МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

перший (бакалаврський)
(рівень вищої освіти)

на тему Комп'ютерна система візуалізації моніторингу проєктної діяльності

Виконав: студент 4 курсу, групи 6.1211-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Програмне забезпечення систем
(код і назва освітньої програми)
М.О. Посунько
(ініціали та прізвище)
Керівник доцент кафедри ЕІС та ПЗ
А.І. Безверхий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Дискус»
Р.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти _____перший (бакалавський)_____
Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)
Освітня програма ___Програмне забезпечення систем_____
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Тетяна КРИТСЬКА
“_01_” __березня_____2025_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

____Посуньку Микиті Олександровичу
(прізвище, ім’я, по батькові)

1.Тема роботи Комп’ютерна система візуалізації моніторингу проєктної діяльності

керівник роботи _____Безверхий Анатолій Ігорович_____
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 27.12.2024 р. № 2029-с
2. Строк подання студентом кваліфікаційної роботи_____01.06.2025__
3. Вихідні дані кваліфікаційної роботи бакалавра
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми збору та аналізу даних з веб-сторінок;
- створення програмного продукту та його опис;
- дослідження поставленої проблеми та розробка висновків.

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

6. Консультанти розділів бакалаврської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата
		Завдання прийняв

7. Дата видачі завдання 01.03.2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів бакалаврської роботи	Примітка
1	Аналіз предметної області	01.03 – 10.03.25	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	11.03 – 12.03. 25	виконано
3	Аналіз існуючих методів рішення	13.03 – 14.03.25	виконано
4	Дослідження засобів реалізації системи	15.03 – 20.03.25	виконано
5	Узгодження подальших дій з науковим керівником	21.03 – 26.03.25	виконано
6	Проєктування архітектури системи	27.03 – 28.03.25	виконано
7	Планування бази даних	29.03 – 13.04.25	виконано
8	Програмна реалізація backend	14.04 – 16.04.25	виконано
9	Програмна реалізація frontend	17.04 – 19.04.25	виконано
10	Інтеграція frontend та backend	20.04 – 01.05.25	виконано
11	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	02.05 – 17.05.25	виконано
12	Тестування системи	17.05 – 02.06.25	виконано

Студент _____ М.О. Посунько
(підпис) (прізвище та ініціали)

Керівник роботи _____ А.І. Безверхий
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок — 75

Рисунків — 23

Джерел — 33

Посунько М.О. Комп’ютерна система візуалізації моніторингу проєктної діяльності: кваліфікаційна робота бакалавра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник А. І. Безверхий. Запоріжжя : ЗНУ, 2025. 75 с.

Мета полягає у створення комп'ютерної системи для допомоги людині організувати, структурувати та візуалізувати проєктну діяльність.

У процесі розробки була розглянута проблема уваги людини, у період інформаційного перевантаження. У результаті був розроблений застосунок, створений з використання Tauri та SolidJS, а також серверна частина з використання FastAPI та PostgreSQL. Клієнтська частина забезпечує робочий простір для користувача, який повинен забезпечити усі його потреби під час планування та роботи з інформацією. Серверна частина обслуговує клієнтські запити, обробляє і зберігає дані.

Ключові слова: увага людини, ментальна втома, кросплатформовий застосунок, drag-and-drop, Class Table Inheritance, Tauri, SolidJS, FastAPI, SQLAlchemy.

ABSTRACT

Pages — 75

Drawings — 23

Sources — 33

Posunko M.O. Computer System for Visualization and Monitoring of Project Activity: Bachelor's Qualification Work in the Specialty 121 "Software Engineering" / Scientific Advisor A.I. Bezverkhyi. Zaporizhzhia: ZNU, 2025. 75 p.

The goal is to create a computer system to assist individuals in organizing, structuring, and visualizing project activities.

During development, the problem of human attention during periods of information overload was addressed. As a result, an application was developed using Tauri and SolidJS, along with a backend implemented with FastAPI and PostgreSQL. The client-side provides a workspace for the user, aimed at meeting all their needs during planning and information management. The server-side handles client requests, processes data, and ensures its storage.

Keywords: human attention, mental fatigue, cross-platform application, drag-and-drop, Class Table Inheritance, Tauri, SolidJS, FastAPI, SQLAlchemy.

ЗМІСТ

ВСТУП 8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 12
1.1 Аналіз існуючих рішень 12
1.1.1 Notion 12
1.1.2 Obsidian 14
1.1.3 ClickUp 15
1.1.4 Todoist 16
1.2 Постановка завдання 18
1.3 Висновки до розділу 1 18
2 ВИМОГИ ДО ОТОЧЕННЯ 19
2.1 Аналіз та вибір технологій для розробки 19
2.1.1 Фреймворк для клієнтської частини 19
2.1.2 Фреймворки для побудови UI 21
2.1.3 Фреймворки для серверної частини 22
2.1.4 Вибір бази даних 23
2.1.5 Допоміжні бібліотеки та інструменти 26
2.2 Вимоги до апаратного забезпечення 28
2.3 Вимоги до програмного забезпечення 28
2.4 Висновки до розділу 2 29
3 РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ 30
3.1 Загальна архітектура 30
3.2 Solidjs frontend 31
3.3 База даних 47
3.4 Серверна частина 52
3.5 Функціональні вимоги 56
3.6 Вимоги до інтерфейсу 57
3.7 Реалізація і тестування 61
3.7.1 Unit тести для тестування серверної частини 61

3.7.2 Тестування клієнтської частини 68
3.8 Висновки до розділу 3 71
ВИСНОВКИ 72
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 73

ВСТУП

Актуальність теми

Зараз у світі швидкість інформації навколо кожної людини є найбільш реактивною за всю історію людства. Всі люди, які знаходяться в інформаційному просторі, кожен день відчувають безперервний потік завдань з неоднозначними пріоритетами, проблеми, пов'язані з різними проєктами та напрямами діяльності, новини, які провокують постійні зміни та оновлення. Найбільш критично це відчувають люди, які багато часу працюють з інформацією, планують та організовують виконання персональних та\або командних проєктів, повідомляють або пояснюють інформацію іншим людям.

Увага людини впродовж дня — дуже лімітований ресурс. Усе це призводить до багатьох проблем та складностей.

Проблема пріоритезації: кожна людина, яка працює над одним або декількома проєктами, повинна тримати під контролем постійний потік задач та проблем, їх розбиття на підзадачі, групування, розподіл та пріоритезацію. Нові дані, інформація, термінові задачі та проблеми приходять з багатьох різних джерел.

Людина може знаходитись в різних умовах впродовж дня, і для кожної з умов відрізняється набір задач, який найбільш зручно та ефективно виконувати у конкретних умовах, і в той самий час пам'ятати про пріоритети. Наприклад, під час виконання конкретного та визначенного завдання потрібно максимізувати увагу тільки на речах, пов'язаних з цим завданням, і зменшити усі відволікаючі фактори, але під час пошуку нових ідей або дослідження теми краще робити навпаки: максимізувати виявлення різноманітних шматків інформації у пошуках релевантних, а не фокусуватись на чомусь одному.

Ще однією проблемою є складність “входження” нових людей в існуючий проєкт або пояснення комплексних речей. Складність полягає в тому, що людині, яка розуміється на темі та деталях, дуже складно відчути себе у стані нерозуміння цих речей. Через це пояснення вимагає складних

процесів виявлення саме тих шматків, які не розуміє людина, що розбирається, в той час як для людини, яка знає все, здається очевидним. Вирішити це можна шляхом візуалізації та схематичного пояснення, але трудоємність створення таких пояснень вимагає багато часу.

Усі намагаються оптимізувати та прискорити все навколо, не звертаючи увагу на особливості та механізми роботи людського розуму.

Створення комп'ютерної системи для вирішення усіх цих проблем буде рішенням, яке полегшить інформаційний тиск на людину впродовж дня. Полегшить процес сприйняття комплексних речей, групування, пріоритезацію та прискорить процеси розуміння та пояснення комплексних взаємопов'язаних речей.

Для контролю за прогресом проєкту та розподілом пріоритетів та порядком виконання його частин дуже сильно може допомогти добре структурована та інформативна візуалізація усіх задач, процесів, ресурсів та можливість як побачити проєкт в цілому, так і більш глибоко у конкретних гілках.

Система буде намагатись оптимізувати те, що не потребує людської уваги, та сприяти максимально комфортним умовам для людського розуму в процесах, які потребують людської уваги.

Мета дослідження

Створення комп'ютерної системи для допомоги людині організувати, структурувати та візуалізувати проєктну діяльність.

Завдання дослідження

Проаналізувати існуючі рішення, їх переваги та недоліки. Обрати найбільш оптимальний tech stack для розробки цієї системи. Дослідити алгоритми AI, які допоможуть зробити процесс людської діяльності найбільш комфортним та ефективним.

Об’єкт дослідження

Об'єктом дослідження є процес аналізу та виявлення складностей під час проєктної діяльності.

Предмет дослідження

Предметом дослідження є створення комп’ютерної системи для візуалізації моніторингу діяльності як для персональних, так і для командних проєктів, яка потенційно полегшить та оптимізує складності під час роботи над проєктами.

Методи дослідження

Проведення огляду та аналізу літературних джерел, книг, популярних найкращих практик щодо максимізації ефективності людської та командної діяльності, а також факторів, які погіршують цю ефективність; аналіз та випробування існуючих рішень, сфокусованих на цій проблемі, з метою виявлення проблемних та вдалих підходів, а також функціоналу, якого може не вистачати для найкращої ефективності.

Практичне значення одержаних результатів

1. Практично здобуті знання про фреймворки, бібліотеки, та підходи, які найкраще підійдуть для розробки подібних систем.
2. Застосунок буде корисним як для власного використання (з можливістю швидкої адаптації під потреби), так і для подальшого впровадження при повному доопрацюванні.

Глосарій

Гіперфокус (англ. Hyperfocus) — стан інтенсивної й тривалої концентрації уваги на одній справі, під час якого людина максимально ефективно виконує цю справу.

Скаттерфокус (англ. Scatterfocus) — навмисна розсіяна увага, коли думки вільно блукають без конкретного фокусу, що допомагає мозку відпочити й створити сприятливий простір для ідей.

dnd (drag-and-drop) — поведінка інтерфейсу, яка дозволяє користувачу переміщати об'єкти на екрані шляхом натискання і перетягування курсором.

fine-grained reactivity — модель реактивності, де оновлюються лише потрібні частини UI, завдяки відстеженню залежностей у використаних реактивних утилітах.

DOM (англ. Document Object Model) — ієрархічне представлення HTML/XML-документа, що забезпечує інтерфейс для доступу та модифікацій через скрипти (наприклад, JavaScript).

віртуальна DOM (англ. Virtual DOM) — віртуальна копія DOM в оперативній пам'яті.

ASGI (англ. Asynchronous Server Gateway Interface) — асинхронний аналог WSGI — специфікація для зв’язку Python-серверів із фреймворками й додатками, що підтримують як синхронні, так і асинхронні операції, включно з WebSocket.

masterless architecture — архітектура розподілених систем без виділеного головного вузла; кожен вузол рівноправний і може обробляти запити, що підвищує відмовостійкість і масштабованість.

ACID — акронім, що позначає чотири властивості транзакцій у СУБД: Atomicity (атомарність), Consistency (узгодженість), Isolation (ізоляція) і Durability (стійкість).

WAL (англ. Write-Ahead Log) — техніка, де всі зміни спочатку записуються у додатковий журнал, а вже потім застосовуються до бази даних; забезпечує атомарність і стійкість, підтримуючи ACID.

HMR (англ. Hot Module Replacement) — механізм, що дозволяє під час розробки обмінювати, додавати або змінювати модулі в вже запущеному додатку без повного перезапуску або перезавантаження сторінки, зберігаючи стан програми.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз існуючих рішень

В цьому розділі буде розглянуто чотири популярні системи, які надають схожі рішення.

1.1.1 Notion

На рисунку 1 представлено головне меню застосунка Notion.

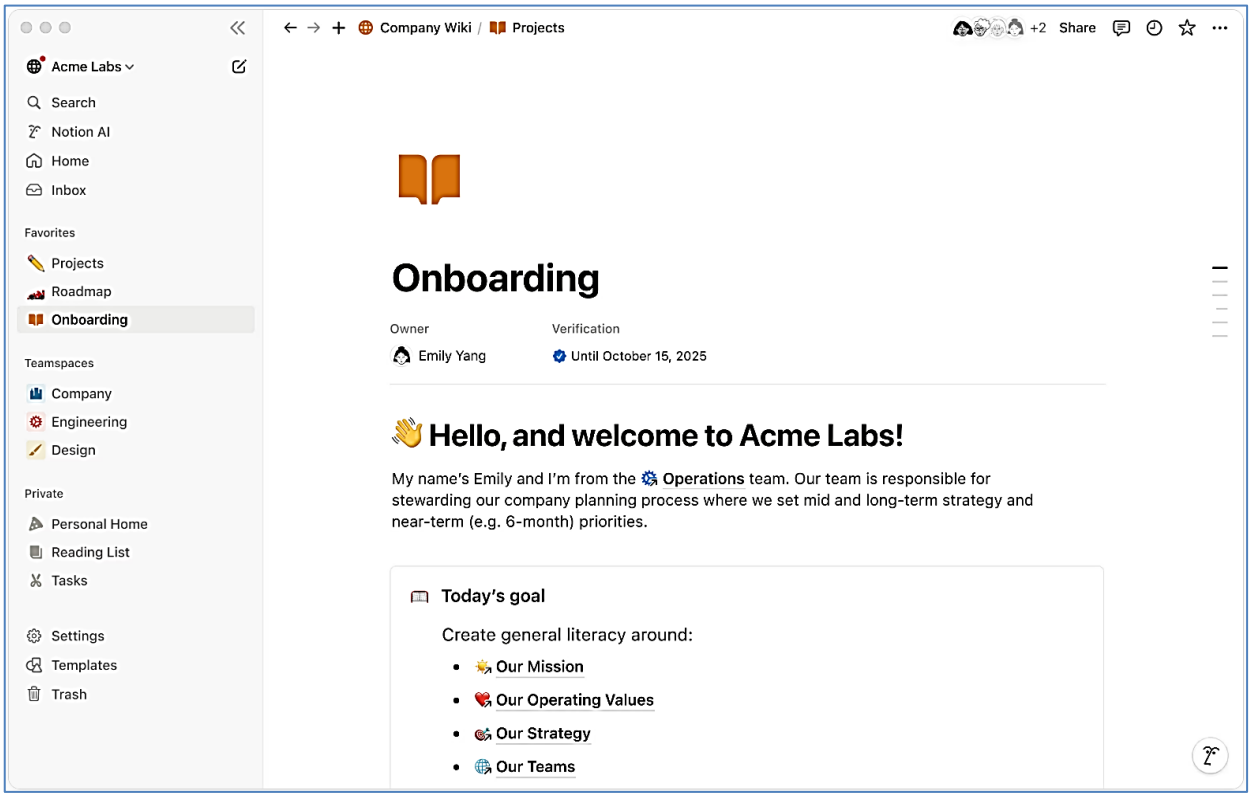


Рисунок *1 — Головне меню Notion*

Notion — дуже популярний гнучкий кросплатформовий інструмент, який можна використовувати як для власних потреб, так і для командної роботи. Він може стати заміною для Google Docs, Trello та Google Drive. Більшість функціональності входить у безкоштовний план, але для повного доступу до всіх AI-функцій та поліпшених інструментів для команд є платні

плани. Система продовжує розвиватись та зростати, Notion нещодавно створили додатковий застосунок Notion Calendar, а також анонсують Notion Mail.

Notion надає наступні можливості:

- Зберігання інформації у вигляді ієрархічних сторінок та блоків (текст, задача, список, медіа та інше), які можна редагувати та впорядковувати.
- Підтримка баз даних з різними видами відображення (таблиця, таймлайн і т.д.) та власними полями.
- Можливість створити тригери та не дуже складні автоматизації.
- Підтримка дуже великої кількості інтеграцій з різними системами та спеціальні блоки для того, щоб додавати їх на сторінки.
- Можливість зберегти сторінку або проєкт як Wiki.
- Можливість створювати шаблони для різних потреб, а також велика кількість шаблонів як від розробників, так і від ком'юніті.
- Підтримка декількох робочих пристроїв (персональних або командних).
- Співпраця в реальному часі між користувачами, налаштування прав доступу та ролей.
- Підтримка можливості публікації сторінки як сайту в один клік.
- Активне впровадження різноманітних AI інструментів.

Недоліки:

- Відсутність деяких типів блоків, наприклад, таких як Toggle Task для більш тонкого управління завданнями.
- Невелика можливість кастомізації інтерфейсу, часткова модульність.
- Немає підтримки відображення у вигляді інтерактивної дошки.
- Якщо мати дуже велику ієрархію сторінок, може бути складно швидко знайти потрібну інформацію.
- Відсутня можливість тегувати блоки, для гнучкої централізації блоків з різних сторінок.

- Почати роботу та налаштувати робочий простір для себе може бути складно людям, які не знайомі з таком застосунків, або не мають навичок планування та організації.
- Немає режиму фокусу, у якому користувач зможе приховати усе, що не стосується поточного завдання.
- Не оптимізований та не дуже зручний мобільний застосунок.

1.1.2 Obsidian

На рисунку 2 представлено головне меню застосунка Obsidian.

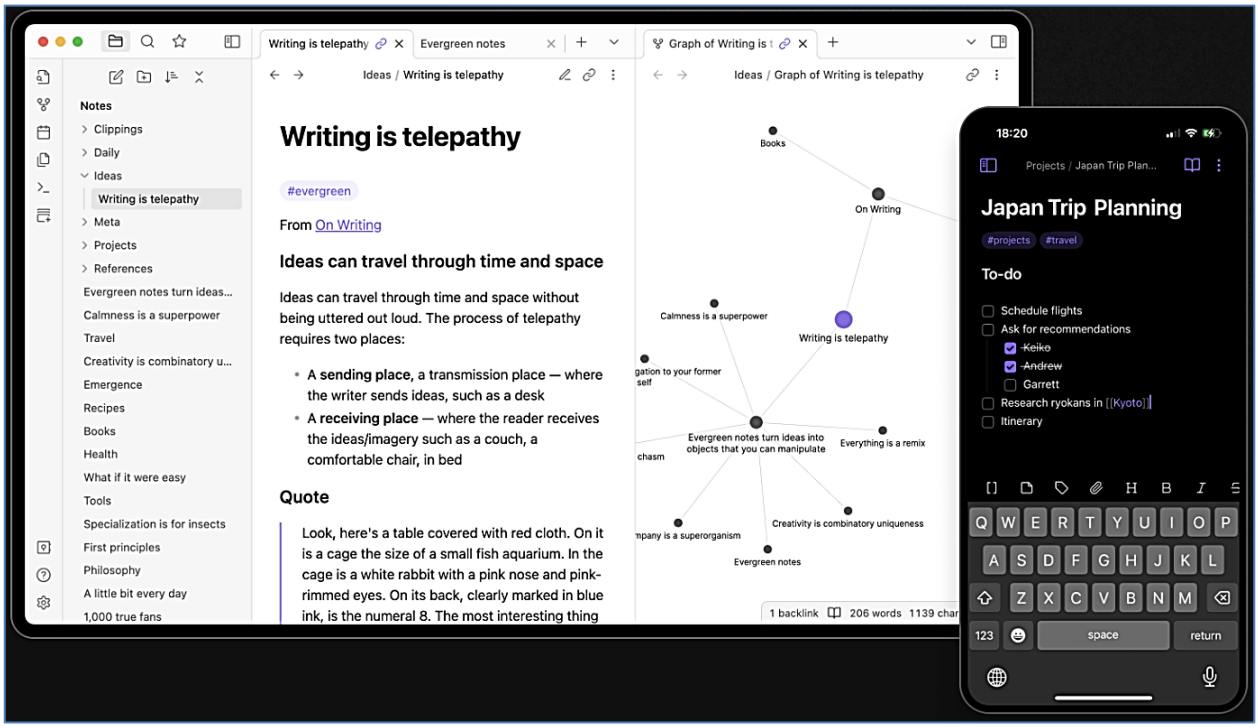


Рисунок *2 — Головне меню Obsidian*

Obsidian – це кросплатформовий текстовий редактор для створення зв’язаних нотаток, орієнтований на персональне використання. Інформація зберігається у форматі Markdown, що забезпечує різноманітне форматування тексту, а також імпорт/експорт у цьому форматі. Ключовою особливістю Obsidian є можливість створення зв’язків між нотатками, що дозволяє будувати графи знань та візуалізувати їх.

Obsidian надає наступні можливості:

- Локальне зберігання даних.
- Велика кількість плагінів для розширення функціоналу.
- Модульність та кастомізація інтерфейсу.
- Підтримка інтерактивної дошки.
- Платна підписка надає хмарне зберігання даних та синхронізацію між пристроями.

Недоліки:

- Проєктна діяльність не є основною функцією.
- Немає вбудованих засобів управління завданнями і таск-менеджментом.
- Whiteboard має тільки базові функції.

1.1.3 ClickUp

ClickUp — це багатофункціональна (“all-in-one”) платформа для управління завданнями та проєктами, яка орієнтована на командну роботу. Розробники системи намагаються зібрати функціональність для усіх типів завдань в одному місці для централізації та зменшення перемикання між додатками. ClickUp пропонує широкий набір інструментів та об’єднує в одному середовищі діаграми, календарі, тайм-трекер та внутрішню комунікацію.

ClickUp надає наступні можливості:

- Поєднує функції Notion, Google Docs, Miro board, Dropbox, Trello, Loom та інші. Це може зберегти бюджет компанії – одна підписка замість багатьох.
- Має велику кількість шаблонів та типів робочих просторів для організації будь яких типів проєктів, будь яких типів компаній (Enterprise, Startup, Non-Profit) та відділів (Development, Marketing, Sales).
- Комплексне управління проєктами в одному інтерфейсі та миттєві повідомлення про зміни відповідним учасникам проєктів.

- Можливості автоматизації процесів та інтеграції з багатьма сервісами.

На рисунку 3 представлено функціональність платформи ClickUp.

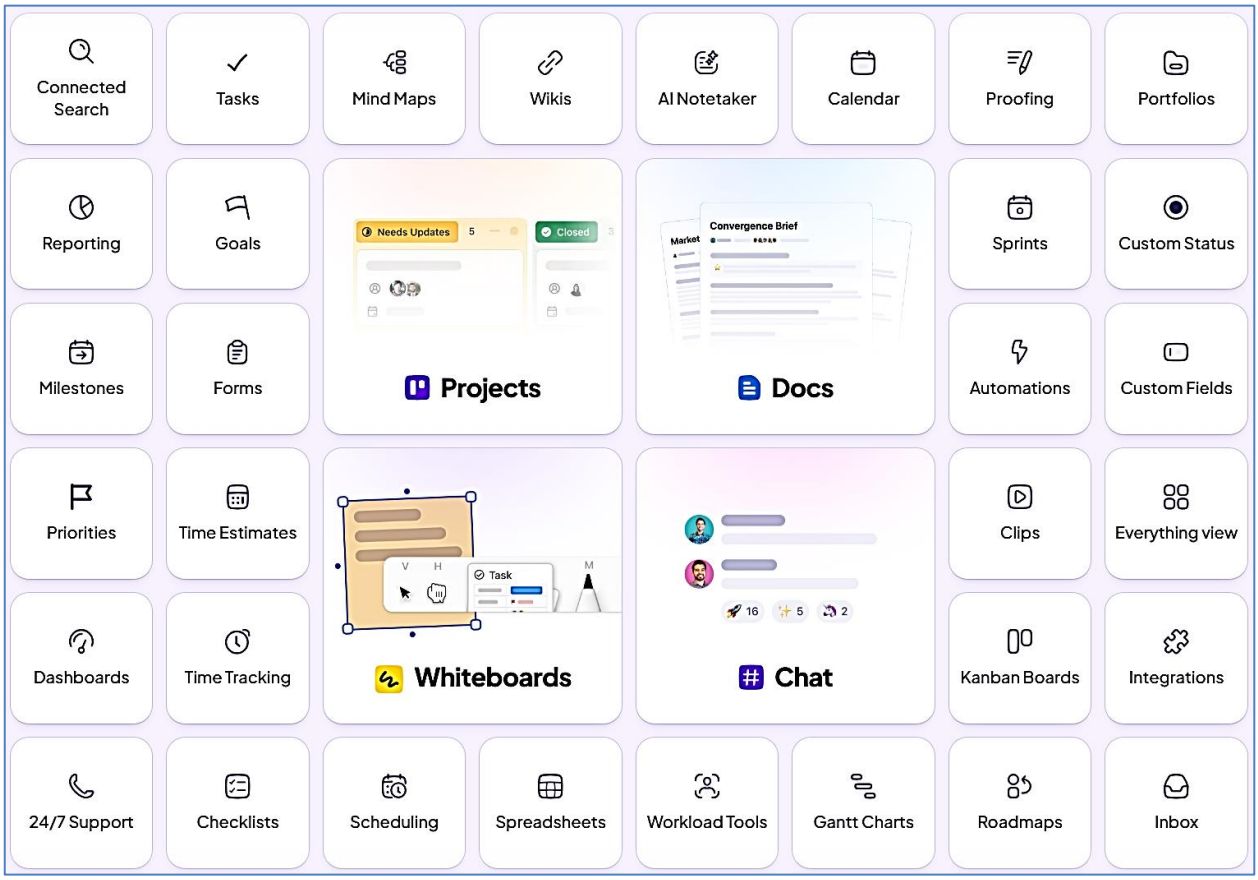


Рисунок *3 — Перелік функцій ClickUp*

Недоліки:

- Не найкращий варіант для персональних проєктів.
- Можливість інформаційного перевантаження через комплексний інтерфейс та велику кількість функцій.
- Повний функціонал потребує підписки.

1.1.4 Todoist

Todoist — це mobile-first мінімалістичний застосунок для управління завданнями, який підходить як для особистого, так і для командного використання.

На рисунку 4 представлено головне меню застосунка Todoist.

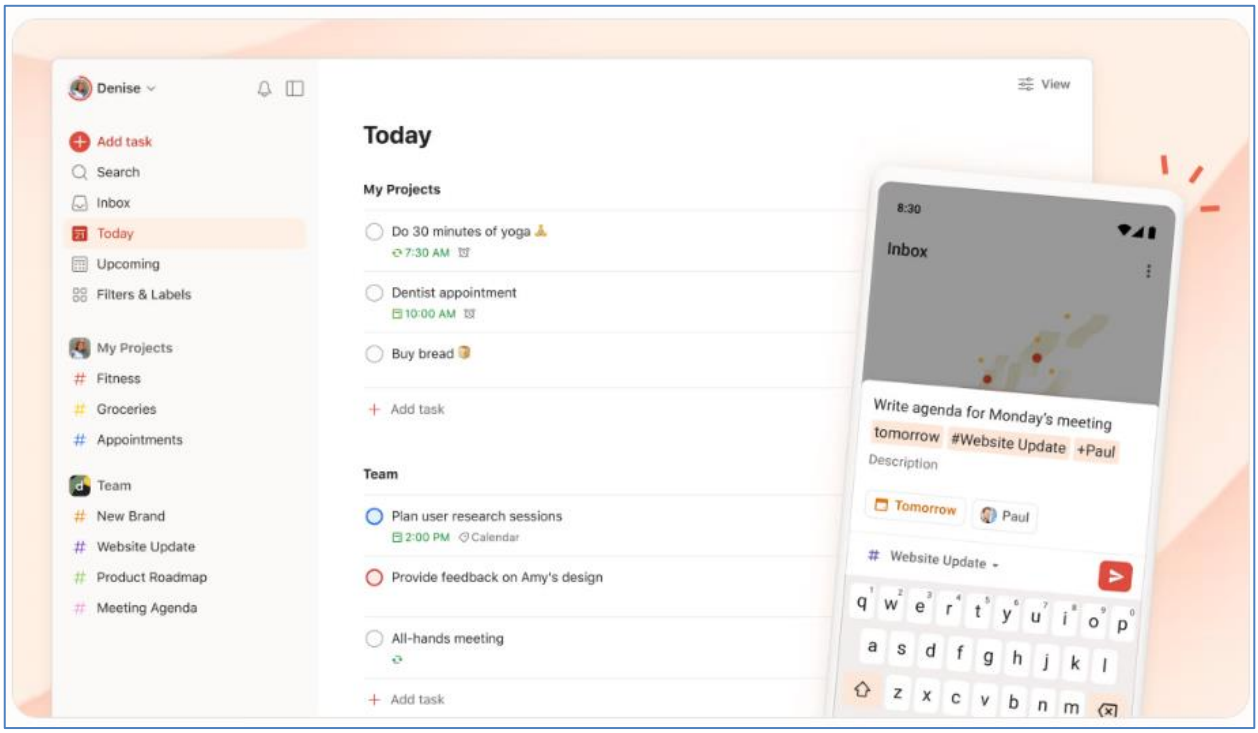


Рисунок *4 — Головне меню Todoist*

Todoist надає наступні можливості:

- Мінімалістичний дизайн, що знижує відволікання.
- Зручне використання системи тегування для категоризації.
- Встановлення дедлайнів, повторюваних завдань, нагадування про час.

Недоліки:

- Обмежені можливості для керування комплексними проєктами.
- Немає таких можливостей як “задача повинна відображатись після виконання попередньої”, які б ще більше зменшили відволікання.
- Обмежена можливість кастомізації інтерфейсу.

1.2 Постановка завдання

Мета – розробити систему, яка візьме найкраще від усіх конкурентів: зручне ієрархічне структурування інформації (Notion), взаємопов'язаність різних компонентів та візуалізацію (Obsidian), поєднання різних функцій в одному застосунку (ClickUp), мінімалізм та пріоритезацію (Todoist). Постановка завдання включає наступні етапи:

1. Аналіз потреб і вимог: дослідити існуючі системи, їх переваги та недоліки. Проаналізувати особливості уваги людини. Визначити, яким повинен бути застосунок, щоб він був найбільш комфортним для користувачів.
2. Вибір технологій: обрати найбільш вдалий технологічний стек для розробки, проаналізувавши існуючі фреймворки, технології та бібліотеки.
3. Розробка архітектури: планування структури проєкту, бази даних, інтерфейсу. Узгодження частин архітектури (взаємодія клієнтської та серверної частин). Врахування потенційного розширення, масштабування та гнучкості системи.
4. Реалізація функціоналу: розробка можливостей, які повинна надавати система.

1.3 Висновки до розділу 1

1. Було проведено аналіз предметної області, розглянуто популярні системи для організації інформації та управління завданнями.
2. Недоліки розглянутих систем вказують на те, що потрібно врахувати у системі, яка розробляється.
3. Визначена мета та постановка завдання.

2 ВИМОГИ ДО ОТОЧЕННЯ

2.1 Аналіз та вибір технологій для розробки

Вибір інструментів для розробки є одним з найважливіших етапів. У цьому розділі виконано аналіз та порівняння технологій. Обрані інструменти повинні дати змогу розробити застосунок з наступними критеріями:

- Кросплатформовість: люди працюють на різних пристроях та різних операційних системах, і застосунок повинен враховувати це та надавати можливість виконувати потрібні дії так, як зручно для людини.
- Гнучкість та масштабування архітектури: система повинно легко розширюватись та витримувати збільшення навантаження, функціональність повинна бути модульно згрупована і добре структурована.
- Модульність та гнучкість інтерфейсу: інтерфейс повинен бути зручним, динамічним та адаптивним, давати змогу користувачу змінювати та коригувати його під власні потреби.
- Мінімізація down-time та цілісність даних: технологічний стек повинен дозволяти організовувати безперервну роботу застосунку, резервні копіювання, захист від втрати даних під час збоїв.
- Зручність розробки та підтримки застосунку: всі інструменти повинні бути документовані та підтримувані. Також буде плюсом активна спільнота розробників та різні інтеграції.

2.1.1 Фреймворк для клієнтської частини

Серед доступних рішень досліджувались фреймворк Qt 6, біліотеки Dear pygui (python), Nuclear (C++) та веб-орієнтовані підходи (Electron, Tauri). Qt 6 [8] забезпечує найбагатший набір готових віджетів і стабільне кросплатформне середовище, проте комерційна ліцензія значно збільшує вартість проєкту. Dear pygui [6] та Nuclear [7] – цікаві варіанти клієнтської

частини, але вони мають середню популярність, меншу універсальність, ніж веб-технології, кращу продуктивність ціною довшого часу розробки. Electron [9] і Tauri [3] – фреймворки, які використовують веб-технології для інтерфейсу користувача, що дає ще одну перевагу запускати застосунки у будь якому браузері з мінімальними змінами у коді. Electron був розроблений раніше і використовує підхід вбудовування веб-переглядача під час білду застосунку. Через це розміри застосунків, написаних на Electron, великі, а споживання ресурсів не найбільш оптимальне. У свою чергу, Tauri використовує системний веб-переглядач, що дозволяє створювати застосунки з використанням будь-яких веб-технологій для UI, а також Rust шар для звертання до системного API. Така комбінація забезпечує невеликий розмір кінцевого пакету, високу швидкодію та кращу безпеку завдяки строгій системі прав доступу Rust. Враховуючи всі критерії, **Tauri** було обрано як оптимальний фреймворк для клієнтської частини (див. рисунок 5).

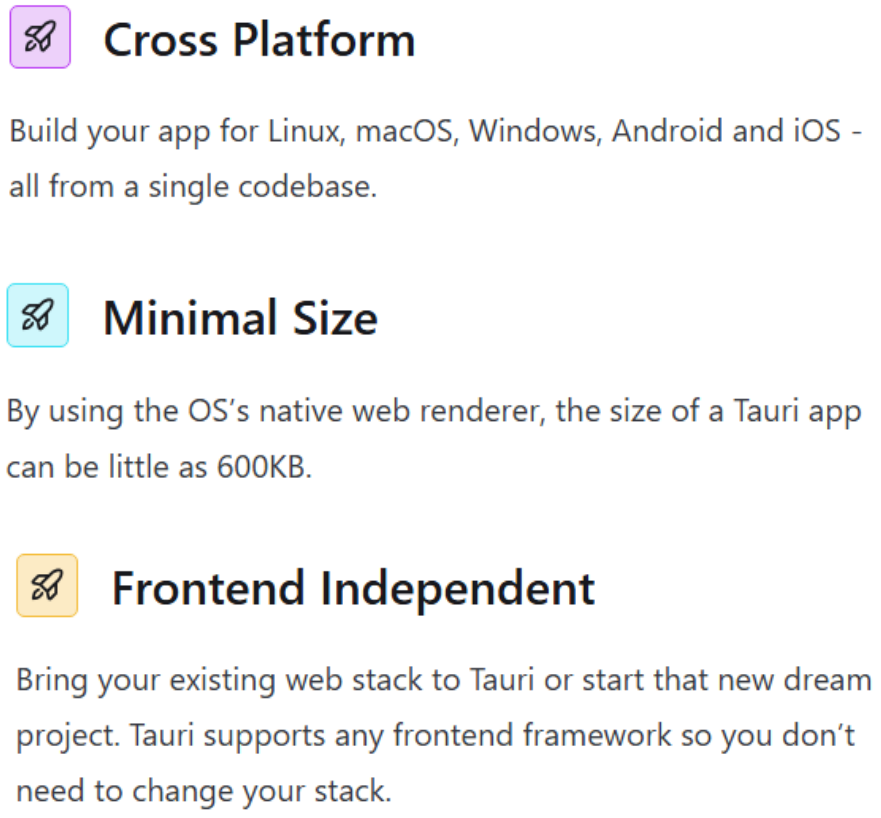


Рисунок *5 — Переваги фреймворку Tauri*

2.1.2 Фреймворки для побудови UI

Tauri надає можливість розробляти UI, використовуючи веб-технології, отже треба вибрати, що саме ми будемо використовувати. Розглянемо три фреймворки для побудови реактивного інтерфейсу: React, Vue і SolidJS.

React [10] має найширше ком'юніті розробників та є найпопулярнішим, має багатий набір плагінів і добре відпрацьовані практики керування станом, але вимагає оптимізацій віртуальної DOM і часто призводить до зайвих оновлень інтерфейсу. React можна використати для швидкого написання простих та різноманітних інтерфейсів, але під час написання більш комплексного динамічного та інтерактивного інтерфейсу може страждати продуктивність.

Vue [11] – ще один популярний UI фреймворк. Він має кращі показники продуктивності, проте оптимізація реактивності виконується вбудовано, і в складних додатках вона також може генерувати зайві перерахунки.

SolidJS [5] використовує fine-grained реактивність, яка дозволяє оновлювати лише ті частини UI, де фактично змінилися дані, також SolidJS не використовує віртуальну DOM. Це призводить до майже «vanilla» JS рівня продуктивності та знижує кількість обчислень на оновлення великої кількості компонентів. Оскільки наша система передбачає динамічний та інтерактивний інтерфейс, різні види елементів та компонентів, роботу з інтерактивним канвасом – використання SolidJS зможе забезпечити найкращу продуктивність нашого застосунку.

На рисунку 6 представлено метрики для оцінки продуктивності різних фреймворків.

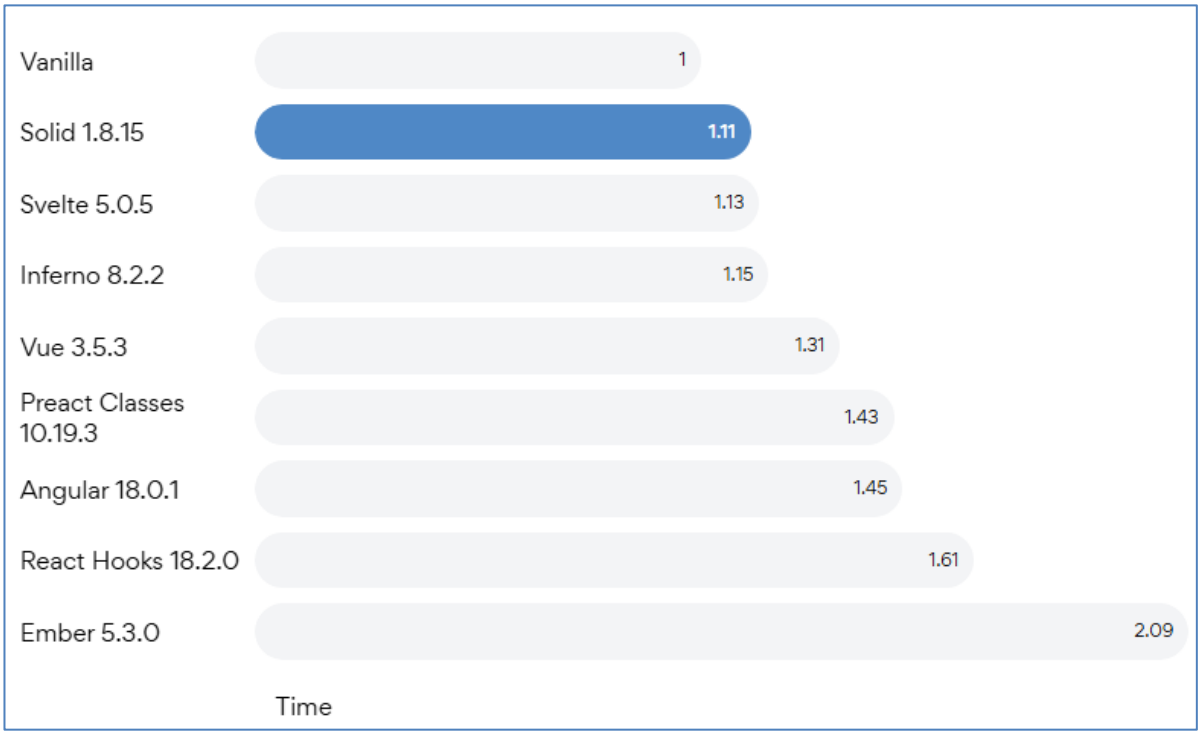


Рисунок *6 — Метрики продуктивності SolidJS порівняно з іншими фреймворками*

2.1.3 Фреймворки для серверної частини

На стороні бекенду було порівняно Laravel, ExpressJS та FastAPI. Laravel [12] є одним із найпопулярніших PHP фреймворків завдяки фокусу на комфортності розробки та широким можливостям. Інструмент надає вбудовані функції для швидкої реалізації поширених потреб, таких як авторизація, кешування, але має менш гнучку архітектуру. ExpressJS [13] є популярним рішенням для створення бекенду на JavaScript, що працює поверх Node.js [14]. Інструмент забезпечує гнучкість, однак потребує додаткових дій та бібліотек для побудови структури проєкту та валідації. **FastAPI** [4] на базі Python поєднує асинхронний веб-фреймворк Starlette [15] та швидкий ASGI-сервер Uvicorn [16], автоматично генерує інтерактивну Swagger UI [17] документацію (рисунок 7) і дозволяє легко інтегрувати AI-моделі, написані на Python. Додатковою перевагою є валідація даних на рівні опису Pydantic моделей. Через високу швидкість розробки REST API та всі вище перераховані переваги було обрано FastAPI для серверної частини.

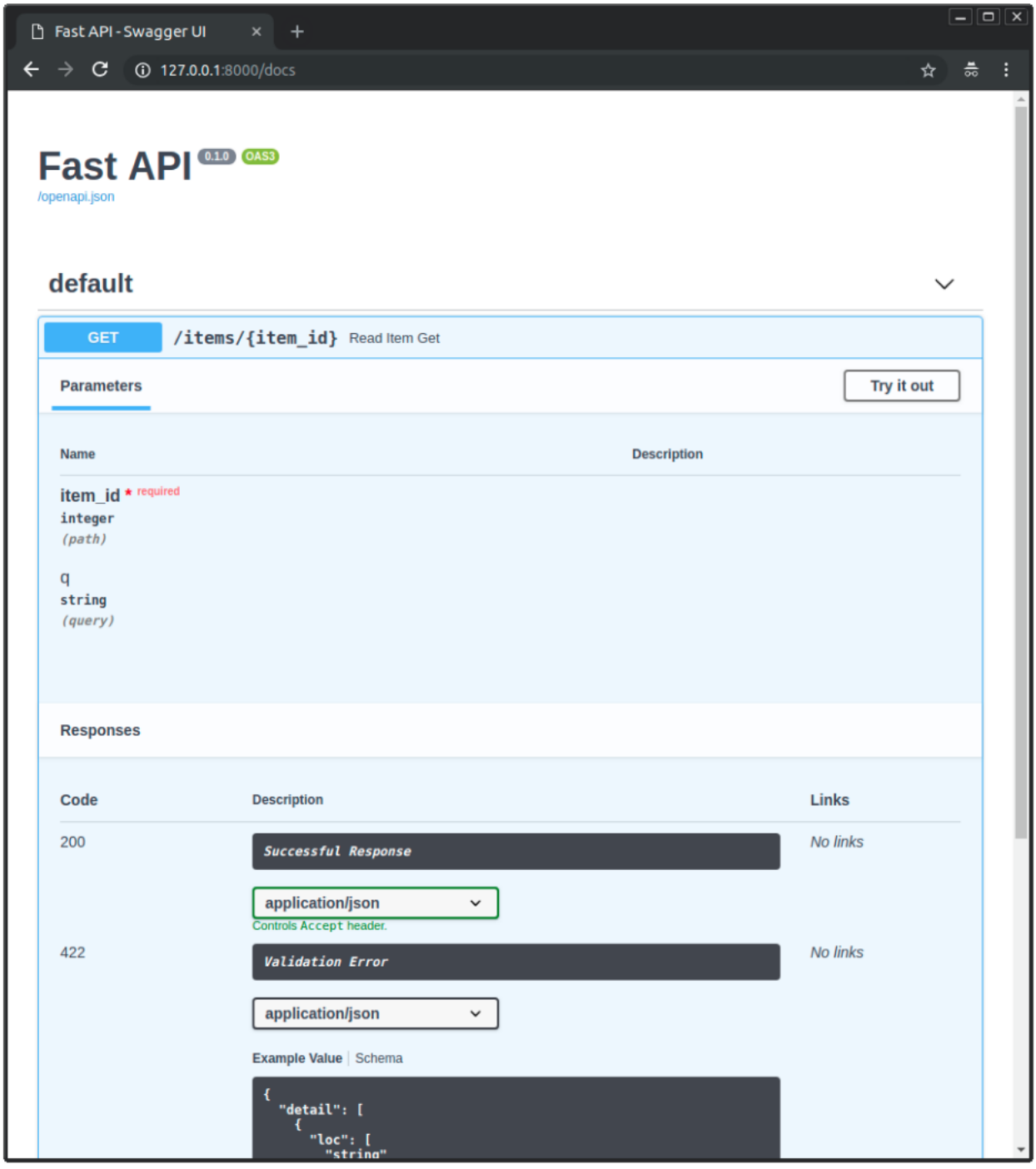


Рисунок *7 — Приклад згенерованої Swagger UI докумнтації*

2.1.4 Вибір бази даних

Під час оцінки сховищ даних були розглянуті чотири категорії СУБД:

- Графові: Neo4j, JanusGraph, OrientDB, Dgraph.
- NoSQL: Cassandra, MongoDB.
- Реляційні: PostgreSQL, MySQL.

Усі варіанти оцінювалися за критеріями надійності, масштабованості, стійкості до відмов, зручності використання та управління, вартості ліцензії, швидкості виконання запитів, а також наявності систем, які успішно використовують БД.

Графові бази зручні для складних зв'язків між сутностями та швидких запитів типу "сусідні вузли", або які зводяться до пошуків у графі. Більшість відкритих БД такого типу не дуже популярні і не мають широкого розповсюдження, тому багато чого доведеться налаштовувати та ретельно тестувати, що може зайняти певний час. Але є і популярні варіанти, наприклад, Dgraph. Ця БД дозволяє масштабування, порівняне з NoSQL, і в той же час виконання транзакцій та запитів як у реляційних базах. Стабільні продакшн-системи, наприклад, Neo4j чи ArangoDB, у безкоштовній (GPL-подібній) версії не надають функціоналу більше, ніж для навчальних проєктів, а вартість комерційної ліцензії виправдовує себе лише в дуже специфічних випадках або для великих систем, яким потрібна оптимізація. В подальшому можна розглянути **Dgraph [20]** для додаткових можливостей або функцій, але як основну БД краще вибрати більш надійний та універсальний варіант.

Apache Cassandra [19] — це розподілена NoSQL база даних, спроєктована для роботи з великими обсягами даних у режимах високого навантаження. Вона орієнтована на горизонтальне масштабування та забезпечення відмовостійкості в розподілених системах. Cassandra працює без централізованого керівного вузла (masterless architecture), усі вузли в кластері рівноправні, а додавання або вихід з ладу вузлів виконується дуже легко. В противагу перевагам ця база має такі недоліки, як високі вимоги до ресурсів, складність запитів, потреба в детально запланованому дизайні моделі даних. Цю базу використовують такі компанії, як Netflix — для зберігання метаданих та історії переглядів, Apple — для метаданих для файлів та резервних копій iCloud і Siri, Instagram — для зберігання даних користувачів, таких як лайки, коментарі та пости.

MongoDB – ще одна NoSQL документно-орієнтована база, що дозволяє зберігати дані у вигляді JSON або BSON. Так само, як і Cassandra, підтримує горизонтальне масштабування та реплікацію для високої доступності та відмовостійкості. Має вбудовану підтримку гео-даних, що може бути корисним в деяких випадках. Цю базу використовують такі компанії, як Uber,– для зберігання даних про маршрути, замовлення і клієнтів, Coinbase – обробка запитів у реальному часі для забезпечення швидких фінансових операцій.

Такі бази даних, як Apache Cassandra або MongoDB, скоріше використовуються як покращення або оптимізація вже працюючих систем або для систем, де немає багато взаємопов'язаних даних. Тому це – не найкращий варіант для нашої системи.

Далі розглянемо реляційні бази.

MySQL є однією з найпоширеніших реляційних СУБД, відомою своєю швидкістю виконання простих транзакцій та широкою підтримко. Завдяки архітектурі Master–Slave і індексуванню InnoDB MySQL забезпечує високу пропускну здатність для операцій читання і запису, що робить її популярним вибором для веб додатків із великою кількістю запитів. Проте в порівнянні з PostgreSQL MySQL має трохи менше можливостей. Популярні компанії, які використовують MySQL: YouTube, GitHub, Boeing, Spotify.

PostgreSQL [18] було обрано завдяки збалансованості між транзакційною стійкістю (ACID), продуктивністю для складних SQL-запитів та підтримкою розширень.

- Масштабування і відмовостійкість.
    - Citus: розширення, що забезпечує горизонтальний шардінг таблиць і розподілені запити, дозволяючи збільшувати ємність кластеру шляхом додавання нових вузлів.
    - Patroni: система управління кластерами з високою доступністю, що автоматично переключає лідера в разі збою та підтримує синхронну реплікацію.

    - Streaming replication: вбудована функція PostgreSQL для трансляції WAL-журналів на репліки, що забезпечує баланс читання/запису та швидке відновлення після збоїв.
- Аналітика та пошук.
    - Вбудований повнотекстовий пошук на основі tsvector і tsquery.
    - Розширення pg_trgm для нечіткого пошуку й індексації за подібністю рядків.
    - pgvector для векторного пошуку в СУБД.
- Гнучке зберігання даних.
    - Тип JSONB дозволяє зберігати напівструктуровані документи з індексацією полів.
    - Розширення PostGIS для роботи з геоданими.
    - Можливість використовувати ltree для організації ієрархічних структур.

PostgreSQL використовують Microsoft, Red Hat, Reddit.

Для доступу з Python використовуємо бібліотеку **SQLAlchemy**, яка має 2 API: Core та ORM. SQLAlchemy Core дозволяє програмно будувати SQL-запити через типізовані класи, водночас зберігаючи можливість низькорівневих запитів. Для керування міграціями використаємо бібліотеку **Alembic**, яка працює з SQLAlchemy.

Таким чином, PostgreSQL забезпечує необхідну гнучкість, надійність, потенційну масштабованість і широкий функціонал без додаткових витрат на комерційне програмне забезпечення.

2.1.5 Допоміжні бібліотеки та інструменти

Drag-and-drop. Розглянуті бібліотеки solid-dnd і dragula; обрано бібліотеку Pragmatic dnd [21] від розробників компанії Atlassian, яка має найбільше можливостей та найкращу продуктивність.

Стилізація. Розглянуті Tailwind CSS і Emotion; обрано Emotion через гнучкість CSS-in-JS.

Бібліотеки для візуалізації.

- Canvas API. Нативний інструмент для створення графіки в браузері. Забезпечує високу продуктивність, але вимагає ручної реалізації базових функцій, таких як обробка подій і управління об'єктами.
- ECharts [23]. Бібліотека для створення діаграм і графіків із підтримкою інтерактивності. Потужна для аналітичних візуалізацій, але не підходить для створення кастомних інтерактивних елементів.
- D3.js [24]. Бібліотека для побудови складних візуалізацій даних із гнучким API. Дає максимальний контроль, але не призначена для роботи зі сценами або канвасом.
- Cytoscape.js. Спеціалізована бібліотека для роботи з графами та мережами. Чудова для графів, але не універсальна для кастомних елементів чи інших типів візуалізацій.
- Fabric.js. Бібліотека для роботи з Canvas, яка додає зручності для малювання та трансформації елементів. Простий API, але менш продуктивна для складних сцен із багатьма елементами.
- KonvaJS [22]. Бібліотека для створення інтерактивних графічних сцен на Canvas. Простий API для роботи з шарами і подіями. Висока продуктивність для інтерактивної графіки. Зручно створювати динамічні сцени із можливістю роботи з drag-and-drop. Легко інтегрується з іншими бібліотеками, такими як D3.js, для складних візуалізацій.

Обрано бібліотеку KonvaJS, D3.js та ECharts можуть бути додатково використані для динамічних візуалізацій даних або аналітики, якщо потрібно буде додати графіки чи інші специфічні візуалізації.

HTTP-клієнт і збірка. Використання Axios гарантує зручні запити до REST-API, а Vite забезпечує швидкий HMR та ефективний production-бандл.

Backend: asyncpg – PostgreSQL Database Client бібліотека для Python/asyncio.

Усе перераховане в поєднанні створює надійну, масштабовану та продуктивну архітектуру, здатну задовольнити вимоги системи організації інформації та завдань.

2.2 Вимоги до апаратного забезпечення

Розроблений застосунок може бути використаний на будь-якій системі (за умови доопрацювання деяких специфічних частин), яка має веб переглядач, тобто практично на будь-якій сучасній ОС. Як мінімум, можна назвати наступні вимоги:

- будь-яка ОС із встановленим сучасним веб-переглядачем;
- мінімум 2 GB оперативної пам’яті.

Сервер, який хостить бекенд застосунку, повинен забезпечити достатню кількість оперативної та дискової пам’яті в залежності від навантаження, а також від кількості даних, на які буде розрахована БД. Як мінімум, можна назвати наступні вимоги:

- мінімум 4 GB оперативної пам’яті;
- мінімум 20 GB вільного дискового простору;
- мінімум 10 Мбіт/с швидкість інтернет підключення.

2.3 Вимоги до програмного забезпечення

Нижче наведено мінімальний набір ПЗ, яке повинно бути встановлене на сервері для стабільної роботи бекенд-застосунку (FastAPI + SQLAlchemy + PostgreSQL):

- Python 3.11+.
- FastAPI.
- SQLAlchemy 2.0+.

- PostgreSQL 14+.

2.4 Висновки до розділу 2

1. Проведено аналіз та обрані технології та бібліотеки для розробки бази даних, клієнтської частини, серверної частини.
2. Розглянуті загальні вимоги до апаратного та програмного забезпечення для клієнтського застосунку, а також до сервера.
3. Вказані мінімальні параметри, що є достатніми для забезпечення роботи системи.

3 РОЗРОБКА ПРОГРАМНОЇ СИСТЕМИ

3.1 Загальна архітектура

Архітектура системи складається з двох частин: застосунок, який встановлює користувач або звертається через браузер, та сервер, який обробляє запити та виконує збереження, видалення та зміну даних у БД (див. рисунок 8).

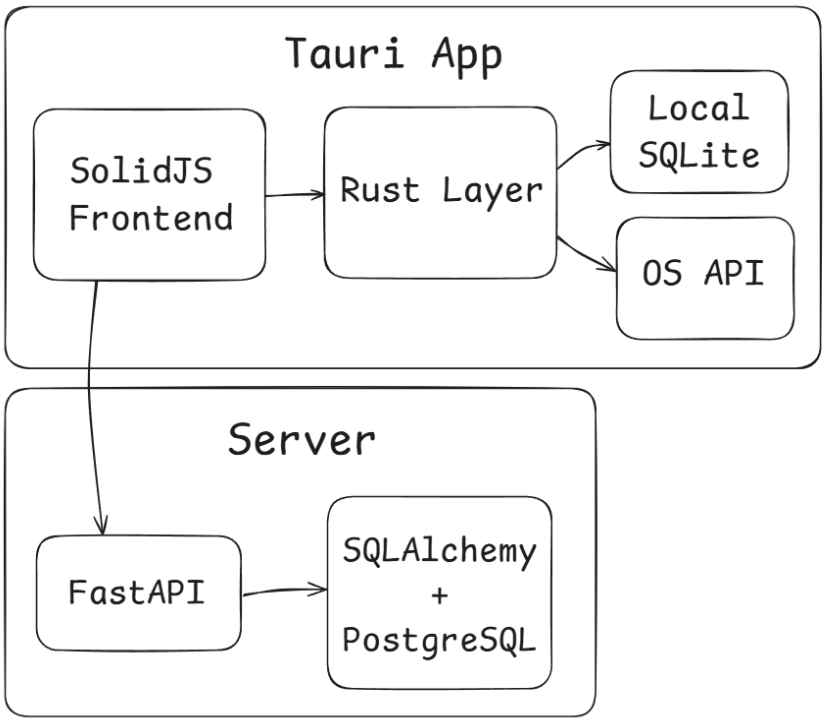


Рисунок 8 — *Загальна архітектура системи*

Застосунок складається з SolidJS Frontend частини, яка відповідає за інтерфейс користувача. Також frontend може звертатися до Rust шару, який може забезпечити доступ до API операційної системи, на якій встановлено додаток, або виконувати обробку запитів до локальної БД, якщо застосунок повинен підтримувати офлайн режим роботи.

Сервер використовує UV project manager [25], веб фреймворк FastAPI для обробки HTTP-запитів, фреймворк SQLAlchemy для створення схеми БД та побудови запитів, і також базу даних PostgreSQL.

3.2 Solidjs frontend

Для стилізації більшості компонентів була використана бібліотека @emotion/css для написання стилів з використанням JavaScript. Для дуже динамічних стилів використовувались або inline css, або custom properties.

Основна частина клієнтського застосунку написана з використанням web-фреймворку SolidJS. Розглянемо компоненти, контролери та моделі (рисунок 9).

Container – перший компонент, який створюється всередині App. Він створює *multiPanelController* та *sidebarController*, після цього передає функції з цих контролерів до *multiPanelContext* та *sidebarContext*, до яких будуть мати доступ усі компоненти, які рендерятся нижче по ієрархії компонентів. *Container* рендерить *Sidebar* та *PanelContainer.*

Sidebar – компонент, який містить меню зліва. Ширину цього компонента можна змінювати динамічно. Всередині знаходяться кореневі сторінки і сторінки, помічені як улюблені. При натисканні на них викликається метод з *multiPanelContext*, який відкриває цю сторінку в активній панелі у новій вкладці.

PanelContainer – компонент для відображення панелей та вкладок на кожній панелі, і який також перерендерить панелі, якщо їх кількість або порядок буде змінено. Цей компонент рендерить *ControlButtons (minimize, maximize, close)* у правому верхньому куті, які ми налаштовуємо за допомогою API, до якого надає доступ Rust. Tauri має верхню панель вікна за замовчуванням, але якщо її кастомізувати, потрібно відповідно налаштувати події, які повинна обробляти верхня панель вікна на кожній системі (контрольні кнопки та місце, за яке можна перетягнути вікно). Якщо

відкривати застосунок не як Tauri app, а у браузері, де немає доступу до системного API, потрібно це перевірити за допомогою функції *isTauri* і пропустити рендер *ControlButtons.* Також цей компонент рендерить *Panel* та *TabsBar* для кожної відкритої панелі.

TabsBar – компонент приймає позицію у grid (rowStart, colStart, rowSpan, colSpan), реактивний сигнал зі сховища про те, яка вкладка відкрита, і список id усіх вкладок. *TabsBar* ми розташовуємо над кожною панеллю для відображення відкритих вкладок і активної вкладки. Якщо користувач відкриває більше вкладок, ніж відведено місця, то вони спочатку зменшуються, а потім з'являється скроллбар. Також планується реалізувати логіку перетягування вкладок для зміни порядку на поточній панелі, а також між різними панелями.

Panel – компонент приймає позицію у grid (rowStart, colStart, rowSpan, colSpan), реактивний сигнал зі сховища про те, яка вкладка відкрита, і список id усіх вкладок. Цей список нам потрібен для того, щоб рендерити усі сторінки, але залишати видимою тільки активну. Це зробить перемикання між вкладками більш плавним, тому що не потрібно рендерити та ініціалізувати кожну сторінку з нуля.

Для управління даними та операціями відкритої сторінки є *pageStateController,* а також 2 допоміжних контролера *pageControllerHelperFactory і pageApiControllerFactory*. Ці контроллери використовують наступні моделі за допомогою бібліотеки arktype [26] та typescript типи.

Спочатку ми визначаємо інтерфейси, за допомогою яких ми будемо будувати кожен з типів блоків (Лістинг 1).

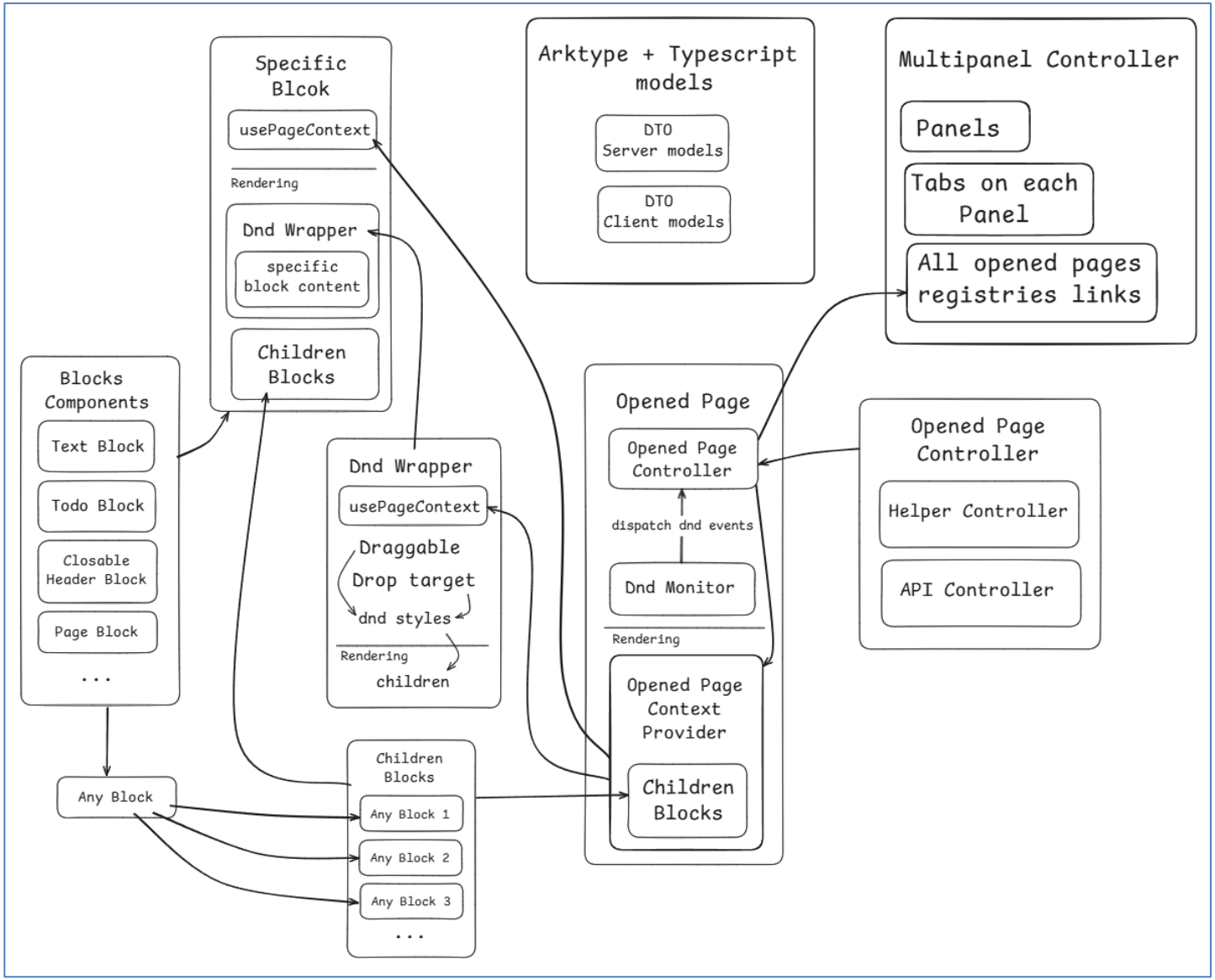


Рисунок *9 — Frontend architecture*

Лістинг *1 — Інтерфейси, за допомогою яких будується кожен з типів блоків.*

```
const types = scope({
  // Base types
  HasText: {
    text: "string",
  },
  HasParent: {
    parentId: "string",
  },
  HasOptionalParent: {
    parentId: "string?",
  },
  HasChildren: {
    childrenIds: "string[]",
  },
  Closable: {
    isOpen: "boolean",
  },
  HasCompletionStatus: {
    isDone: "boolean",
  },
  HasHeaderLevel: {
    level: HeaderLevelSchema,
  },
  HasUrl: {
    url: "string.url",
  },

  // Composite types
  HasTextChildren: ["HasChildren", "&", "HasText"],
  HasTextParent: ["HasText", "&", "HasParent"],
  HasTextChildrenParent: ["HasTextChildren", "&",
"HasParent"],

  //
}).export();
```

Після цього ми визначаємо моделі для даних кожного з типів блоків.

Лістинг *2 — Модель та тип для даних блоку*

```
const ClosableTextDataSchema =
types.HasTextChildrenParent.and(
  types.Closable,
```

```
).and({
  type: BlockDataType.ClosableText,
});
export type ClosableTextData = typeof
ClosableTextDataSchema.infer;
```

Лістинг *3 — Модель та тип для даних блоку для відправки на сервер*

```
const ClosableTextDataServerSchema =
ClosableTextDataSchema.omit("parentId")
  .omit("childrenIds")
  .onUndeclaredKey("delete");
type ClosableTextDataServer = typeof
ClosableTextDataServerSchema.infer;
```

Лістинг *4 — Модель та тип для блоку кожного типу*

```
export const ClosableTextBlockSchema = type({
  id: type.string,
  kind: BlockKind.Nested,
  data: ClosableTextDataSchema,
});
export type ClosableTextBlock = typeof
ClosableTextBlockSchema.infer;
```

Лістинг *5 — Guard блоку кожного типу*

```
// ===== Guard

export function isClosableTextBlock(
  block: unknown,
): block is ClosableTextBlock {
  const closableTextBlock = ClosableTextBlockSchema(block);
  return closableTextBlock instanceof type.errors ===
false;
}
```

Лістинг *6 — Функції для створення нових даних кожного типу*

```
// ===== New Empty Closable Text Data
export function getNewEmptyClosableTextData() {
```

```
  const newEmptyClosableTextData: ClosableTextData = {
    type: "closable_text",
    text: "",
    childrenIds: [],
    parentId: "",
    isOpen: true,
  };
  return newEmptyClosableTextData;
}
```

Лістинг *7 — Модель та тип для масиву блоків кожного типу*

```
const BlocksByTypeSchema = type({
  [BlockDataType.Task.infer]: TaskBlockSchema.array(),
  [BlockDataType.ClosableTask.infer]:
ClosableTaskBlockSchema.array(),
  [BlockDataType.Text.infer]: TextBlockSchema.array(),
  [BlockDataType.ClosableText.infer]:
ClosableTextBlockSchema.array(),
  [BlockDataType.Header.infer]: HeaderBlockSchema.array(),
  [BlockDataType.ClosableHeader.infer]:
ClosableHeaderBlockSchema.array(),
  [BlockDataType.Link.infer]: LinkBlockSchema.array(),
  [BlockDataType.Page.infer]:
PagePreviewBlockSchema.array(),
});
export type BlocksByType = typeof BlocksByTypeSchema.infer;
```

Лістинг *8 — Модель та тип для даних відкритого блоку*

```
export const OpenedItemDataBaseSchema =
types.HasTextChildren.and(
  types.HasOptionalParent,
);
export type OpenedItemDataBase = typeof
OpenedItemDataBaseSchema.infer;
```

Лістинг *9 — Модель та тип для відкритої сторінки*

```
export const OpenedItemSchema = type({
  id: type.string,
  kind: BlockKind.Open,
```

```
  data: OpenedItemDataBaseSchema,
  blocksByType: BlocksByTypeSchema,
});
export type OpenedItem = typeof OpenedItemSchema.infer;
```

Лістинг *10 — Модель та тип для команд, які ми можемо виконувати на відкритій сторінці*

```
export const PageActionSchema = type.or(
  {
    type: "'instruction'",
    instruction: InstructionSchema,
    itemId: type.string,
    itemParentId: type.string,
    targetId: type.string,
    targetParentId: type.string,
  },
  {
    type: "'add-empty-item-after'",
    itemId: type.string,
    itemParentId: type.string,
  },
  {
    type: "'add-block-after'",
    targetId: type.string,
    targetParentId: type.string,
    newBlockData: NestedBlockDataSchema,
  },
  {
    type: "'delete-block'",
    itemId: type.string,
    itemParentId: type.string,
  },
  {
    type: "'edit-text'",
    itemId: type.string,
    newText: type.string,
  },
  {
    type: "'edit-url'",
    itemId: type.string,
    newUrl: type.string,
  },
  {
    type: "'toggle-done'",
    itemId: type.string,
```

```
  },
  {
    type: "'toggle'",
    itemId: type.string,
  },
  {
    type: "'expand'",
    itemId: type.string,
  },
  {
    type: "'collapse'",
    itemId: type.string,
  },
);

export type PageAction = typeof PageActionSchema.infer;
```

pageControllerHelperFactory – closure factory, яка містить утилітні методи для обробки сховища стану відкритої сторінки, такі як:

- *newUniqueUUID* – Метод, який генерує новий UUID для нового блока.
- *getItemById* – Метод, який повертає блок зі сховища контролера за вказаним Id.
- *reparentItem* – Метод, який перевизначає новий parent для вказаного блока.
- *reorderItem* – Метод, який змінює порядок обраного блока відносно іншого.
- *insertNewBlock* – Метод додавання нового блока до сховища.
- editText – Метод зміни тексту обраного блока.
- *toggleIsOpen* – Метод зміни стану IsOpen обраного блока.
- *getLevelOfItemById* – Метод, який знаходить рівень у ієрархії для обраного блока.
- та інші функції для маніпуляцій стану відкритої сторінки.

pageApiControllerFactory – closure factory, яка містить методи, що виконують запити до сервера. Використовується компонент *createResource*, який приймає запит на отримання даних і надає сигнал про результат

виконання запиту (loading, error) і отримані дані запиту, коли результат буде отримано.

pageStateController – closure factory, яка приймає *pageControllerHelperFactory* і *pageApiControllerFactory*, і виконує роль оркестрації цими контролерами для виконання усіх потрібних запитів від компонентів. Спочатку виконується позитивне оновлення у SolidJS store, яке реактивно оновлює усі компоненти, що використовують цей контролер. Після позитивного оновлення одразу виконується запит на сервер з такою ж дією і очікується відповідь. Якщо сервер повертає позитивну відповідь, то більше ніяких дій не потрібно, якщо сервер повертає помилку, ми виконуємо refetch даних стану сторінки, а також повідомлення користувачу про помилку. SolidJs store має утиліту reconcile, яка дозволяє під час отримання нового повного стану даних сторінки виконати реактивне оновлення тільки тих UI елементів, які змінились. Такий підхід забезпечує мінімальну кількість затримок для всіх дій користувача. Містить наступні складові:

- *[pageState, updatePageState] = createStore<OpenedItemState>* – реактивне сховище стану відкритої сторінки.
- *[idToIndexStore, updateIdToIndexStore] = createStore();* – допоміжне сховище для швидшого пошуку блоків за Id.
- Ініціалізація *helperFactory*.
- Ініціалізація *apiFactory*.
- *isInHierarchy* – метод для визначення, чи знаходится один блок у ієрархії іншого.
- *isInstructionAllowed* – допоміжний метод для запобігання виконання недопустимих операцій, наприклад, перетягування блока всередину самого себе.
- *dispatch* – метод для виконання всіх операцій на сторінці, які передаются як discriminated union параметр (type and payload)

Компонент **PageContent (tabId, isActive)** відповідає за відображення відкритої сторінки у вкладці з tabId. Основний контролер **pageStateController**

створюється у компоненті **PageContent** і після створення надається всім компонентам нижче за допомогою контексту. Також **PageContent** містить **createEffect,** який моніторить події перетягування у відкритій сторінці і виконує **dispatch** потрібної команди, коли відбувається подія onDrop. **PageContent** виконує рендер елементів, які занаходятся всередини цієї сторінки, за допмогою компонента **BlockChildren.**

Лістинг *11 — Ефект, який моніторить події перетягування блоків всередині відкритої сторінки*

```
createEffect(() => {
    return combine(
      monitorForElements({
        canMonitor: ({ source }) => {
          const payload =
BlockDraggablePayloadSchema.assert(source.data);
          return payload.uniqueContextId ===
contextValue.uniqueContextId;
        },
        onDrop(args) {
          const { location, source } = args;
          // didn't drop on anything
          if (!location.current.dropTargets.length) {
            return;
          }

          const draggablePayload =
BlockDraggablePayloadSchema.assert(
            source.data,
          );
          const targetPayload =
DropTargetPayloadSchema.assert(
            location.current.dropTargets[0].data,
          );

          if (draggablePayload.type === "tree-item") {
            const itemId = draggablePayload.id;
            const itemParentId = draggablePayload.parentId;

            const targetId = targetPayload.id;
            const targetParentId = targetPayload.parentId;

            const instruction: Instruction | null =
              extractInstruction(targetPayload);
```

```
            if (instruction !== null) {
              const execute = () =>
                contextValue.dispatch({
                  type: "instruction",
                  instruction,
                  itemId,
                  itemParentId,
                  targetId,
                  targetParentId,
                });

              // restrict dropping inside own hierarchy
              if (

contextValue.isInstructionAllowed(instruction, itemId,
targetId)
              ) {
                execute();
              }
            }
          }
        },
      }),
    );
  });
```

Компонент **BlockChildren (***id, hasChildren, isOpen, level***)** рендерить список блоків **AnyBlock** на рівні *level* дерева блоків (блоки всередині відкритої сторінки – 0, всередині них – 1, і т.д.), блок отримує дані з контексту за *id*, а також має стани *hasChildren*, *isOpen,* завдяки яким визначається, чи відображати блоки нижче чи ні.

Лістинг *12 — Компонент BlockChildren*

```
export function BlockChildren(props: {
  id: string;
  hasChildren: () => boolean;
  isOpen: () => boolean;
  level: number;
}) {
  const { getChildrenById } = usePageContext();

  return (
    <Show when={props.hasChildren() && props.isOpen()}>
      <For each={getChildrenById(props.id)}>
        {(id, index) => {
```

```
            return <AnyBlock id={id} level={props.level + 1}
index={index()} />;
          }}
        </For>
      </Show>
    );
}
```

Компонент **AnyBlock** виконує роль визначення, якого типу блок ми маємо за *id*, за допомогою guard для кожного типу блоку та рендерить відповідний компонент. Також в цьому компоненті визначається значення *getBlockMode ("standard" | "expanded" | "last-in-group")* в залежності від розташування блоку в ієрархії. Це значення потрібно для бібліотеки pragmatic dnd, щоб визначати відповідну команду в процесі перетягування блоків (див. рисунок 10).

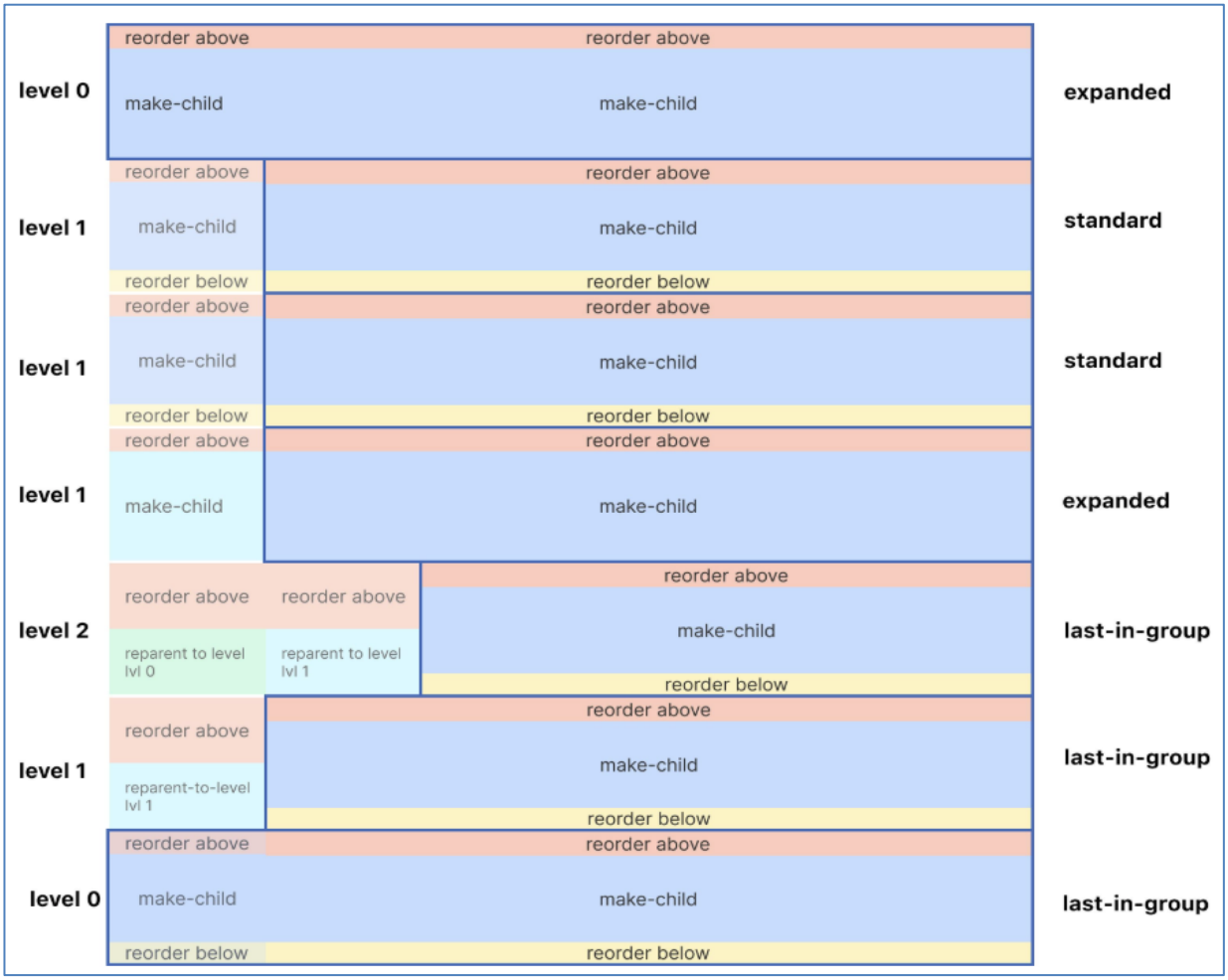


Рисунок *10 — Правило виклику команд бібліотеки pragmatic dnd для ієрархічної структури блоків*

Компонент **BlockDndWrapper** має наступні частини:

- поточний стан *"idle"* | *"dragging"* | *"parent-of-instruction"*. *"idle"* – базовий стан, *"dragging"* – блок зараз перетягується, *"parent-of-instruction"* – блок є батьківським для інструкції, яку зараз намагається виконати користувач.
- Сигнал про те, яку команду зараз намагається виконати користувач – *[instruction, setInstruction] = createSignal<Instruction | null>(null);*
- createEffect, який визначає логіку того, що буде відбуватися, коли поточний об'єкт буде перетягуватися, а також, коли на цей об'єкт буде щось перетягуватися, відображення відповідних стилів.
- рендеринг компонента **DropIndicatorTree,** який сигналізує про те, яка операція буде виконана, якщо користувач завершить перетягування в місці, де зараз знаходиться курсор.

Лістинг *13 — Ефект, який визначає логіку draggable та dropTargetForElements*

```
createEffect(() => {
    if (!props.refDragHandler()) {
      return;
    }

    const dragHandler = props.refDragHandler();
    invariant(refElement);
    invariant(dragHandler);

    return combine(
      draggable({
        element: dragHandler,
        getInitialData: () => {
          const data: BlockDraggablePayload = {
```

```
            id: props.id,
            parentId: props.parentId,
            type: "tree-item",
            uniqueContextId,
          };
          return data;
        },
        onGenerateDragPreview: ({ nativeSetDragImage }) =>
{
          setCustomNativeDragPreview({
            getOffset: pointerOutsideOfPreview({ x: "16px",
y: "8px" }),
            render: ({ container }) => {
              const dispose = render(
                () => <DragPreview text={props.previewText}
/>,
                container,
              );
              return () => dispose();
            },
            nativeSetDragImage,
          });
        },
        onDragStart: () => {
          setState("dragging");
        },
        onDrop: () => {
          setState("idle");
        },
      }),
      dropTargetForElements({
        element: refElement,
        getData: ({ input, element }) => {
          const data: DropTargetPayload = {
            id: props.id,
            parentId: props.parentId,
          };

          return attachInstruction(data, {
            input,
            element,
            indentPerLevel: INDENT_PER_LEVEL,
            currentLevel: props.level,
            mode: props.mode(),
            block: [],
          });
        },
        canDrop: ({ source }) => {
```

```
          const payload =
BlockDraggablePayloadSchema.assert(source.data);
          return (
            payload.type === "tree-item" &&
            payload.uniqueContextId === uniqueContextId
          );
        },
        getIsSticky: () => true,
        onDragEnter: onChange,
        onDrag: onChange,
        onDragLeave: () => {
          setInstruction(null);
        },
        onDrop: () => {
          setInstruction(null);
        },
      }),
      monitorForElements({
        canMonitor: ({ source }) => {
          const payload =
BlockDraggablePayloadSchema.assert(source.data);
          return payload.uniqueContextId ===
uniqueContextId;
        },
        onDragStart: ({ location, source }) => {
          updateIsParentOfInstruction({ location });
          updateDraggableHierarchyStyles({ source });
        },
        onDrag: updateIsParentOfInstruction,
        onDrop: () => {
          clearParentOfInstructionState();
          clearDraggingState();
        },
      }),
    );
  });
```

Компоненти **ToggleTodoForPage, UrlLinkForPage, TextForPage** та інші типи блоків мають схожу структуру. Кожен з них має *BlockDndWrapper*, всередині якого розташовані відповідні дані для кожного блоку, а також *BlockChildren.*

Лістинг *14 — Рендеринг компонента ToggleTodoForPage*

```
<>
```

```
      <BlockDndWrapper
        id={item.id}
        parentId={item.data.parentId}
        level={level}
        mode={mode}
        previewText={item.data.text}
        refDragHandler={refDragHandler}
      >
        <IconsWrapperBlock>
          <PlusAndHandlerBlock
            onClickPlus={addEmptyItemAfter}
            refHandler={setRefDragHandler}
          />
          <PageBlockToggleButton
            hasChildren={hasChildren()}
            isOpen={isOpen()}
            onToggle={toggle}
          />
          <CustomCheckbox value={isDone()}
onChange={toggleDone} />
        </IconsWrapperBlock>
        <EditableTextBlock
          type={{ type: "text" }}
          initialText={item.data.text}
          onSave={editText}
          onEnterClick={addEmptyToggleTodoAfter}
        />
      </BlockDndWrapper>
      <BlockChildren
        id={item.id}
        hasChildren={hasChildren}
        isOpen={isOpen}
        level={level}
      />
    </>
```

Таким чином, ми можемо забезпечити використання будь-яких типів блоків на сторінці, вкладених сторінок, перегрупування та зміну інформації кожного з блоків, миттєве позитивне оновлення інтерфейсу. Також після доопрацювання можна буде використати весь потенціал мультипанельного інтерфейсу.

3.3 База даних

База даних серверної частини (рисунок 11) має тип **Class Table Inheritance.** Ця стратегія є зручною в нашому випадку для зберігання id базового блоку та його типу в спільній таблиці, а інформації, специфічної для кожного типу блоку, в окремій таблиці, кожна з яких приєднується до базової за спільним UUID. Визначення схеми, створення БД, видалення БД, міграції виконується за допомогою *SQLAlchemy ORM API* та бібліотек *Alembic*.

Спочатку визначаємо клас *_Base(DeclarativeBase)*, який наслідується від декларативної бази. Кожна таблиця, яку ми визначаємо, повинна наслідуватися від *_Base* для того, щоб ми могли через цей клас виконувати *create* або *drop* команди. Також для кожної таблиці ми додаємо анотації типів для більш зручної побудови запитів. Далі розглянемо основні таблиці в БД.

Лістинг *15 — Таблиця для централізованого зберігання типів блоків. Містить id та опціональний опис*

```
class BlockType(_Base):
    __tablename__ = "block_types"

    name: Mapped[str] = mapped_column(String,
primary_key=True)
    description: Mapped[str | None] = mapped_column(String,
nullable=True)
```

Лістинг *16 — Таблиця base_blocks. Містить id та тип блоку, прив'язаний до block_types*

```
class BaseBlock(_Base):
    __tablename__ = "base_blocks"

    id: Mapped[PyUUID] = mapped_column(
        UUID, primary_key=True,
server_default=func.uuid_generate_v4()
```

```
    )
    type: Mapped[str] = mapped_column(
        String, ForeignKey("block_types.name",
ondelete="RESTRICT"), nullable=False
    )
    data: Mapped[dict] = mapped_column(JSON, nullable=True)
```

Таблиця *block_children* містить інформацію, ієрархічне відношення блоків, а також порядок блоків всередині батьківського блока. Таблиця зберігає parent_id, child_id, order_index. order_index — це дробове число, яке представляє порядок блока. Коли нам потрібно змінити порядок, наприклад, вставити блок між індексами 1 та 2, то новий блок отримує індекс 1.5. Завдяки такому підходу ми можемо зменшити кількість перерахунків при зміні порядку, але треба слідкувати за тим, щоб точність не стала замалою. У цьому випадку ми перерахуємо індекси і зробимо їх знову 1.0, 2.0, 3.0 і т.д.

Лістинг *17 — Таблиця block_children:*

```
class BlockChildren(_Base):
    __tablename__ = "block_children"

    parent_id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("base_blocks.id",
ondelete="RESTRICT"), primary_key=True
    )
    child_id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("base_blocks.id",
ondelete="RESTRICT"), primary_key=True
    )
    order_index: Mapped[float] = mapped_column(Double,
nullable=False)
    is_origin: Mapped[bool] = mapped_column(Boolean,
nullable=False, default=False)
```

Лістинг *18 — Таблиця page_blocks містить інформацію про будь яку сторінку. На даному етапі це заголовок(text) та id базового блока*

```
class PageBlock(_Base):
    __tablename__ = "page_blocks"
```

```
    id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("base_blocks.id",
ondelete="CASCADE"), primary_key=True
    )
    text: Mapped[str] = mapped_column(String,
nullable=False)
```

Лістинг *19 — Таблиця top_level_pages містить інформацію про ті сторінки, які є кореневими, поля id та order_index*:

```
class TopLevelPage(_Base):
    __tablename__ = "top_level_pages"

    id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("page_blocks.id",
ondelete="CASCADE"), primary_key=True
    )
    order_index: Mapped[float] = mapped_column(Double,
nullable=False)
```

Лістинг *20 — Таблиця favorite_pages містить інформацію про ті сторінки, які помічені, як улюблені. Це можуть бути як кореневі, так і вкладені сторінки. Таблиця також містить поля id та order_index*

```
class FavoritePage(_Base):
    __tablename__ = "favorite_pages"

    id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("page_blocks.id",
ondelete="CASCADE"), primary_key=True
    )
    order_index: Mapped[float] = mapped_column(Double,
nullable=False)
```

Далі визначаються таблиці для кожного типу блоку, як приклад наведено таблицю *closable_task_blocks* з полями *id, text, is_done, is_open* і таблицю *link_blocks* з полями *id, text, url:*

Лістинг *21 — Таблиця closable_task_blocks з полями id, text, is_done, is_open*

```
class ClosableTaskBlock(_Base):
    __tablename__ = "closable_task_blocks"

    id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("base_blocks.id",
ondelete="CASCADE"), primary_key=True
    )
    text: Mapped[str] = mapped_column(String,
nullable=False)
    is_done: Mapped[bool] = mapped_column(Boolean,
nullable=False)
    is_open: Mapped[bool] = mapped_column(Boolean,
server_default="false")
```

Лістинг *22 — Таблиця link_blocks з полями id, text, url*

```
class LinkBlock(_Base):
    __tablename__ = "link_blocks"

    id: Mapped[PyUUID] = mapped_column(
        UUID, ForeignKey("base_blocks.id",
ondelete="CASCADE"), primary_key=True
    )
    url: Mapped[str] = mapped_column(String,
nullable=False)
    text: Mapped[str] = mapped_column(String,
nullable=True)
```

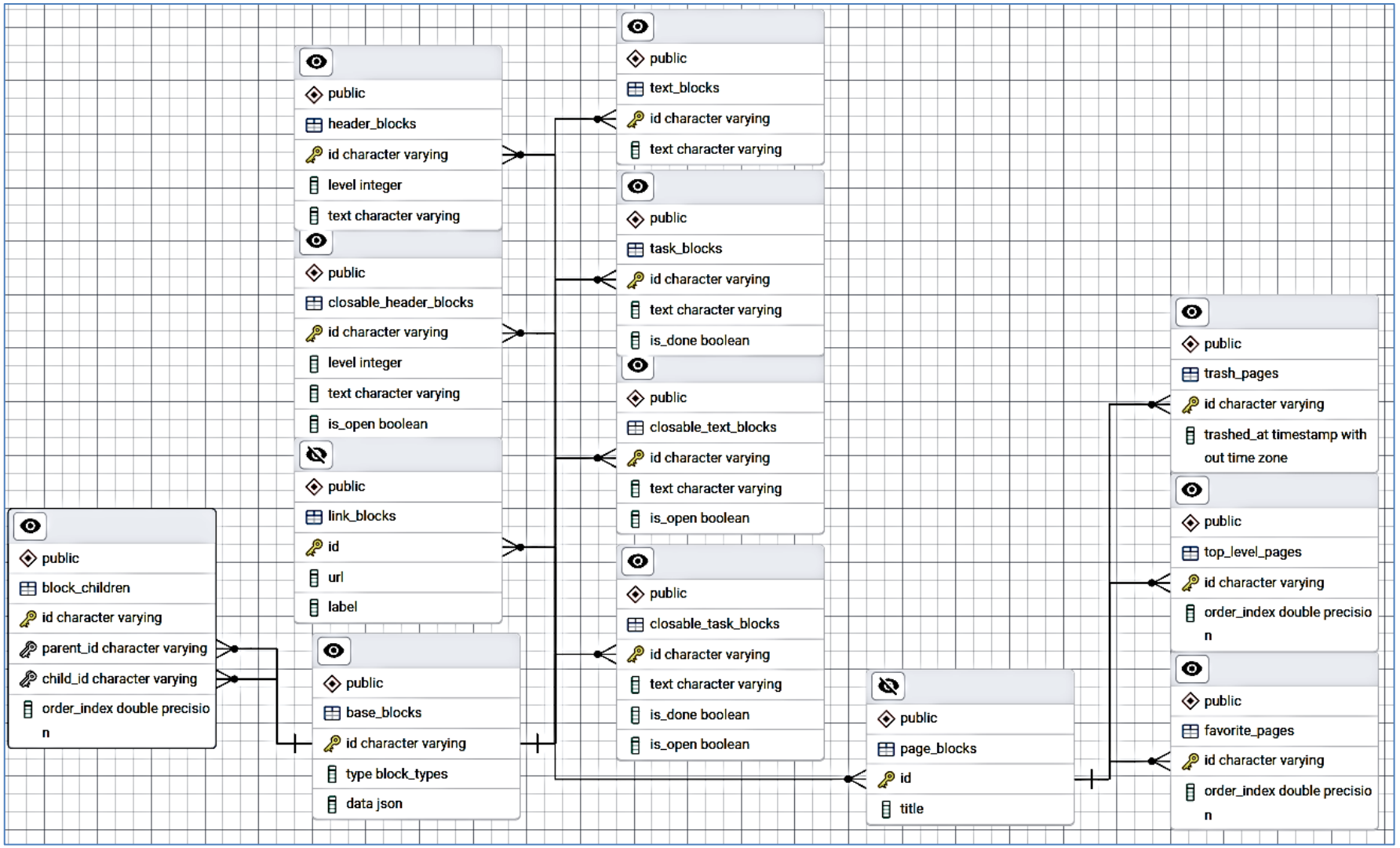


Рисунок *11* — *База даних серверної частини*

3.4 Серверна частина

У цьому розділі розглянемо, як веб сервер обробляє запити від клієнта. Для кожного запита GET, POST, DELETE, UPDATE визначаємо відповідні обробники.

Лістинг *23 — Обробник запиту GET для отримання вмісту кореневої сторінки за id*

```
@router.get("/content/root-page/{item_id}",
response_model=OpenedItem)
async def root_page_content(item_id: str):
    result = await get_opened_root_page_content_db(item_id)
    if result == DBOperationResult.FAILURE:
        raise HTTPException(

status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
            detail="Failed to get root page content",
        )
    if result == DBOperationResult.NOT_FOUND:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
detail="Root page not found"
        )
    return result
```

Для кожного обробника HTTP запиту у FastAPI нам потрібно визначити моделі даних для запиту, якщо це запит POST, та для відповіді.

Лістинг *24 — response_model для запиту на отримання вмісту кореневої сторінки за id*:

```
class OpenedItem(CamelModel):
    id: str
    kind: Literal[BlockKindEnum.open] = BlockKindEnum.open
    data: OpenedItemDataClient
    blocks_by_type: BlocksByTypeMap

    model_config = ConfigDict(
        extra="ignore",  # silently drop any extra keys
```

```
    )
```

Після того, як обробник провалідував формат даних запиту на коректність, виконується тіло функції, де ми викликаємо потрібні нам запити до БД. У випадку отримання вмісту кореневої сторінки це — await *get_opened_root_page_content_db (item_id).* Між запитами до БД під час виконання дані передаються у вигляді server DTO-моделей.

Лістинг *25 — Метод, який розпочинає транзакцію для отримання вмісту кореневої сторінки*

```
async def get_opened_root_page_content_db(page_id: str):
    try:
        async with get_async_engine().begin() as conn:
            return await get_opened_root_item(conn,
page_id)
    except Exception as e:
        logger.error(f"Error getting opened root page
content: {e}")
        return DBOperationResult.FAILURE
```

Кожен запит, який закінчується на *_db*, спочатку виконує отримання конекту до БД, після цього розпочинає транзакцію, всередині якої вже викликається метод без закінчення *_db*, який виконує запит. Кожен такий запит в свою чергу викликає деяку кількість допоміжних запитів до БД, які виконують одну визначену дію з однією таблицею. Якщо це запит на отримання даних, то результат збирається до моделі client dto у форматі, в якому дані очікує клієнт. У запиті з лістингу 26 це — модель *OpenedItem.* Після цього, якщо запит до БД успішний, відповідь відправляється назад клієнту. На рисунку 12 наведено візуально-схематичне пояснення того, як це відбувається.

Лістинг *26 — Запит на отримання даних opened_root_item всередині розпочатої транзакції*

```
async def get_opened_root_item(conn: AsyncConnection,
block_id: str):
```

```
    # 1 - get data of root page with block_id
    opened_item_data = await get_root_page_data(conn,
block_id)
    if opened_item_data == DBOperationResult.NOT_FOUND:
        return DBOperationResult.NOT_FOUND

    # 3 - get all children blocks in dict by type
    blocks = await get_all_blocks_model(conn, block_id)
    if blocks == DBOperationResult.NOT_FOUND:
        return DBOperationResult.NOT_FOUND

    # 4 - building response for client of type OpenedBlock
    opened_block = OpenedItem.model_validate(
        {
            "id": block_id,
            "data": opened_item_data.model_dump(),
            "blocks_by_type": blocks,
        }
    )

    return opened_block
```

Розглянемо ще один запит, POST для зміни порядку блоків.

Лістинг *27 — POST для зміни порядку блоків*

```
@router.post("/reorder-item",
response_model=ResponseSuccess)
async def reorder_item(reorder_item_request:
ReorderItemRequest):
    result = await reorder_block_db(reorder_item_request)
    match result:
        case DBOperationResult.SUCCESS:
            return ResponseSuccess(
                success=True, message="Blocks reordered
successfully"
            )
        case DBOperationResult.FAILURE:
            raise HTTPException(

status_code=status.HTTP_500_INTERNAL_SERVER_ERROR,
                detail="Failed to reorder blocks",
            )
```

В цьому запиті ми маємо модель запиту на reorder. Якщо валідація якоїсь з цих Pydantic моделей не проходить, то FastAPI одразу відправляє відповідь про помилку валідації, і обробка запиту на цьому закінчується.

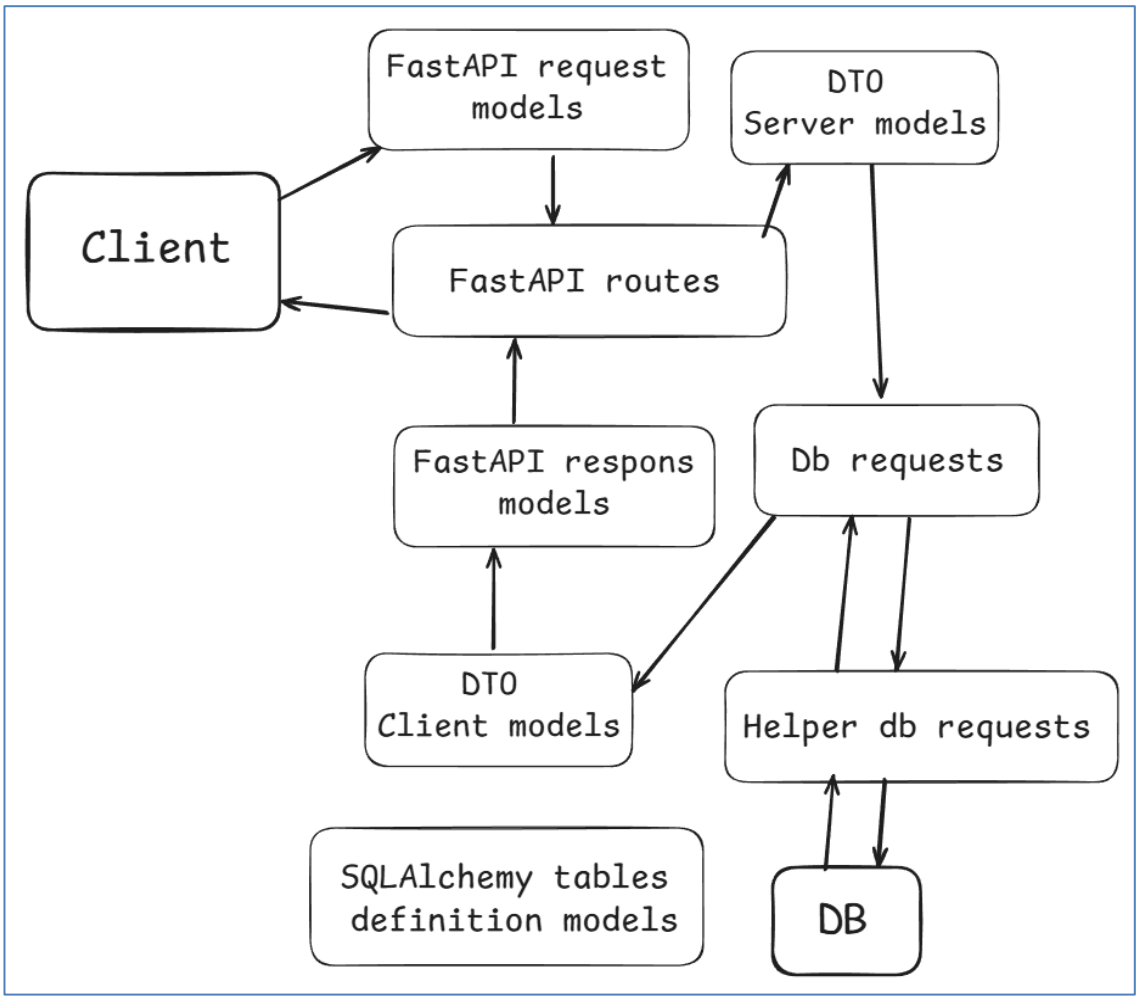


Рисунок 12 — *Візуально-схематичне представлення обробки запитів на сервері*

Лістинг *28 — Модель запиту на reorder*

```
class ReorderItemRequest(CamelModel):
    item_id: str
    item_parent_id: str
    target_id: str
    target_parent_id: str
    reorder_position: ReorderPositionOption
```

Якщо валідація успішна, то все відбувається за порядком, наведеним вище: ініціація конекту до БД, початок транзакції, запит, який використовує

деяку кількість допоміжних запитів, відповідь про успішність транзакції або відкат, у випадку необробленої помилки, і відповідь про помилку:

Лістинг *29 — Запит на виконання reorder всередині розпочатої транзакції*

```
async def reorder_block(
    conn: AsyncConnection,
    item_id: str,
    item_parent_id: str,
    target_id: str,
    target_parent_id: str,
    reorder_option: ReorderPositionOption,
):
    """
    Detach `item_id` from its current parent, and attach it
to `target_parent_id`.
    """

    # 1 Delete the existing link
    await delete_parent_child(conn, item_parent_id,
item_id)

    # 2 connect to parent of target
    await connect_to_parent_of_target(
        conn, item_id, target_id, target_parent_id,
reorder_option
    )
```

3.5 Функціональні вимоги

Система розробляється для того, щоб:

- Полегшити розподіл та групування безперервного потоку інформації та завдань.
- Полегшити процес пріоритизації та розподіл завдань між учасниками, якщо проєкт командний.
- Полегшити групування задач та розбиття їх на підзадачі.
- Покращити ефективність планування проєктів, пошук рішень для проблем проєкту та генерацію ідей.

- Мінімізувати перемикання між різними додатками та екранами, що розсіює увагу та призводить до ментальної втоми.
- Зробити перемикання між текстовими та візуальними представленнями інформації легкими та зручними.
- Забезпечити можливість легко презентувати проєкти, ідеї або схеми учасникам команди чи зацікавленим сторонам.

Система буде корисною як для персональних, так і для командних проєктів різної складності. Вона включатиме:

1. Режим візуального планування та організації. Цей режим допоможе групувати, пов'язувати та пріоритезувати завдання [1], а також відображення зв’язків між задачами та ресурсами.
2. Гнучкий інтерфейс із модульними вікнами. Інтерфейс системи побудований за принципом модульності. Користувачі зможуть адаптувати інтерфейс під власні потреби, використовуючи готові або власні шаблони.
3. Режим презентацій. Це дозволить легко представляти проєкти учасникам команди чи зацікавленим сторонам.
4. Фокусування. Система сприятиме використанню двох основних видів фокусу [2]:
   - Гіперфокус – для виконання конкретних завдань з мінімальними відволіканнями.
   - Скаттерфокус – для генерації ідей і пошуку рішень.
5. Забезпечити можливість зручно та однозначно визначити різні типи інформації та керувати зв’язками між ними.
6. Забезпечити зручний режим керування командами, розподілення завдань та проєктів між учасники команд, трекінг навичок та знань учасників команд, контроль ролей.

3.6 Вимоги до інтерфейсу

На рисунку 13 зображено головний екран застосунку з відкритою сторінкою Focus.

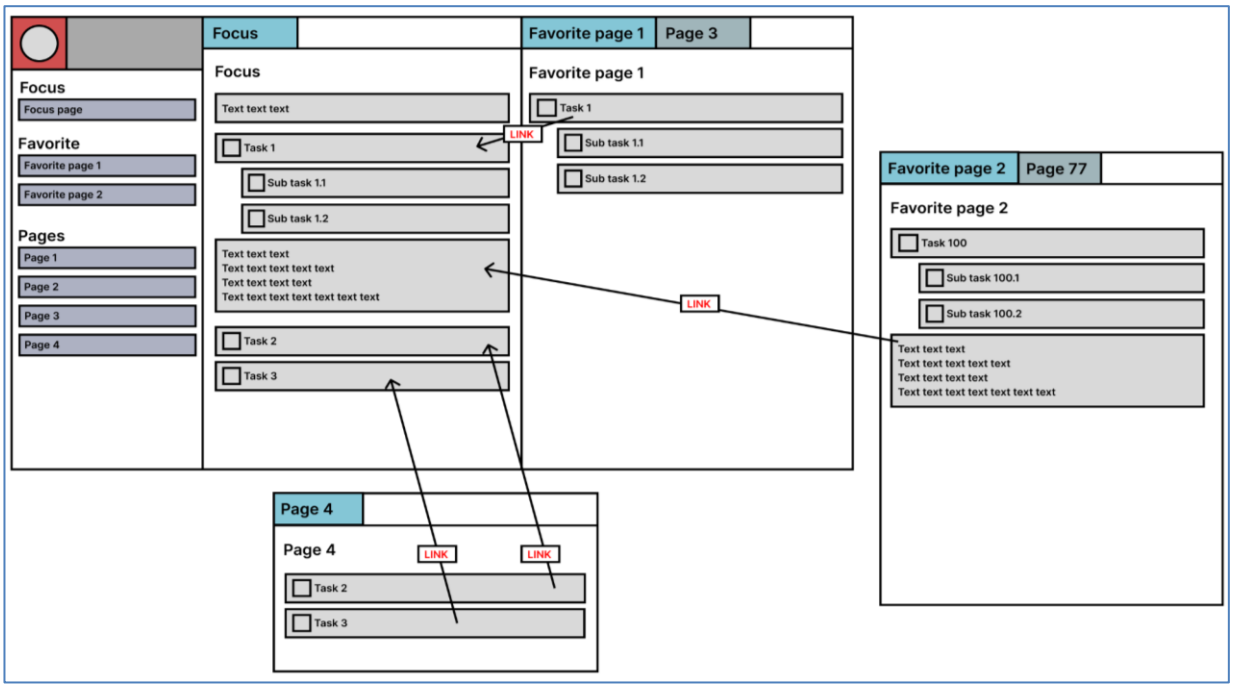


Рисунок *13 — Головний екран застосунку*

На рисунку 14 зображено можливість додавання до блоків тегів, дедлайнів, а також фільтрація на основі цієї інформації.

На рисунку 15 зображено схематичне представлення візуального встановлення порядку та групування для завдань.

На рисунку 16 зображено схематичне представлення функції для виділення частини тексту для відокремлення частини блоку як нового самостійного блоку з можливістю додати додаткову інформацію в процесі цієї команди.

На рисунку 17 зображено схематичне представлення функції для створення зв'язків між завданнями, блоками та проєктами.

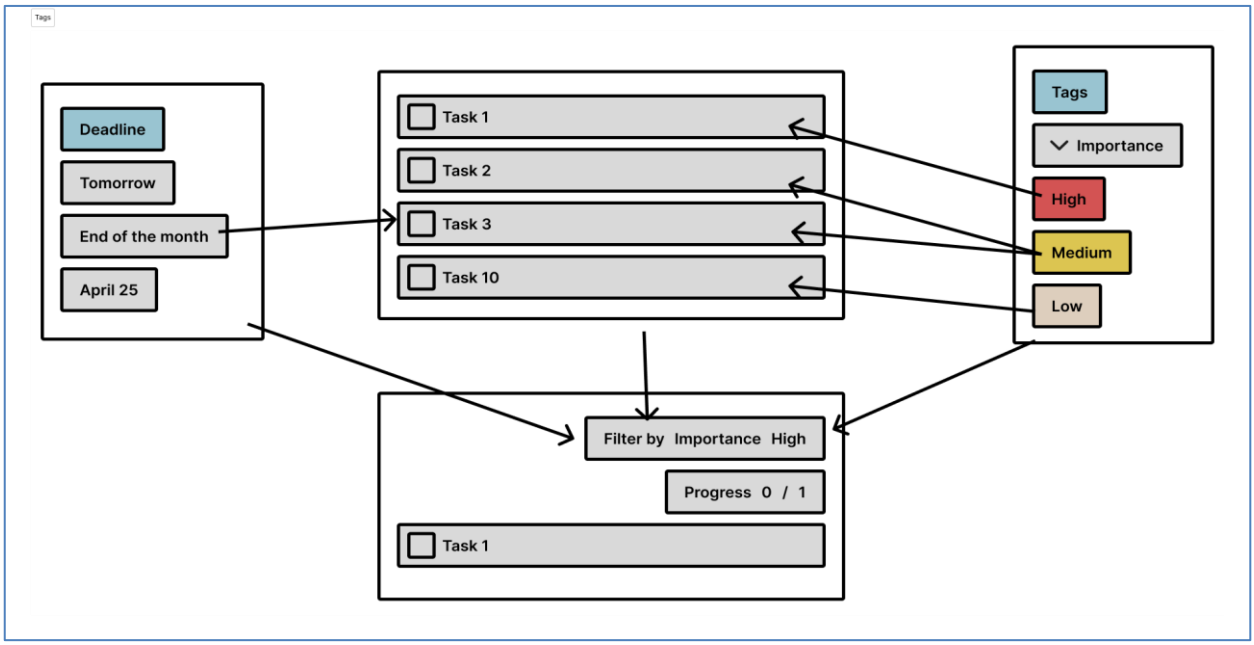


Рисунок *14 — Схематичне представлення тегів та дедлайнів*

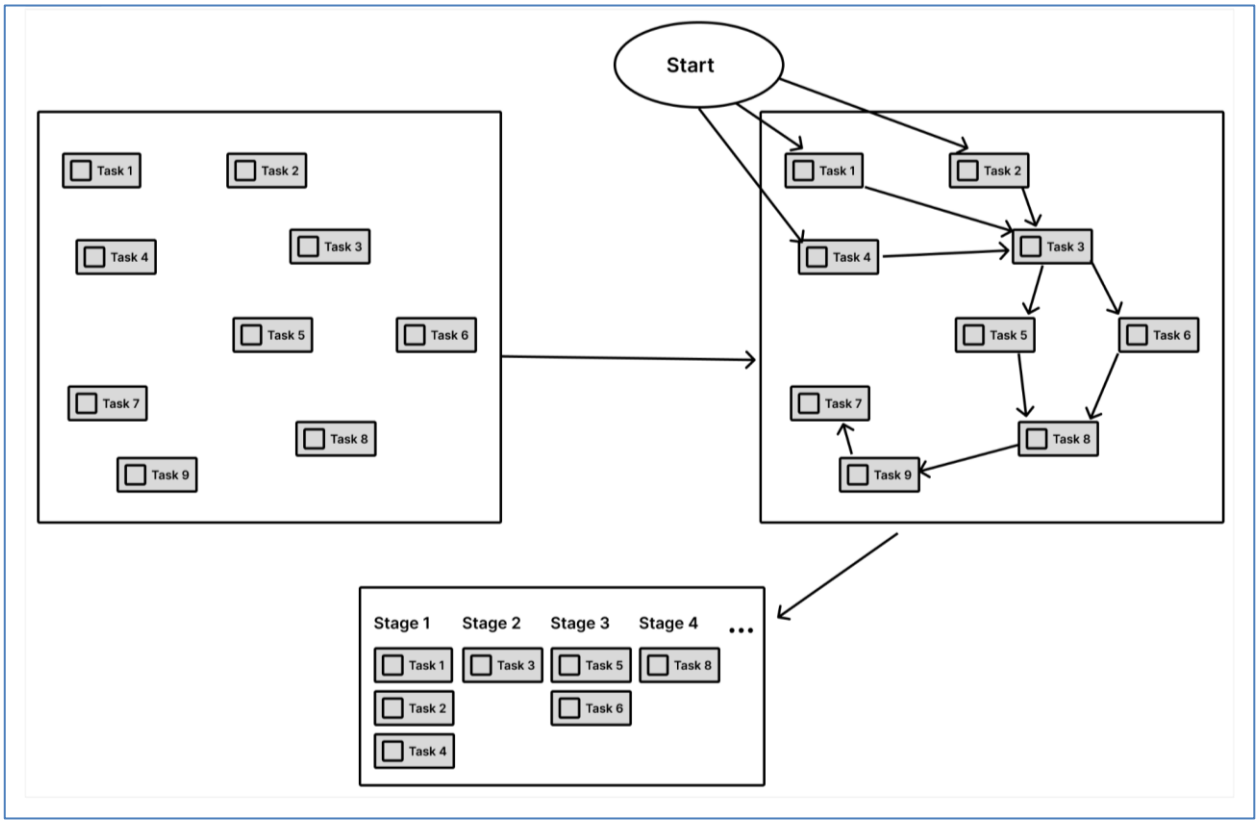


Рисунок *15 — Схематичне представлення візуального встановлення порядку та групування для завдань*

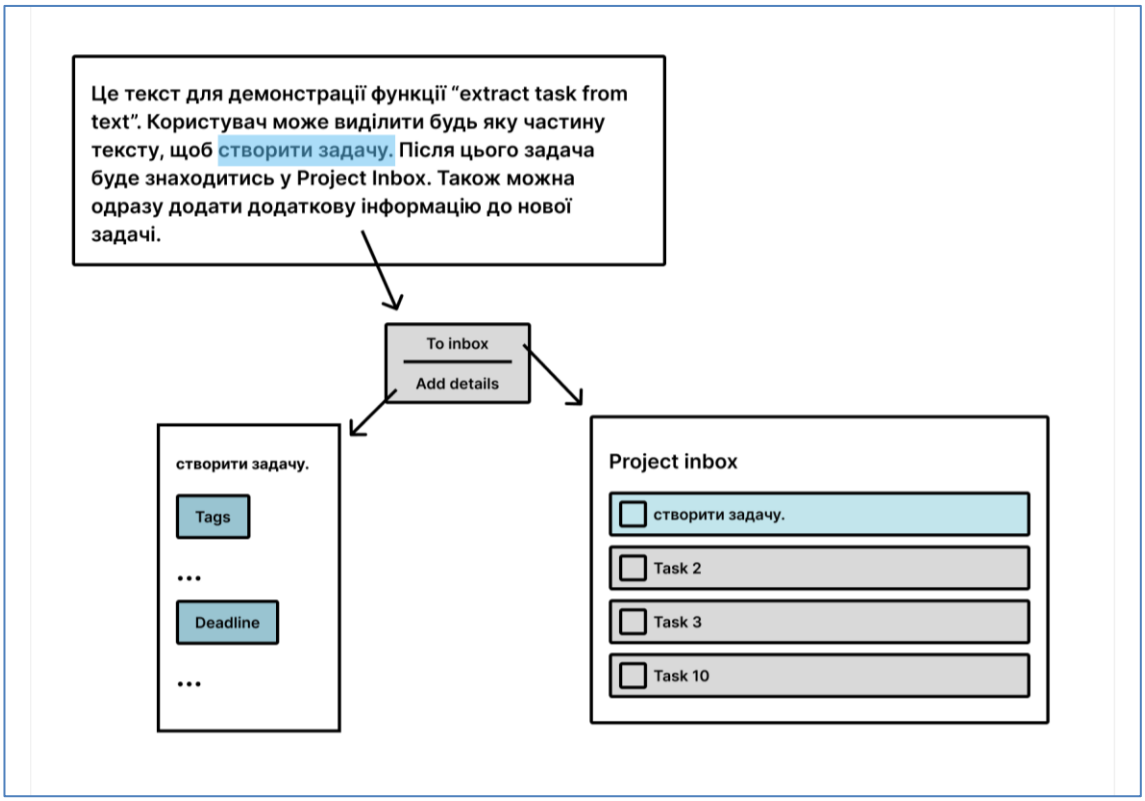


Рисунок *16 — Схематичне представлення функції для виділення частини тексту для виклику однієї з команд*

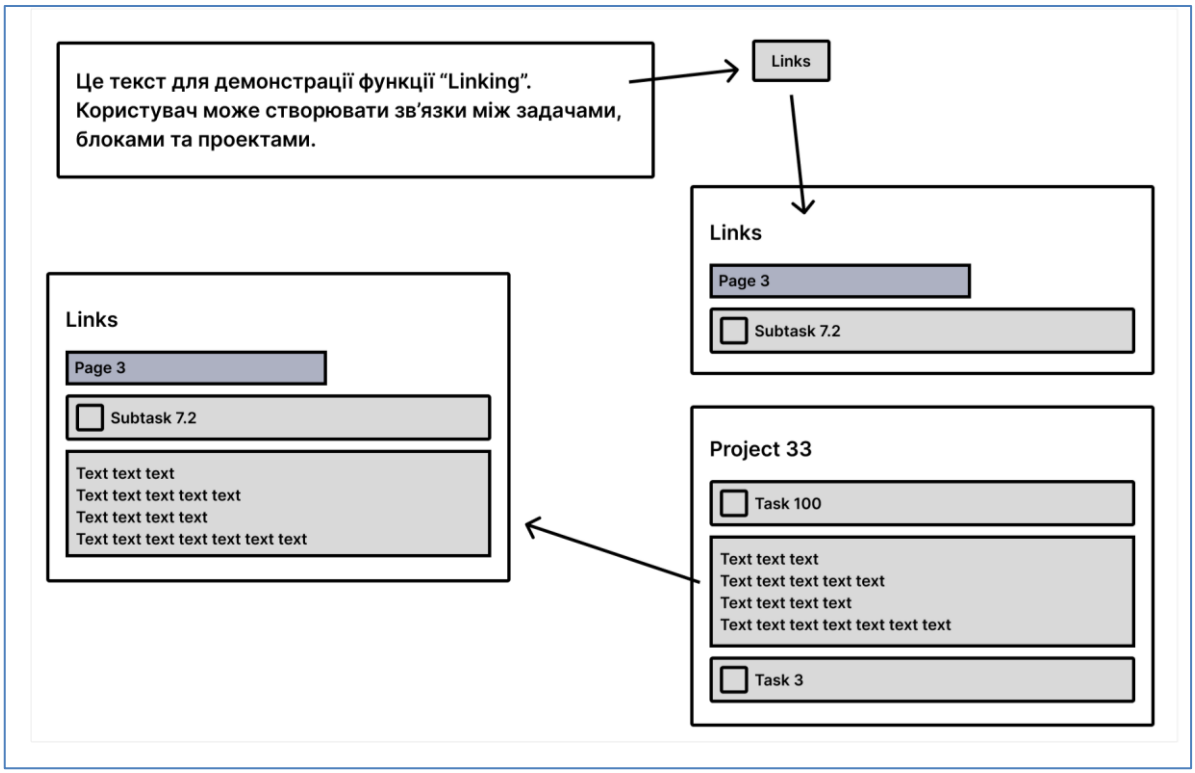


Рисунок *17 — Схематичне представлення функції для створення зв'язків між завданнями, блоками та проєктами.*

3.7 Реалізація і тестування

3.7.1 Unit тести для тестування серверної частини

Для тестування розгортання БД та запитів до неї, щоб гарантувати цілісність даних, а також впевнитись у правильності комплексних запитів, які змінюють одразу декілька таблиць або мають складну структуру з використанням recursive CTE, були використані наступні інструменти:

1. pytest [30] – framework для написання Unit тестів для python проєкту. Однією з переваг pytest є *Fixtures*. *Fixtures* надають фіксований, надійний та масштабований контекст для тестів. Вони можуть включати налаштування оточення або визначені датасети для тестів. Fixtures можуть мати scope тестової сесії, тестового класу або окремого тесту, що визначає частоту повторного виклику функції *fixture*. У pytest fixtures визначаються як функції з декоратором *@pytest.fixture*.

2. Плагіни pytest-cov та pytest-asyncio для тестового покриття та підтримки асинхронних тестів.

3. Hypothesis [31] – інструмент для генерації різних “стратегій” вхідних даних, що спрощує тестування різних варіантів вхідних параметрів.

4. Podman [27] і Testcontainers [28] для ізольованого запуску контейнер-середовища з PostgreSQL DB. Podman (short for pod manager) — це опенсорс інструмент для створення, керування та запуску контейнерів, розроблений Red Hat. Pods — це групи контейнерів, які працюють одночасно та ділять спільні ресурси аналогічно до Kubernetes pods. Podman CLI підтримує стандарт Open Container Initiative (OCI) для контейнерів. Особливістю Podman є те, що він має daemonless архітектуру, яка означає, що йому не потрібен кореневий процес для керування контейнерами. Кожен Pod може сам мати як кореневі, так і не кореневі права в системі в залежності від наших налаштувань і потреб. Також Podman часто може бути взаємозамінний з Docker через схожість API та підтримку стандарту OCI. Ще однією зручністю є наявність Podman Desktop – графічний інтерфейс для використання Podman у локальному середовищі.

Testcontainers – те, заради чого нам був потрібен сетап контейнера, це відкрита бібліотека для використання легких інстансів БД для зручного тестування. [29] Під час кожної тестової сесії ми створюємо нову БД, а після закінчення все очищається.

Такий набір інструментів дозволяє протестувати запити до БД, створення та видалення БД повністю автоматизовано.

Fixture для запуску чистого контейнеру з *postgres* та *asyncpg* python драйвера, який використовується для запитів. Після створення ми отримуємо connection link та додаємо її до тимчасових змінних оточення, які будуть використані під час тестування. Fixture має scope="session", це означає що запуск тестового контейнера виконується на початку тестової сесії, а cleanup— перед її завершенням.

Лістинг *30 — Fixture для запуску чистого контейнеру*

```
@pytest.fixture(scope="session", autouse=True)
def postgres_container(request: pytest.FixtureRequest):
    """Start a single Postgres container for all tests in
this session."""
    postgres = PostgresContainer("postgres:16-alpine",
driver="asyncpg")
    postgres.start()

    """
    For each test class:
    • Set ENV=test
    • Point DB_URL (and other vars) at the container
    • Clear Pydantic/engine caches so they re-read these
values
    """
    mp = pytest.MonkeyPatch()
    mp.setenv("ENV", "test")
    mp.setenv("DB_URL", postgres.get_connection_url())
    # …etc for HOST, PORT, USER, PASS…
    get_settings.cache_clear()
    get_async_engine.cache_clear()

    def _cleanup():
        mp.undo()
```

```
        get_settings.cache_clear()
        get_async_engine.cache_clear()

    request.addfinalizer(_cleanup)

    return postgres
```

Лістинг *31 — Приклад Hypothesis стратегії передавання в тест рядків рандомної довжини у вказаному діапазоні*

```
def short_text_strategy(min_size=0, max_size=50):
    """
    Returns a Hypothesis strategy for generating random
short texts.

    Args:
        min_size: The minimum length of the generated text.
        max_size: The maximum length of the generated text.

    Returns:
        A Hypothesis text strategy.
    """
    return st.text(
        alphabet=st.characters(
            whitelist_categories=["L", "Nd", "Zs", "P"]
        ),  # Letters, Digits, Spaces, Punctuation
        min_size=min_size,
        max_size=max_size,
    )
```

Для тестування додавання нової кореневої сторінки у кінець використовуємо стратегію, яка дає нам як параметр рандомний рядок від 0 до 50 символів, задану у hypothesis кількість разів. У тесті ми створюємо нову чисту БД у контейнері, який активований, і тестуємо, що запит виконується правильно:

Лістинг *32 — Приклад тесту, який тестує додавання нової кореневої сторінки у кінець*

```
@given(page_text=short_text_strategy())
    async def test_insert_top_level_page_to_end(
        self, async_engine_fix: AsyncEngine, page_text
```

```
    ):
        """
        Test that one first top level page is inserted
correctly.
        """
        await create_tables_script(drop=True)

        test_page_id = await
insert_top_level_page_to_end_db(page_text)

        # 1 - test that base block is inserted correctly
        assert test_page_id != DBOperationResult.FAILURE,
"Insertion failed"
        base_block = await
select_base_block_by_id_db(async_engine_fix, test_page_id)
        assert isinstance(base_block, BaseBlockModel), (
            "base_block is not a BaseBlockModel"
        )
        logger.info(base_block)
        assert (base_block.id, base_block.type) == (
            test_page_id,
            BlockTypesEnum.ROOT_PAGE,
        ), "Wrong block"
        all_base_blocks = await
select_all_base_blocks_db(async_engine_fix)
        assert isinstance(all_base_blocks, list),
"all_base_blocks is not a list"
        assert base_block in all_base_blocks, "Base block
not found"
        assert len(all_base_blocks) == 1, "Wrong number of
base blocks"

        # 2 - test that page details are inserted correctly
        page_details = await select_page_details_by_id_db(
            async_engine_fix, test_page_id
        )
        assert isinstance(page_details, PageDataModel), (
            "page_details is not a PageDataModel"
        )
        assert page_details.text == page_text, "Wrong page
details text"
        all_page_details = await
select_all_page_details_db(async_engine_fix)
        assert isinstance(all_page_details, list),
"all_page_details is not a list"
        assert page_details in all_page_details, "Page
details not found"
```

```
        assert len(all_page_details) == 1, "Wrong number of
page details"

        # 3 - test that top level page is inserted
correctly
        root_page_details = await
select_root_page_details_by_id_db(
            async_engine_fix, test_page_id
        )
        assert root_page_details is not None, "Root page
details not found"

        root_pages_ordered = await
select_all_root_pages_in_order_db(async_engine_fix)
        assert isinstance(root_pages_ordered, list),
"root_pages_ordered is not a list"
        assert root_page_details in root_pages_ordered,
"Root page details not found"
        assert len(root_pages_ordered) == 1, "Wrong number
of root page details"

        # 4 - test select all top level pages function
        all_top_level_pages = await
select_all_top_level_pages_db()
        assert isinstance(all_top_level_pages, list), (
            "all_top_level_pages is not a list"
        )
        assert len(all_top_level_pages) == 1, "Wrong number
of top level pages"
        assert isinstance(all_top_level_pages[0],
RootPagePreviewBlock), (
            "all_top_level_pages[0] is not a
RootPagePreviewBlock"
        )
        inserted_page = all_top_level_pages[0]
        assert inserted_page.id == test_page_id, "Wrong
page id"
        assert inserted_page.data.type ==
BlockTypesEnum.ROOT_PAGE, "Wrong block type"
        assert inserted_page.data.text == page_text, "Wrong
page text"
```

Таким чином, ми тестуємо усі основні запити до бази даних.

Результати тестування запитів до бази даних за покриттям statement та branches представлені на рисунках 18, 19. На рисунку 20 представлені результати виконання тестів.

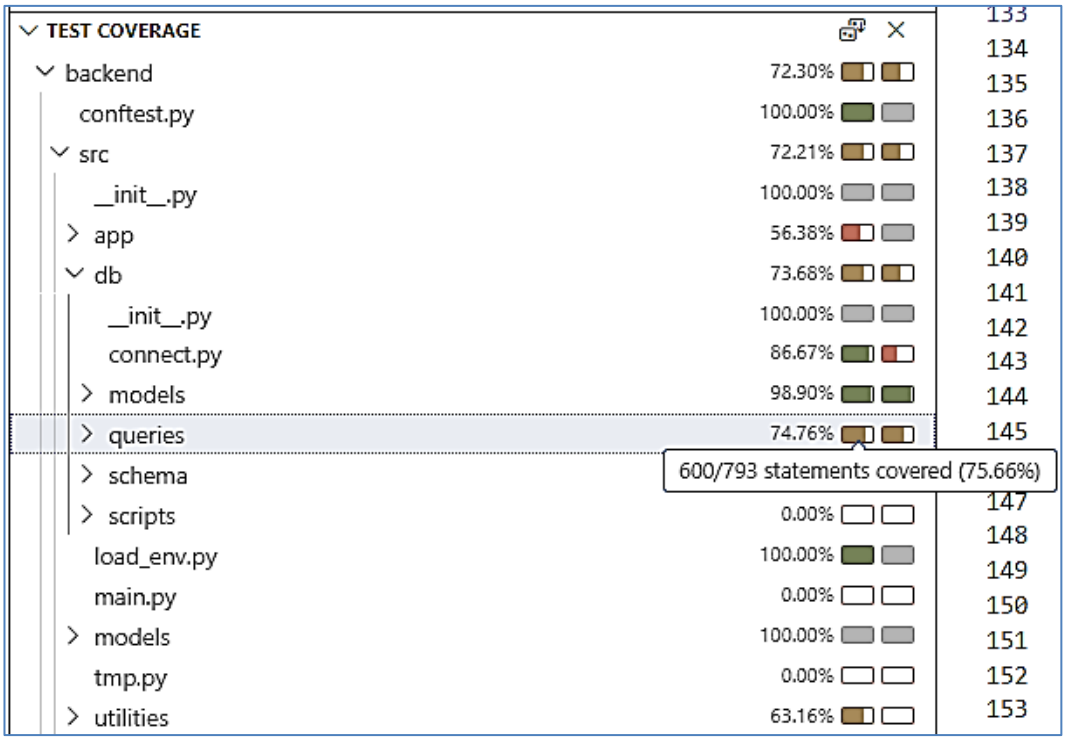


Рисунок *18 — Результати тестування запитів до бази даних за покриттям statement backend-частини*

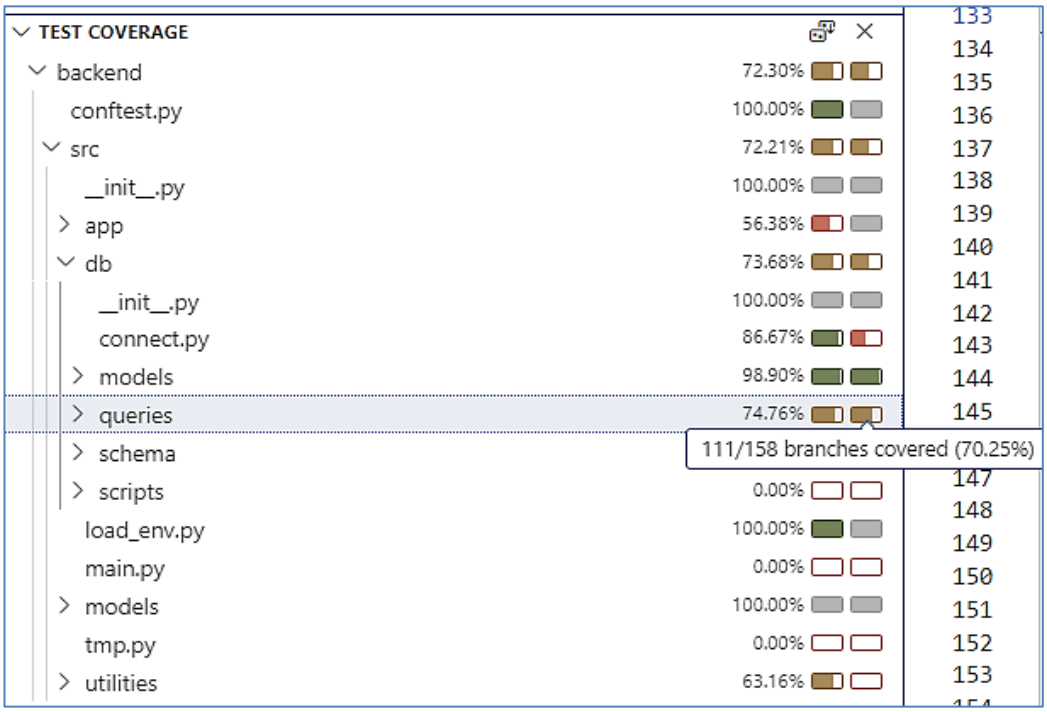


Рисунок *19 — Результати тестування запитів до бази даних за покриттям branches backend-частини*


test_db_setup.py
test_wrong_connection_string
TestBlocksBasicRequestsDb
test_insert_parent_child_relationship
test_delete_parent_child_relationship
test_delete_selected_parent_child_relationships
test_get_last_order_index_of_parent_block
test_get_new_last_order_index_of_parent_block
test_get_order_index_of_block_in_parent
test_get_previous_order_index
test_get_next_order_index
test_get_order_index_before_bounds
test_get_order_index_before
test_get_order_index_before_with_recalculation
test_get_order_index_after_bounds
test_get_order_index_after
test_get_order_index_after_with_recalculation
test_connect_to_parent_of_target
test_root_children_reorder
test_reorder_blocks
test_reparent_blocks
test_insert_new_block
test_build_subtree_cte
test_build_parent_counts_cte
test_build_to_delete_cte
test_delete_block_1
test_delete_block_2
test_delete_block_3
test_update_block_text
test_toggle_block_is_done
test_toggle_block_is_open
test_get_root_page_data_model
test_get_one_block_data_model
test_get_opened_root_item
test_get_opened_root_item_2
test_get_opened_item_content
TestRootAndFavoritePagesDb
test_insert_top_level_page_to_end
test_insert_many_top_level_pages_to_end
test_delete_specific_top_level_page
test_delete_top_level_page
test_add_page_to_favorites
test_remove_page_from_favorites
test_select_no_favorites
test_root_and_favorites_query_failure

Рисунок *20 — Результати виконання тестів backend-частини*

3.7.2 Тестування клієнтської частини

Для unit тестів допоміжного контролера відкритої сторінки клієнтської частини був використаний фреймворк Vitest [32]. Vitest надає наступні можливості:

- Налаштування конфігурацій тестів у Vite's config файлі.
- Тестування компонентів різних UI бібліотек.
- Сумісність з Chai and Jest expect.
- Mocking.
- Тестове покриття.
- Сумісність з Playwright для end-to-end тестування.

Були часткова протестовані основні операції, які виконуються у контролері відкритої сторінки для більшої впевненості у правильності обробки даних та команд контролером, а також для швидкого пошуку помилок при їх виникнені та швидшому визначенні джерела помилки (frontend or backend).

Результати тестування за покриттям statement та branches представлені на рисунках 21, 22, результати виконання тестів — на рисунку 23.

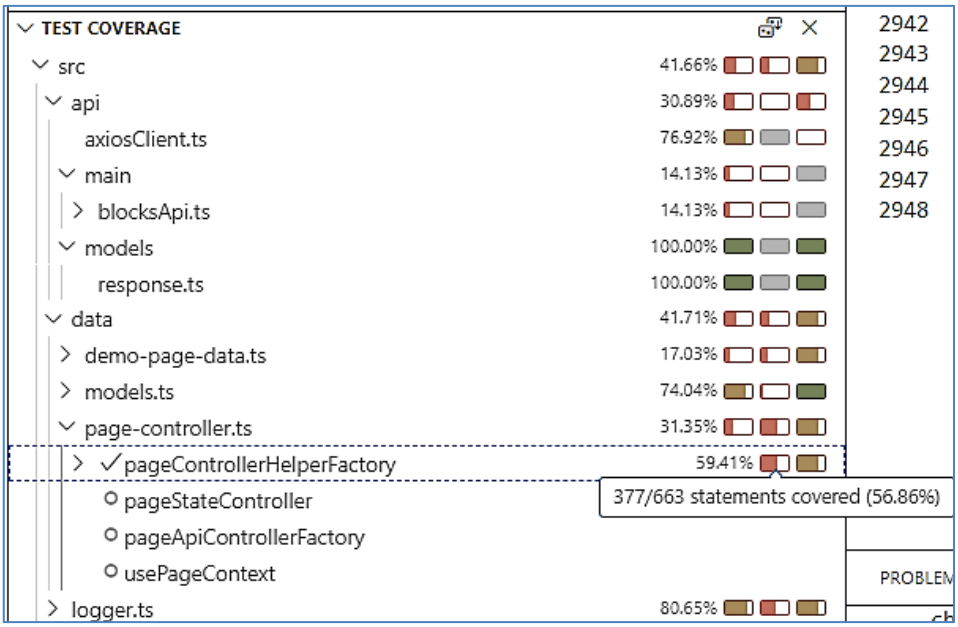


Рисунок *21 — Результати тестування запитів до бази даних за покриттям statement frontend-частини*

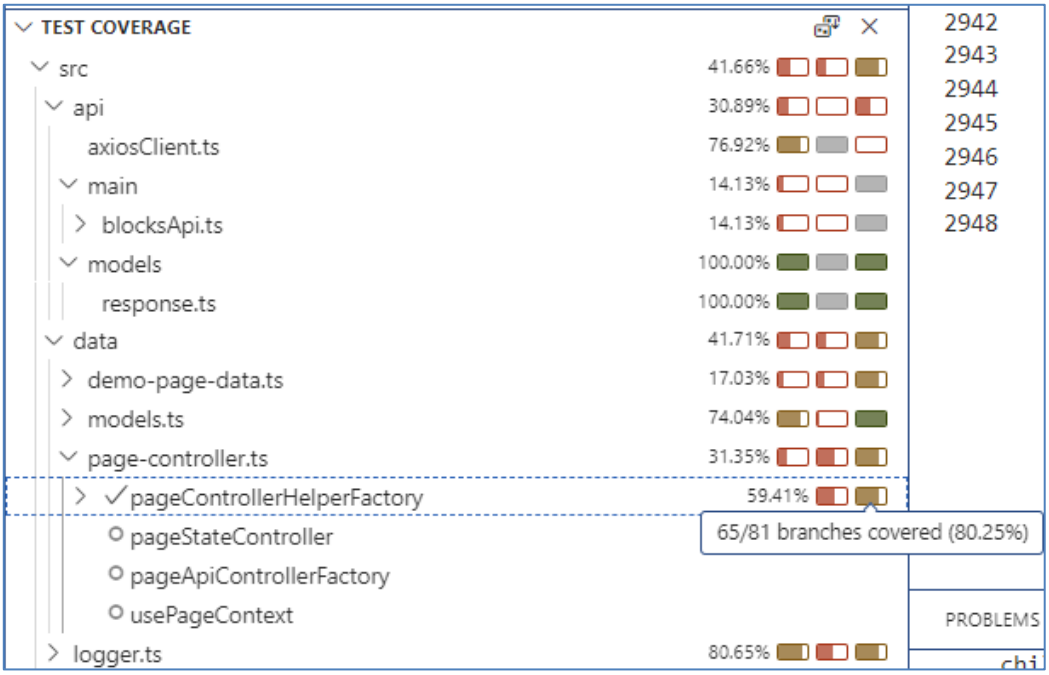


Рисунок *22 — Результати тестування запитів до бази даних за покриттям branches frontend-частини*

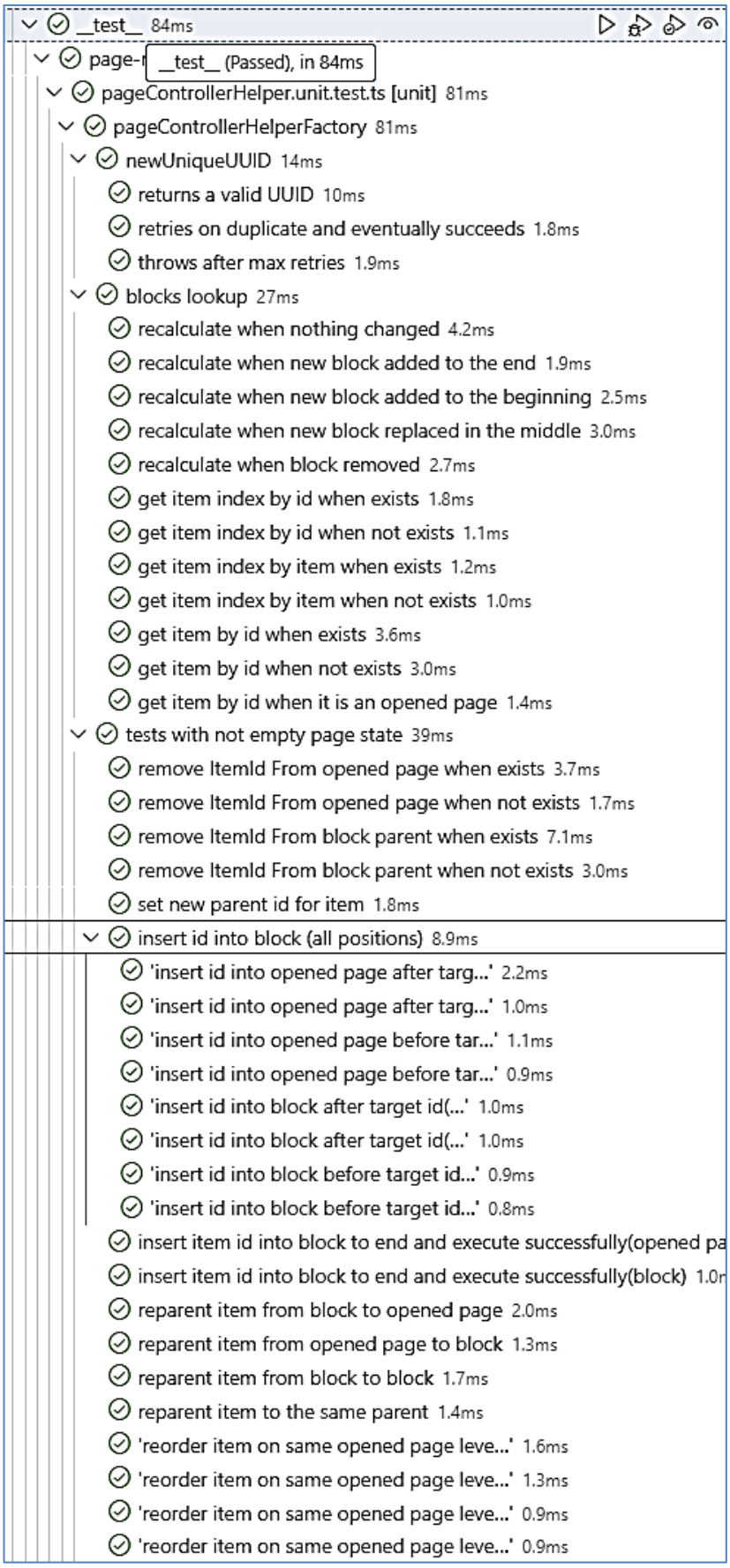


Рисунок *23 — Результати виконання тестів frontend-частини*

3.8 Висновки до розділу 3

1. Архітектура системи складається з двох частин: застосунок, який встановлює користувач та сервер, який обробляє запити.
2. Була описана архітектура клієнтського застосунку з використанням Solidjs.
3. Була описана схема БД та її тип Class Table Inheritance.
4. Був описаний процес обробки запитів на сервері.
5. Визначені функціональні вимоги, а також вимоги до інтерфейсу.
6. Проведено часткове тестування серверної та клієнтської частин.

ВИСНОВКИ

1. Результати аналізу літературних джерел, а також власний досвід свідчать про те, що у період інформаційного перевантаження людині потрібні інструменти для полегшення організації та візуалізації проєктної діяльності.
2. Проведено аналіз існуючих систем: Notion, Obsidian, ClickUp, Todoist. Визначені їх позитивні та негативні сторони, їх можливості та особливості.
3. Розроблено архітектуру системи, архітектуру клієнтської та серверної частин та їх взаємодію, сплановано базу даних.
4. Створено робочий варіант програмної системи, який демонструє можливість підвищити продуктивність працівників і зменшити інформаційне перевантаження та ментальну втому. Система розроблена з використанням фреймворків Tauri, SolidJS, FastAPI

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Greg Mckeown (2014). Essentialism: The Disciplined Pursuit of Less. Virgin Books. 272 p.
2. Chris Bailey (2018). Hyperfocus: How to Be More Productive in a World of Distraction. Pan MacMillan. 256 p.
3. Tauri documentation. URL: https://v2.tauri.app (дата звернення: 26.03.2025 р.).
4. FastAPI documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 26.03.2025 р.).
5. Solidjs documentation. URL: https://www.solidjs.com/ (дата звернення: 26.03.2025 р.)
6. DearPyGui documentation. URL: https://github.com/hoffstadt/DearPyGui (дата звернення: 27.04.2025 р.)
7. Nuklear documentation. URL: https://github.com/Immediate-Mode-UI/Nuklear (дата звернення: 27.04.2025 р.)
8. Qt 6 documentation. URL: https://www.qt.io/ (дата звернення: 26.03.2025 р.).
9. Electron documentation. URL: https://www.electronjs.org/ (дата звернення: 26.03.2025 р.).
10. React documentation. URL: https://react.dev/ (дата звернення: 26.03.2025 р.).
11. Vue documentation. URL: https://vuejs.org/ (дата звернення: 26.03.2025 р.).
12. Laravel documentation. URL: https://laravel.com/ (дата звернення: 26.03.2025 р.).
13. ExpressJS documentation. URL: https://expressjs.com/ (дата звернення: 26.03.2025 р.).
14. Node.js documentation. URL: https://nodejs.org/en (дата звернення: 26.03.2025 р.).
15. Starlette documentation. URL: https://www.starlette.io/(дата звернення: 26.03.2025 р.).
16. Uvicorn documentation. URL: https://www.uvicorn.org/

17. Swagger-ui GitHub repository. URL: https://github.com/swagger-api/swagger-ui (дата звернення: 26.03.2025 р.).
18. PostgreSQL documentation. URL: https://www.postgresql.org/ (дата звернення: 26.03.2025 р.).
19. Cassandra documentation. URL: https://cassandra.apache.org/doc/latest/ (дата звернення: 26.03.2025 р.).
20. Dgraph documentation. URL: https://docs.hypermode.com/dgraph/overview (дата звернення: 26.03.2025 р.).
21. Pragmatic drag and drop GitHub repository. URL: https://atlassian.design/components/pragmatic-drag-and-drop/about (дата звернення: 26.03.2025 р.).
22. Konva documentation. URL: https://konvajs.org/docs/index.html (дата звернення: 26.03.2025 р.).
23. ECharts documentation. URL: https://echarts.apache.org/en/index.html
24. D3 documentation. URL: https://d3js.org/ (дата звернення: 26.03.2025 р.).
25. UV project manager GitHub repository. URL: https://github.com/astral-sh/uv (дата звернення: 26.03.2025 р.).
26. ArkType documentation. URL: https://arktype.io/ (дата звернення: 26.03.2025 р.).
27. What is Podman? article. URL: https://www.redhat.com/en/topics/containers/what-is-podman (дата звернення: 26.03.2025 р.).
28. What is Testcontainers? article. URL: https://testcontainers.com/(дата звернення: 26.03.2025 р.).
29. Setup Testcontainers with Podman documentation. URL: https://podman-desktop.io/tutorial/testcontainers-with-podman (дата звернення: 26.03.2025 р.).
30. pytest documentation. URL: https://docs.pytest.org/en/stable/index.html (дата звернення: 26.03.2025 р.).

31. Hypothesis documentation. URL: https://hypothesis.readthedocs.io/en/latest/ (дата звернення: 26.03.2025 р.).
32. Vitest documentation. URL: https://vitest.dev/(дата звернення: 26.03.2025 р.).
33. Посунько М., бакаласр, Безверхий А.І. доцент — науковий керівник. Комп’ютерна система візуалізації моніторингу проєктної діяльності. // ЗБІРНИК наукових праць студентів, здобувачів PhD, докторантів і молодих вчених «Молода наука-2025». Запоріжжя : Запорізький національний університет, 2025. Т. 5. С. 408–409.