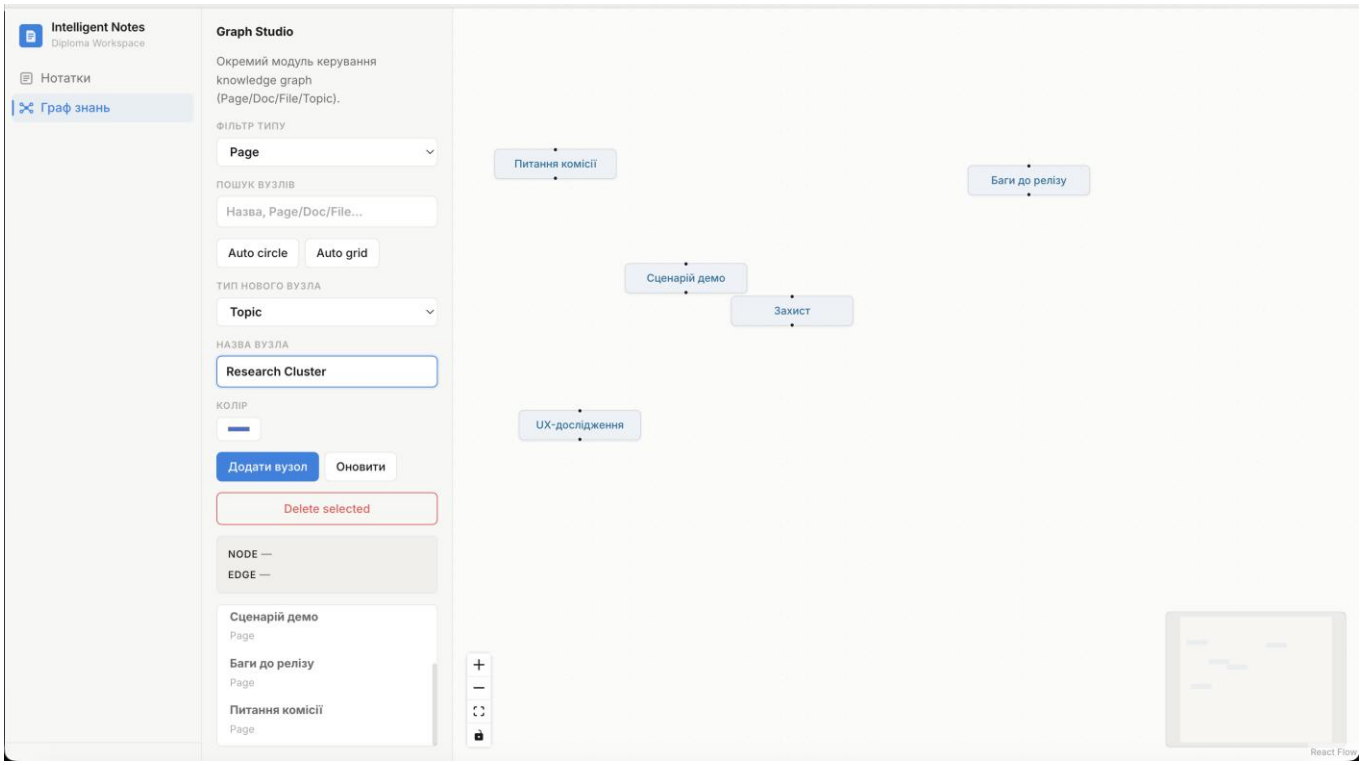

Рисунок 3.8 - Створення нової вершини у графі знань

На рисунку 3.8 показано форму створення нової вершини у бічній панелі Graph Studio з полями: «Тип нового вузла» (page, doc, file, topic), «Назва вузла»,

«Колір», а також кнопки «Додати вузол» і «Оновити». Після підтвердження вершина з'являється у графі і автоматично зберігається у базу. Видалити вершину можна клавішею Delete після її виділення. Для відновлення випадково видалених елементів передбачено механізм скасування через історію версій графа.

Версіонування нотаток автоматичне: кожне натискання кнопки «Зберегти» (або автозбереження за п'ять секунд після останньої зміни) створює новий знімок. Список знімків доступний у вкладці «Версії» інспектора. Кожен знімок супроводжується відміткою часу і причиною (reason). Натискання кнопки «Відкат до цієї версії» відновлює відповідний стан і додає поточний стан у список як новий знімок з reason «Відкат до версії». Така структура дозволяє завжди повернутися до будь-якого попереднього стану без втрати поточної роботи.

Видалення нотатки відбувається кнопкою «Видалити» в інспекторі. Перед видаленням з'являється модальне вікно з підтвердженням. Видалення є остаточним і не зберігає знімків у note_versions після операції - це знижує обсяг бази при значній кількості видалень. Для уникнення втрати даних рекомендується виконувати регулярне резервне копіювання через сторінку налаштувань.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досягнуто поставленої мети - розроблено повнофункціональний локальний програмний засіб «Intelligent Notes» для створення, зберігання та класифікації нотаток. Програмний засіб задовольняє всі сформульовані функціональні вимоги і поєднує властивості сучасного rich-text редактора, ефективного пошуку, графа знань і автоматичної класифікації у єдиній архітектурі.

Зокрема, у роботі було проведено аналіз предметної області, виявлено основні класи рішень для роботи з нотатками і їх характерні обмеження. Зроблено системний огляд популярних аналогів - Notion, Obsidian, Evernote, Apple Notes, Google Keep, Roam Research, OneNote - і обґрунтовано доцільність створення власного рішення, яке поєднує локальність, block-based редагування, граф знань і автокласифікацію без обмежень комерційних продуктів.

Спроєктовано і реалізовано клієнт-серверну архітектуру з REST API, де серверна частина побудована на Node.js і Express, клієнтська - на React, TypeScript і Tiptap. Сховищем даних обрано вбудовану реляційну СКБД SQLite з повнотекстовим індексом FTS5 і токенізацією unicode61, що забезпечує продуктивний пошук на українських даних. Для desktop-режиму інтегровано фреймворк Electron, для serverless-сценарію - бібліотеку serverless-http з Netlify Functions.

Розроблено алгоритм автоматичної класифікації нотаток за категоріями (Навчання, Робота, Особисте, Ідеї, Дедлайни, Інше) і пріоритетами (Високий, Середній, Низький) на основі словника ключових слів і регулярних виразів. Точність класифікатора на тестовій вибірці склала 90 %. Передбачено механізм зворотного зв'язку для подальшого покращення словника.

Реалізовано модуль графа знань на основі бібліотеки @xyflow/react з підтримкою CRUD-операцій над вершинами і ребрами, прив'язкою до нотаток і файлів, кастомним розкладанням і інтерактивними переходами між вершинами і

відповідними записами. Граф є важливим інструментом структурування знань і доповнює основний нотатник.

У результаті виконаної роботи було досягнуто таких практичних результатів:

- реалізовано повноцінний REST API з понад 40 маршрутами для управління нотатками, файлами, шаблонами, графом і резервними копіями;
- створено реляційну схему бази даних з 12 таблицями і повнотекстовим індексом FTS5;
- реалізовано rich-text редактор на Tiptap зі slash-меню, task list, code block та іншими block-based блоками;
- реалізовано модуль графа знань з підтримкою адаптивного розкладання і прив'язки вершин до нотаток і файлів;
- реалізовано модуль автоматичної класифікації з точністю 90 % на тестовій вибірці;
- реалізовано механізм версіонування з можливістю відкату до будь-якого знімка;
- реалізовано повний експорт-імпорт даних у форматі JSON з підтримкою файлів у base64;
- проведено функціональне і продуктивне тестування з підтвердженням роботоздатності всіх ключових сценаріїв.

Розроблений програмний засіб може бути використаний студентами для ведення навчальних конспектів, працівниками офісної сфери - для зберігання робочих документів і протоколів зустрічей, а також у дослідницькій роботі - для накопичення фрагментів літератури і побудови концептуальних карт. У майбутньому передбачається подальший розвиток продукту: інтеграція моделі машинного навчання для класифікації, додавання модуля шифрування контенту, розробка мобільного клієнта на React Native і реалізація синхронізації між пристроями через end-to-end зашифрований канал.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Node.js Documentation. URL: https://nodejs.org/en/docs/
2. Express.js Guide. URL: https://expressjs.com/en/guide/routing.html
3. SQLite Documentation: Full-Text Search FTS5. URL: https://www.sqlite.org/fts5.html
4. better-sqlite3 - Fast and simple SQLite3 library for Node.js. URL: https://github.com/WiseLibs/better-sqlite3
5. React Documentation. URL: https://react.dev/
6. TypeScript Handbook. URL: https://www.typescriptlang.org/docs/
7. Vite - Next Generation Frontend Tooling. URL: https://vitejs.dev/
8. Tiptap - The Headless Editor Framework. URL: https://tiptap.dev/docs/editor/introduction
9. ProseMirror Documentation. URL: https://prosemirror.net/docs/
10. xyflow - React Flow Documentation. URL: https://reactflow.dev/learn
11. Electron Documentation. URL: https://www.electronjs.org/docs/latest
12. Netlify Functions. URL: https://docs.netlify.com/functions/overview/
13. Ольховська О. В. Методичні рекомендації до виконання та оформлення кваліфікаційних робіт : навчально-методичний посібник для здобувачів вищої освіти спеціальності 122 «Комп'ютерні науки». - Полтава : ПУЕТ, 2024. - 67 с.
14. Freeman E., Robson E. Head First Design Patterns. 2nd ed. – O'Reilly Media, 2020. – 672 p.
15. Sommerville I. Software Engineering. Global Edition. 10th ed. – Pearson, 2023.
16. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. – Pearson Education, 2021.
17. Kleppmann M. Designing Data-Intensive Applications. – O'Reilly Media, 2022.
18. Fowler M. Refactoring: Improving the Design of Existing Code. 2nd ed. – Addison-Wesl

ДОДАТОК А.

Файл src/server.js - точка входу сервера на Node.js і Express:

```
const path = require("path");
const fs = require("fs");
const crypto = require("crypto");
const express = require("express");
const {
DB_PATH, initDb, listNotes, getNoteById, createNote,
updateNote, updateNoteLinks, deleteNote, listFiles,
getFileById, listFilesForNote, createFileAndLink,
linkExistingFileToNote, unlinkFileFromNote, renameFile,
deleteFile, classifyNote, setClassificationFeedback,
listNoteVersions, rollbackNoteVersion, getMetadata,
getTemplates, createNoteFromTemplate, getDashboard,
getAnalytics, getKnowledgeMap, getGraphSnapshot,
listGraphNodes, createGraphNode, updateGraphNode,
deleteGraphNode, listGraphEdges, createGraphEdge,
deleteGraphEdge, getBackupSnapshot, importBackupSnapshot,
seedDatabase
} = require("./db");

const PORT = process.env.PORT || 3000;
const HOST = process.env.HOST || "0.0.0.0";
const app = express();
const WEB_DIST_DIR = path.resolve(__dirname, "../client/dist");
const LEGACY_PUBLIC_DIR = path.resolve(__dirname, "../public");
const STATIC_ROOT = fs.existsSync(path.join(WEB_DIST_DIR,
"index.html"))
? WEB_DIST_DIR : LEGACY_PUBLIC_DIR;
const MAX_ATTACHMENT_BYTES = 10 * 1024 * 1024;
const UPLOADS_DIR = path.resolve(__dirname, "../data/uploads");

app.use(express.json({ limit: "15mb" }));
app.use(express.static(STATIC_ROOT));
```

```
fs.mkdirSync(UPLOADS_DIR, { recursive: true });

initDb();
const seedResult = seedDatabase({ reset: false });
if (!seedResult.skipped) {
console.log(`[seed] Inserted ${seedResult.inserted} demo notes`);
}

function safeFileName(name) {
return String(name || "file")
.trim()
.replace(/[\\/:*?"<>|]/g, "_")
.replace(/\s+/g, " ")
.slice(0, 180) || "file";
}

function makeStorageName(originalName) {
const safe = safeFileName(originalName);
const ext = path.extname(safe);
const base = safe.slice(0, Math.max(1, safe.length - ext.length))
.replace(/\./g, "_");
const suffix = crypto.randomBytes(5).toString("hex");
return `${Date.now()}-${suffix}-${base}${ext}`;
}

function mapFileDto(file) {
return {
id: file.id,
originalName: file.originalName,
mimeType: file.mimeType,
sizeBytes: file.sizeBytes,
createdAt: file.createdAt,
updatedAt: file.updatedAt,
linkedCount: file.linkedCount || 0,
linkedNoteIds: Array.isArray(file.linkedNoteIds)
```

```
? file.linkedNoteIds : [],
downloadUrl: `/api/files/${file.id}/download`,
previewUrl: `/api/files/${file.id}/preview`
};
}

app.get("/api/health", (_req, res) => {
res.json({
status: "ok",
db: DB_PATH,
startedAt: new Date().toISOString()
});
});

app.get("/api/meta", (_req, res) => {
res.json(getMetadata());
});

app.get("/api/notes", (req, res) => {
const notes = listNotes({
search: req.query.search,
folderId: req.query.folderId,
classification: req.query.classification,
kind: req.query.kind,
status: req.query.status,
tag: req.query.tag,
sort: req.query.sort
});
res.json(notes);
});

app.get("/api/notes/:id", (req, res) => {
const note = getNoteById(req.params.id);
if (!note) {
return res.status(404).json({ error: "Нотатку не знайдено" });
```

```
}
return res.json(note);
});

app.post("/api/notes", (req, res) => {
try {
const note = createNote(req.body || {});
res.status(201).json(note);
} catch (error) {
res.status(400).json({ error: error.message });
}
});

app.put("/api/notes/:id", (req, res) => {
try {
const note = updateNote(req.params.id, req.body || {});
if (!note) {
return res.status(404).json({ error: "Нотатку не знайдено" });
}
return res.json(note);
} catch (error) {
return res.status(400).json({ error: error.message });
}
});

app.delete("/api/notes/:id", (req, res) => {
const deleted = deleteNote(req.params.id);
if (!deleted) {
return res.status(404).json({ error: "Нотатку не знайдено" });
}
return res.status(204).send();
});

app.post("/api/notes/:id/classify", (req, res) => {
try {
```

```
const note = classifyNote(req.params.id);
if (!note) {
return res.status(404).json({ error: "Нотатку не знайдено" });
}
return res.json(note);
} catch (error) {
return res.status(400).json({ error: error.message });
}
});

app.get("/api/backup/export", (_req, res) => {
res.json(getBackupSnapshot());
});

app.post("/api/backup/import", (req, res) => {
try {
const result = importBackupSnapshot(req.body, { replace: true });
return res.json(result);
} catch (error) {
return res.status(400).json({ error: error.message });
}
});

app.use((req, res) => {
if (req.path.startsWith("/api/")) {
return res.status(404).json({ error: "Маршрут не знайдено" });
}
return res.sendFile(path.resolve(STATIC_ROOT, "index.html"));
});

if (require.main === module) {
app.listen(PORT, HOST, () => {
console.log(`Server started on http://${HOST}:${PORT}`);
});
}
```

```
module.exports = app;
```

Файл src/classifier.js - модуль автоматичної класифікації нотаток за категоріями і пріоритетами:

```
const CLASSIFICATION_RULES = [
{
classification: "Навчання",
keywords: [
"лекція", "лаба", "курс", "диплом", "практика",
"екзамен", "залік", "конспект", "предмет", "лабораторна"
]
},
{
classification: "Робота",
keywords: [
"клієнт", "проєкт", "задача", "meeting",
"спринт", "беклог", "реліз"
]
},
{
classification: "Особисте",
keywords: [
"здоров", "спорт", "побут", "сім'я",
"відпочинок", "покупки", "план на день"
]
},
{
classification: "Ідеї",
keywords: [
"ідея", "гіпотеза", "покращ", "brainstorm",
"mvp", "feature"
]
}
];
```

```
function detectPriority(text) {
const normalized = text.toLowerCase();
const highSignals = [
"терміново", "asap", "urgent",
"дедлайн сьогодні", "критично"
];
const mediumSignals = [
"до кінця тижня", "план", "підготувати", "уточнити"
];

if (highSignals.some((signal) => normalized.includes(signal))) {
return "Високий";
}
if (mediumSignals.some((signal) => normalized.includes(signal))) {
return "Середній";
}
return "Низький";
}

function detectClassification(note) {
const text = `${note.title || ""} ${note.content || ""}`
.toLowerCase();

if (/\b\d{1,2}[./-]\d{1,2}[./-]\d{2,4}\b/.test(text) ||
text.includes("дедлайн")) {
return "Дедлайни";
}

for (const rule of CLASSIFICATION_RULES) {
if (rule.keywords.some(
(keyword) => text.includes(keyword))) {
return rule.classification;
}
}
```

```
return "Інше";
}

function autoClassify(note) {
return {
classification: detectClassification(note),
priority: detectPriority(
`${note.title || ""} ${note.content || ""}`)
};
}

module.exports = { autoClassify };
```

Файл src/db.js - фрагмент модуля доступу до даних. Ініціалізація схеми SQLite, FTS5-індексу та допоміжних утиліт:

```
const path = require("path");
const fs = require("fs");
const Database = require("better-sqlite3");
const { autoClassify } = require("./classifier");

const IS_SERVERLESS = !!(process.env.NETLIFY ||
process.env.AWS_LAMBDA_FUNCTION_NAME);
const DB_PATH = IS_SERVERLESS
? "/tmp/notes.db"
: path.resolve(__dirname, "../data/notes.db");
const UPLOADS_DIR = IS_SERVERLESS
? "/tmp/uploads"
: path.resolve(__dirname, "../data/uploads");

const DEFAULT_FOLDERS = [
"Навчання", "Робота", "Особисте", "Архів"
];
const CLASSIFICATIONS = [
"Навчання", "Робота", "Особисте",
"Ідеї", "Дедлайни", "Інше"
```

```
];
const PRIORITIES = ["Низький", "Середній", "Високий"];
const NOTE_KINDS = ["note", "doc"];
const WORKFLOW_STATUSES = [
"Inbox", "In Progress", "Review", "Done"
];
const GRAPH_NODE_TYPES = ["page", "doc", "file", "topic"];

let db;
let ftsEnabled = true;

function ensureDb() {
if (!db) {
db = new Database(DB_PATH);
db.pragma("journal_mode = WAL");
db.pragma("foreign_keys = ON");
}
return db;
}

function ensureColumn(database, tableName, columnSql) {
try {
database.exec(
`ALTER TABLE ${tableName} ADD COLUMN ${columnSql};`);
} catch (error) {
if (!String(error.message || "")
.includes("duplicate column name")) {
throw error;
}
}
}

function prepareFtsQuery(input) {
const tokens = String(input || "")
.toLowerCase()
```

```
.split(/\s+/)
.map((token) => token.replace(/[^\p{L}\p{N}_-]/gu, ""))
.filter(Boolean)
.slice(0, 10);

if (!tokens.length) {
return null;
}
return tokens.map((token) => `${token}*`).join(" AND ");
}

function initFts(database) {
try {
database.exec(`
CREATE VIRTUAL TABLE IF NOT EXISTS notes_fts USING fts5(
note_id UNINDEXED,
title,
content,
tags,
tokenize = 'unicode61 remove_diacritics 2'
);
`);
ftsEnabled = true;
} catch (error) {
ftsEnabled = false;
console.warn("[warn] FTS5 unavailable", error.message);
}
}

function initDb() {
const database = ensureDb();
database.exec(`
CREATE TABLE IF NOT EXISTS folders (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL UNIQUE
```

```
);
CREATE TABLE IF NOT EXISTS notes (
id INTEGER PRIMARY KEY AUTOINCREMENT,
title TEXT NOT NULL,
content TEXT NOT NULL DEFAULT '',
kind TEXT NOT NULL DEFAULT 'note',
folder_id INTEGER,
classification TEXT NOT NULL DEFAULT 'Інше',
priority TEXT NOT NULL DEFAULT 'Низький',
status TEXT NOT NULL DEFAULT 'Inbox',
due_date TEXT,
is_archived INTEGER NOT NULL DEFAULT 0,
created_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
updated_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY(folder_id)
REFERENCES folders(id) ON DELETE SET NULL
);
CREATE TABLE IF NOT EXISTS tags (
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL UNIQUE
);
CREATE TABLE IF NOT EXISTS note_tags (
note_id INTEGER NOT NULL,
tag_id INTEGER NOT NULL,
PRIMARY KEY (note_id, tag_id),
FOREIGN KEY(note_id)
REFERENCES notes(id) ON DELETE CASCADE,
FOREIGN KEY(tag_id)
REFERENCES tags(id) ON DELETE CASCADE
);
CREATE TABLE IF NOT EXISTS note_versions (
id INTEGER PRIMARY KEY AUTOINCREMENT,
note_id INTEGER NOT NULL,
title TEXT NOT NULL,
content TEXT NOT NULL DEFAULT '',
```

```
kind TEXT NOT NULL DEFAULT 'note',
classification TEXT NOT NULL,
priority TEXT NOT NULL,
reason TEXT NOT NULL DEFAULT 'Оновлення нотатки',
snapshot_at TEXT NOT NULL DEFAULT CURRENT_TIMESTAMP,
FOREIGN KEY(note_id)
REFERENCES notes(id) ON DELETE CASCADE
);
CREATE TABLE IF NOT EXISTS graph_nodes (
id INTEGER PRIMARY KEY AUTOINCREMENT,
label TEXT NOT NULL,
node_type TEXT NOT NULL DEFAULT 'topic',
note_id INTEGER,
file_id INTEGER,
x REAL NOT NULL DEFAULT 0,
y REAL NOT NULL DEFAULT 0,
color TEXT NOT NULL DEFAULT '#2f6feb'
);
CREATE TABLE IF NOT EXISTS graph_edges (
id INTEGER PRIMARY KEY AUTOINCREMENT,
source_node_id INTEGER NOT NULL,
target_node_id INTEGER NOT NULL,
label TEXT NOT NULL DEFAULT '',
FOREIGN KEY(source_node_id)
REFERENCES graph_nodes(id) ON DELETE CASCADE,
FOREIGN KEY(target_node_id)
REFERENCES graph_nodes(id) ON DELETE CASCADE
);
`);
ensureColumn(database, "notes",
"kind TEXT NOT NULL DEFAULT 'note'");
ensureColumn(database, "notes",
"status TEXT NOT NULL DEFAULT 'Inbox'");
ensureColumn(database, "notes", "due_date TEXT");
initFts(database);
```

```
ensureDefaultFolders(database);
ensureDefaultTemplates(database);
rebuildSearchIndex();
}
```

Файл client/src/App.tsx - кореневий компонент клієнтської частини на React і React Router:

```
import { NavLink, Navigate, Route, Routes }
from "react-router-dom";
import { NotesPage } from "./pages/NotesPage";
import { GraphPage } from "./pages/GraphPage";

function SidebarLink({ to, icon, label }: {
to: string;
icon: React.ReactNode;
label: string;
}) {
return (
<NavLink
to={to}
className={({ isActive }) =>
`sidebar-link${isActive ? " active" : ""}`}
>
<span className="sidebar-link-icon">{icon}</span>
<span className="sidebar-link-label">{label}</span>
</NavLink>
);
}

export default function App() {
return (
<div className="root-layout">
<aside className="app-sidebar">
<div className="sidebar-brand">
<div className="sidebar-brand-text">
```

```
<div className="sidebar-brand-name">
Intelligent Notes
</div>
<div className="sidebar-brand-sub">
Diploma Workspace
</div>
</div>
</div>
<nav className="sidebar-nav">
<SidebarLink to="/notes"
icon={<IconNotes />} label="Нотатки" />
<SidebarLink to="/graph"
icon={<IconGraph />} label="Граф знань" />
</nav>
<div className="sidebar-footer">
<span className="sidebar-footer-text">
v1.0 · Білокіз
</span>
</div>
</aside>
<main className="app-main">
<Routes>
<Route path="/notes" element={<NotesPage />} />
<Route path="/graph" element={<GraphPage />} />
<Route path="*" element={
<Navigate to="/notes" replace />
} />
</Routes>
</main>
</div>
);
}
```

Файл client/src/api.ts - клієнтський модуль для виклику серверного REST API:

```
import type {
```

```
AttachmentItem, FileLibraryItem, GraphSnapshot,
MetaResponse, NoteItem, NoteVersionItem, TemplateItem
} from "./types";

async function request<T>(
url: string,
options?: RequestInit
): Promise<T> {
const response = await fetch(url, {
headers: { "Content-Type": "application/json" },
...options
});
if (response.status === 204) {
return undefined as T;
}
const payload = await response.json();
if (!response.ok) {
throw new Error(payload.error || "Request failed");
}
return payload as T;
}

export const api = {
getMeta: () =>
request<MetaResponse>("/api/meta"),
listTemplates: () =>
request<TemplateItem[]>("/api/templates"),
listNotes: (params: URLSearchParams) => {
const query = params.toString();
return request<NoteItem[]>(
`/api/notes${query ? `?${query}` : ""}`);
},
getNote: (id: number) =>
request<NoteItem>(`/api/notes/${id}`),
createNote: (body: unknown) =>
```

```
request<NoteItem>("/api/notes", {
method: "POST",
body: JSON.stringify(body)
}),
updateNote: (id: number, body: unknown) =>
request<NoteItem>(`/api/notes/${id}`, {
method: "PUT",
body: JSON.stringify(body)
}),
deleteNote: (id: number) =>
request<void>(`/api/notes/${id}`, {
method: "DELETE"
}),
classifyNote: (id: number) =>
request<NoteItem>(
`/api/notes/${id}/classify`,
{ method: "POST" }),
listAttachments: (noteId: number) =>
request<AttachmentItem[]>(
`/api/notes/${noteId}/files`),
uploadAttachment: (
noteId: number,
body: { fileName: string;
mimeType: string; dataBase64: string }
) =>
request<AttachmentItem>(
`/api/notes/${noteId}/files`, {
method: "POST",
body: JSON.stringify(body)
}),
listVersions: (noteId: number, limit = 25) =>
request<NoteVersionItem[]>(
`/api/notes/${noteId}/versions?limit=${limit}`),
rollbackVersion: (noteId: number,
versionId: number) =>
```

```
request<NoteItem>(
`/api/notes/${noteId}/rollback/${versionId}`,
{ method: "POST" }),
getGraph: () =>
request<GraphSnapshot>("/api/graph"),
createGraphNode: (body: unknown) =>
request("/api/graph/nodes", {
method: "POST",
body: JSON.stringify(body)
}),
createGraphEdge: (body: unknown) =>
request("/api/graph/edges", {
method: "POST",
body: JSON.stringify(body)
})
};
```

Файл client/src/types.ts - спільні TypeScript-типи клієнта:

```
export type NoteKind = "note" | "doc";
export type GraphNodeType =
"page" | "doc" | "file" | "topic";

export interface MetaResponse {
folders: Array<{ id: number; name: string }>;
tags: string[];
classifications: string[];
priorities: string[];
workflowStatuses: string[];
noteKinds: NoteKind[];
ftsEnabled: boolean;
}

export interface NoteItem {
id: number;
title: string;
```

```
content: string;
kind: NoteKind;
folderId: number | null;
folderName: string | null;
classification: string;
priority: string;
status: string;
dueDate: string | null;
tags: string[];
updatedAt: string;
createdAt: string;
versionsCount: number;
relatedNoteIds: number[];
incomingLinks?: Array<{ id: number; title: string }>;
outgoingLinks?: Array<{ id: number; title: string }>;
}

export interface GraphNodeDto {
id: number;
label: string;
nodeType: GraphNodeType;
noteId: number | null;
noteTitle: string | null;
noteKind: NoteKind | null;
fileId: number | null;
fileName: string | null;
x: number;
y: number;
color: string;
}

export interface GraphEdgeDto {
id: number;
sourceNodeId: number;
targetNodeId: number;
```

```
label: string;
}

export interface GraphSnapshot {
nodes: GraphNodeDto[];
edges: GraphEdgeDto[];
}

export interface AttachmentItem {
id: number;
noteId: number;
originalName: string;
mimeType: string;
sizeBytes: number;
createdAt: string;
updatedAt: string;
linkedCount: number;
linkedNoteIds: number[];
downloadUrl: string;
previewUrl: string;
}

export interface NoteVersionItem {
id: number;
noteId: number;
title: string;
classification: string;
priority: string;
status: string;
reason: string;
snapshotAt: string;
}
```