

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп'ютерних наук, фізики та математики
Кафедра комп'ютерних наук та програмної інженерії**

**Розробка вебзастосунку для спільної роботи над завданнями та
нотатками у режимі реального часу**

**Кваліфікаційна робота (проект)
на здобуття ступеня вищої освіти “бакалавр”**

Виконав: здобувач 4 курсу 12-461 групи

Спеціальності 126 Інформаційні системи та
технології

Освітньо-професійної програми

Інформаційні системи та технології

Кузнєцов Євген Миколайович

Керівник:

Черненко Ірина Євгенівна

Рецензент:

Middle Frontend Developer Linkup ST

Борейко О. І.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	5
ВСТУП.....	6
РОЗДІЛ 1 ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	9
1.1. Дослідження сучасних підходів до менеджменту персональної та командної інформації	9
1.2. Аналіз існуючих інформаційних систем та їх обмежень	11
1.2.1. Google Docs	11
1.2.2. Notion.....	12
1.2.3. Obsidian.....	13
1.2.4. Trello	14
1.3. Порівняння актуальних технологій розробки	15
1.3.1. Реляційні та нереляційні СУБД	15
1.3.2. Аналіз середовищ виконання та протоколів обміну даними в реальному часі	17
1.3.3. Огляд сучасних клієнтських фреймворків для веб-інтерфейсів	19
1.3.4. Побудова кросплатформних десктопних застосунків із веб-технологій	20
РОЗДІЛ 2 МЕТОДОЛОГІЯ ТА ПРОЄКТУВАННЯ ЗАСТОСУНКУ “VAULT”	23
2.1. Архітектура та системні рішення	23
2.1.1. Багатошарова архітектура серверної частини	23
2.1.2. Обмін даними у реальному часі через WebSockets.....	24

2.1.3. Реактивна модель клієнтської частини та управління локальним станом.....	24
2.1.4. Локалізація інтерфейсу.....	26
2.2. Моделювання інформаційних потоків	27
2.3. Синхронізація змін під час спільного редагування нотатки	28
2.4. Концепція self-hosting та відкритість вихідного коду як засоби контролю даних	29
РОЗДІЛ 3 ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБ-ДОДАТКУ “VAULT”	30
3.1. Структура та схеми бази даних.....	30
3.1.1. Модель користувача	30
3.1.2. Модель проєкту.....	31
3.1.3. Моделі даних модуля нотаток.....	32
3.1.4. Моделі даних модуля Kanban-дошок.....	34
3.2. Архітектура серверної частини застосунку	36
3.2.1. Авторизація та права доступу	38
3.2.2. Socket.IO.....	43
3.2.3. Логіка модуля нотаток.....	44
3.2.4. Логіка модуля Kanban-дошок.....	48
3.3. Візуальна частина інтерфейсу та взаємодія з користувачем.....	48
3.3.1. Компоненти інтерфейсу	49
3.3.2. UX-рішення для навігації та спільної роботи	51
3.4. Архітектура клієнтної частини застосунку	52
3.4.1. Структура SPA на основі SvelteKit	52
3.4.2. Управління локальним станом.....	53

3.4.3. Взаємодія з API та Socket.IO Client.....	54
3.4.4. Форми та валідація даних.....	55
3.4.5. Редактор нотаток на основі Markdown	56
3.4.6. Інтерактивні Kanban-дошки	57
3.5. Організація середовища розгортання та процеси збірки застосунку	59
3.6. Перспективи вдосконалення та напрямки розвитку застосунку.....	60
ВИСНОВКИ	61
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	63
ДОДАТКИ	66
Додаток А. Повний цикл створення нотатки.....	66
Додаток Б. Файлова структура серверної частини додатку.....	67

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

UI/UX	User Interface/User Experience
SPA	Single Page Application
API	Application Programming Interface
DOM	Document Object Model
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
CRDT	Conflict-free Replicated Data Type
ODM	Object-Document Mapping
DTO	Data Transfer Object
XML	eXtensible Markup Language
JSON	JavaScript Object Notation
CSS	Cascading Style Sheets
OS	Operating System
DnD	Drag-and-Drop
CI/CD	Continuous Integration/Continuous Delivery

ВСТУП

У сучасному інформаційному суспільстві ефективне управління персональними та командними знаннями стає критичним фактором продуктивності. Постійне зростання обсягів вхідної інформації вимагає від користувача використання інструментів для її структурування, проте існуючі рішення часто мають суттєві обмеження.

Основною перешкодою є функціональна розрізненість інструментів: швидкі нотатки, складні бази знань та системи управління задачами зазвичай існують як окремі застосунки. Це створює бар'єри при трансформації ідей у конкретні плани дій. Популярні платформи, такі як Notion, Obsidian або Trello, часто є або занадто складними для швидкої фіксації думок, або обмеженими лише одним форматом представлення даних (наприклад, тільки Kanban-дошки). До того ж більшість цих сервісів є хмарними рішеннями із закритим кодом, що обмежує контроль користувача над безпекою власних даних.

Додатковим викликом є складність інструментів для спільної роботи. У корпоративних системах на кшталт Jira процес координації часто перевантажений складними налаштуваннями прав доступу та ієрархіями, що знижує оперативність взаємодії в невеликих командах.

Для розв'язання цих проблем є актуальним проектування застосунку “Vault” – середовища, що працює в режимі реального часу та поєднує модулі ведення нотаток та Kanban-дошок. Застосунок “Vault” має бути орієнтований на баланс між швидкістю роботи, приватністю даних (через концепт self-hosting) та ефективною командною співпрацею без накладних адміністративних витрат.

Об’єкт дослідження – процеси менеджменту персональних та командних знань, організації завдань та спільної роботи над цифровим контентом.

Предмет дослідження – методи та технології побудови архітектури вебзастосунків з низькою затримкою, механізми синхронізації даних без конфліктів та принципи інтеграції модулів ведення нотаток і Kanban-дошок у єдиний інформаційний простір.

Мета роботи – проєктування та розробка вебзастосунку як єдиного середовища для управління знаннями, що забезпечить інтеграцію модулів нотаток та Kanban-дошок з пріоритетом на приватність даних та self-hosting.

Завдання дослідження:

- Проаналізувати сучасні підходи до менеджменту персональної та командної інформації та виявити обмеження існуючих рішень.
- Визначити та обґрунтувати вибір технологічного стека для забезпечення високого рівня продуктивності.
- Спроектувати архітектуру застосунку, врахувавши систему авторизації та прав доступу. Реалізувати серверну частину застосунку.
- Розробити візуальну концепцію та UX-рішення в середовищі Figma.
- Реалізувати клієнтську частину застосунку з акцентом на мінімізацію накладних витрат на рендеринг.
- Розробити модуль спільного редагування Markdown нотаток.
- Розробити модуль інтерактивних Kanban-дошок.
- Організувати процес збірки вебзастосунку у нативний додаток та розгортання серверної частини відповідно до концепції self-hosting.
- Описати перспективи розвитку системи та її подальшого розширення.

Структура роботи: робота складається з переліку умовних позначень, вступу, трьох розділів, висновків, списку використаних джерел та додатків.

РОЗДІЛ 1

ДОСЛІДЖЕННЯ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Дослідження сучасних підходів до менеджменту персональної та командної інформації

Ефективність управління інформацією в сучасних інформаційних системах базується на виборі правильної методології обробки та передачі даних. З існуючих підходів до менеджменту інформації можна виділити декілька ключових методологій, які визначають архітектуру сучасного програмного забезпечення:

- **Knowledge Codification (кодифікація знань)** - процес перетворення неявних ідей та досвіду у структуровані інформаційні об'єкти (текстові нотатки, документи, схеми), які можна зберігати, редагувати та передавати іншим учасникам команди.

- **Knowledge Mapping (картографування знань та процесів)** - візуалізація потоків та зв'язків між об'єктами. Сучасним втіленням цього підходу є Kanban-дошки, де кожна картка є не просто завданням, а носієм контекстної інформації. Це дозволяє команді бачити цілісну картину проекту та стан кожного окремого етапу в реальному часі.

- **Taxonomy Design (таксономічний дизайн)** - побудова гнучких систем класифікації за допомогою тегів, категорій та метаданих. Це вирішує проблему інформаційного хаосу, дозволяючи користувачу швидко фільтрувати інформацію.

- **After Action Review (аналіз та зворотній зв'язок)** - методологія, що полягає в аналізі завершених етапів роботи. У застосунках менеджмент типу це реалізується через систему коментарів, історію змін та архіви виконаних завдань, що створює базу “вивчених уроків” для команди.

- **Sensemaking (спільне осмислення)** - підхід, що фокусується на колективній роботі над інформацією. Він передбачає створення середовища для взаємодії між користувачами у реальному часі, що дозволяє одночасно редагувати інформацію, формуючи спільне розуміння цілей проєкту.

Зазначені методології командної взаємодії наряду залежать від того, наскільки ефективно кожен учасник працює з інформацією. Командний менеджмент знань є логічною ітерацією персонального менеджменту, оскільки будь-яка спільна діяльність починається з індивідуальної ідеї або результатів персонального дослідження. Сучасні системи персонального менеджменту знань базуються на циклічному підході до обробки даних, найбільш відомим з яких є фреймворк CODE, започаткований Тіаго Форте [1]:

- **Capture (захоплення)** - швидка фіксація ідей у єдиному вхідному просторі. Це дозволяє розвантажити пам'ять користувача.
- **Organize (організація)** - структурування зібраного матеріалу за темами та/або контекстом використання.
- **Distill (дистиляція)** - виділення головної суті з великих обсягів інформації. У нотатках це реалізується через форматування.
- **Express (вираження)** - перетворення накопичених знань у конкретний результат: виконання завдання, написання коду або створення нового проєкту.

Описаний цикл відображає послідовність дій, які користувач проходить під час роботи з інформацією на практиці. Окрім цього, у межах персонального менеджменту знань також виділяють два архітектурні підходи:

- **Networked Thought (мережеве мислення)** - концепція побудови зв'язків між інформаційними блоками за допомогою двосторонніх посилок, що імітує асоціативну роботу людського мозку. Замість жорсткої

ієрархії папок, інформація організовується у вигляді графа знань, де кожна нотатка може бути точкою входу до декількох контекстів одночасно.

– **Outlining (окреслення)** - підхід до структурування інформації через створення багаторівневих списків або ієрархії заголовків. Це критично важливий інструмент, що дозволяє розбити масштабну ідею на послідовність атомарних кроків.

Аналіз сучасних підходів до менеджменту інформації показує, що більшість інструментів для персональної продуктивності поступово інтегруються з командними сервісами. Найбільшу ефективність демонструють системи, що здатні інтегрувати процеси накопичення знань (через нотатки та мережеві зв'язки) із процесами їх реалізації (через ієрархію та Kanban-дошки). Зазначений підхід дозволяє мінімізувати втрати контексту при переході від індивідуального планування до групового виконання, що обґрунтовує доцільність проєктування комплексних веб-застосунків для менеджменту інформації з підтримкою спільної роботи в режимі реального часу.

1.2. Аналіз існуючих інформаційних систем та їх обмежень

Сучасні інформаційні системи виникли як відповідь на зростаючі потреби користувачів та команд у зручному зберіганні, обміні та організації знань. Раніше інформація часто розпорошувалася між паперовими документами, електронною поштою та локальними файлами, що ускладнювало спільну роботу та контроль за змінами. Сьогодні для вирішення цих завдань існує кілька популярних інструментів, серед яких Google Docs, Notion, Obsidian та Trello.

1.2.1. Google Docs

Система Google Docs (рис. 1.1) виділяється тим, що дозволяє спільно редагувати текстові документи в реальному часі та залишати коментарі [2]. Водночас її функціональні обмеження проявляються у неможливості

організовувати завдання у Kanban-стилі або будувати комплексну систему управління проектами. Крім того, оскільки дані зберігаються на серверах Google, користувачі мають обмежений контроль над інформацією, що обмежує використання Docs як універсального інструменту.

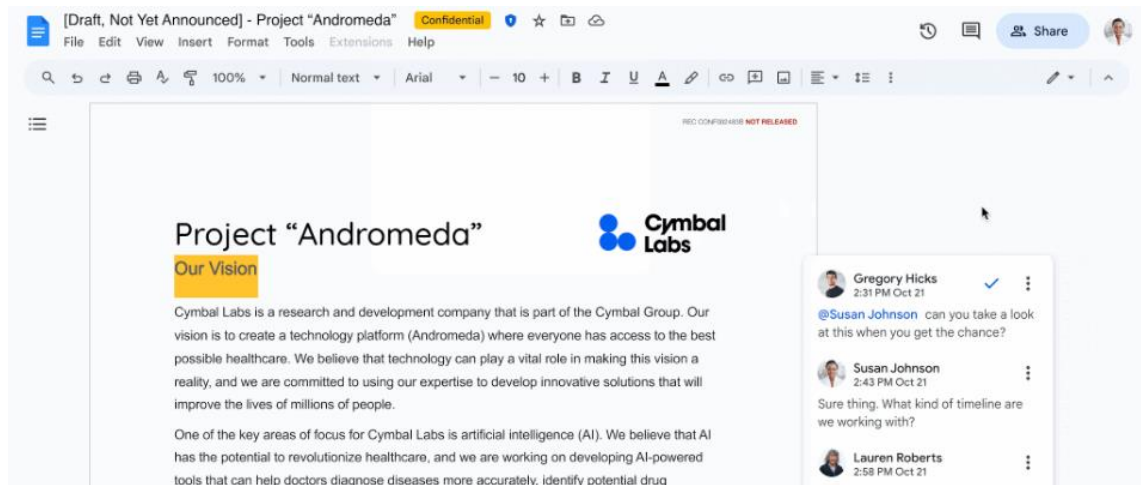


Рис. 1.1. Інтерфейс Google Docs

1.2.2. Notion

Платформа Notion (рис. 1.2) пропонує широкі можливості для створення баз даних, сторінок та завдань, що дозволяє користувачам організовувати інформацію різними способами [3]. Проте саме через багатфункціональність система стає складною для освоєння, а при роботі з великими обсягами даних відчутно сповільнюється. Як і у випадку з Google Docs, Notion є пропрієтарною системою, що обмежує її адаптацію під специфічні потреби користувачів.

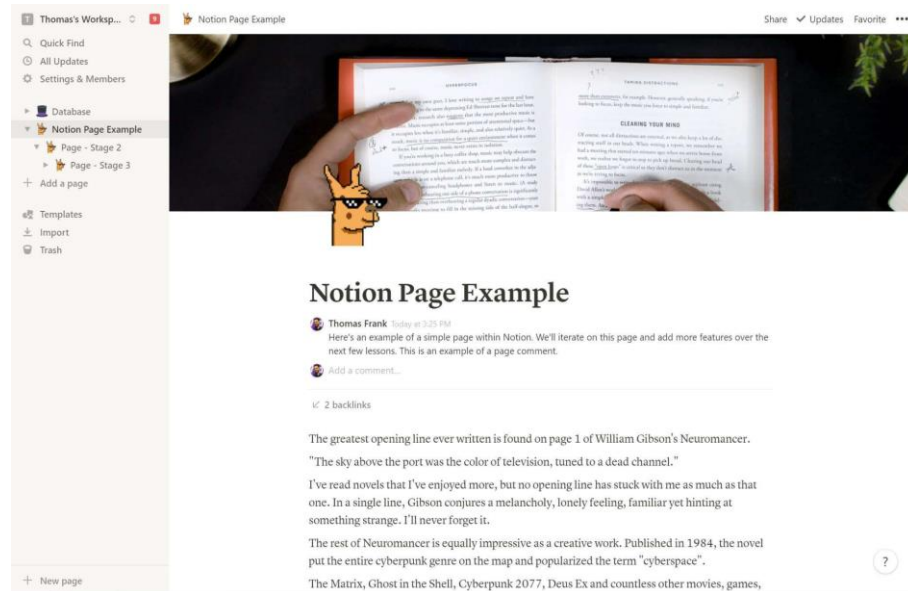


Рис. 1.2. Інтерфейс Notion

1.2.3. Obsidian

Obsidian (рис. 1.3) забезпечує ефективну персональну організацію знань з локальним зберіганням і графами зв'язків між нотатками [4]. Проте без додаткових плагінів неможливо реалізувати спільну роботу або інтеграцію із системами управління завданнями [5]. Таким чином, Obsidian ефективний лише для індивідуальної роботи, але не замінює командні інструменти.

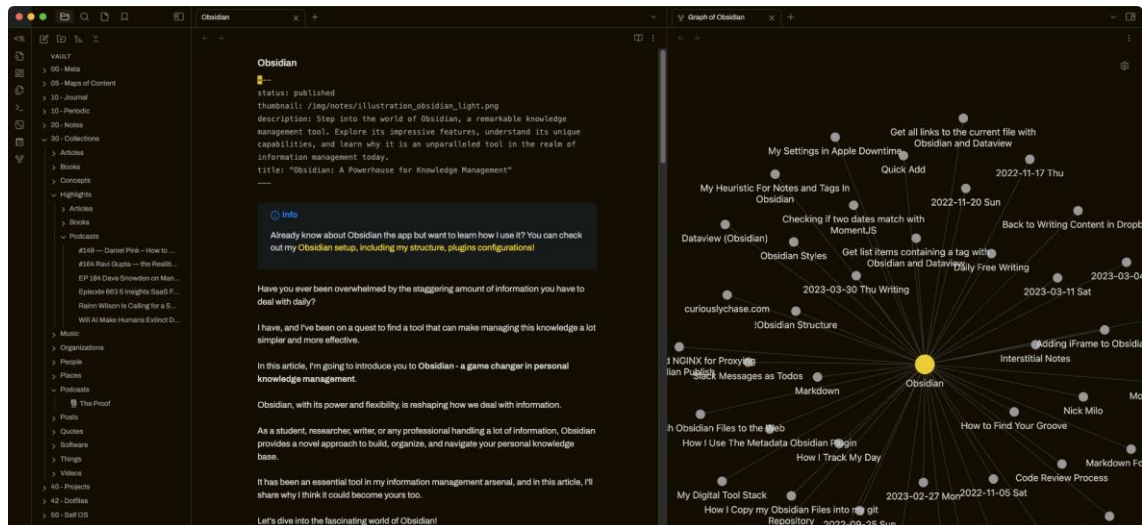


Рис. 1.3. Інтерфейс Obsidian

1.2.4. Trello

Застосунок Trello (рис 1.4) орієнтований на просту організацію за допомогою Канбан-дошок [6]. Його зручність для управління проектами не поширюється на ведення текстових нотаток або створення баз знань, а пропріетарність і залежність від зовнішніх серверів також обмежують контроль користувача над даними.

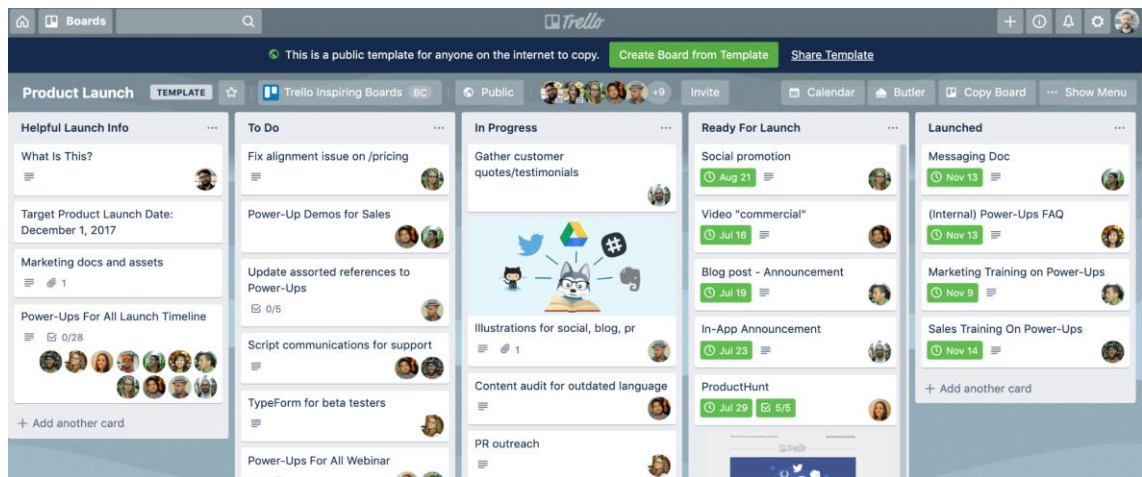


Рис. 1.4. Інтерфейс Trello

З огляду на аналіз сучасних інформаційних систем, можна зробити висновок, що кожен із розглянутих інструментів має свої сильні сторони та обмеження. Google Docs ефективний для спільного редагування текстів,

Notion дозволяє створювати бази знань і організовувати інформацію, Obsidian забезпечує персональну структуру нотаток, а Trello спрощує управління проектами через Kanban-дошки. Проте жодна система не поєднує всі ці функції одночасно. Крім того, більшість із них є пропрієтарними, що зменшує контроль користувача над даними та обмежує їхню адаптацію під специфічні потреби команд і індивідуальних користувачів.

1.3. Порівняння актуальних технологій розробки

Основна цінність веб-застосунку “Vault” полягає в усуненні бар’єрів між персональною фіксацією думок та командним виконанням задач. Це вимагає архітектури, яка підтримує миттєве перемикавання між режимами роботи без затримок на синхронізацію чи рендеринг. Вибір технологій обґрунтований необхідністю забезпечити швидкість рівня нативних застосунків в межах веб-технологій.

1.3.1. Реляційні та нереляційні СУБД

При проектуванні систем, орієнтованих на менеджмент неструктурованої інформації та колаборацію в реальному часі, критичним є вибір між реляційною та документоорієнтованою моделями даних. Реляційні системи, такі як PostgreSQL, базуються на жорсткій типізації та нормалізації, що вимагає декомпозиції складних об’єктів на множину пов’язаних між собою таблиць [7]. Для інформаційної системи типу “Vault”, де базовою одиницею даних є нотатка з динамічною структурою, такий підхід є менш доцільним через високу вартість операцій з’єднання таблиць JOIN та складність підтримки цілісності при постійній зміні формату контенту.

Документоорієнтована модель СУБД MongoDB виглядає більш адекватною для поставлених задач завдяки використанню формату BSON [8] для збереження даних. Це дозволяє зберігати дані у вигляді, що

максимально наближений до об'єктної моделі в мовах програмування. У контексті роботи з нотатками, які можуть містити довільну кількість вкладених блоків, NoSQL підхід дозволяє отримувати цілісні об'єкти за одну операцію читання без виконання складних з'єднань таблиць. У такий спосіб суттєво знижується затримка під час отримання даних на боці клієнта.

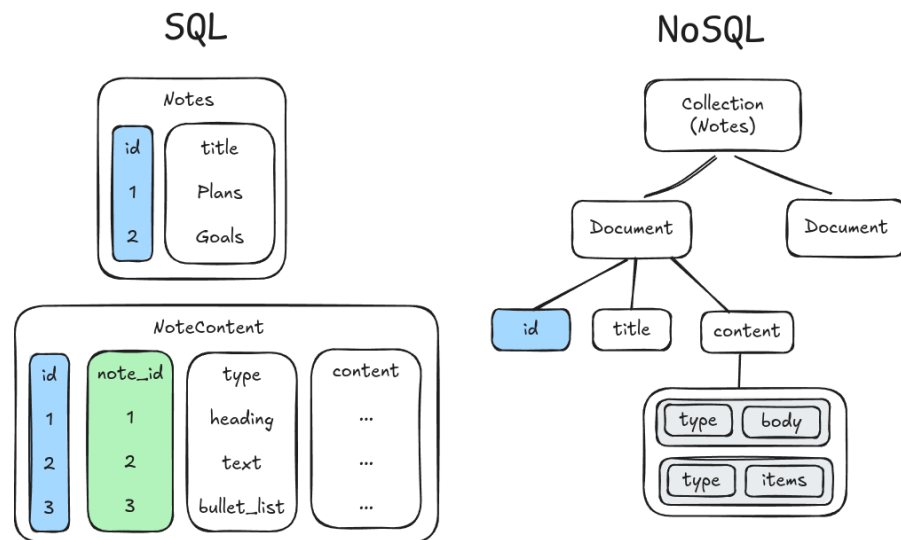


Рис. 1.5. Декомпозиція об'єкта в SQL проти цілісного документа в NoSQL

Сучасні реляційні СУБД підтримують роботу з типом даних JSONB, проте їх використання як основного сховища для динамічного контенту є компромісним рішенням. Робота з таким типом даних вимагає додаткових обчислювальних ресурсів для парсингу та індексації ключів, тоді як MongoDB є нативним середовищем для таких структур.

Іншим важливим аспектом також є масштабованість та гнучкість схеми. У процесі розвитку продукту додавання нових функціональних блоків у реляційних базах вимагає виконання операцій `ALTER TABLE` для зміни структури існуючої таблиці, що, у навантажених системах, призводить до простоїв. Документна модель дозволяє змінювати структуру

даних безпосередньо в кодї застосунку. Це сприяє безперервній доступності сервісу та швидкій ітерації розробки.

Саме безсхемна архітектура MongoDB дозволяє реалізувати динамічне середовище веб-застосунку “Vault” без накладних витрат та реструктуризацію бази даних. Використання документоорієнтованого підходу забезпечує пряму відповідність між структурами в пам’яті сервера та їхнім фізичним представленням у сховищі. Це мінімізує обчислювальні витрати на серіалізацію, сприяє зменшенню затримок та спрощує масштабування додатка в умовах безперервного потоку оновлень.

1.3.2. Аналіз середовищ виконання та протоколів обміну даними в реальному часі

Реалізація веб-застосунків з підтримкою спільної роботи в режимі реального часу висуває підвищені вимоги до середовища виконання серверної частини та протоколів обміну даними між клієнтом і сервером. Ключовими характеристиками таких систем є низька затримка передачі даних, можливість обробки великої кількості одночасних підключень та ефективна синхронізація стану між учасниками взаємодії.

Традиційний підхід до клієнт-серверної взаємодії на основі HTTP-запитів є малоприсадабленим для реалізації сценаріїв використання застосунку у реальному часі. Модель “запит-відповідь” потребує ініціації кожної взаємодії клієнтом, що призводить до надлишкових запитів або необхідності використання технік довгого опитування (long polling). Це, у свою чергу, збільшує навантаження на сервер та негативно впливає на затримки між оновленнями даних.

Більш ефективним рішенням для синхронізації даних у реальному часі є використання протоколу WebSocket [9], який забезпечує двонапрямлений постійний канал зв’язку між клієнтом та сервером (рис. 1.6). На відміну від HTTP, WebSocket дозволяє серверу самостійно

надсилати оновлення клієнту, що забезпечує ефективну синхронізацію змін між користувачами в режимі реального часу.

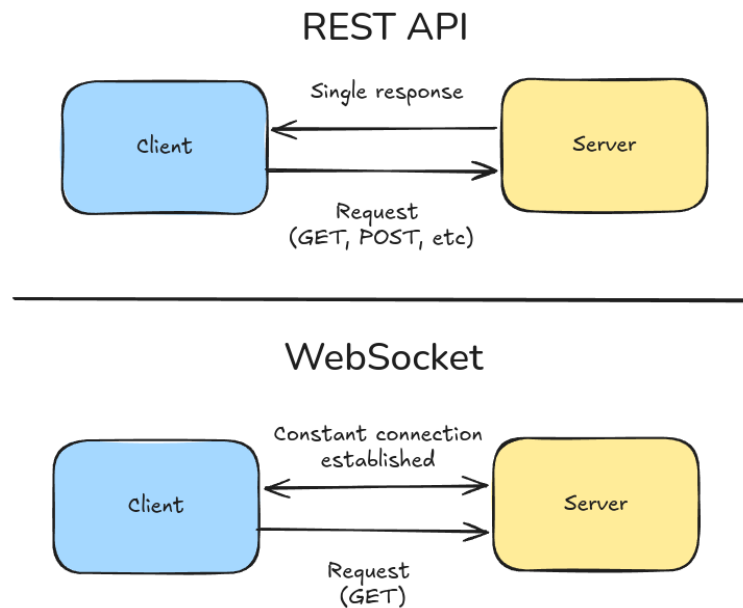


Рис. 1.6. Порівняльна схема моделей взаємодії клієнта та сервера за протоколами HTTP та WebSocket

Серед серверних середовищ, що розглядаються для реалізації таких систем, здебільшого використовують багатопотокові платформи на базі Java, Python або PHP, де для кожного підключення створюється окремий потік. Така модель є відносно простою для реалізації, але значне збільшення кількості одночасних користувачів призводить до високих витрат пам'яті та навантаження на процесор. Альтернативним підходом є Node.js, яке працює на основі подієво-орієнтованого циклу виконання (event loop) [10]. Це дозволяє обробляти тисячі одночасних підключень в межах одного потоку без значного збільшення ресурсних витрат, що робить його в парі з WebSocket оптимальним набором для застосунків із підтримкою колаборативної роботи в режимі реального часу.

1.3.3. Огляд сучасних клієнтських фреймворків для веб-інтерфейсів

Сучасні веб-застосунки, особливо з підтримкою роботи в реальному часі, потребують ефективних клієнтських фреймворків для організації компонентної архітектури та управління станом. Найпопулярнішими технологіями на ринку є React, Angular, Vue, а також більш нові фреймворки, такі як Svelte і Solid.js. Кожен із них пропонує свої підходи до рендерингу, управління даними та інтеграції зі сторонніми бібліотеками.

React є найпоширенішим рішенням, що надає компонентно-орієнтовану архітектуру та велику екосистему інструментів. Його сильні сторони – гнучкість та підтримка великої кількості бібліотек для управління станом, маршрутизації та рендерингу на стороні сервера [11]. Однак React потребує додаткового налаштування для оптимальної продуктивності, а обсяг коду та складність абстракцій можуть ускладнити невеликі проєкти.

Angular пропонує комплексне рішення “з коробки”, включаючи маршрутизацію, форми, HTTP-сервіси та систему dependency injection [12]. Це робить його потужним для великих корпоративних застосунків, але високий поріг входу та складність фреймворку можуть бути надмірними для легких веб-застосунків із швидким циклом розробки.

Svelte відрізняється тим, що перетворює компоненти в чистий JavaScript на етапі компіляції, зменшуючи обсяг коду, що виконується у браузері, та значно прискорюючи завантаження і рендеринг сторінок [13]. Його підхід дозволяє мінімізувати накладні витрати на віртуальний DOM, що особливо важливо для веб-застосунків із великою кількістю інтерактивних компонентів і спільної роботи в реальному часі.

Solid.js поєднує підхід компіляції з реактивним оновленням стану без віртуального DOM, що забезпечує дуже низькі накладні витрати та високу

продуктивність навіть у великих SPA. Він ідеально підходить для застосунків із швидкою зміною стану та високою інтенсивністю оновлень. Однак екосистема Solid ще менша, ніж у React чи Angular, що може створювати певні обмеження при використанні сторонніх бібліотек [14].

Вибір Svelte для веб-застосунку “Vault” обґрунтовується простотою синтаксису, високою продуктивністю та легкою інтеграцією з WebSocket і Node.js на боці сервера. Це дозволяє створити швидкий, легкий та масштабований інтерфейс користувача для роботи з динамічним контентом, зокрема нотатками та Kanban-дошками, із мінімальними накладними витратами на рендеринг та управління станом.

1.3.4. Побудова кросплатформних десктопних застосунків із веб-технологій

Для створення нативних десктопних застосунків із веб-інтерфейсом ключовим аспектом є процес білдингу, який перетворює фронтенд-код у самостійний виконуваний файл для конкретної операційної системи. Сучасними популярними підходами є використання фреймворків типу Electron та Tauri, забезпечуючи кросплатформну сумісність та доступ до системних ресурсів.

Electron [15] надає повноцінне середовище Chromium та Node.js всередині пакету, що забезпечує високу сумісність і простоту доступу до нативних можливостей. Проте такий підхід призводить до значних розмірів кінцевого дистрибутива та підвищених системних вимог, що може негативно впливати на швидкість запуску застосунку та споживання пам'яті.

Tauri [16] вирізняється компактністю та продуктивністю. В його роботі використовується системний веб-двигун операційної системи для рендерингу інтерфейсу та бекенд на мові програмування Rust. Під час білдингу Tauri створює легкий виконуваний файл, який інтегрує веб-код із

нативними API, забезпечуючи ефективну роботу інтерфейсу та мінімальні витрати ресурсів [17]. Крім того, Tauri підтримує механізм підпису та оновлення дистрибутива, що полегшує підтримку версійності та доставку оновлень користувачу.

Порівняння споживання оперативної пам'яті порожніх додатків на різних платформах наведено в табл. 1.1.

Таблиця 1.1

Порівняння споживання оперативної пам'яті (Release)

Операційна система	Electron	Tauri
Windows (x64)	≈4 МБ	≈4 МБ
MacOS (arm64)	≈300 МБ	≈72 МБ
Linux (x64)	≈620 МБ	≈93 МБ

Ці дані демонструють, що Tauri значно ефективніший у використанні пам'яті, особливо на MacOS та Linux. Різницю у розмірі дистрибутиву для порожніх додатків подано у табл. 1.2.

Таблиця 1.2

Порівняння ваги дистрибутиву

Операційна система	Electron	Tauri
Windows (x64)	≈345 МБ	≈3 МБ
MacOS (arm64)	≈463 МБ	≈4 МБ
Linux (x64)	≈302 МБ	≈3 МБ

Отримані дані демонструють суттєву різницю у споживанні ресурсів між Electron та Tauri. Особливо помітною є різниця на MacOS та Linux, де Electron створює значно більші накладні витрати на пам'ять. Переваги використання Tauri для застосунку “Vault” полягають у зменшенні розміру

дистрибутиву, високій продуктивності інтерфейсу, легкій інтеграції з серверними компонентами через API та підтримці кросплатформності.

Водночас варто зазначити, що Taui має певні обмеження щодо сумісності та поведінки інтерфейсу на різних платформах. Наприклад, деякі властивості CSS можуть працювати інакше в залежності від системи та/або версії системного веб-двигуна. Такі нюанси вимагають додаткового тестування та, за потреби, умовних перевірок у коді для забезпечення однакового досвіду користувача на різних ОС.

Попри це, Taui забезпечує оптимальний баланс між продуктивністю, вагою дистрибутива та доступністю на різних ОС, що робить його доцільним рішенням для білдингу веб-інтерфейсу у нативний додаток для “Vault”, дозволяючи ефективно поєднати легкість фронтенду з надійністю та інтеграцією з серверними компонентами.

РОЗДІЛ 2

МЕТОДОЛОГІЯ ТА ПРОЄКТУВАННЯ ЗАСТОСУНКУ “VAULT”

2.1. Архітектура та системні рішення

На основі проведеного аналізу сучасних підходів до менеджменту інформації та порівняння технологій розробки, було визначено ключові вимоги до архітектури застосунку “Vault”. Головними критеріями є реактивність інтерфейсу, низька затримка при синхронізації даних та модульність компонентів.

2.1.1. Багатошарова архітектура серверної частини

Для забезпечення масштабованості та легкості підтримки серверного коду було обрано багатошарову архітектуру (Layered Architecture). Основним принципом цього підходу є ізоляція бізнес-логіки від способів передачі даних та особливості збереження інформації. Багатошарова архітектура складається з наступних рівнів:

- Транспортний шар (Controllers/Handlers): відповідає за ініціалізацію з’єднань, авторизацію та маршрутизацію. Він не містить бізнес-логіки, а лише делегує виконання запитів відповідним сервісам.
- Сервісний шар (Services): центральна частина архітектури, де реалізовується логіка маніпуляції даними, перевірку прав доступу та валідацію.
- Шар доступу до даних (Repositories): забезпечує взаємодію з СУБД. Використання репозиторіїв дозволяє приховати складність запитів до бази даних і забезпечує легку зміну запитів без впливу на бізнес-логіку.
- Шар трансформації даних (Mappers): відповідає за перетворення внутрішніх моделей бази даних у об’єкти передачі даних (DTO), що мінімізує обсяг інформації, яка передається через мережу.

2.1.2. Обмін даними у реальному часі через WebSockets

Оскільки застосунок “Vault” орієнтований на колаборативну роботу, ключовим елементом архітектури є постійне двостороннє з’єднання на базі протоколу WebSocket, реалізоване за допомогою бібліотеки Socket.IO [18]. Основні архітектурні рішення на цьому рівні включають:

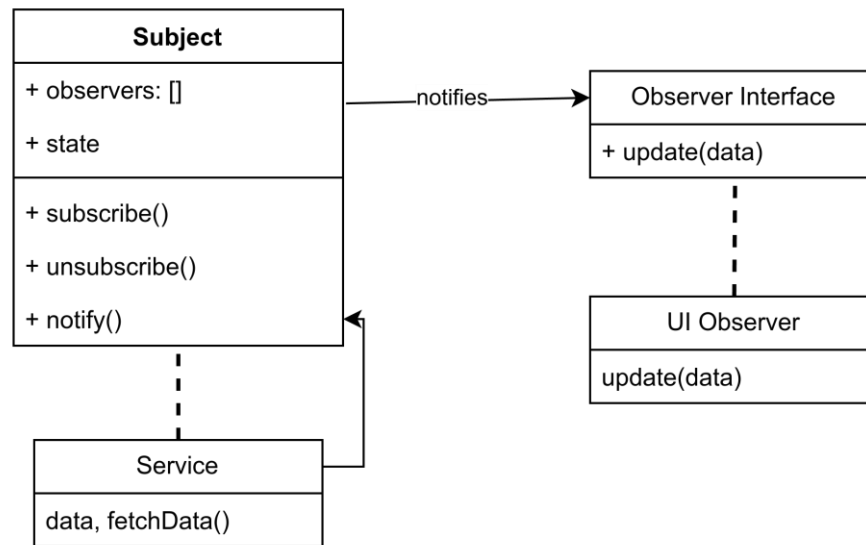
- Механізм підтверджень (Acknowledgments): використання зворотніх викликів (callbacks) для миттєвого підтвердження успішності операцій (наприклад, створення нотатки). Це дозволяє клієнт-частині застосунку миттєво реагувати на результат дії без очікування загальної розсилки подій (наприклад, з об’єктом даних створеної нотатки).
- Кімнати (Rooms): логічний розподіл користувачів за знаходженням у додатку. У такий спосіб події редагування нотаток або переміщення карток на дошці будуть доставлені лише тим учасникам, які переглядають відповідний об’єкт.
- Подієва модель синхронізації: сервер виступає “джерелом істини”, який після валідації змін транслює оновлений стан всім клієнтам у контексті, забезпечуючи ідентичність даних в учасників сесії.

2.1.3. Реактивна модель клієнтської частини та управління локальним станом

Архітектурне рішення для клієнтської частини застосунку базується на сигналах, що реалізується фреймворком Svelte 5. Основна мета - забезпечити синхронізацію інтерфейсу із серверним станом за мінімальних накладних витрат на обчислення.

П’ята версія фреймворку Svelte впровадила концепцію runes - набір спеціальних символів, що керують реактивністю через систему сигналів на етапі виконання (runtime reactivity) [19]. На відміну від попередніх версій фреймворку, де реактивність була обмежена межами svelte файлів, руни

використовують механізм Проху-об'єктів, а також сигналів, що побудовані на основі концепту Observable (рис. 2.1).



State changes -> Events -> Updates

Рис. 2.1. Схема концепту Observable

Механізм Проху дозволяє Svelte 5 перехоплювати операції читання та запису властивостей об'єктів, що автоматично реєструє залежності в момент доступу до них. Це забезпечує дрібнозернисту реактивність (fine-grained reactivity), де замість перевірки всього дерева компонентів система оновлює лише конкретні вузли DOM, безпосередньо пов'язані зі зміненим сигналом. Головною перевагою такого підходу є вихід реактивності за межі svelte файлів, що дозволяє описувати логіку управління станом у звичайних модулях JavaScript або TypeScript. Це дає можливість створювати універсальні сховища даних та складну бізнес-логіку, яка залишається реактивною незалежно від місця її використання в архітектурі застосунку.

Для реалізації локального стану в застосунку “Vault” ключову роль відіграють руни \$state, \$derived та \$effect. У спрощеному вигляді, вони

представляють собою Subject, Middleware відносно Subject, та Observer відповідно.

Руна `$state` ініціалізує реактивне джерело даних, створюючи об'єкт, за змінами якого фреймворк стежить на етапі виконання. Якщо певні дані в інтерфейсі залежать від базового стану, використовується руна `$derived`, яка автоматично обчислює похідні значення та кешує їх до моменту зміни першоджерела. Для виконання побічних дій використовується руна `$effect`, яка відстежує звернення до реактивних змінних всередині свого блоку та автоматично запускає побічний ефект при зміні залежностей.

Використання такої моделі дозволяє автоматизувати процес відстеження зв'язків між даними, що зводить до мінімуму ризик помилок при оновленні інтерфейсу та забезпечує оптимальне використання ресурсів системи користувача.

2.1.4. Локалізація інтерфейсу

Для локалізації інтерфейсу застосунку використовуватиметься бібліотека Paraglide JS [20].

Paraglide JS - сучасна бібліотека для локалізації веб-додатків, яка використовує компіляторний підхід. На відміну від традиційних runtime-бібліотек, Paraglide JS під час збірки проєкту компілює усі повідомлення у типізовані функції, що дозволяє уникнути помилок у ключах перекладів та параметрах (рис 2.2).



Рис. 2.2. Скомпільовані у типізовані функції ключі перекладів та параметри

Однією з ключових переваг бібліотеки є можливість tree-shaking'у функцій, тобто видалення непотрібних перекладів під час збірки. Завдяки такому підходу, Paraglide JS може зменшити розмір i18n-бандла до 70% у порівнянні з класичними runtime-бібліотеками, що поставляють усі переклади без винятку.

Крім того, бібліотека підтримує локалізовані маршрути сторінок, плюралізації, а також інтеграцію з Visual Studio Code розширеннями. Це робить її зручною як для розробників, так і команд перекладачів.

2.2. Моделювання інформаційних потоків

Веб-застосунок “Vault” ґрунтується на клієнт-серверній моделі, де основна логіка зосереджена на сервері, а клієнт відповідає за інтерфейс користувача та взаємодію з ним. Оскільки система передбачає роботу з нотатками в реальному часі, важливо змодельовати процеси, через які проходять дані застосунку. Для опису процесу створення нотатки користувачем було побудовано діаграму послідовностей повного циклу створення нотатки, надану у додатку А. Вона показує, які частини застосунку беруть участь у обробці запиту та в якому порядку між ними відбувається обмін даними.

Процес починається з введення основних даних нотатки в інтерфейс користувача. Після підтвердження дії, клієнт Socket.IO надсилає подію створення нотатки (“note:create”) на сервер через WebSocket-з’єднання. На сервері подія обробляється відповідним обробником (NoteHandler), який передає дані в сервісний шар (NotesService). Сервісний шар, в свою чергу, виконує перевірку прав доступу користувача до проєкту та у випадку відсутності необхідних прав перериває операцію та повертає клієнту повідомлення про помилку. Якщо перевірка проходить успішно, дані передаються в репозиторій, який відповідає за роботу з СУБД MongoDB. Після створення запису отриманий документ перетворюється у формат, зручний для передачі клієнту (NotesMapper), щоб не передавати зайві технічні поля та внутрішню структуру моделі. Після перетворення сервер повертає підтвердження дії клієнту-ініціатору, а також надсилає подію про створення нової нотатки всім підключеним клієнтам, які знаходяться у відповідному проєкті. Клієнтська частина застосунку оновлює локальний реактивний стан списку нотаток, після чого нова нотатка автоматично з’являється в інтерфейсі завдяки механізмам реактивності Svelte.

Побудована модель інформаційних потоків дозволяє чітко виділити відповідальність між окремими частинами системи: клієнт відповідає за взаємодію з користувачем, сервер - за бізнес-логіку та перевірку доступу, база даних - за збереження інформації, а WebSocket-з’єднання - за синхронізацію стану між клієнтами. Такий підхід робить поведінку системи передбачуваною.

2.3. Синхронізація змін під час спільного редагування нотатки

Однією з можливостей застосунку “Vault” стане спільне редагування нотатки у режимі реального часу. Для цього планується використати Yjs [21] - CRDT-фреймворк, який забезпечує автоматичну синхронізацію змін між декількома клієнтами без конфліктів.

На практиці для кожної нотатки можна створювати довільну кількість незалежних сесій Yjs. У рамках сесії користувачі вносять локальні зміни, які Yjs інтегрує у глобальний стан цієї сесії для всіх підключених клієнтів. Після завершення роботи з документом, сесію можна або закрити без збереження, або зберегти, інтегрувавши їх в основну нотатку. Такий підхід забезпечує гнучку та безпечну колаборацію, дозволяючи одночасно вести кілька експериментальних або паралельних потоків редагування без ризику втрати даних.

2.4. Концепція self-hosting та відкритість вихідного коду як засоби контролю даних

Окрім технічних рішень для спільного редагування, важливим аспектом для додатку “Vault” є контроль над даними користувача. Одним із ключових підходів для цього є self-hosting - розгортання сервера застосунку на власному обладнанні користувача або організації. Такий спосіб дозволяє повністю контролювати зберігання та обробку нотаток, уникнути залежності від сторонніх хмарних сервісів та забезпечити приватність даних [22].

Паралельно, відкритість вихідного коду забезпечує прозорість роботи системи. Будь-який користувач або розробник може перевірити, як застосунок обробляє інформацію, інтегрує зміни в спільні сесії Yjs, застосовує синхронізацію. Такий підхід не тільки підвищує довіру до системи, а й дозволяє вносити власні поліпшення або інтеграції, не чекаючи на оновлення від сторонніх розробників.

РОЗДІЛ 3

ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБ-ДОДАТКУ “VAULT”

3.1. Структура та схеми бази даних

Зберігання даних у застосунку “Vault” організовано через набір взаємопов’язаних колекцій, кожна з яких відповідає за окремий функціональний модуль системи. Головним архітектурним рішенням є використання ідентифікаторів ObjectId для встановлення зв’язків між сутностями. Хоча MongoDB належить до NoSQL-систем, такий підхід дозволяє вибудувати чітку структуру відносин: “один-до-багатьох”, “багато-до-багатьох” тощо.

Для забезпечення типізації та валідації даних на рівні програмного коду використано бібліотеку Mongoose [23]. Вона виступає як рівень об’єктно-документного відображення (ODM), що дозволяє описувати схеми даних безпосередньо у середовищі Node.js. Це гарантує, що кожен документ, який потрапляє до БД, відповідає визначеній схемі, та дозволяє використовувати механізми створення часових міток (timestamps) для відстеження змін.

3.1.1. Модель користувача

Колекція User є базовою для функціонування системи авторизації та ідентифікації учасників колаборативного середовища. Схема моделі, реалізована за допомогою Mongoose, включає наступні поля:

- name - поле імені користувача, яке використовується для відображення в інтерфейсі та може бути змінено користувачем у налаштуваннях профілю;
- username - унікальний ідентифікатор, який є обов’язковим для заповнення та використовується у процесі авторизації;

- password - хешована версія пароля користувача. Для забезпечення безпеки збереження даних використовується бібліотека bcrypt, яка виконує шифрування пароля перед його записом у базу даних;
- avatar - необов'язкове поле, що зберігає шлях до зображення профілю користувача.

Код даної схеми та моделі, що використовує її, наведено у лістингу 3.1.

Лістинг коду 3.1

Код User

```
const userSchema = new mongoose.Schema({
  name: { type: String, required: true, trim: true },
  username: { type: String, required: true, unique: true, trim: true },
  password: { type: String, required: true },
  avatar: { type: String },
});
const User = mongoose.model('User', userSchema);
```

3.1.2. Модель проєкту

Колекція Project виступає центральною сутністю, яка об'єднує нотатки та дошки в єдиний робочий простір. Схема проєкту описує наступні поля:

- name - назва проєкту;
- description - необов'язковий опис проєкту;
- icon - необов'язкова іконка проєкту;
- members - масив об'єктів, де кожен елемент містить посилання на User (у вигляді ObjectId), роль та дату приєднання до проєкту.
- link - об'єкт, що містить унікальний код для запрошень до проєкту з визначеним рівнем прав для нових учасників та посиланням на автора коду.
- Автоматичні системні поля createdAt (дата створення) та updatedAt (дата останнього оновлення), визначені параметром timestamps.

Код схеми моделі Project наведено у лістингу 3.2.

Лістинг коду 3.2

Код Project

```
const projectSchema = new mongoose.Schema(
  {
    name: { type: String, required: true },
    description: { type: String, required: false },
    icon: { type: String, required: false },
    members: [
      {
        user: {
          type: mongoose.Schema.Types.ObjectId,
          ref: 'User',
          required: true,
        },
        permission: {
          type: String,
          enum: [PERMISSIONS.OWNER, PERMISSIONS.EDIT, PERMISSIONS.VIEW],
          default: PERMISSIONS.VIEW,
        },
        joinedAt: { type: Date, default: Date.now },
      },
    ],
    link: {
      token: { type: String },
      permission: { type: String, enum: [PERMISSIONS.EDIT, PERMISSIONS.VIEW] },
      createdBy: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
    },
  },
  { timestamps: true, versionKey: false },
);
const Project = mongoose.model('Project', projectSchema);
```

3.1.3. Моделі даних модуля нотаток

Логіка збереження текстової інформації розподілена між двома колекціями Note та NoteFolder, що дозволяє вибудовувати ієрархічні структури знань.

Колекція NoteFolder відповідає за логічне групування нотаток в межах проєкту та описана схемою з наступними полями:

- projectId - посилання на проєкт, у якому створено папку;
- name - назва папки;
- description - необов'язковий опис папки;
- Автоматичні системні поля createdAt та updatedAt, визначені параметром timestamps.

Код схеми моделі NoteFolder наведено у лістингу 3.3.

Код NoteFolder

```
const noteFolderSchema = new mongoose.Schema (
  {
    projectId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Project',
      required: true,
    },
    name: { type: String, required: true },
    description: { type: String, required: false },
  },
  { timestamps: true, versionKey: false },
);
const NoteFolder = mongoose.model('NoteFolder', noteFolderSchema);
```

Колекція Note описує сутність для збереження метаданих та контенту нотатки. Колекція має наступні поля:

- `projectId` - посилання на проєкт, у якому створено нотатку;
- `folderId` - посилання на папку, у якому знаходиться нотатка. Може мати значення `null`, що вказує на знаходження нотатки у корені проєкту;
- `icon` - необов'язкова іконка нотатки;
- `title` - назва нотатки;
- `content` - неструктурований вміст нотатки. Тип `Object` для поля обрано для сумісності з форматом JSON-дерева Markdown-редактора.
- Автоматичні системні поля `createdAt` та `updatedAt`, визначені параметром `timestamps`.

Код схеми моделі Note наведено у лістингу 3.4.

Код Note

```
const noteSchema = new mongoose.Schema (
  {
    projectId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Project',
      required: true,
    },
    folderId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'NoteFolder',
    },
  },
);
```

```

    default: null,
  },
  icon: { type: String, required: false },
  title: { type: String, required: true },
  content: { type: Object, required: false },
},
{ timestamps: true, versionKey: false },
);
const Note = mongoose.model('Note', noteSchema);

```

3.1.4. Моделі даних модуля Kanban-дошок

Модуль організації завдань у Kanban-дошки базується на трьох взаємопов'язаних колекціях: Board, Column та Task.

Колекція Board містить загальні дані про дошку та теги, описані наступними полями схеми:

- projectId - посилання на проєкт, у якому створена дошка;
- name - назва дошки;
- description - необов'язковий опис дошки;
- tags - масив вкладених об'єктів, що містять назву, колір та індекс для впорядкування;
- Автоматичні системні поля createdAt та updatedAt, визначені параметром timestamps.

Код схеми моделі Board наведено у лістингу 3.5.

Лістинг коду 3.5

Код Board

```

const boardSchema = new mongoose.Schema(
{
  projectId: {
    type: mongoose.Schema.Types.ObjectId,
    ref: 'Project',
    required: true,
  },
  name: { type: String, required: true },
  description: { type: String, required: false },
  tags: [
    {
      name: { type: String, required: true, trim: true },
      color: { type: String, required: false },
      orderIndex: { type: Number, required: true },
    },
  ],
},
);

```

```
{ timestamps: true, versionKey: false },
);
```

Колекція `Column` описує вертикальні списки на дошці. Схема моделі описана наступними полями:

- `boardId` - посилання на дошку, у якій знаходиться колонка;
- `name` - назва колонки;
- `orderIndex` - порядковий номер колонки у дошці;
- `color` - необов'язковий колір колонки;
- Автоматичні системні поля `createdAt` та `updatedAt`, визначені параметром `timestamps`.

Код схеми моделі `Column` наведено у лістингу 3.6.

Лістинг коду 3.6

Код Column

```
const columnSchema = new mongoose.Schema (
  {
    boardId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Board',
      required: true,
    },
    name: { type: String, required: true },
    orderIndex: { type: Number, required: true },
    color: { type: String, required: false },
  },
  {
    timestamps: true,
    versionKey: false,
  },
);
const Column = mongoose.model('Column', columnSchema);
```

Колекція `Task` є найбільш деталізованою сутністю модуля. Вона містить наступні поля:

- `boardId` - посилання на дошку, у якій знаходиться завдання;
- `columnId` - посилання на колонку, у якій знаходиться завдання;
- `title` - назва завдання;
- `description` - необов'язковий опис завдання;
- `orderIndex` - порядковий номер завдання у колонці;

- color - необов'язковий колір завдання;
- tags - масив посилань;
- dueDate - кінцева дата виконання;
- assignedTo - посилання на користувача-виконавця;
- createdBy - посилання на користувача-автора;
- Автоматичні системні поля createdAt та updatedAt, визначені параметром timestamps.

Код схеми моделі Task наведено у лістингу 3.7.

Лістинг коду 3.7

Код Task

```
const taskSchema = new mongoose.Schema (
  {
    boardId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Board',
      required: true,
    },
    columnId: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'Column',
      required: true,
    },
    title: { type: String, required: true, trim: true },
    description: { type: String, required: false },
    orderIndex: { type: Number, required: true },
    color: { type: String, required: false },
    tags: [{ type: mongoose.Schema.Types.ObjectId }],
    dueDate: { type: Date, required: false },
    assignedTo: [{ type: mongoose.Schema.Types.ObjectId, ref: 'User' }],
    createdBy: {
      type: mongoose.Schema.Types.ObjectId,
      ref: 'User',
      required: true,
    },
  },
  { timestamps: true, versionKey: false },
);
const Task = mongoose.model('Task', taskSchema);
```

3.2. Архітектура серверної частини застосунку

Реалізація серверної частини веб-додатку “Vault” базується на платформі Node.js та фреймворку Express.js [24] із використанням

архітектурного шаблону розділення відповідальності (Separation of Concerns). Основною метою обраної структури є створення модульної системи, де кожен компонент виконує чітко визначену роль, що спрощує подальшу підтримку, тестування та розширення функціоналу.

Відповідно до обраного шаблону було реалізовано структуру проєкту, візуалізовану в додатку Б. Кожна з директорій виконує специфічну роль у загальному циклі роботи серверної частини:

- **config/** та **constants/** - містять конфігурійні змінні середовища та статичні значення (наприклад, рівні прав доступу), що забезпечує централізоване керування налаштуваннями;
- **routes/** та **controllers/** - формують шар обробки HTTP-запитів. Шляхи визначають кінцеві точки (endpoints), а контролери відповідають за валідацію вхідних даних та виклик відповідних сервісів;
- **middlewares/** - містить проміжні етапи для обробки запитів, зокрема логіку авторизації та обробки помилок;
- **errors/** - містить власні класи помилок (наприклад, NotFoundError, UnauthorizedError), що дозволяє стандартизувати відповіді сервера у разі виникнення відповідних ситуацій;
- **permissions/** - реалізує логіку перевірки прав користувача для доступу до конкретних ресурсів (проєктів, нотаток, дошок);
- **models/** - описує схеми даних для Mongoose, які було розглянуто у підрозділі 3.1;
- **repositories/** - реалізує шар доступу до даних, де зосереджені запити до СУБД;
- **services/** - містить ядро бізнес-логіки застосунку. Кожен сервіс реалізований як клас-синглтон, що гарантує наявність лише одного екземпляра об'єкта в пам'яті сервера протягом усього часу роботи процесу;

- **mappers/** - відповідає за перетворення внутрішніх об'єктів бази даних у формат, придатний для відправки клієнт-частині застосунку;
- **utils/** - містить допоміжні функції загального призначення, які використовуються в різних частинах системи.

Для покращення процесу розробки та уникнення складних шляхів імпорту, у проєкті використовуються аліаси (aliases). Node.js надає нативне рішення для реалізації аліасів через вбудовані псевдоніми підшляхів (subpath imports). Використання нативного рішення дозволяє відмовитись від сторонніх бібліотек та додаткових етапів компіляції, що позитивно впливає на швидкість серверної частини. Конфігурацію псевдонімів наведено у лістингу 3.8.

Лістинг коду 3.8

Код package.json

```
{
  "imports": {
    "#config/*": "./src/config/*.js",
    "#constants": "./src/constants/index.js",
    "#controllers": "./src/controllers/index.js",
    "#middlewares": "./src/middlewares/index.js",
    "#errors": "./src/errors/index.js",
    "#mappers": "./src/mappers/index.js",
    "#handlers": "./src/sockets/handlers/index.js",
    "#models": "./src/models/index.js",
    "#permissions": "./src/permissions/index.js",
    "#repositories": "./src/repositories/index.js",
    "#routes": "./src/routes/index.js",
    "#services": "./src/services/index.js",
    "#sockets": "./src/sockets/index.js",
    "#utils": "./src/utils/index.js",
    "#validators/*": "./src/validators/*.js"
  }
}
```

3.2.1. Авторизація та права доступу

Контроль доступу до REST-ендпоінтів реалізовано за допомогою проміжного шару auth middleware. Його функціонування базується на перевірці заголовка Authorization у форматі Bearer-токена. Процес обробки запиту включає вилучення JWT, його верифікацію за допомогою

секретного ключа та дешифрацію контенту (payload). У разі успішної перевірки дані користувача інтегруються в об'єкт запиту `req.user`, що дозволяє контролерам та сервісам ідентифікувати суб'єкта дії. Будь-яка невідповідність токена або його відсутність викине помилку `UnauthorizedError`, яка перериває виконання запиту. Код цього шару наданий у лістингу 3.9.

Лістинг коду 3.9

Код auth

```
const auth = (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (!authHeader) throw new UnauthorizedError('No token provided');

  const token = authHeader.split(' ')[1];
  if (!token) throw new UnauthorizedError('Malformed token');

  try {
    const decoded = jwt.verify(token, JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    throw new UnauthorizedError('Invalid or expired token');
  }
};
```

Цей механізм інтегрується при визначенні маршрутів Express.js, що гарантує виконання бізнес логіки лише авторизованими користувачами, забезпечуючи цілісність та приватність даних у системі. Приклад використання механізму описано у лістингу 3.10.

Лістинг коду 3.10

Код boardsRoutes

```
const router = express.Router();

router.get('/:id', auth, BoardsController.getBoard);
router.get('/:id/columns', auth, BoardsController.getBoardColumns);
router.get('/:id/tasks', auth, BoardsController.getBoardTasks);
```

Для забезпечення безпеки системи, механізм перевірки токенів реалізований не лише для HTTP-запитів, а й для протоколу WebSocket. Авторизація відбувається на етапі “рукоштовування” (handshake), після

встановлення сокет-з'єднання. Для авторизації розроблено проміжний шар `socketAuth`, який інтегрується в конфігурацію сокет-сервера. На відміну від `Express.js auth middleware`, цей шар отримує дані користувача з поля `auth` об'єкта сокета. Після успішної верифікації JWT ідентифікатор користувача зберігається в екземплярі `socket.userId`. Це дозволяє подальшим обробникам миттєво отримувати доступ до даних автора без необхідності повторної дешифрації токена при кожній події. Приклад реалізації шару сокет-авторизації наведено у лістингу 3.11.

Лістинг коду 3.11

Код `socketAuth`

```
export const socketAuth = (socket, next) => {
  const token = socket.handshake.auth.token;

  if (!token) return next(new UnauthorizedError());

  try {
    const userData = jwt.verify(token, JWT_SECRET);
    socket.userId = userData.userId;

    next();
  } catch (err) {
    next(new UnauthorizedError());
  }
};
```

Підключення даного шару відбувається на етапі ініціалізації сокет-сервера. Це створює обов'язковий фільтр для кожного нового підключення: якщо клієнт не надав валідний токен у полі `auth`, сервер відхиляє спробу з'єднання ще до моменту активації будь-яких функціональних обробників.

У підрозділі 3.1 зазначено, що проєкти налічують поле `members` з різними рівнями прав. Через розрізнення рівнів прав серед учасників проєкту, виникає необхідність у перевірці цих прав. Для вирішення цього завдання було розроблено сервіс `hasPermissions`. На відміну від базової перевірки токена, цей сервіс реалізує логіку перевірки прав на рівні конкретних об'єктів. Наприклад, чи має поточний учасник проєкту з роллю `view` право на видалення нотатки, чи зміни структури Kanban-дошки.

Головною особливістю сервісу `hasPermissions` є його модульність та здатність до автоматичного визначення контексту. Замість написання окремих перевірок для кожного типу запиту, сервіс використовує заздалегідь описаний об'єкт, де для кожного ідентифікатора ресурсу (наприклад, `noteId`, `boardId` тощо) призначено відповідний обробник. Код об'єкту з обробниками надано у лістингу 3.12.

Лістинг коду 3.12

Код PERMISSION_HANDLERS

```
export const PERMISSION_HANDLERS = {
  noteId: NotePermissionHandler,
  folderId: NoteFolderPermissionHandler,
  boardId: BoardPermissionHandler,
  projectId: ProjectPermissionHandler,
};
```

Кожен обробник реалізує методи `resolve` та `check` та параметр `needs`.

Метод `resolve` аналізує вхідні дані та, за відсутності необхідних для перевірки ідентифікаторів, звертається до репозиторію, щоб отримати їх. , який автоматично знаходить батьківський об'єкт сутності. Таким чином цей метод відновлює повний контекст ієрархії, необхідний для подальшої перевірки.

Метод `check`, після того як контекст встановлено, виконує запит до бази даних, перевіряючи наявність документа відповідно вказаному ідентифікатору та присутність і рівень прав поточного користувача у масиві `members` проєкту.

Параметр `needs` вказує на залежність об'єктів. Якщо перевірка на рівні конкретної сутності (наприклад, нотатки) не дала результату, сервіс рекурсивно піднімається на рівень вище і перевіряє права користувача на рівні папки або всього проєкту. Це забезпечує гнучке успадкування прав доступу: користувач, що має доступ до проєкту, автоматично отримує доступ до всіх його вкладених ресурсів. Також це створює можливість у

майбутньому розширити налаштування прав доступу на глибших рівнях за проєкт.

Приклад коду обробника описано у лістингу 3.13.

Лістинг коду 3.13

Код NotePermission

```
const NotePermission = {
  resolve: async (input) => {
    if (!input.noteId) throw new BadRequestError('noteId is required');

    if (input.folderId !== undefined && input.projectId) return input;

    const note = await NotesRepository.findById(input.noteId);
    if (!note) throw new NotFoundError('Note not found');

    if (input.folderId === undefined) input.folderId = note.folderId;
    if (input.projectId === undefined) input.projectId = note.projectId;

    return input;
  },

  check: async ({
    userId,
    noteId,
    folderId,
    projectId,
    allowedPermissions,
  }) => {
    if (!noteId) return false;

    const note = await NotesRepository.findOne({
      _id: noteId,
      projectId,
      folderId,
      members: {
        $elemMatch: { user: userId, permission: { $in: allowedPermissions }
      },
    },
  });
    return !!note;
  },

  needs: ['folderId', 'projectId'],
};
```

Код головної частини сервісу `hasPermissions`, що використовує подібні обробники, наданий у лістингу 3.14.

Лістинг коду 3.14

Код hasPermissions

```

export const hasPermissions = async (input) => {
  if (!input.allowedPermissions || !input.allowedPermissions.length)
    input.allowedPermissions = DEFAULT_ALLOWED_PERMISSIONS;

  const key = Object.keys(PERMISSION_HANDLERS).find(
    (k) => input[k] !== undefined,
  );
  if (!key) throw new InternalServerError('Unable to detect handler type');

  const handler = PERMISSION_HANDLERS[key];

  if (handler.resolve) input = await handler.resolve(input);

  if (await handler.check(input)) return true;

  if (handler.needs) {
    for (const depKey of handler.needs) {
      const depHandler = PERMISSION_HANDLERS[depKey];
      if (depHandler && (await depHandler.check(input))) return true;
    }
  }

  throw new ForbiddenError('No permission');
};

```

3.2.2. Socket.IO

Після проходження авторизації на етапі `socketAuth`, взаємодія з даними переходить до транспортного шару сокетів. Основна логіка зосереджена в обробниках подій (`handlers`), які, як і сервіси, реалізовані сінглтонами. Приклад підключення до обробників до сокет-серверу надано у лістингу 3.15.

Лістинг коду 3.15

Код `initSockets`

```

export const initSockets = (server) => {
  io.use(socketAuth);

  io.on('connection', (socket) => {
    const { userId } = socket;

    JoinHandler(io, socket, userId);
    ProjectsHandler(io, socket, userId);
    NoteFoldersHandler(io, socket, userId);
    NotesHandler(io, socket, userId);
    BoardsHandler(io, socket, userId);
    ColumnsHandler(io, socket, userId);
    SessionHandler(io, socket, userId);
    TasksHandler(io, socket, userId);
  });
};

```

Важливу частину логіки відіграють сокет-кімнати, завдяки яким інформація відправляється лише конкретним клієнтам за контекстом. За логіку взаємодії користувача з кімнатами відповідає JoinHandler, який доєднує/від'єднує сокет користувача до/від бажаної кімнати. Типи кімнат зазначені у папці constants основної директорії проєкту та налічують PROJECT, NOTE_FOLDER, NOTE, SESSION, BOARD, BOARDS. Відповідно до кожного типу кімнати, користувачі отримуватимуть інформацію про створення, оновлення та видалення відповідного об'єкту.

Приклад коду обробника підписок до кімнат описано у лістингу 3.16.

Лістинг коду 3.16

Код JoinHandler

```
const JoinHandler = (_, socket) => {
  const joinRoom = (type, id, cb) => {
    try {
      const room = `${type}:${id}`;
      socket.join(room);
      console.log(`[room:join] ${socket.id} joined ${room}`);
      cb?>({ success: true });
    } catch (error) {
      callbackError(cb, error);
    }
  };

  const leaveRoom = (type, id, cb) => {
    try {
      const room = `${type}:${id}`;
      socket.leave(room);
      console.log(`[room:leave] ${socket.id} left ${room}`);
      cb?>({ success: true });
    } catch (error) {
      callbackError(cb, error);
    }
  };

  socket.on('room:join', ({ type, id }, cb) => joinRoom(type, id, cb));
  socket.on('room:leave', ({ type, id }, cb) => leaveRoom(type, id, cb));
};
```

3.2.3. Логіка модуля нотаток

Для модуля нотаток описані сервіси, репозитарії та контролери у відповідних за структурою папках.

За бізнес-логіку відповідає клас `NotesService`, у якому описані методи для взаємодії з сутностями нотаток, а саме створення, редагування метаданих (назви, іконки тощо), переміщення та видалення. Приклад бізнес-логіки створення нотатки наведено у лістингу коду 3.17.

Лістинг коду 3.17

Код `NotesService`

```
create = async ({ userId, projectId, folderId = null, noteData }) => {
  const project = await ProjectsRepository.findById(projectId);
  if (!project) throw new NotFoundError('Project not found');

  if (folderId) {
    const folder = await NoteFoldersRepository.findById(folderId);
    if (!folder) throw new NotFoundError('Folder not found');
  }

  await hasPermissions({
    userId,
    folderId,
    projectId: project._id,
  });

  const note = await NotesRepository.create({
    ...noteData,
    projectId,
    folderId,
    members: [{ user: userId, permission: PERMISSIONS.OWNER }],
  });

  return noteMapper.toNoteDTO(note);
};
```

Метод `create` описує наступні етапи:

- метод приймає об'єкт з полями `userId`, `projectId`, `folderId` та `noteData`;
- перевіряє чи існує проєкт за вказаним у об'єкті `projectId`;
- перевіряє чи існує папка за вказаним у об'єкті `folderId`;
- дізнається, чи існують у користувача за вказаним `userId` права власності або редагування;
- створює сутність нотатки у СУБД через відповідний репозитарій;
- повертає дані про створену нотатку з використанням мапперу.

Контролером на основі `Socket.IO` виступає `NotesHandler`, який описує

аналогічні сервісу операції відносно сутностей нотаток. Приклад обробника події створення нотатки надано у лістингу коду 3.18.

Лістинг коду 3.18

Код NotesHandler

```
socket.on(
  'note:create',
  async ({ projectId, folderId = null, noteData }, cb) => {
    try {
      const note = await NotesService.create({
        userId,
        projectId,
        folderId,
        noteData,
      });

      const room = note.folderId
        ? `${ROOM_TYPES.NOTE_FOLDER}:${note.folderId}`
        : `${ROOM_TYPES.PROJECT}:${note.projectId}`;

      io.to(room).emit('note_list:created', note);

      cb?.({ success: true, message: 'Successfully created the note' });
    } catch (error) {
      callbackError(cb, error, 'Failed to create the note');
    }
  },
);
```

Обробник подій отримує об'єкт параметрів з протоколу WebSocket, викликає метод створення нотатки з відповідного сервісу та розсилає повідомлення про успішне створення нотатки у цільові кімнати.

Описаний обробник дає можливість для базових операцій відносно сутностей нотаток, але не налічує методів для редагування контенту нотатки. Для відповідного функціоналу описано підмодуль Sessions керування сесіями редагування нотаток, а саме їх створення, перетворення сесії у колаборативну, зберігання та закриття. Його архітектурна роль полягає у забезпеченні синхронізації контенту нотаток за допомогою протоколу Yjs та інтеграції з Markdown-редактором.

Управління сесіями базується на використанні структури даних Map, де ключем виступає ідентифікатор проєкту, а значенням - вкладена мапа

активних сесій. Це дозволяє ізолювати сесії в межах конкретних проєктів. Код ініціалізації мапи та допоміжної функції пошуку сесій наведено у лістингу 3.19.

Лістинг коду 3.19

Структура зберігання та доступу до активних сесій

```
export const projectsSessionMap = new Map();

const getSession = (projectId, sessionId) => {
  const projectSessions = projectsSessionMap.get(projectId);
  if (!projectSessions) throw new NotFoundError('Project sessions not found');

  const session = projectSessions.get(sessionId);
  if (!session) throw new NotFoundError('Session not found');

  return { projectSessions, session };
};
```

Процес створення сесії передбачає трансформацію JSON-контенту нотатки з БД у формат XML-фрагменту Yjs, що є необхідним для подальшої роботи фреймворку. Для цього використовуються утиліта `prosemirrorJSONtoYXmlFragment`. Кожна сесія отримує унікальний ідентифікатор через бібліотеку `nanoid` та `namespace` у `Socket.IO`, після чого зберігається у відповідну мапу проєкта.

Важливою особливістю є розмежування типів сесій на приватні, яка доступна лише автору, та колаборативні, доступ до якої мають всі учасники проєкту. При доєднанні до сесії використовується подія `session:member_join`, яка виконує подвійну перевірку: спочатку на рівні приватності сесії, а потім через сервіс `hasPermission` на рівні прав доступу до об'єкта нотатки. Це гарантує, що сторонній користувач не зможе отримати доступ до даних.

Завершення роботи з сесією реалізовано через подію `session:close`, яка підтримує два режими:

- **save** - виконується зворотня конвертація контенту нотатки у XmlFragment у JSON-структуру, оновлюється документ у СУБД через репозитарій та видаляється сесія;

- **discard** - сесія видаляється без синхронізації з БД.

У такий підхід активна робота з текстом відбувається в оперативній пам'яті з низькою затримкою, а сховище оновлюється лише за явною командою або за логікою завершення сесії.

3.2.4. Логіка модуля Kanban-дошок

Архітектурно модуль організації завдань у форматі Kanban-дошок розділений на три основні підмодулі: Boards, Columns та Tasks.

Boards керує високорівневими сутностями дошок та системою тегів у них. Основною задачею обробника BoardsHandler є менеджмент метаданих дошки, метаданих тегів та їх порядку. Події створення та редагування дошки транслуються у кімнату проєкта та відповідної дошки, завдяки чому інші користувачі можуть побачити зміни без оновлення сторінки.

Columns відповідає за структурування дошки у колонки. Оскільки порядок колонок є критичним для методології Kanban, обробник ColumnsHandler використовує поле orderIndex. При зміні порядку колонок сервер виконує перерахунок індексів та розсилає оновлений масив усім клієнтам у кімнаті дошки.

Tasks обробляє створення, редагування, призначення виконавців та переміщення карток. Логіка переміщення завдання task:move включає:

- перевірку прав доступу користувача до цільової дошки;
- зміну columnId та orderIndex у базі даних;
- трансляцію оновленого об'єкта завдання іншим учасникам дошки.

3.3. Візуальна частина інтерфейсу та взаємодія з користувачем

Проектування інтерфейсу застосунку ґрунтувалось на принципах мінімалізму та функціональної щільності. Головною метою було створення

середовища, яке не відволікає від контенту, але надає швидкий доступ до інструментів організації.

Візуальна концепція була розроблена в графічному середовищі Figma. За основу було взято темну кольорову схему та бібліотеку компонент на основі shadcn [25] у якості компонентної бази. При розробці клієнтської частини це дозволило використовувати перевірені UX-патерни (до прикладу, доступність з клавіатури), зберігаючи повний контроль над стилізацією кожного елемента. Для графічних символів (іконок) було використано бібліотеку Lucide.

3.3.1. Компоненти інтерфейсу

Архітектура інтерфейсу побудована за атомарним принципом Бред Фроста [26]:

- Атоми: кнопки, поля введення інформації, іконки;
- Молекули: картки завдань, елементи списку нотаток, контекстуальні меню;
- Організми: контекстуальне бокове меню, інтерактивні колонки Kanban.

Використання такої ієрархічної моделі дозволяє досягти високого рівня перевикористання коду. Наприклад, один і той самий “атом” може бути частиною декількох молекул, що згодом можуть бути частиною ще більшої кількості організмів (рис. 3.1-3.2).

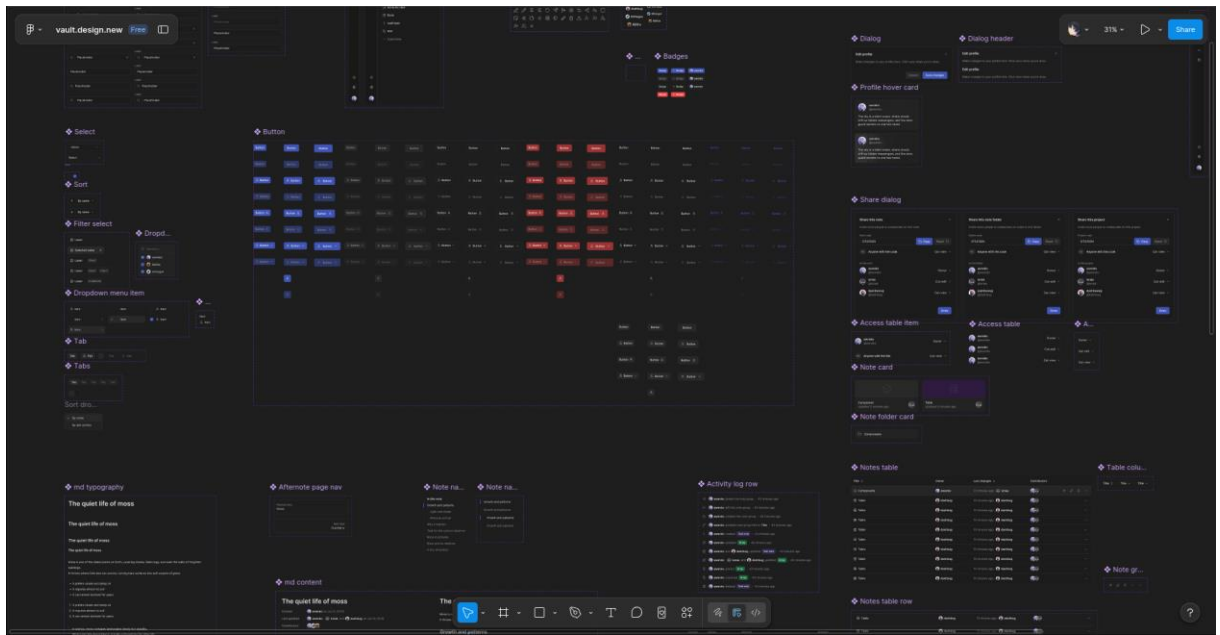


Рис. 3.1. Атомарні компоненти системи

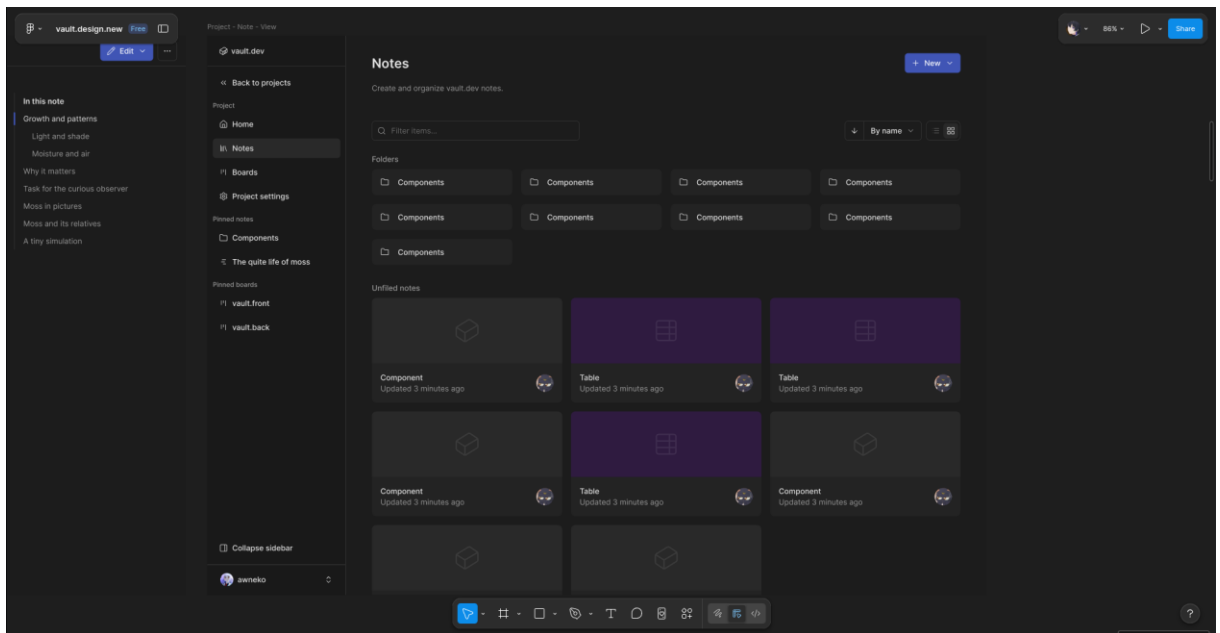


Рис. 3.2. Використання компонент у “організмі” - сторінці зі списком нотаток

Для забезпечення візуальної консистентності всіх компонентів було впроваджено систему відступів за сіткою 4 пікселя та типографіки на основі шрифту Inter. Усі атомарні компоненти підтримують динамічну

систему станів, що гарантує передбачувану поведінку інтерфейсу при взаємодії з користувачем.

У такий спосіб була сформована бібліотека компонентів, що стала фундаментом для складних інтерфейсів модулів нотаток та Kanban-дошок, забезпечивши їх швидке візуальне проєктування.

3.3.2. UX-рішення для навігації та спільної роботи

Ключовим UX-рішенням стала реалізація контекстуально-залежної навігації у вигляді бокового меню - сайдбару. Логіка полягає у автоматичній адаптації вмісту під поточний робочий простір користувача: при відкритій сторінці проєкту, сайдбар наповнюється навігаційними елементами проєкту (рис. 3.3).

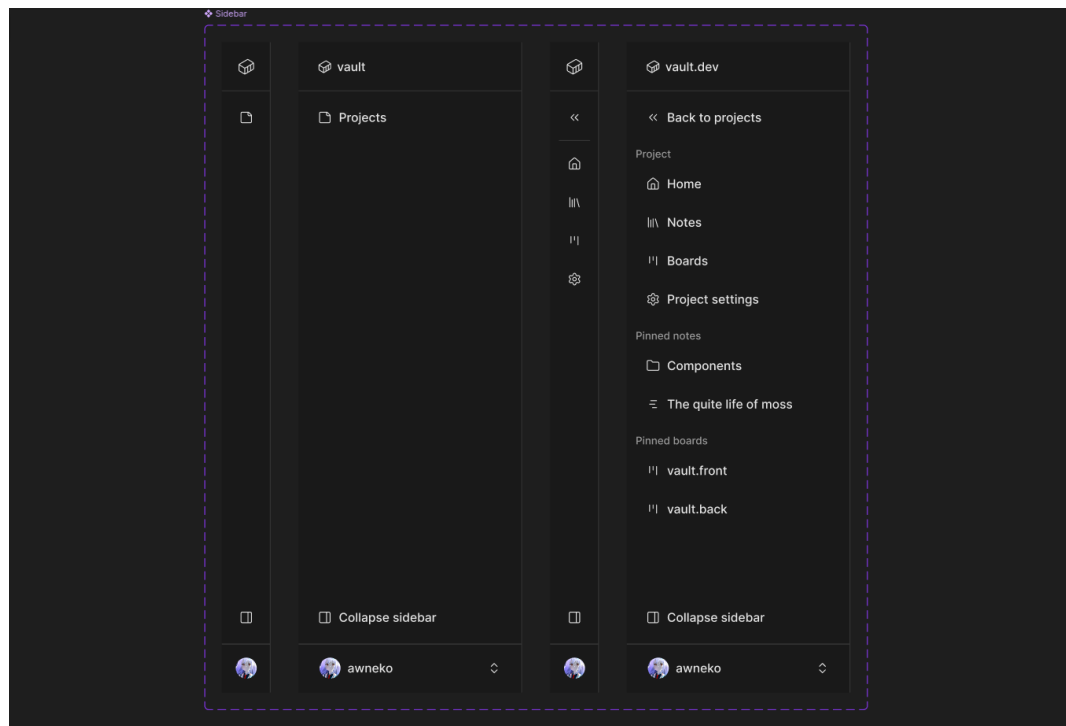


Рис. 3.3. Наповнення бокового меню навігації

Взаємодія між користувачами у реальному часі супроводжується візуальними підказками присутності. У сесіях спільного редагування нотаток відображені курсори інших учасників із кольоровою ідентифікацією, що не лише запобігає конфліктам при одночасному

редагуванні, а й створює ефект “спільної кімнати” (рис. 3.4). Аналогічно, у модулі Канбан-дошок будь-яке переміщення картки відображається з використанням плавних анімацій стану, що забезпечує передбачуваність змін у спільному просторі.

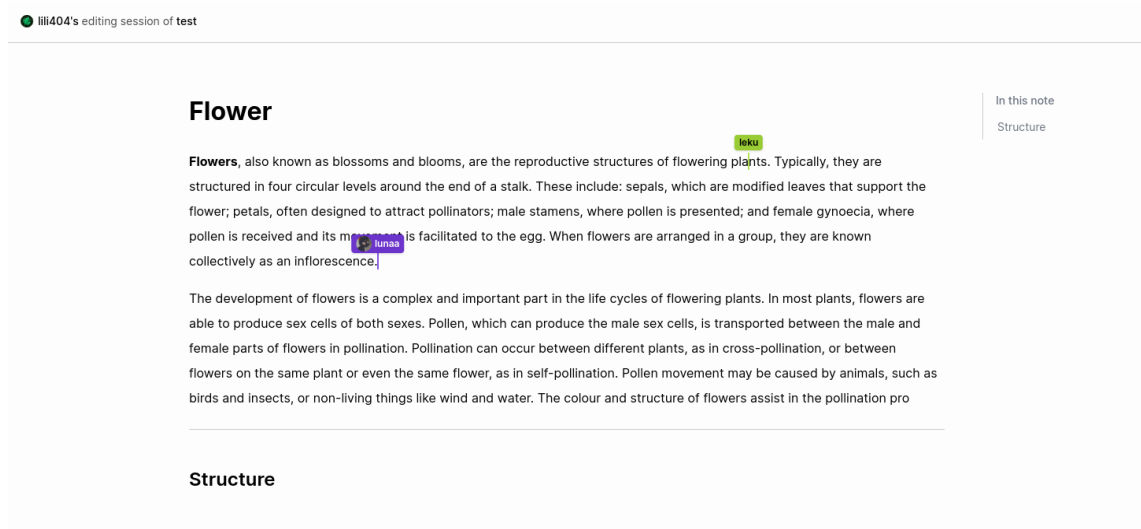


Рис. 3.4. Відображення курсорів інших учасників сесії редагування нотатки

3.4. Архітектура клієнтної частини застосунку

Для реалізації клієнтської частини застосунку “Vault” використовувався фреймворк SvelteKit у поєднанні з бібліотекою Svelte 5. Такий вибір дозволив побудувати високопродуктивний SPA, де інтерфейс миттєво реагує на зміни даних без повного перезавантаження сторінок.

3.4.1. Структура SPA на основі SvelteKit

Архітектура клієнтської частини організована за принципом файлової маршрутизації, яку використовує фреймворк SvelteKit. Основна структура проєкту включає наступні ключові директорії:

- **src/routes/(private)/** та **src/routes/(public)/** - логічне групування маршрутів залежно від стану авторизації користувача;

- **src/lib/api/** - шар абстракції над HTTP-запитами, реалізований за допомогою бібліотеки Axios. Для роботи у десктопному середовищі Tauri використовується спеціальний адаптер `axios-tauri-api-adapter`;
- **src/lib/stores/** - централізовані сховища стану, побудовані на класах TypeScript з використанням рун;
- **src/lib/sockets/** - підсистема обробки подій реального часу, що включає обробники та еміттери подій.

Використання вбудованого у SvelteKit механізму Layouts дозволило реалізувати спільні інтерфейсні елементи (наприклад, контекстуальний сайдбар) лише один раз на рівні кореневого макету, що забезпечує безшовний перехід між елементами проєкту без зайвого рендерингу.

3.4.2. Управління локальним станом

Управління станом у додатку ґрунтується на використанні класів-сингльтонів, що використовують реактивні зміни на основі рун `$state` та `$derived`.

Приклад реалізації `KanbanState`, який керує станом Kanban-дошок, наведено у лістингу 3.20.

Лістинг коду 3.20

Код KanbanState

```
class KanbanState {
  query = $state("");
  view = $state<ViewType>(ViewType.Grid);
  list = $state<Board[]>([]);
  current = $state<Board | null>(null);

  filteredList = $derived.by(() => {
    if (!this.query) return this.list;
    const regex = new RegExp(escapeRegExp(this.query), "i");
    return this.list.filter(board =>
      regex.test(board.name) || (board.description
regex.test(board.description))
    );
  });

  get isGridView() {
    return this.view === ViewType.Grid;
  }
}
```

```

    addBoard(board: Board) {
      this.list.push(board);
    }
  }
}

export default new KanbanState();

```

Використання синглтона гарантує синхронізацію стану між різними частинами інтерфейсу, забезпечуючи цілісність даних у межах всього застосунку.

3.4.3. Взаємодія з API та Socket.IO Client

Взаємодія з сервером у застосунку розділена на два рівні: REST API та Socket.IO. Для виконання HTTP-запитів використовується клієнт Axios [27], який у середовищі десктопного додатка доповнюється адаптером `axios-tauri-api-adapter` для коректної роботи з нативними мережевими запитами.

Особливість архітектури взаємодії з сервером є механізм управління WebSocket-подіями. Для запобігання витокам пам'яті та зайвому навантаженню на клієнт, підписки на події реалізовані через систему підрахунку посилань (Reference Counting). Приклад логіки менеджера подій наведено у лістингу 3.21.

Лістинг коду 3.21

Код менеджера подій

```

const refCounts = new Map<SocketEvent, number>();

export const events = {
  subscribe(keys: SocketEvent[]) {
    keys.forEach((key) => {
      const currentCount = refCounts.get(key) ?? 0;

      if (currentCount === 0) {
        const handler = HANDLERS[key];
        socket.on(key, handler);
      }

      refCounts.set(key, currentCount + 1);
    });
  },
  return () => {
    keys.forEach((key) => {

```

```

const currentCount = refCounts.get(key) ?? 0;

if (currentCount <= 1) {
  socket.off(key, HANDLERS[key]);
  refCounts.delete(key);
} else {
  refCounts.set(key, currentCount - 1);
}
});
},
};
};

```

Такий підхід вирішує проблему життєвого циклу обробників у складному SPA-інтерфейсі. Оскільки одна й та сама подія сокета може бути потрібна декільком незалежним компонентам одночасно, механізм гарантує, що реальна деактивація прослуховування каналу відбудеться лише тоді, коли кількість активних підписників дорівнюватиме нулю.

Це забезпечує автоматичне очищення ресурсів при розмонтуванні компонентів, запобігає дублюванню обробників та виключає витрати ресурсів на оновлення станів, які в даний момент не відображаються в інтерфейсі.

3.4.4. Форми та валідація даних

Будь-яка взаємодія користувача з інтерфейсом застосунку, від створення нотатки до налаштування Kanban-дошки, супроводжується використанням форм для введення інформації. Для цього у проєкті використано зав'язку бібліотек Zod [28] та SvelteKit Superforms [29].

Така комбінація дозволяє описати єдину схему валідації, яка одночасно перевіряє типи в коді, підсвічує помилки користувачу в реальному часі та захищає API на стороні сервера. Superforms автоматизує обробку станів завантаження та відображення повідомлень про помилки, що значно покращує UX. Приклад схеми для створення дошки наведено у лістингу 3.22.

Лістинг коду 3.22

Код схеми валідації для створення Kanban-дошки

```
const boardBaseSchema = z.object({
  name: z
    .string()
    .min(3, { error: "Board name must be at least 3 characters long" })
    .max(50, { error: "Board name cannot exceed 50 characters" }),
  description: z
    .string()
    .max(300, { error: "Description is too long (max 300 characters)" })
    .nullable(),
});
```

Для ініціалізації форми на клієнті використовується функція `superForm`, яка зв'язує схему валідації з логікою відправки даних. Це дозволяє виконувати перевірку та обробляти відповідь сервера в єдиному блоці коду. Приклад використання форми для створення дошки наведено у лістингу 3.23.

Лістинг коду 3.23

Код обробки форми для створення Kanban-дошки

```
const form = superForm(defaults(zod4(createBoardSchema)), {
  SPA: true,
  validators: zod4(createBoardSchema),
  onUpdate({ form }) {
    if (form.valid)
      KanbanBoardsSocket.create(
        { projectId, boardData: $formData },
        {
          onSuccess: () => {
            open = false;
          },
        },
      );
  },
});
```

3.4.5. Редактор нотаток на основі Markdown

Частиною модуля редагування нотатків є безпосередньо Markdown редактор, розроблений на основі бібліотеки Markdown. Для поточної архітектури застосунку було розроблено 2 окремі редактори для перегляду та редагування відповідно.

Редактор для перегляду `ViewEditor` оптимізований для швидкого рендерингу статичного контенту та генерує зміст на основі ієрархії заголовків Markdown.

Редактор для редагування CollabEditor є більш комплексним за структурою, бо інтегрує редактор з фреймворком Yjs для редагування у сесіях у реальному часі. Ключовим елементом реалізації є поєднання бібліотеки Yjs та провайдера y-socket.io.

Приклад реалізації логіки редактора для редагування наведено у лістингу 3.24.

Лістинг коду 3.24

Код реалізації логіки CollabEditor

```
const doc = new Doc();
const provider = new SocketIOProvider(configStore.serverUrl, sessionId,
doc);

const awareness = provider.awareness;
if (authStore.user) {
  awareness.setLocalStateField("user", {
    id: authStore.user.id,
    name: authStore.user.name,
    color: randomHexColor(),
  });
}

collabService
  .bindDoc(doc)
  .setAwareness(awareness)
  .setOptions({
    yCursorOpts: {
      cursorBuilder: (user) => createCustomCursor(user),
      selectionBuilder: (user) => ({ style: `background-color:
${user.color}` })
    }
  })
  .connect();
```

Особливістю даної реалізації є використання механізму Awareness, який дозволяє передавати метадані користувачів (наприклад, ім'я, колір та аватар), що підключені до поточної сесії. У коді компонента реалізовано власний cursorBuilder, який динамічно створює DOM-елементи для візуалізації курсорів інших учасників.

3.4.6. Інтерактивні Kanban-дошки

Модуль організації завдань за методологією Kanban побудований на використанні бібліотеки svelte-dnd [30], яка забезпечує високу

продуктивність механізму Drag-and-Drop (рис. 3.5) навіть при великій кількості об'єктів.

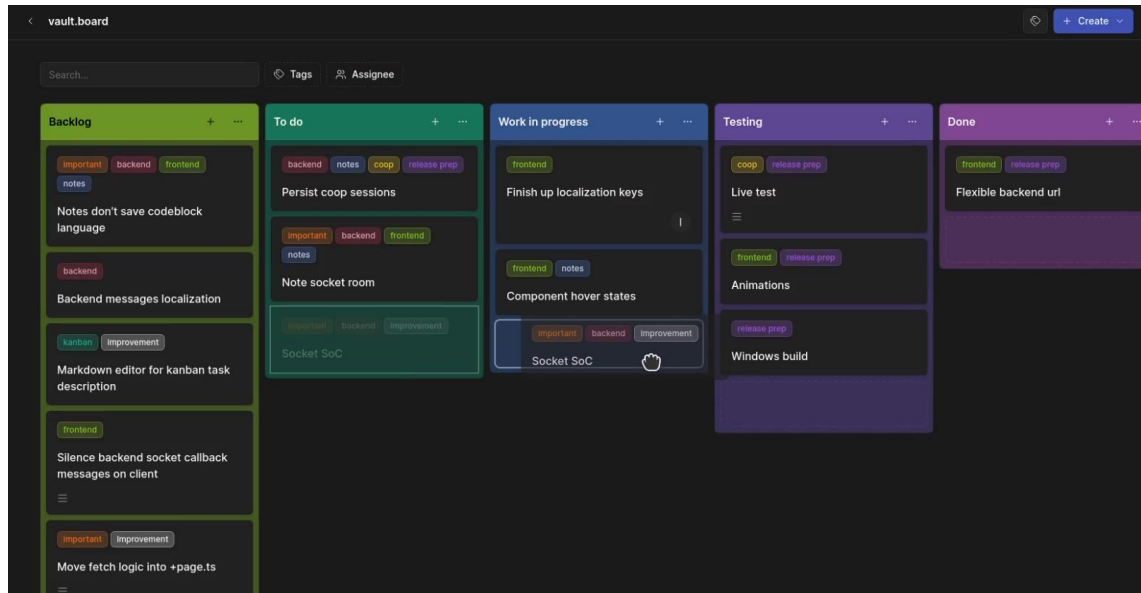


Рис. 3.5. Механізм Drag-and-Drop на прикладі картки Kanban-дошки

Компоненти колонки та картки завдання використовують надані бібліотекою директиви `use:draggable` та `use:draggable` для опису можливості перетягування та правил викидання. Система підтримує одночасне перетягування як окремих карток, так і цілих колонок. Для запобігання конфліктів у момент перетягування картки тимчасово блокується можливість зміщення колонок, та навпаки. При відпусканні елемента в новій позиції спрацьовує функція, яка обчислює індекс нової позиції елемента та цільовий контейнер, інформація про які відправляється на сервер через протокол WebSocket.

Логіку переміщення завдання між колонками надано у лістингу 3.25.

Лістинг коду 3.25

Код реалізації логіки переміщення завдання між колонками

```
const handleTaskDrop = (state: DragDropState<Task>, orderIndex: number) =>
{
  const { draggedItem, targetContainer } = state;

  const samePosition = draggedItem.orderIndex === orderIndex &&
```

```

        draggedItem.columnId === targetContainer;

    if (!targetContainer || samePosition) return;

    KanbanTasksSocket.move({
        columnId: targetContainer,
        orderIndex,
        taskId: draggedItem.id
    });
};

```

Важливу роль у зручності використання дошок має система фільтрації. Використання руни `$derived` у поєднанні з компонентами поля пошуку та випадаючого меню дозволяє реалізувати пошук та фільтрацію за назвою, тегами та виконавцями без звернення до сервера.

3.5. Організація середовища розгортання та процеси збірки застосунку

Процес підготовки застосунку до використання кінцевим користувачем базується на принципах автоматизації та контейнеризації, що відповідає концепції `self-hosting`.

Для розгортання серверної частини та бази даних MongoDB обрано Podman [31]. На відміну від традиційного Docker, Podman дозволяє запускати контейнери від імені звичайного користувача без `root`-привілеїв, що суттєво підвищує рівень безпеки при розгортанні. Керування стеком сервісів у Podman реалізовано через специфікацію `podman-compose`, що забезпечує легке розгортання на будь-якій Linux-системі.

Збірка клієнтської частини додатку повністю автоматизована з використанням GitHub Actions. Відповідно до конфігурації у репозитарії, процес CI/CD включає наступні етапи:

- підготовка середовища з використанням Node.js та Rust;
- генерація статичних елементів SvelteKit;
- білдинг ядра через Tauri для цільової платформи (Windows, Linux);
- формування та вивантаження готового інсталяційного пакету у форматах `.msi` та `.AppImage`.

Завдяки відкритому вихідному коду будь-який користувач або організація має можливість не лише розгорнути готові рішення, а й самостійно модифікувати компоненти серверної чи клієнтської частини під власні потреби. Процес збірки на власному пристрої гарантує відсутність сторонніх втручань у виконуваних файли та забезпечує повний контроль над життєвим циклом застосунку.

3.6. Перспективи вдосконалення та напрямки розвитку застосунку

Розроблена система “Vault” є масштабованою платформою, архітектура якої дозволяє впроваджувати нові інструменти для менеджменту знань без змін вже існуючих модулів. Основними напрямками подальшого розвитку застосунку є:

- Мобільна адаптація, що забезпечить доступ до інструментів з мобільних пристроїв;
- Вдосконалення редактора нотаток, а саме версіонування та інструменти взаємодії з існуючим контентом;
- Гнучкість спільної роботи на основі публічних посилань на окремі сутності, що дозволить ділитися інформацією з користувачами поза межами конкретного проєкту;
- Аналітичні інструменти для відстеження та візуалізації продуктивності команди в межах проєкту;
- Персоналізація з можливістю персоналізації інтерфейсу, а саме створення власних тем.

Реалізація зазначених напрямків дозволить перетворити “Vault” із функціонального інструмента на універсальну екосистему для управління знаннями, що поєднує високу продуктивність інтерфейсу та принципи безпеки та контролю над власними даними.

ВИСНОВКИ

У межах кваліфікаційної роботи було проведено аналіз предметної області на наявність функціонального розриву між інструментами для швидкої фіксації думок та менеджменту існуючих знань. Виявлено, що більшість існуючих рішень обмежують користувача або складністю інтерфейсу, або закритістю вихідного коду, що перешкоджає контролю над приватністю даних.

Для вирішення виявленої проблеми було обрано технологічний стек на основі Svelte 5 для клієнтської частини, Node.js для серверної частини та MongoDB у якості СУБД. Для створення цільної візуальної концепції застосунку, заснованої на принципах атомарного дизайну, було обрано середовище Figma.

З використанням зазначених технологій було спроектовано та розроблено модульну архітектуру застосунку з системою рекурсивної перевірки прав доступу до створених сутностей, реалізовано механізми синхронізації у реальному часі за допомогою WebSocket та створено модулі спільного редагування Markdown нотаток та менеджменту Kanban-дошок.

Для забезпечення кросплатформності та високої швидкості нативного застосунку було використано фреймворк Tauri, що значно зменшує споживання системних ресурсів у порівнянні з популярними рішеннями. Підсумком роботи стала реалізація концепції self-hosting через контейнеризацію серверної частини, що разом із відкритістю вихідного коду гарантує кінцевому користувачу повний контроль над функціоналом застосунку, приватністю та безпекою власних даних.

Реалізована архітектура застосунку “Vault” закладає фундамент для подальшого вдосконалення та розширення системи, зокрема у напрямках мобільної адаптації, інструментах аналітики та персоналізації інтерфейсу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Forte T. Building a Second Brain: A Proven Method to Organize Your Digital Life and Unlock Your Creative Potential. Atria Books, 2022. 242 с.
2. Google Docs: Editor | Google Workspace. Google Workspace. URL: <https://workspace.google.com/products/docs/> (дата звернення: 04.04.2026).
3. What is Notion?. Notion. URL: <https://www.notion.com/help/guides/what-is-notion> (дата звернення: 04.04.2026).
4. Obsidian - Sharpen your thinking. Obsidian. URL: <https://obsidian.md/> (дата звернення: 04.04.2026).
5. Introduction to Obsidian Sync - Obsidian Help. Obsidian - Sharpen your thinking. URL: <https://obsidian.md/help/sync> (дата звернення: 04.04.2026).
6. Capture, organize, and tackle your to-dos from anywhere | Trello. Capture, organize, and tackle your to-dos from anywhere | Trello. URL: <https://trello.com/> (дата звернення: 04.04.2026).
7. Concepts. PostgreSQL Documentation. URL: <https://www.postgresql.org/docs/current/tutorial-concepts.html> (дата звернення: 04.04.2026).
8. BSON (Binary JSON) Serialization. BSON (Binary JSON) Serialization. URL: <https://bsonspec.org/> (дата звернення: 04.04.2026).
9. RFC 6455: The WebSocket Protocol. IETF Datatracker. URL: <https://datatracker.ietf.org/doc/html/rfc6455> (дата звернення: 04.04.2026).
10. The Node.js Event Loop | Node.js v24.14.1 Documentation. Node.js – Run JavaScript Everywhere. URL: <https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick> (дата звернення: 04.04.2026).
11. React Reference Overview – React. React. URL: <https://react.dev/reference/react> (дата звернення: 04.04.2026).

12. What is Angular? • Angular. Home • Angular. URL: <https://angular.dev/overview> (дата звернення: 04.04.2026).
13. Svelte vs React: I Built the Same App in Both – Here's What I Learned. Medium. URL: <https://navanathjadhav.medium.com/svelte-vs-react-i-built-the-same-app-in-both-heres-what-i-learned-57f64cc0e9ca> (дата звернення: 04.04.2026).
14. Solid Docs. Solid Docs. URL: <https://docs.solidjs.com/> (дата звернення: 04.04.2026).
15. Why Electron | Electron. Build cross-platform desktop apps with JavaScript, HTML, and CSS | Electron. URL: <https://www.electronjs.org/docs/latest/why-electron> (дата звернення: 04.04.2026).
16. What is Tauri?. Tauri. URL: <https://v2.tauri.app/start/> (дата звернення: 04.04.2026).
17. GitHub - Elanis/web-to-desktop-framework-comparison: An objective comparison of multiple frameworks that allow us to "transform" our web apps to desktop applications. GitHub. URL: <https://github.com/Elanis/web-to-desktop-framework-comparison> (дата звернення: 04.04.2026).
18. Introduction | Socket.IO. Socket.IO. URL: <https://socket.io/docs/v4/> (дата звернення: 04.04.2026).
19. Introducing runes. Svelte • Web development for the rest of us. URL: <https://svelte.dev/blog/runes> (дата звернення: 04.04.2026).
20. Paraglide JS | inlang. inlang. URL: <https://inlang.com/m/gerre34r/library-inlang-paraglideJs> (дата звернення: 04.04.2026).
21. GitHub - yjs/yjs: Shared data types for building collaborative software. GitHub. URL: <https://github.com/yjs/yjs> (дата звернення: 04.04.2026).
22. Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World. Norton & Company, Incorporated, W. W., 2016. 400 с.

23. Mongoose ODM v9.4.1. Mongoose ODM v9.4.1. URL: <https://mongoosejs.com/> (дата звернення: 04.04.2026).
24. Express - Node.js web application framework. Express - Node.js web application framework. URL: <https://expressjs.com/> (дата звернення: 04.04.2026).
25. The Foundation for your Design System - shadcn/ui. The Foundation for your Design System - shadcn/ui. URL: <https://ui.shadcn.com/> (дата звернення: 04.04.2026).
26. Frost B. Atomic Design. Brad Frost, 2016.
27. Getting Started | Axios Docs. Axios. URL: <https://axios-http.com/docs/intro> (дата звернення: 04.04.2026).
28. Basic usage | Zod. Zod. URL: <https://zod.dev/basics> (дата звернення: 04.04.2026).
29. Superforms - Form library for SvelteKit. Superforms - Form library for SvelteKit. URL: <https://superforms.rocks/> (дата звернення: 04.04.2026).
30. @thisux/svelte-dnd - Modern Drag and Drop for Svelte 5. @thisux/svelte-dnd. URL: <https://svelte-dnd.thisux.com/> (дата звернення: 04.04.2026).
31. Podman. Podman. URL: <https://podman.io/> (дата звернення: 04.04.2026).

ДОДАТКИ

Додаток А. Повний цикл створення нотатки

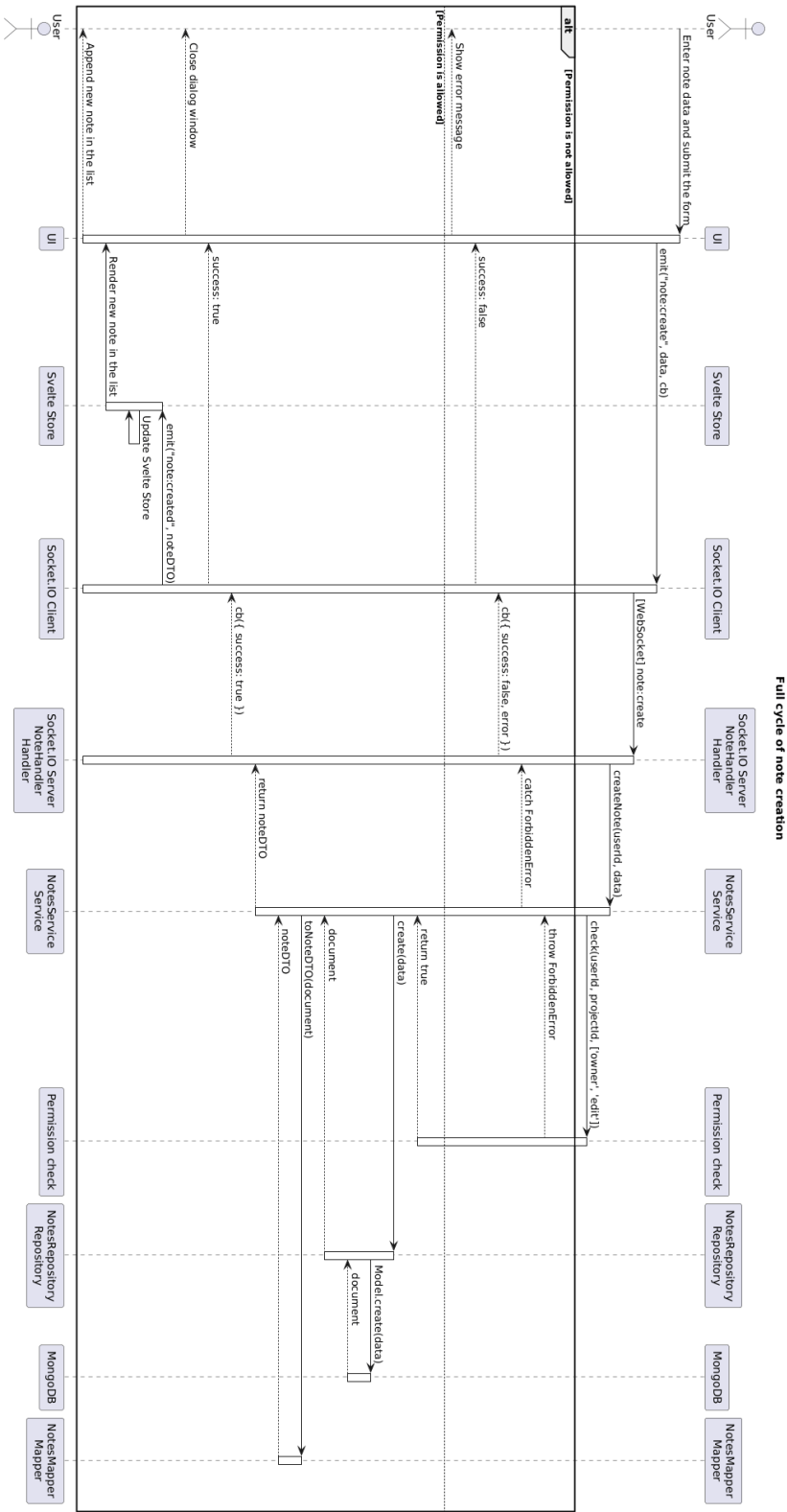


Рис. А.1. Повний цикл створення нотатки

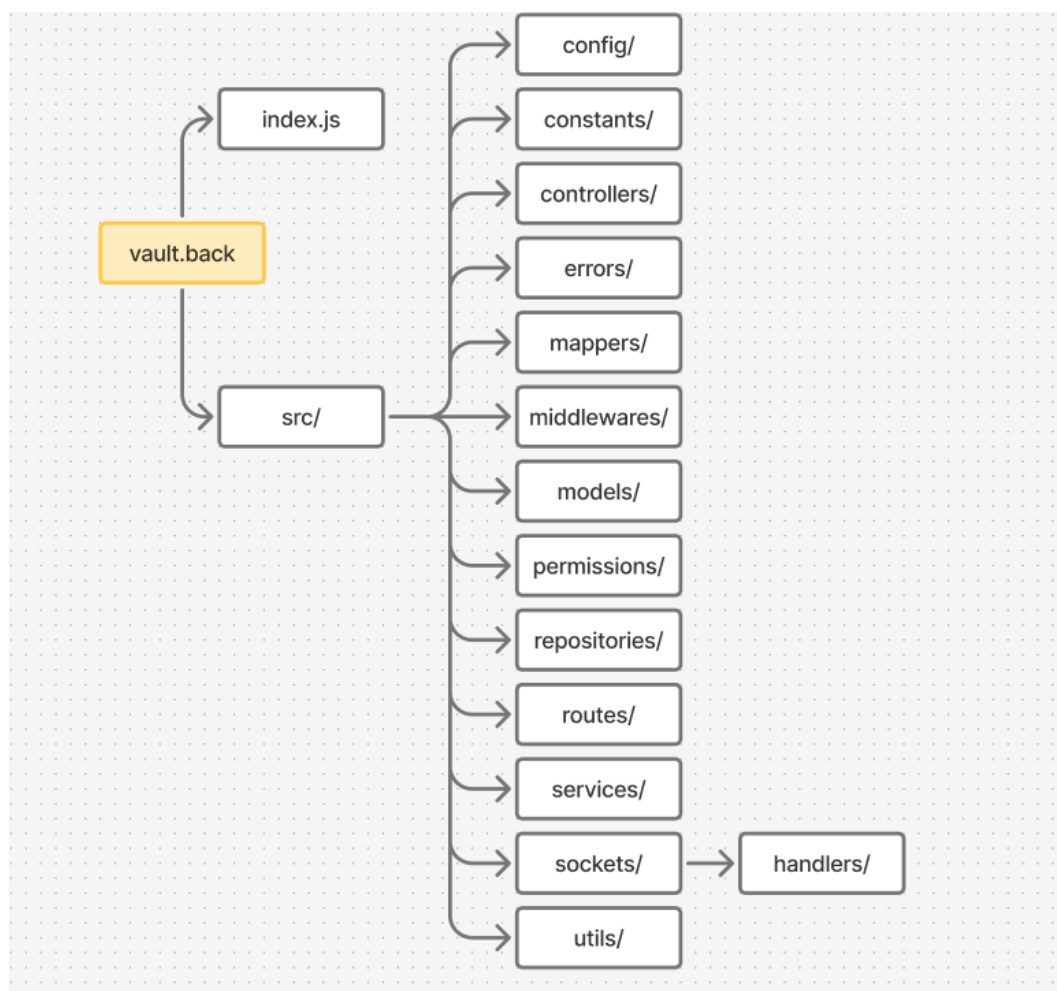
Додаток Б. Файлова структура серверної частини додатку

Рис. Б.1. Файлова структура серверної частини