

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

Факультет комп'ютерних наук і технологій

(повне найменування факультету)

Кафедра програмних засобів

(повне найменування кафедри)

Пояснювальна записка

до дипломного проекту (роботи)

бакалавр

(ступінь вищої освіти)

на тему ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ
ПЛАНУВАННЯ ЗАВДАНЬ ІЗ ЗАБЕЗПЕЧЕННЯМ ОФЛАЙН РОБОТИ
DESIGN AND DEVELOPMENT OF A WEB APPLICATION TO PLAN
TASKS WITH OFFLINE MODE OPTION

Виконав(ла): студент(ка) 4 курсу, групи КНТ-141

Спеціальності 121 Інженерія програмного

(код і найменування спеціальності)

забезпечення

Освітня програма (спеціалізація)

Інженерія програмного забезпечення

МІЩЕНКО В. М.

(ПРИЗВИЩЕ та ініціали)

Керівник ШИТІКОВА О.В.

(ПРИЗВИЩЕ та ініціали)

Рецензент ТИМЕНКО К.І.

(ПРИЗВИЩЕ та ініціали)

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
 (повне найменування закладу вищої освіти)

Факультет КНТ
 Кафедра програмних засобів
 Ступінь вищої освіти бакалавр
 Спеціальність 121 Інженерія програмного забезпечення
(код і найменування)
 Освітня програма (спеціалізація) Інженерія програмного забезпечення
(назва освітньої програми (спеціалізації))

ЗАТВЕРДЖУЮ

Завідувач кафедри ПЗ, д.т.н, проф.
Сергій СУББОТІН
 “ ” 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА(КИ)

МІЩЕНКА Володимира Михайловича
(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Проектування та розробка вебзастосунку для
планування завдань із забезпеченням офлайн роботи. Design and Development of
a Web Application to Plan Tasks with Offline Mode Option

керівник проєкту (роботи) канд. техн. наук, ШИТИКОВА Олена Вікторівна,
(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом закладу вищої освіти від “28” квітня 2025 року № 209

2. Строк подання студентом проєкту (роботи) 02 червня 2025 року

3. Вихідні дані до проєкту (роботи) рекомендована література, технічне
завдання

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно
 розробити) 1. Аналіз. 2. Обрання засобів та середовища розробки.
3. Проектування системи. 4. Реалізація.

5. Перелік графічного матеріалу (з з точним зазначенням обов'язкових креслень,
 кількість слайдів, плакатів)

Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРИЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4 Основна частина	ШИТІКОВА О.В., доцент		
Нормоконтроль	КАЛІНІНА М.В., асистент		

7. Дата видачі завдання « 14 » квітня 2025 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Постановка завдання роботи.	1 тиждень	Завдання, ТЗ
2	Аналіз предметної області.	1 тиждень	Розділ 1
3	Вибір мови програмування та інших технологій розробки.	2 тиждень	Розділ 2
4	Розробка архітектури програми.	2 тиждень	Розділ 3
5	Розробка програми.	3-4 тижні	Розділ 3, 4
6	Тестування та експериментальне дослідження програмного забезпечення.	5 тиждень	Розділ 4
7	Оформлення пояснювальної записки та документів до неї.	6 тиждень	Додатки
8	Нормоконтроль та рецензування.	7 тиждень	
9	Захист роботи.	8 тиждень	

Студент(ка)

(підпис) Володимир МІЩЕНКО
(Ім'я ПРИЗВИЩЕ)

Керівник проєкту (роботи)

(підпис) Олена ШИТІКОВА
(Ім'я ПРИЗВИЩЕ)

РЕФЕРАТ

Пояснювальна записка до дипломної кваліфікаційної роботи бакалавра: 86 с., 5 табл., 27 рис., 2 дод., 24 джерела.

ВЕБЗАСТОСУНОК, PWA, УПРАВЛІННЯ ЗАВДАННЯМИ, REACT, NODE.JS.

Об'єкт дослідження – процес організації управління завданнями у вебсередовищі.

Предмет дослідження – програмна реалізація вебзастосунку для керування задачами з підтримкою автономного режиму роботи (PWA).

Мета роботи – розробити зручний та адаптивний вебзастосунок для створення, редагування, планування і контролю виконання завдань, здатний підвищити ступінь автономності завдяки технології PWA.

Матеріали, методи та технічні засоби: технології front-end та back-end розробки, технологія PWA (Service Worker, IndexedDB, localStorage), бібліотеки JavaScript (React, @hello-pangea/dnd, framer-motion), Node.js, Express.js, СКБД MongoDB або PostgreSQL, персональний комп'ютер з ОС Windows 11 та середовище розробки Visual Studio Code.

Результати. Розроблено вебзастосунок у форматі PWA, що забезпечує автономність, ефективну організацію завдань і можливість роботи без постійного з'єднання з мережею, а також надає користувачам контроль над власними даними.

Висновки. Створений застосунок ефективно вирішує проблему управління задачами з підтримкою автономного та онлайн режимів роботи, є простим у використанні та забезпечує високий рівень приватності даних користувача.

Галузь використання – індивідуальне та командне управління завданнями для навчання, роботи, особистого планування в умовах нестабільного або відсутнього Інтернет-з'єднання.

ABSTRACT

Explanatory note to the diploma qualifying work of the bachelor: 86 pages, 5 tables, 27 figures, 2 appendixes, 24 sources.

WEB APPLICATION, PWA, TASK MANAGEMENT, REACT, NODE.JS.

The object of research is the process of organizing task management in the web environment.

The subject of research is the software implementation of a web application for task management with support for offline operation (PWA).

The purpose of the study is to develop a convenient and adaptive web application for creating, editing, scheduling and monitoring the execution of tasks that can increase the degree of autonomy through PWA technology.

Materials, methods and techniques: front-end and back-end development technologies, PWA technology (Service Worker, IndexedDB, localStorage), JavaScript libraries (React, @hello-pangea/dnd, framer-motion), Node.js, Express.js, MongoDB or PostgreSQL databases, a personal computer with Windows 11 and the Visual Studio Code development environment.

Results. A web application in the PWA format has been developed that provides autonomy, efficient organization of tasks and the ability to work without a constant connection to the network, as well as gives users control over their own data.

Conclusions. The created application effectively solves the problem of task management with support for offline and online modes of operation, is easy to use and provides a high level of user data privacy.

The field of application is individual and team task management for study, work, personal planning in conditions of unstable or absent Internet connection.

ЗМІСТ

	С.
Перелік скорочень та умовних познач.....	7
Вступ.....	9
1 Аналіз предметної області.....	11
1.1 Опис предметної області	11
1.2 Аналіз аналогів	12
1.3 Постановка завдання.....	20
2 Обрання засобів та середовища розробки	21
2.1 Аналіз технологій front-end.....	21
2.2 Докладний порівняльний аналіз технологій	24
2.3 Інструментарій та середовище розробки	27
2.4 Аналіз технологій бекенду	27
2.5 Аналіз баз даних	32
2.6 Обґрунтування вибору технологічного стеку	33
3 Проєктування системи.....	36
3.1 Призначення та мета системи	36
3.2 Архітектура вебзастосунку	37
3.3 Проєктування бази даних	40
3.4 Структура програмного забезпечення	44
4 Реалізація та тестування	49
4.1 Реалізація серверної частини (API)	49
4.2 Реалізація клієнтської частини (інтерфейс)	54
4.3 Оновлення у реальному часі через WebSocket	61
4.4 Тестування функціональності застосунку	62
Висновки	67
Перелік джерел посилання	69
Додаток А Приклади коду програми	71
Додаток Б Слайди презентації	81

ПЕРЕЛІК СКОРОЧЕНЬ ТА УМОВНИХ ПОЗНАК

API	– Application Programming Interface;
ACID	– Atomicity, Consistency, Isolation, Durability;
CLI	– Command Line Interface;
CLR	– Common Language Runtime;
CORS	– Cross-Origin Resource Sharing;
CRUD	– Create, Read, Update, Delete;
CSS	– Cascading Style Sheets;
CSRF	– Cross-Site Request Forgery;
CSR	– Client-Side Rendering;
DB	– DataBase;
DI	– Dependency Injection;
DOM	– Document Object Model;
ENUM	– Enumerated Type
GIL	– Global Interpreter Lock;
HTML	– HyperText Markup Language;
HTTP	– HyperText Transfer Protocol;
ID	– Identifier;
IDE	– Integrated Development Environment;
I/O	– Input/Output;
JS	– JavaScript;
JSON	– JavaScript Object Notation;
JSONB	– JavaScript Object Notation Binary;
MVC	– Model-View-Controller;
MVVM	– Model-View-ViewModel;
MVT	– Model-View-Template,
npm	– Node Package Manager;
NoSQL	– Not Only SQL;
OAuth	– Open Authorization;

ORM	– Object-Relational Mapping;
PWA	– Progressive Web Application;
REST	– Representational State Transfer;
RSS/Atom	– Rich Site Summary / Atom Syndication Format;
SPA	– Single Page Application;
SQL	– Structured Query Language;
SSR	– Server-Side Rendering;
TLS	– Transport Layer Security;
TS	– TypeScript;
UI	– User Interface;
UML	– Unified Modeling Language;
URI	– Uniform Resource Identifier;
URL	– Uniform Resource Locator;
UUID	– Universally Unique Identifier;
UX	– User Experience;
WS	– WebSocket
XSS	– Cross-Site Scripting;
БД	– База Даних;
СУБД	– Система Управління Базами Даних.

ВСТУП

У сучасних умовах, коли значна кількість завдань виконується через вебінтерфейси, особливо актуальними стають застосунки для ефективного керування особистими й робочими справами. Таск-менеджери, що забезпечують створення, планування і контроль виконання завдань, є необхідними інструментами для широкого кола користувачів — від студентів і фрилансерів до фахівців у сфері інформаційних технологій. Такі вебзастосунки суттєво допомагають оптимізувати процеси роботи й підвищують продуктивність діяльності.

Попри різноманітність існуючих програмних засобів для управління завданнями, більшість з них мають низку суттєвих недоліків, серед яких — надлишкова складність інтерфейсу, обмежений функціонал у безкоштовних версіях і головне — залежність від постійного та стабільного підключення до мережі Інтернет. У реальних умовах використання, коли стабільність інтернет-з'єднання може бути обмежена або відсутня, користувачі стикаються з проблемами, що знижують ефективність таких інструментів.

У зв'язку з цим ідея створення вебзастосунку для керування завданнями з можливістю повноцінної роботи в автономному режимі (офлайн-режимі) набуває особливої актуальності. Вирішення цієї задачі можливо завдяки використанню технології прогресивних вебзастосунків (PWA), що дозволяє забезпечити стабільну та незалежну від інтернет-з'єднання роботу програми.

Об'єктом дослідження цієї роботи є процес організації управління завданнями у вебсередовищі. Предметом дослідження виступає програмна реалізація вебзастосунку з підтримкою автономного режиму роботи.

Метою роботи є розробити зручний та адаптивний вебзастосунок для створення, редагування, планування і контролю виконання завдань, здатний підвищити ступінь автономності завдяки технології PWA. Для досягнення поставленої мети передбачено виконати аналіз існуючих аналогів, визначити

оптимальний технологічний стек, спроектувати архітектуру застосунку, реалізувати основний функціонал та перевірити працездатність створеного рішення.

Практична цінність роботи полягає у створенні адаптивного, зручного та надійного вебінструменту, який надає користувачу можливість ефективно планувати й керувати завданнями незалежно від доступності інтернет-з'єднання, що є особливо важливим у сучасних умовах динамічного життя й роботи.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Опис предметної області

Сучасне цифрове середовище характеризується швидким темпом життя, великою кількістю завдань та необхідністю ефективної організації часу як для особистих, так і для робочих цілей. У таких умовах особливої актуальності набувають вебзастосунки, які дозволяють зручно створювати, планувати та контролювати виконання завдань на різноманітних пристроях з доступом до мережі Інтернет.

Традиційні вебзастосунки мають істотні недоліки, серед яких найбільш помітною є їх залежність від стабільного інтернет-з'єднання. В умовах нестабільного зв'язку або відсутності доступу до мережі, користувачі стикаються з суттєвими труднощами, що значно погіршує комфортність та продуктивність роботи [1].

Рішенням цієї проблеми стали прогресивні вебзастосунки (Progressive Web Applications, PWA), які завдяки спеціальним технологіям, таким як Service Workers, IndexedDB і localStorage, можуть забезпечувати повноцінну роботу в офлайн-режимі. Service Workers виступають посередниками між браузером і мережею, забезпечуючи кешування ресурсів та доступ до інформації без підключення до Інтернету. IndexedDB дозволяє локально зберігати складні дані, забезпечуючи високий рівень ефективності роботи, а localStorage використовується для зберігання простіших даних в межах браузера [2].

Таким чином, використання технологій PWA дозволяє створити вебзастосунок, який працює ефективно навіть без доступу до Інтернету, з подальшою синхронізацією даних, коли з'єднання стає доступним. Це робить PWA особливо актуальним для застосунків управління завданнями, де безперервність і доступність інформації відіграють ключову роль у зручності та продуктивності.

Ілюстративним прикладом ефективної роботи PWA є схема взаємодії між браузером, Service Worker і кешем (рис. 1.1), яка демонструє принцип роботи технології та її переваги для кінцевого користувача [2].

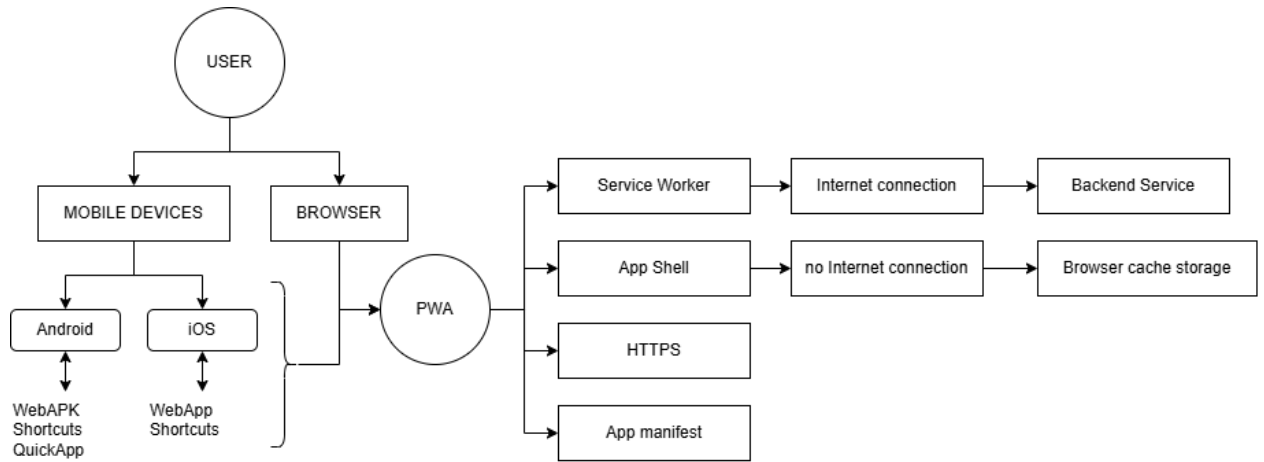


Рисунок 1.1 — Схема роботи PWA [2]

Отже, створення прогресивного вебзастосунку для керування завданнями, який ефективно функціонує в офлайн-режимі, є актуальним та відповідає сучасним потребам користувачів, забезпечуючи високу автономність та стабільність роботи незалежно від умов мережевого підключення.

1.2 Аналіз аналогів

На сьогодні існує значна кількість вебзастосунків для управління завданнями, які активно використовуються у різних сферах діяльності. Розглянемо найбільш популярні та поширені серед користувачів: Todoist, Trello, Google Tasks, Microsoft To Do та Any.do.

Todoist

Todoist є одним із лідерів серед інструментів для управління завданнями. Це кросплатформенний органайзер з багатьма функціями: він підтримує розбивку завдань на проєкти й підпроєкти, пріоритети, мітки

(теги), фільтри та повторення завдань, приклад стандартного головного екрану Todoist можна переглянути на ілюстрації (рисунок 1.2). Можна швидко додавати завдання за допомогою обробки природньої мови, налаштовувати дедлайни, чек-листи й коментарі до завдань. Доступне спільне використання проєктів і призначення завдань колегам. Серед сильних сторін – простий та зрозумілий інтерфейс, синхронізація на всіх пристроях, зручна статистика виконання та багато вбудованих інтеграцій (з календарями, нотатками, чатами тощо). Todoist може працювати офлайн: у вебверсії та настільному застосунку режим офлайн активується автоматично при відсутності мережі [3].

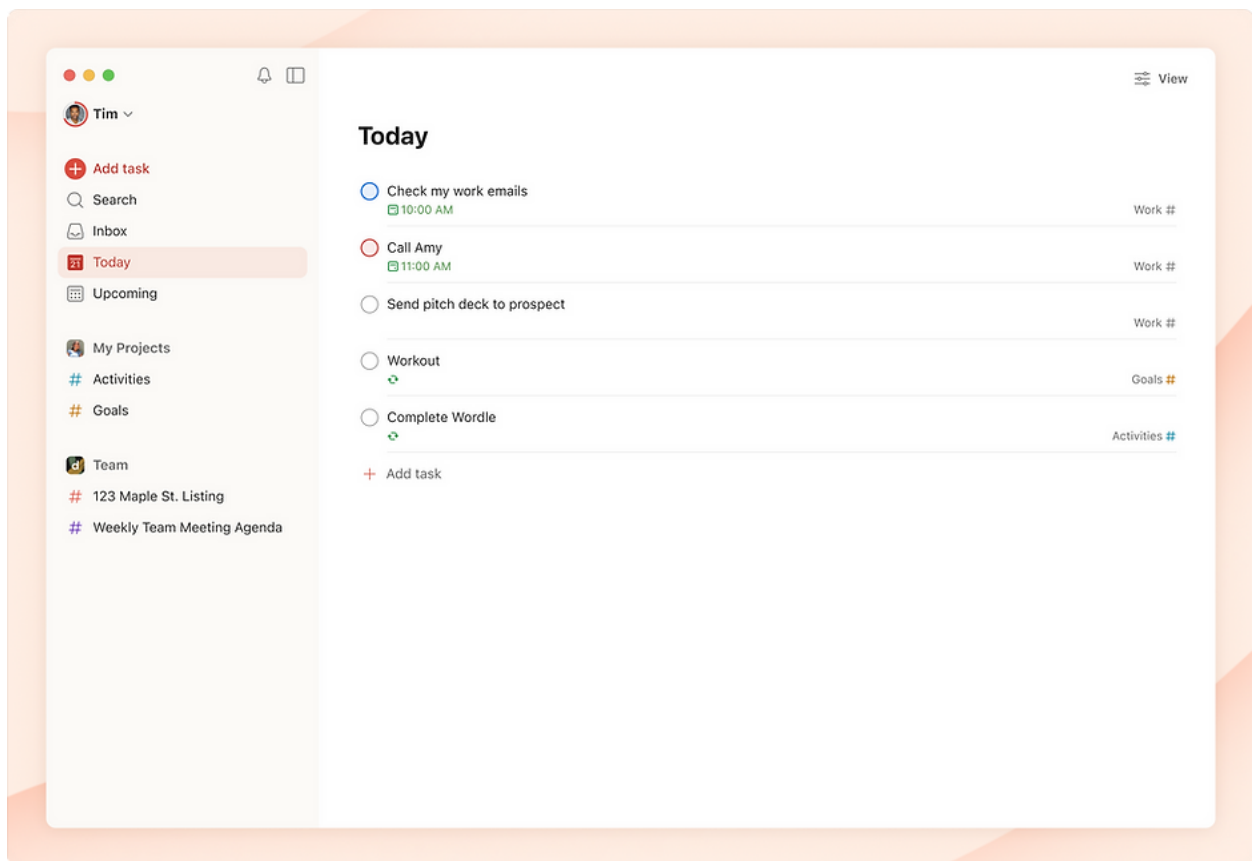


Рисунок 1.2 — Приклад інтерфейсу Todoist [4]

Це зручно для мобільного та десктопного використання без доступу до мережі. Інтеграції – дуже широкі: застосунок підтримує зв’язок зі службами Google (Календар, Gmail), Slack, Outlook, Zapier та багатьма

іншими. Безкоштовна версія обмежена: напр., у ній можна створювати обмежену кількість активних проєктів, відсутні нагадування та повна історія завдань; деякі важливі функції (напр. кілька шаблонів чи інтеграції) доступні тільки у платній версії. Вхід можна виконувати через електронну пошту чи OAuth від Google. Але варто пам'ятати, що дані зберігаються на серверах Doist (європейська компанія), і їх безпека залежить від загальних практик хмарного сховища [4].

Недоліками є необхідність попередньої авторизації для повноцінної офлайн-роботи, обмежена функціональність безкоштовної версії та низька конфіденційність через централізоване зберігання даних, безкоштовні обмеження. Базова робота з підзавданнями досить обмежена. Також деякі користувачі вважають корпоративне ліцензування – доволі дорогим.

Trello

Trello здобув популярність завдяки простій та ефективній системі канбан-дошок (рисунок 1.3), яка дозволяє легко візуалізувати процеси виконання завдань та організовувати командну роботу. Робочий простір побудований навколо канбан-дошок: у ньому є дошки, списки та картки, що імітують візуальне планування задач. Користувачі можуть перетягувати картки між колонками, додавати описи, коментарі, чеклісти, терміни, файли та призначати відповідальних. Основними перевагами Trello є інтуїтивно зрозумілий інтерфейс, можливість командної співпраці у реальному часі та велика кількість сторонніх інтеграцій з такими сервісами, як Google Drive, Dropbox та Jira [5].

Багато корисних інтеграцій у Trello реалізовано через Power-Ups: календар, таблиці, чати в Slack, додатки Atlassian тощо. Безкоштовна версія надає практично необмежену кількість карток і бордів, але обмежує кількість Power-Up на дошку (часто один), обсяг завантажень і відсутність корпоративних ролей (Teams). Усі нові борди за замовчуванням є приватними, тобто видимі лише учасникам, а профіль користувача показує тільки ім'я та ім'я користувача (email не афішується) [6].

Базова організація даних (тільки канбан), та обмежені можливості безкоштовного плану (обмежена кількість Power-Ups та автоматизацій). Trello не підтримує офлайн у веб/десктопній версії: на сайті Trello потрібне підключення до Інтернету, і якщо з'єднання обірвалося, застосунок лише повідомить про відсутність мережі і дочекається відновлення для синхронізації.

Натомість останні версії мобільних застосунків (Android/iOS) мають розвинений офлайн-режим: у них можна створювати дошки, картки, змінювати статуси та інше без доступу до мережі, а при поверненні онлайн всі зміни синхронізуються автоматично [7].

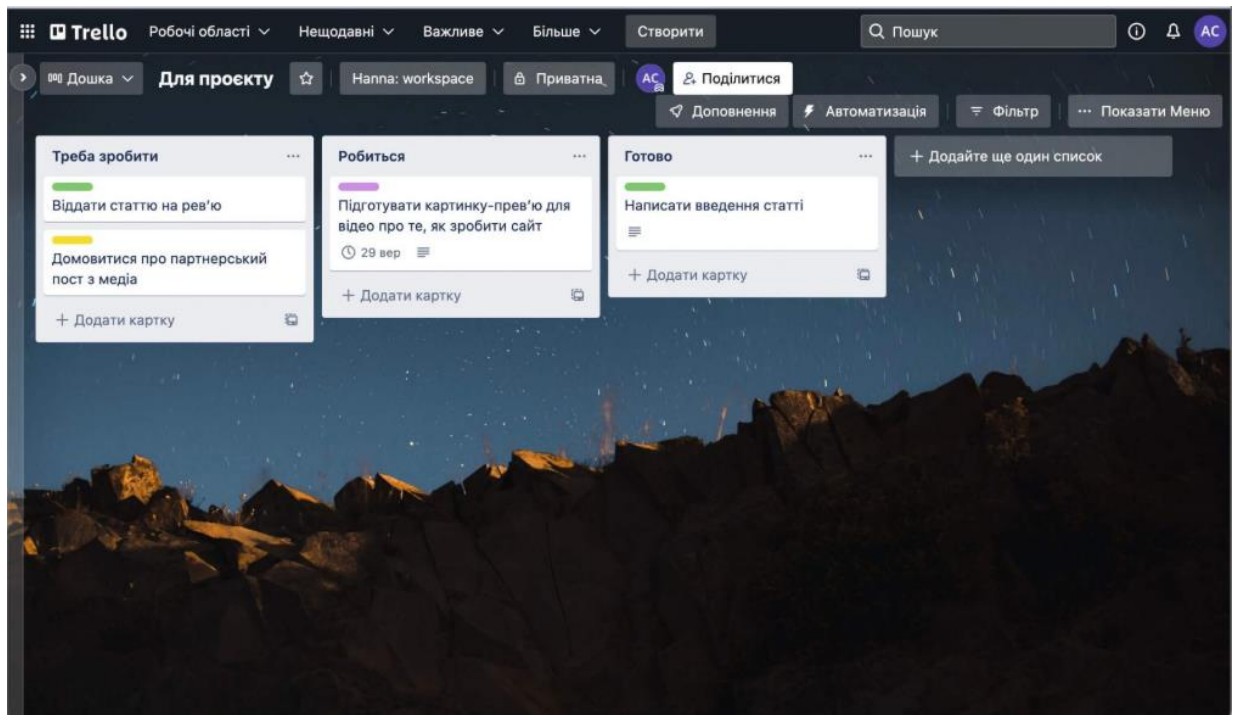


Рисунок 1.3 — Приклад інтерфейсу Trello [8]

Google Tasks

Google Tasks – найпростіший зі списку вебзастосунків, також інтегрований у Gmail і Google Calendar. Він дозволяє створювати нескладні списки завдань і підзавдань, встановлювати дедлайни та нагадування (рисунок 1.4). Особливість – робота в рамках екосистеми Google: задачі видно у бічному меню Gmail та Calendar, а також у мобільному застосунку

Google Tasks. Сильні сторони – повна безкоштовність і глибока інтеграція з Google (автоматична синхронізація з Gmail/Calendar та Google Assistant).

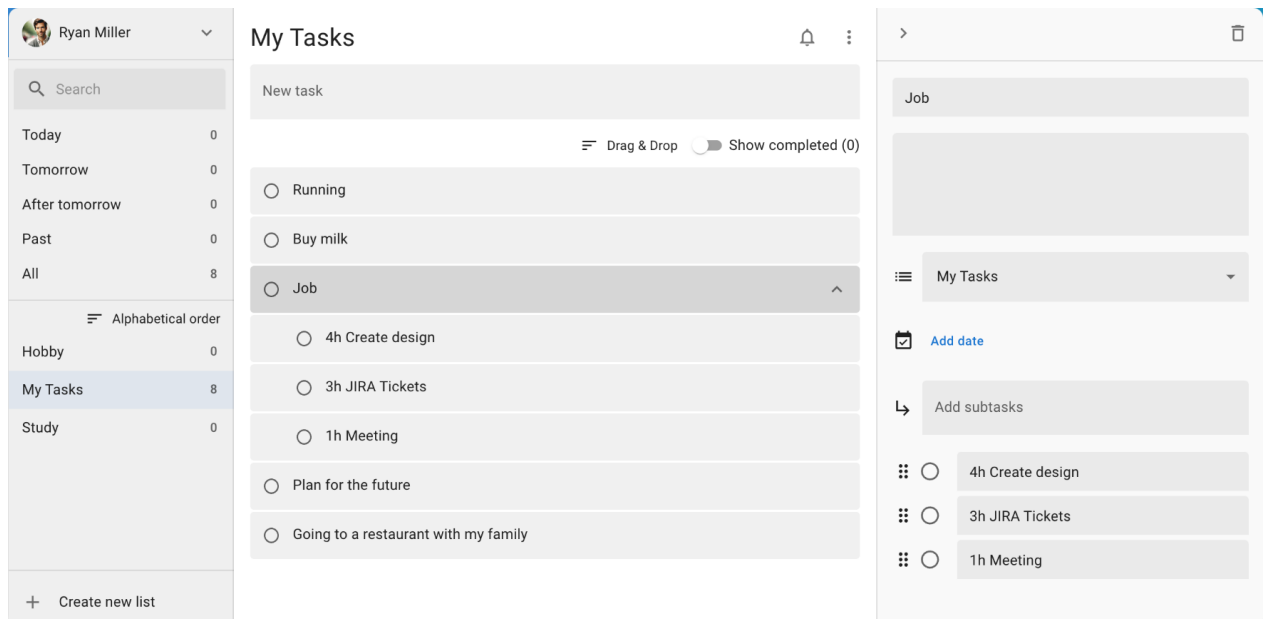


Рисунок 1.4 — Приклад інтерфейсу Google Tasks [9]

Офлайн-режим у Google Tasks офіційно не документовано, але деякі джерела стверджують, що доступ до задач можна отримати офлайн, якщо увімкнено офлайн-доступ у Gmail/Drive. З іншого боку, оскільки Tasks підпорядковано обліковому запису Google, типовий сценарій – використання з підключенням до Інтернету. Інтеграції обмежуються Google-сервісами: задачі можна переглядати в Calendar та інших додатках Workspace, але інші сервіси (Slack, Trello) не підключаються напряму.

Google Tasks безкоштовний для всіх користувачів Google, але має дуже обмежений функціонал (немає кольорових міток, розширених фільтрів, коментування чи спільного редагування списків). Перевага – повністю безоплатне довічне використання й нескінченна історія виконаних завдань (на відміну від Todoist Free, що зберігає виконані завдання лише тиждень) [10].

З погляду приватності, Google забезпечує шифрування даних у дорозі та «на зберіганні», а доступ до нього мають лише ви (крім випадків, коли ви

вручну ділитися задачею через простори Gmail Chat). Недоліком є те, що Tasks не має власного інтерфейсу за межами Gmail/Calendar, і інформація зберігається в хмарі Google, що може викликати сумніви щодо приватності у деяких користувачів (хоча Google підтримує прозорість політик, освітньо-пояснює доступ до даних [11]).

Функціональність Google Tasks обмежена і не підходить для складних завдань або великомасштабної командної роботи. Вебверсія практично не підтримує офлайн-режим, що суттєво обмежує використання застосунку у випадках нестабільного підключення до мережі.

Ще одним обмеженням Google Tasks є відсутність можливості структурувати завдання за пріоритетністю. Усі задачі зберігаються у вигляді простого списку без ієрархічного поділу. Крім того, Google Tasks не підтримує прикріплення файлів чи зображень до задач, що робить його менш зручним для користувачів, які потребують додаткових контекстуальних даних.

Microsoft To Do

Microsoft To Do – безкоштовний застосунок, тісно інтегрований з Microsoft 365. Він дозволяє створювати прості списки завдань, призначати дати й нагадування, розбивати на кроки (підзавдання) і переносити плани на день «Мій день». Ілюстрація на рисунку демонструє приклад інтерфейсу Microsoft To Do (рисунок 1.5). Сильна сторона – інтеграція з Outlook: завдання можна переглядати і управляти ними як з вебінтерфейсу To Do, так і в бічній панелі Outlook або на телефонах і в Windows. Користувачі можуть ділитися списками з іншими (спільні списки) та присвоювати завдання учасникам. Офлайн-режиму як такого спеціально нема: вебверсія To Do вимагає підключення, а мобільний або десктопний клієнт може кешувати останню синхронізацію (деякі користувачі помічали, що без інтернету мобільний додаток не дає повноцінно працювати). У будь-якому разі дані синхронізуються через хмару Microsoft. Інтеграції нечисленні: To Do вбудований у Microsoft 365 (Outlook, Planner, Teams), але немає можливості

напрямку підключити, скажімо, Google Calendar чи Slack без сторонніх сервісів [12].

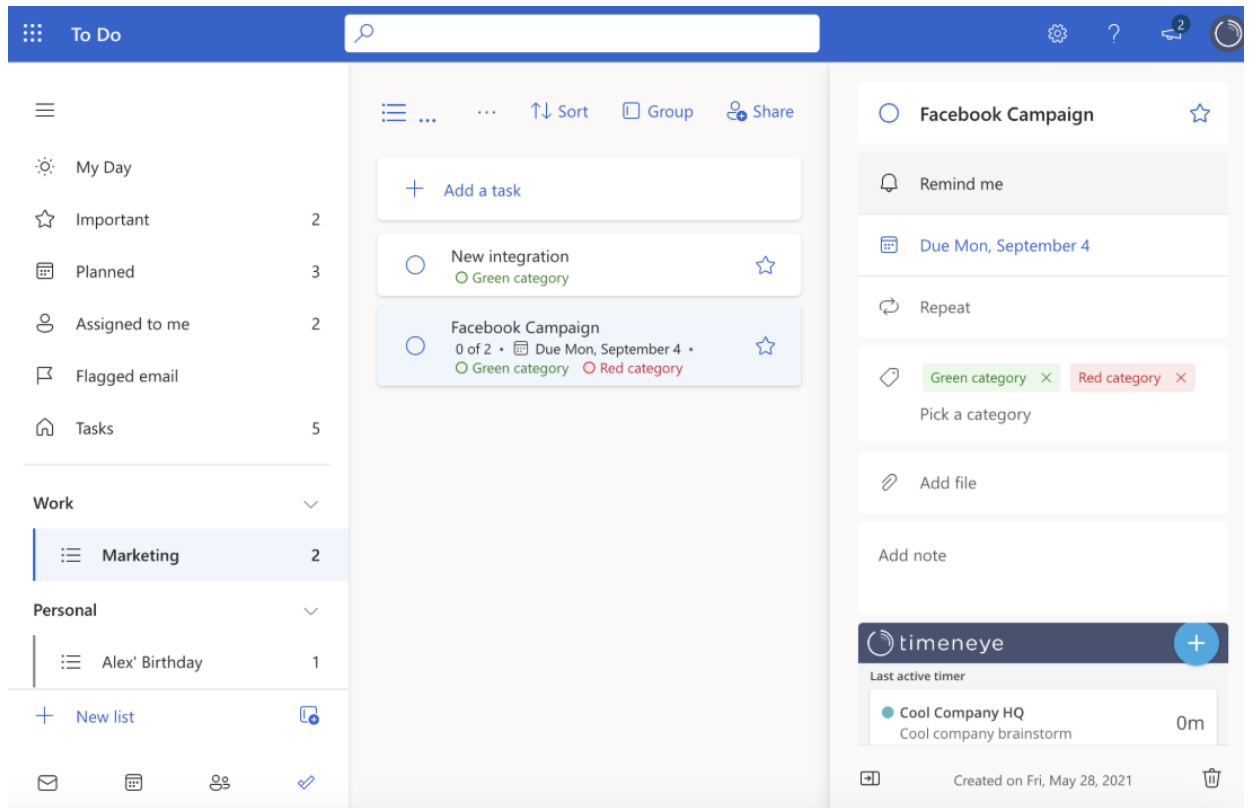


Рисунок 1.5 — Приклад інтерфейсу Microsoft To Do [13]

Безкоштовна версія Microsoft To Do є одна на всіх – достатня для особистого користування. Весь функціонал To Do, зокрема шаблони та спільні списки, розблоковується при наявності акаунта Microsoft (Outlook.com чи корпоративний). Обмежень щодо кількості завдань чи списків зазвичай немає. Щодо приватності та безпеки – Microsoft заявляє, що дані шифруються та зберігаються в Exchange Online, але експертні огляди відзначають проблеми: наприклад, мережеві атаки на сервери Exchange у 2021 році, де фактично зберігаються дані To Do. Загалом Cloudwards дає низьку оцінку конфіденційності Microsoft, зазначаючи, що Windows 10 збирає багато даних і може використовувати їх для реклами. Таким чином, To Do варто використовувати з обережністю, якщо ви плануєте зберігати дуже конфіденційні замітки в хмарі Microsoft. Слабкі місця To Do –

мінімалізм та відсутність розширених функцій: немає таймлайну чи канбан-інструментів, жорстка прив'язка до стека Microsoft, а мобільні клієнти часом «гальмують» при синхронізації чи без зв'язку [14].

Тому мобільні додатки демонструють добру підтримку автономної роботи, проте вебверсія має суттєві обмеження щодо офлайн-функціоналу. Також існує проблема з конфіденційністю даних через централізоване зберігання інформації на серверах Microsoft, що створює ризики для приватності користувачів. Також є обмеження у налаштуваннях безкоштовної версії.

Для налядності порівняння роглянутих вебзастосунків було створено порівняльну таблицю (Табл 1.1).

Таблиця 1.1 — Порівняння основних характеристик вебзастосунків Todoist, Trello, Google Tasks, Microsoft To Do та власного застосунку

Характеристика	Todoist	Trello	Google Tasks	Microsoft To Do	Власний застосунок
Офлайн-режим (веб)	✓	✗	✗	✗	✓
Простота інтерфейсу	✗	✓	✓	✓	✓
Швидкість роботи	✓	✗	✓	✓	✓
Конфіденційність	✗	✗	✗	✗	✓
Можливість розширення	✓	✗	✓	✓	✓

Для застосунку з орієнтацією на офлайн-режим можна виділити наступні переваги порівняно з переліченими аналогами.

Головна — повна автономність. Більшість популярних сервісів потребують доступу до хмари, а це означає залежність від інтернет-з'єднання

та потенційні затримки при синхронізації. Застосунок, що розробляється, працюватиме швидко навіть без мережі, забезпечуючи миттєвий відгук та роботу з даними локально. Це також підвищує конфіденційність: користувачі контролюють свої дані й немає необхідності передавати особисті завдання третім сервісам. Крім того, концентрація на офлайн-режимі дозволяє спростити інтерфейс і уникнути зайвих хмарних функцій, фокусуючись на потрібному мінімумі інструментів. Таким чином, головні конкурентні переваги – стабільна робота без інтернету, швидкість і посилена приватність користувача – забезпечують відчутно кращий досвід у умовах обмеженого зв'язку чи підвищених вимог до безпеки даних.

1.3 Постановка завдання

Проаналізувавши сильні та слабкі сторони існуючих рішень, було визначено такі ключові вимоги:

- максимально простий та зручний інтерфейс;
- повноцінний офлайн-режим завдяки технології PWA;
- надати користувачам максимальний контроль над власними даними, уникаючи залежності від сторонніх сервісів.

Таким чином, метою роботи є проектування та розробка вебзастосунку для планування завдань із забезпеченням ефективної офлайн-роботи та високого рівня приватності та гнучкості для користувачів.

2 ОБРАННЯ ЗАСОБІВ ТА СЕРЕДОВИЩА РОЗРОБКИ

2.1 Аналіз технологій front-end

Фронтенд є ключовою частиною вебзастосунку, що забезпечує взаємодію з користувачем і динамічну зміну інтерфейсу без перезавантаження сторінок. Для розробки клієнтської частини PWA TaskManager ключовим є вибір відповідного JavaScript-фреймворку. Сучасний стан фронтенд-розробки характеризується домінуванням трьох популярних фреймворків: Angular, React та Vue.js. Кожен із них має свою архітектуру, підхід до управління станом, продуктивність та екосистему інструментів, тому необхідно провести детальний порівняльний аналіз, аби обґрунтовано обрати технологію для проєкту.

Angular – повноцінний front-end фреймворк, що дотримується архітектури Model-View-Controller (або близької до неї Model-View-ViewModel). Розроблений Google як наступник AngularJS, Angular надає “з коробки” все необхідне для великого застосунку: двостороннє зв’язування даних, систему керування станом через Service та Dependency Injection, маршрутизацію, вбудовані рішення для форм та валідації тощо. Angular написаний на мові TypeScript (статично-типізованою надмножиною JavaScript) і вимагає від розробника чіткого дотримання структури проєкту. Цей фреймворк має строгу архітектуру та високу навченість: поріг входу в Angular вважається найвищим серед трьох порівнюваних рішень [16].

Водночас для великих командних проєктів Angular забезпечує масштабованість та підтримуваність коду, оскільки нав’язує єдині підходи. Angular використовує реальний DOM для відображення сторінки, тобто при зміні даних відбувається оновлення відповідних елементів у DOM-дереві. Хоча Angular оснащено механізмом змінного виявлення (change detection) для оптимізації, у дуже великих додатках неконтрольовані зміни в стані можуть призводити до зниження продуктивності, тому розробникам

рекомендується застосовувати такі прийоми, як Strategy OnPush та оптимізація зон, для прискорення роботи інтерфейсу [17].

React – популярна JavaScript-бібліотека (не повний фреймворк) для побудови інтерфейсів, спочатку розроблена компанією Facebook. На відміну від Angular, React фокусується лише на View-шарі (відображенні) та керує станом через одностороннє пряме (універсальне) зв'язування даних. Архітектурно React дуже гнучкий: він не диктує суворої структури проєкту – розробник сам обирає бібліотеки для маршрутизації, керування станом (наприклад, Redux, MobX), роботи з формами тощо. Однією з ключових особливостей React є використання Virtual DOM – віртуального представлення DOM у пам'яті, що дозволяє ефективно оновлювати лише ті компоненти, стан яких змінився [17].

Такий підхід мінімізує прямі маніпуляції з реальним DOM і, як наслідок, підвищує продуктивність інтерфейсу у додатках зі значною динамікою. React-компоненти пишуться з використанням JSX (розширення синтаксису JavaScript, що дозволяє поєднувати HTML-подібну розмітку з JS-кодом), що спрощує створення інтерфейсів і підвищує читабельність коду.

Навчальна крива React вважається помірною – базові концепції (компоненти, властивості, стан, життєвий цикл) відносно прості для засвоєння, особливо для розробників із знанням JavaScript, хоча окремі інструменти (як-от Redux чи розширені хуки React) можуть вимагати додаткового навчання. Сучасний React (версії 17-18) підтримує також написання коду на TypeScript (опціонально), що сприяє підвищенню якості та надійності програмного коду [15].

На рисунку 2.1 зображено частку популярності фронтенд-фреймворків за період 2016–2024 рр. Видно, що React стабільно зберігає лідируючі позиції серед розробників порівняно з іншими популярними фреймворками, такими як Angular і Vue.js [18].

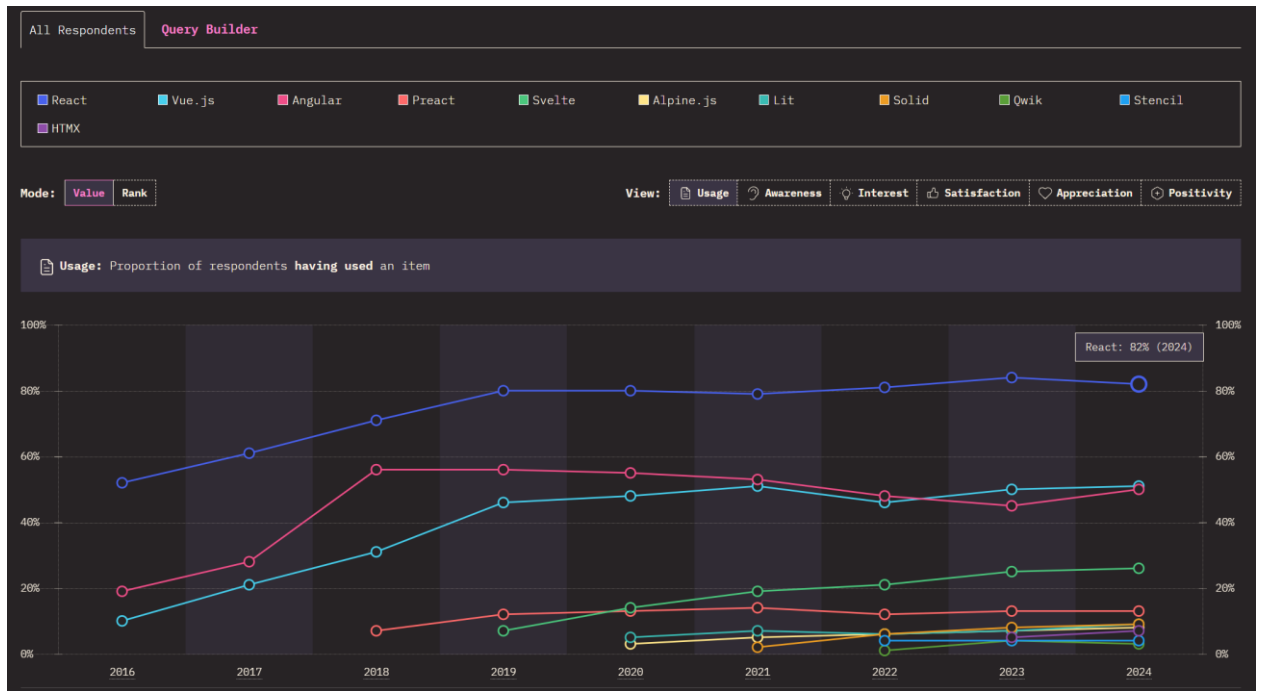


Рисунок 2.1 — Частка популярності фронтенд-фреймворків [18]

Vue.js – “прогресивний” фронтенд-фреймворк, створений Еваном Ю. Vue позиціонується як легкий та поступово інтегрований: його можна впроваджувати у проєкт поступово, починаючи з окремих компонентів інтерфейсу. За філософією, Vue займає проміжне місце між Angular та React. З одного боку, Vue має деякі риси повноцінного фреймворку (на кшталт Angular) – наприклад, підтримує двостороннє зв’язування даних (через директиву `v-model` у шаблонах) і пропонує офіційні додаткові бібліотеки для маршрутизації (Vue Router) та глобального стану (Vuex/Pinia). З іншого боку, за архітектурою Vue ближчий до підходу React: він також використовує компонентний підхід, Virtual DOM та реактивність даних. Важливою особливістю Vue є використання однофайлових компонентів (Single File Components, `*.vue` файли), де в межах одного файлу поєднуються шаблон (HTML-подібна розмітка), скрипт (логіка на JS/TS) і стилі CSS, що забезпечує високу модульність та повторне використання коду [15].

Синтаксис шаблонів Vue є зрозумілим для розробників, знайомих з HTML, і полегшує криву навчання. Загалом, Vue часто називають найбільш

простим для початківців: його навчальна крива – найнижча, базові концепції можна опанувати за короткий час [16].

При цьому Vue забезпечує достатню продуктивність для малих і середніх проєктів; для великих систем може знадобитися додаткова оптимізація та правильна структура (модульність) застосунку [17].

2.2 Докладний порівняльний аналіз технологій

Для вибору оптимального фронтенд-фреймворку проведено глибокий аналіз технологій React, Angular та Vue.js за низкою критеріїв (тобто критерії особливо важливі для PWA-розробки). Нижче наведено основні аспекти порівняння:

Підтримка PWA: PWA є важливим фактором, що впливає на вибір технології для створення сучасних вебзастосунків. React, хоч і не має вбудованої підтримки PWA, має дуже розвинену екосистему з інструментами, такими як Create React App та бібліотека Workbox, що дозволяють легко налаштовувати офлайн-роботу і кешування даних. Angular має вбудований інструментарій для прогресивних вебзастосунків (Angular Service Worker), що автоматично генерує service worker і app shell. Vue CLI також підтримує створення PWA через офіційний плагін (vue-cli-plugin-pwa). Таким чином, усі три фреймворки технічно готові до PWA, але у Angular це найбільш інтегровано, а у Vue і React – через додаткові бібліотеки.

Простота вивчення: React має зрозумілив та порівняно простий синтаксис завдяки JSX і глибоко інтегрованому JavaScript. Це значно полегшує перехід з традиційного JS/HTML на бібліотеку React. Angular навпаки вимагає опанування TypeScript і складної інфраструктури (модулів, декораторів, RxJS), що робить його освоєння важчим. Vue.js вважається доволі простим для новачків через зрозумілий синтаксис і докладні приклади в документації.

Продуктивність: У React і Vue рендеринг відбувається за допомогою оптимізованого Virtual DOM, що забезпечує високу швидкість оновлення інтерфейсу. Angular з більшою вагою фреймворку має вищі початкові накладні витрати, через більш громіздку структуру може бути повільнішим у порівнянні з React. Усі три рішення добре масштабуються при розбивці коду на частини (code splitting) та використанні Lazy Loading.

Спільнота та екосистема: React має найбільшу спільноту серед фронтенд-технологій, що забезпечує постійні оновлення, широкі можливості навчання та велику кількість доступних бібліотек і розширень. React і Angular підтримують великі корпорації (Facebook і Google відповідно), мають активні спільноти та велику кількість сторонніх бібліотек. Vue.js розвивається спільнотою з відкритим кодом і також має широку екосистему (наприклад, Vuex для стану, Nuxt.js для SSR). Популярність React вища, але Angular надає «з коробки» набір рішень (роутінг, форми, i18n) та детальну документацію, що важливо для великих команд.

Це і всі інші важливі компоненти при виборі технології я розгляну в порівняльній таблиці для наглядності (табл. 2.1).

Таблиця 2.1 – Порівняння популярних фронтенд-технологій

Характеристика	React	Angular	Vue.js
Мова/технології	JavaScript (+JSX), підтримка TS	TypeScript, HTML, CSS	JavaScript, HTML- подібний синтаксис
Архітектурна модель	Компоненти, одностороннє зв'язування	MVVM (двостороннє зв'язування)	Компоненти, гнучке зв'язування
Підтримка PWA	Плагіни (Create React App, Workbox)	Вбудована підтримка через Angular Service Worker	Плагіни (Vue CLI PWA Plugin)

Продовження таблиці 2.1

Характеристика	React	Angular	Vue.js
Масштабованість	Гнучка масштабованість: дозволяють будувати як малі, так і великі системи	Відмінна для великих проектів	Добра для середніх проєктів
Рівень складності навчання	середній (JSX)	високий (TypeScript, шаблони)	низько-середній (інтуїтивна структура)
Статична типізація	Опціональна (через TS)	Повна підтримка TypeScript	Опціональна (через TS)
Стан спільноти	Дуже широка	Широка	Зростаюча

Таким чином, React забезпечує високу продуктивність та гнучкість, Vue.js – простоту й легкість освоєння, а Angular – комплексність і повноту функціоналу.

Для проєкту PWA TaskManager важливими є такі фактори: швидкість розробки, продуктивність інтерфейсу при частих оновленнях (наприклад, інтерактивне оновлення списку задач), можливість легкої інтеграції PWA-функціоналу, а також наявність великої спільноти (що спростить пошук рішень та бібліотек). З огляду на це, найдоцільнішим вибором є React. React має найбільшу популярність і підтримку спільноти на 2025 рік, що гарантує доступність численних бібліотек для реалізації PWA (наприклад, готові рішення для Service Workers через Workbox, компоненти інтерфейсу тощо). Крім того, односторонній потік даних у React полегшить відстеження стану застосунку, а Virtual DOM забезпечить швидке відображення навіть при значній кількості елементів (завдань) на екрані [17].

Отже, для фронтенду PWA TaskManager обрано React як оптимальний компроміс між продуктивністю, гнучкістю та багатою екосистемою.

2.3 Інструментарій та середовище розробки

Обрані інструменти мають конкретні переваги, що роблять їх ідеальними для використання у вебзастосунку:

- @hello-pangea/dnd: ця бібліотека дозволяє легко реалізувати drag-and-drop функціонал, має хорошу документацію і високий рівень стабільності;
- framer-motion: надає простий та інтуїтивний спосіб створення плавних анімацій, що значно покращує користувацький досвід;
- localStorage та Service Workers: обрані для ефективної реалізації offline-first підходу, що забезпечує швидку роботу і збереження даних без інтернет-з'єднання.

Середовище розробки включає Visual Studio Code (зручний редактор з багатою підтримкою JavaScript), Node.js (швидкий JavaScript-пушій для серверної розробки) та npm (зручна система керування пакетами).

2.4 Аналіз технологій бекенду

Розробка серверної частини (бекенду) для TaskManager може бути реалізована різними технологічними стеками. Серед найпоширеніших підходів – використання JavaScript-пушія Node.js (у парі з TypeScript для покращення якості коду), платформи .NET (C#) або фреймворку Django (Python). Кожен із цих варіантів має свої переваги та особливості. Розглянемо їх детальніше та проведемо порівняння.

Node.js + TypeScript. Node.js – це серверна платформа, що працює на основі рушія Chrome V8, яка дозволяє виконувати JavaScript на бекенді. Головна відмінність Node.js – неблокуюча, подіє-орієнтована архітектура: введення/виведення здійснюється асинхронно, що дозволяє Node.js

ефективно обробляти велику кількість одночасних запитів без створення надлишкових потоків [19].

Це робить Node.js надзвичайно продуктивним для реального часу та додатків з частим обміном даними (чат-системи, стрімінгові сервіси, оновлення дошки завдань тощо). Додатковою перевагою є те, що і фронтенд, і бекенд використовують одну мову – JavaScript (у нашому випадку із надбудовою TypeScript), що спрощує розробку та підтримку (full-stack JavaScript). Використання TypeScript у Node-проєкті забезпечує статичну типізацію, раннє виявлення помилок та краще автодоповнення в IDE, що позитивно впливає на надійність коду. Node.js має найбільшу спільноту серед серверних рішень – завдяки npm екосистемі доступні тисячі модулів і готових бібліотек для будь-яких задач [20].

Крива навчання Node.js для розробників, знайомих з JS, відносно неглибока; складність може становити розуміння асинхронного програмування (callback-и, Promises, async/await) та конкурентності. Загалом, Node.js вирізняється високою продуктивністю та масштабованістю, особливо добре підходить для мікросервісної архітектури і сценаріїв з інтенсивним використанням WebSocket (наприклад, для миттєвих оновлень статусів завдань) [21].

C#/.NET. Платформа .NET (зокрема її сучасна кросплатформна реалізація .NET 6/7 Core) – це потужний фреймворк від Microsoft для побудови вебзастосунків, служб та інших типів застосунків. Основна мова – C#, об'єктно-орієнтована, статично типізована мова, що компілюється у проміжний код (IL) і виконується під керуванням CLR. Бекенд у .NET традиційно реалізують з використанням ASP.NET Core – інфраструктури для вебAPI та вебзастосунків. Сильна сторона .NET – висока продуктивність та ефективність: сучасний ASP.NET Core демонструє одні з найкращих результатів продуктивності серед популярних фреймворків (завдяки компіляції, оптимізації від JIT/AOT та вдосконаленому вебсерверу Kestrel). .NET також має розвинуту систему безпеки: включає вбудовані механізми

автентифікації, авторизації, захисту від XSS, CSRF, підтримує SSL/TLS тощо. Для розробників з досвідом C# крива навчання ASP.NET є помірною; тим, хто не знайомий з екосистемою Microsoft, може знадобитися час на освоєння Visual Studio, NuGet та інших специфічних інструментів [20].

Варто зазначити, що .NET має дещо вищий поріг входження для новачків у порівнянні з Node.js, а також менш гнучкий у плані розгортання дрібних сервісів (хоча з появою Docker-контейнеризації .NET Core став легшим у деплої). Спільнота .NET є дуже великою, особливо серед корпоративних розробників, і платформа підтримується Microsoft з регулярними оновленнями.

Python/Django. Django – високорівневий вебфреймворк на мові Python, відомий своїм девізом “The web framework for perfectionists with deadlines”. Django дотримується архітектури Model-View-Template (MVT), забезпечуючи розробнику швидкий старт і “батарейки в комплекті” – вбудовані засоби для адміністрування, аутентифікації, ORM для роботи з базою даних, засоби кешування, RSS/Atom та інші часто потрібні компоненти. Однією з головних переваг Django є висока швидкість розробки: принцип DRY (Don’t Repeat Yourself) і генерація типового коду (наприклад, командами `startapp`, `makemigrations`) дозволяють створювати робочий прототип у стислий термін [19].

Django написаний на Python, що означає простий, читабельний синтаксис і можливість швидкого навчання для тих, хто володіє основами Python. Поріг входження в Django вважається відносно низьким, а документація – відмінною. Щодо продуктивності, Python як інтерпретована мова поступається компільованим рішенням (C#, Java) та Node.js у деяких сценаріях; проте Django достатньо оптимізований для обробки значних навантажень, крім того, вертикальне масштабування (додавання ресурсів серверу) та кешування дозволяють вирішувати проблеми продуктивності. Слід відзначити сильні сторони Django у безпеці: фреймворк автоматично

впроваджує захист від найбільш поширених вебвразливостей (SQL-ін'єкції, XSS, CSRF, клікджекінг) і оперативно випускає оновлення безпеки [21].

Django добре підходить для контентно-орієнтованих та CRUD-застосунків – наприклад, новинні портали, CRM-системи, де потрібна швидка розробка і підтримка складних запитів до бази.

Для наочності наведемо порівняння зазначених технологій у вигляді таблиці (табл. 2.2).

Таблиця 2.2 – Порівняння популярних бекенд-технологій

Платформа	Node.js (Express, Nest)	.NET (ASP.NET Core)	Django
Мова	JavaScript/ TypeScript (інтерпретований)	C# (та інші .NET- мови) (компільований)	Python (інтерпретований)
Модель паралельності	Однопоточна з event-loop	Багатопоточна (JVM/CLR)	Багатопоточна (GIL), асинхр. бібліотеки
Кривизна навчання	Низька (JS знайомий фронтендерам)	Висока (різноманіття конфігурацій)	Низько–середня
Архітектурні особливості	Подіє-орієнтована, неблокуюча I/O, однопотокова з циклом подій. Легкий сервер без жорсткої структури (мінімалістична основа Express).	MVC/REST фреймворк зі строгим поділом на контролери, моделі, в'ю. Вбудований DI-контейнер, Middleware pipeline. Статична типізація, компіляція.	MVT фреймворк: чіткий каркас з ORM, шаблонами, адмін-панеллю. Багато автогенерації коду та конфігурацій за замовчуванням. Синхронна архітектура (підтримка async).

Продовження таблиці 2.2

Платформа	Node.js (Express, Nest)	.NET (ASP.NET Core)	Django
Продуктивність	Висока при великій кількості I/O	Висока (JIT-компіляція)	Середня
Екосистема/бібліотеки	Дуже широка (npm)	Дуже широка (Maven, NuGet)	Широка (PyPI)

Аналіз таблиці показує, що Node.js забезпечує найкращу продуктивність у задачах реального часу і дозволяє писати бекенд на тій самій мові, що й фронтенд, але вимагає уважності до безпеки і роботи з асинхронністю. .NET (C#) відзначається вражаючою швидкодією та надійністю, однак орієнтований більше на великі корпоративні проекти та Windows/Azure-інфраструктуру (хоча .NET Core і є кросплатформним) [20].

Django на Python забезпечує швидке створення функціоналу і має виняткову безпеку з коробки, проте може поступатися за продуктивністю під високим навантаженням. Laravel на PHP чудово підходить для швидкого запуску вебслужб і має низький поріг для веброзробників, але в плані обробки тисяч одночасних з'єднань Node.js або .NET можуть перевершувати його.

Node.js добре інтегрується з існуючими фронтенд-технологіями і дозволяє повторно використовувати JS-бібліотеки на сервері. При цьому застосування TypeScript значно підвищує надійність бекенду: статичні типи уловлюють помилки ще на етапі компіляції, а інтелектуальне доповнення коду (IntelliSense) в середовищі розробки полегшує підтримку великого коду. Згідно з аналізом, зазначеним у Framework Training, Node.js легко налаштовується та навчається.

Для PWA TaskManager важливими вимогами до бекенду є: швидка реакція на велику кількість паралельних запитів, підтримка реал-тайм оновлень (наприклад, миттєве розсилання сповіщень про зміну статусу

задач), а також гнучкість у розробці API для зв'язку з фронтом. Зважаючи на це, обрано технологію Node.js (Express) з TypeScript для серверної частини. Такий вибір мотивовано кількома факторами.

По-перше, Node.js чудово справляється з обробкою одночасних запитів завдяки неблокуючій моделі – це буде корисним, якщо багато користувачів одночасно працюватимуть із задачами або отримуватимуть push-сповіщення [21].

По-друге, є можливість легко реалізувати WebSocket-підключення або Server-Sent Events для активної передачі оновлень завдань у режимі реального часу (що узгоджується з концепцією PWA, де бажано надавати динамічний досвід).

По-третє, використання TypeScript на бекенді підвищить надійність коду та полегшить командну розробку, а єдина мова з фронтом (JS/TS) прискорить розробку і зменшить “контекстні перемикання” між технологіями. Крім того, екосистема Node.js пропонує готові рішення для багатьох завдань: зокрема, фреймворк Express для створення REST API, або NestJS як опція, що поєднує переваги TypeScript і Angular-подібної структури проєкту на бекенді. Велика спільнота Node.js гарантує, що будуть доступні численні бібліотеки, у тому числі для роботи з базами даних, авторизації (наприклад, JSON Web Token) і інтеграції з вебзастосунком. Таким чином, Node.js + TS обрано як бекенд-платформу, оскільки вона найкраще відповідає вимогам швидкодії та реального часу, поставленим до PWA TaskManager, і забезпечує ефективну розробку завдяки єдиному стеку технологій.

2.5 Аналіз баз даних

Вибір системи керування базами даних (СУБД) спирається на модель даних додатку та вимоги до надійності. У загальному порівняно реляційні та нереляційні СУБД відрізняються такими властивостями: реляційні (SQL)

СУБД гарантують повні транзакції ACID, чітку схему та підтримку складних запитів (JOIN), а NoSQL-системи (документоорієнтовані, ключ-значення тощо) забезпечують гнучкість схеми, просте горизонтальне масштабування і високу швидкість для певних типів навантажень. До найпоширеніших реляційних СУБД відносяться PostgreSQL, MySQL, Microsoft SQL Server та Oracle. З не реляційних – MongoDB, Redis (як кеш), Firebase Realtime DB тощо. Опитування розробників показало, що PostgreSQL за популярністю випередив MySQL (45,5 % проти 41,1 %), відзначаючи зростання його прийняття. Основні особливості PostgreSQL: це потужна і добре стандартизована система, що підтримує широкий набір типів даних (включаючи JSONB для збереження документів). Крім того, PostgreSQL володіє багатими можливостями розширення (плагіни, додаткові типи даних, тригери, повнотекстовий пошук) [22].

PostgreSQL видається оптимальним вибором для проєкту завдяки наступним фактам: він є однією з найпопулярніших відкритих реляційних СУБД у світі і поєднує в собі надійність SQL з розширюваністю NoSQL через JSONB. Наприклад, JSONB-тип дозволяє гнучко зберігати додаткові дані (категорії, метадані) без зміни схеми, не втрачаючи переваг транзакцій і запитів SQL. У порівнянні з MySQL, PostgreSQL має більш просунуту систему індексів та оптимізатор запитів, що може дати вигоду у продуктивності складних операцій. MongoDB легко масштабувати, але для завдань із важливими транзакціями і суворою консистентністю структура SQL краще відповідає вимогам. Redis використовується як in-memory кеш (наприклад, для зберігання сесій), але він не замінить основної БД через відсутність довготривалої автентифікації даних [23].

2.6 Обґрунтування вибору технологічного стеку

На основі проведеного аналізу фронтенд- і бекенд-технологій, а також систем зберігання даних, сформовано оптимальний стек технологій для

реалізації PWA TaskManager. Результати дослідження та порівняння підтверджують доцільність зробленого вибору, який можна підсумувати такими пунктами:

Фронтенд: React. Попри наявність конкурентних фреймворків (Angular, Vue.js), обрано React у зв'язку з його найбільшою гнучкістю та популярністю у спільноті. Як було показано, React має найбільший відсоток розробників, які бажають і надалі з ним працювати та найвищий попит на ринку.

Це означає кращу підтримку, регулярні оновлення та різноманіття допоміжних бібліотек. Для PWA React надає перевагу у вигляді багатого екосистемою: існують готові рішення для інтеграції service worker, бібліотеки для керування станом (Redux Toolkit або React Context) тощо. Крім того, компонентний підхід React та використання Virtual DOM гарантують високу продуктивність інтерфейсу при динамічних оновленнях списку завдань.

З академічної точки зору, вибір React також демонструє ознайомленість із сучасними підходами (SPA, client-side rendering) та здатність працювати з широко застосовуваною технологією.

Бекенд: Node.js + TypeScript. Проаналізувавши альтернативи, зокрема .NET, Django, ми дійшли висновку, що Node.js найбільше відповідає потребам проєкту, пов'язаним з асинхронністю та реальним часом. Node.js показав найвищу продуктивність у обробці великої кількості одночасних запитів завдяки неблокуючій моделі.

Це критично для TaskManager, якщо планується, що десятки або сотні користувачів можуть одночасно оновлювати або отримувати дані про завдання (особливо при використанні вебсокетів для нотифікацій). Додатковим аргументом стала уніфікація мов: TypeScript на бекенді дозволив використати єдиний стиль і типізацію, полегшуючи взаємодію між фронтендом і сервером. У порівнянні з .NET або Django, Node.js також спростив процес розробки за рахунок менш складного розгортання та налаштування: немає потреби у важкому середовищі виконання, достатньо

встановити Node і необхідні пакети, що особливо зручно при деплої на хмарні платформи. Вибір TypeScript, у свою чергу, обґрунтовано його швидким зростанням популярності – за даними JetBrains, використання TS зросло до ~35% розробників у 2024 році, оскільки статична типізація суттєво знижує кількість помилок у коді та підвищує підтримуваність проєкту [24].

База даних: PostgreSQL. З огляду на потребу зберігати структуровані дані з можливістю гнучко додавати JSON-поля, було обрано PostgreSQL. Ця СУБД є популярною та технологічно розвиненою. Вона гарантує ACID-транзакції для цілісності даних і одночасно підтримує JSONB для збереження неструктурованої інформації. PostgreSQL забезпечує необхідну надійність, багатофункціональність і стабільність для бізнес-логіки Task Manager.

PWA як підхід. Обраний технологічний стек підтверджує прагнення реалізувати саме PWA, а не просто вебсайт. Використання React забезпечує SPA-досвід, Node.js дозволяє налаштувати сервер для push-нотифікацій (або використати Firebase Cloud Messaging), а Firestore гарантує роботу без мережі. Цей синергетичний ефект технологій виявився вирішальним для досягнення мети роботи – створити прогресивний вебзастосунок, здатний конкурувати з рідними додатками. У результаті зробленого вибору підтверджено тезу, що поєднання React + Node.js + хмарна NoSQL БД є ефективним для сучасних вебзастосунків, про що свідчить і практика великих вебплатформ (аналоги, такі як Trello, також використовують Node.js на бекенді та вебфреймворки на фронті для забезпечення інтерактивності).

Таким чином, обраний стек (React, Node.js/TS, PostgreSQL) є результатом всебічного аналізу й відповідає вимогам проєкту “PWA TaskManager”. Він забезпечує швидкодію, офлайн-функціональність, масштабованість та зручність розробки. У наступних розділах роботи (проєктування та реалізація) цей стек буде використано на практиці для побудови системи, а отримані теоретичні висновки – підтверджено шляхом експериментальної перевірки працездатності та ефективності створеного застосунку.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Призначення та мета системи

Розроблений вебзастосунок призначений для планування задач із підтримкою підзадач. Це означає, що користувач може створювати персональні завдання, деталізувати їх на менші підзадачі, встановлювати пріоритети та відстежувати прогрес виконання.

Метою системи є спростити процес управління завданнями: користувач отримує інтуїтивний інтерфейс для додавання нових задач, відмітки виконаних, редагування деталей (опису, дедлайну, пріоритету) та видалення непотрібних задач. Підтримка підзадач дає змогу розбити великі завдання на дрібніші кроки, що підвищує продуктивність та контроль над виконанням роботи. В результаті користувач може ефективніше планувати свій час та ресурси, своєчасно отримувати інформацію про наближення дедлайнів та бачити загальну картину виконання проєкту.

Основні функціональні можливості системи включають:

- створення нового завдання з зазначенням назви, опису, пріоритету, дедлайну (дати та часу);
- додавання довільної кількості підзадач до кожного завдання;
- перегляд списку всіх задач та їх підзадач, відображення статусу виконання;
- редагування властивостей завдання та підзадач (зміна назви, опису, пріоритету, дедлайну);
- відзначення завдань і підзадач як виконаних (та зняття відмітки при потребі);
- видалення завдання (що автоматично призводить до видалення всіх його підзадач);
- реальний час оновлення: зміни, зроблені одним користувачем (додавання, редагування, видалення задач), автоматично передаються всім підключеним клієнтам без перезавантаження сторінки;

– фільтрація та пошук по задачах (наприклад, відображення лише невиконаних або фільтрація за пріоритетом) для зручності користувача.

Таким чином, система покликана забезпечити повний цикл роботи з завданнями – від планування до контролю виконання – у зручному вебінтерфейсі.

3.2 Архітектура вебзастосунку

Архітектура системи побудована за клієнт-серверним принципом, реалізуючи багаторівневу структуру. Застосунок складається з двох основних частин: клієнтської (front-end) та серверної (back-end), які взаємодіють між собою через чітко визначений API та канал WebSocket для реального часу.

Клієнтська частина

Клієнтська частина (Front-end) реалізована як односторінковий вебзастосунок (SPA) на основі фреймворку React. Вона відповідає за інтерфейс користувача: відображає список задач і підзадач, форми для створення/редагування задач, елементи управління (фільтри, пошук, кнопки дій тощо). Front-end надсилає запити до серверної частини для отримання або зміни даних (використовуючи HTTP REST API) та підключається до WebSocket для отримання миттєвих оновлень. Структура клієнтської частини спроектована компонентно: є окремі React-компоненти для шапки застосунку, списку задач, елемента списку задач, модального вікна введення даних, діалогу підтвердження тощо. Це забезпечує підтримку чистого коду та повторного використання компонентів.

Серверна частина

Серверна частина (Back-end) реалізована на платформі Node.js з використанням фреймворка Express. Сервер надає RESTful API для виконання CRUD-операцій над задачами. Логіка серверної частини розподілена за шарами: маршрути (Routes), контролери (Controllers) та сервіси (Services). Маршрути визначають кінцеві точки API (URL-адреси) та

HTTP-методи, контролери обробляють запити (викликають відповідні методи сервісів та формують HTTP-відповідь), а сервіси містять бізнес-логіку і взаємодіють із базою даних. Для зберігання даних використовується реляційна база даних (наприклад, PostgreSQL). Серверна частина також реалізує вебсокет (WebSocket) сервер, який працює паралельно з HTTP-сервером і відповідає за надсилання клієнтам повідомлень у режимі реального часу про зміни (оновлення списку задач).

База даних

Система зберігає інформацію про завдання у базі даних, спроектованій з урахуванням зв'язку «один-до-багатьох» між завданнями та підзадачами. Кожне завдання може мати кілька підзадач, що реалізовано через окремі таблиці (детальний опис структури БД наведено в підрозділі 3.5). База даних взаємодіє із серверною частиною через SQL-запити; для спрощення роботи застосовано бібліотеку для побудови SQL-запитів (конфігурація підключення знаходиться в спеціальному модулі).

Обмін даними

Комунікація між front-end та back-end відбувається двома способами. Основний – це HTTP-запити до REST API (методи GET, POST, PUT, DELETE) для отримання списку задач, створення нових, оновлення та видалення існуючих. Додатково налаштовано WebSocket-з'єднання для надсилання від сервера сповіщень про створення, зміну чи видалення задач. Такий підхід істотно покращує користувацький досвід: усі підключені до сервера клієнти миттєво бачать актуальні дані, що усуває потребу вручну оновлювати сторінку чи постійно опитувати сервер.

На рисунку 3.1 зображено UML-діаграму варіантів використання системи, де показані основні дії користувача (актор – «Користувач») у взаємодії із застосунком: створення завдання, додавання підзадач, перегляд списку, редагування та видалення. Ця діаграма ілюструє функціональні можливості системи та межі системи (система «Менеджер задач» включає сценарії роботи з задачами).

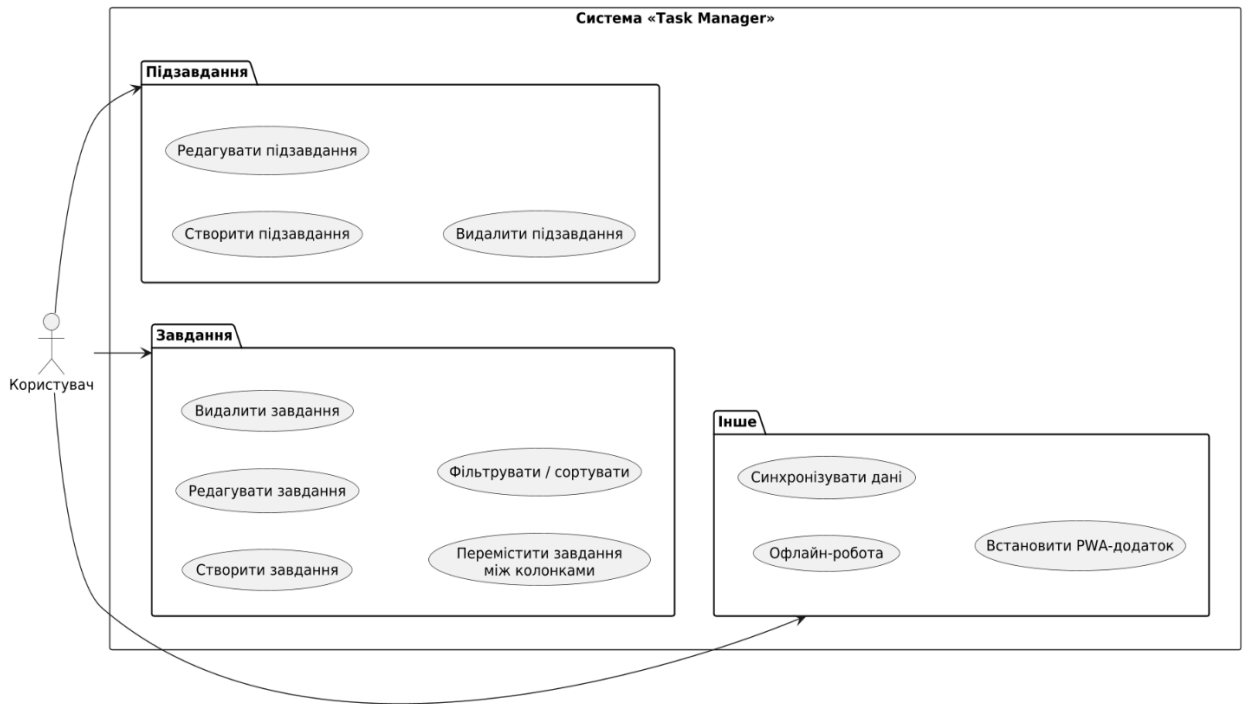


Рисунок 3.1 – UML-діаграма прецедентів (варіантів використання) вебзастосунку

На рисунку 3.2 наведено структурну діаграму системи (діаграма компонентів), що відображає архітектуру рішення. На ній умовно показано поділ на клієнтську та серверну частини, а також внутрішні основні компоненти: на клієнті – компоненти інтерфейсу (формування UI), на сервері – шари маршрутизації, контролерів, сервісів та підсистему роботи з базою даних. Діаграма ілюструє, як клієнт звертається до сервера через HTTP (REST API) та підтримує постійне з'єднання через WebSocket. Серверний компонент «SocketServer» надсилає повідомлення на клієнтські компоненти при подіях створення/оновлення/видалення задач. Компонент «Database» зображено як окремий елемент, який зберігає стан задач та підзадач.

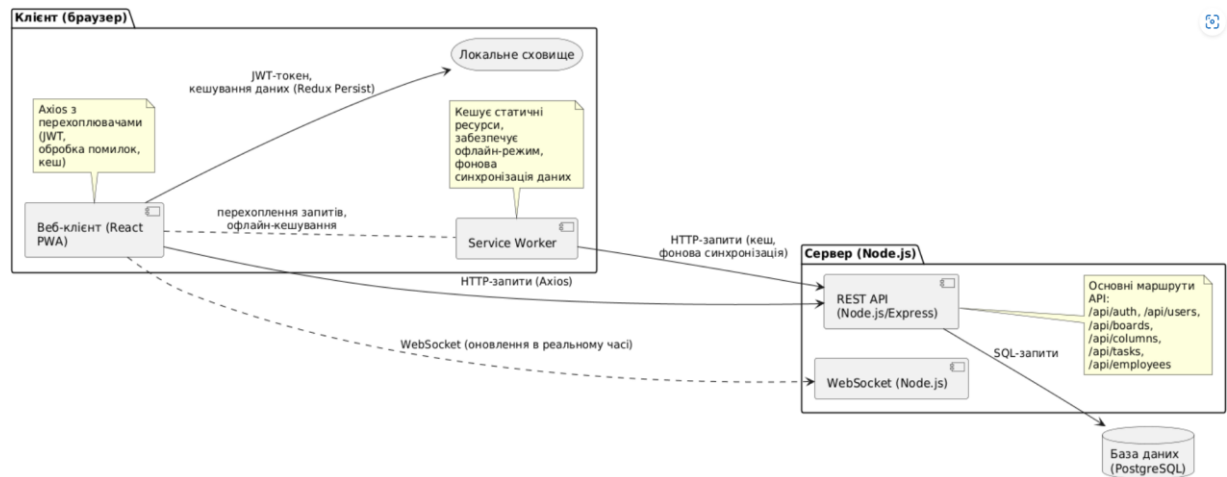


Рисунок 3.2 – UML-діаграма компонентів системи (архітектурна схема клієнт-сервер)

3.3 Проєктування бази даних

Проєктування сховища даних почалося із визначення основних сутностей та їх зв'язків. У системі виділено дві основні сутності: Task (Завдання) та Subtask (Підзадача).

Для них створені відповідні таблиці в базі даних – tasks та subtasks. Кожен запис у таблиці tasks представляє окреме завдання, а записи у subtasks є підзадачами, пов'язаними з певним завданням. Відношення між таблицями – 1 до N: одному запису в tasks може відповідати багато записів у subtasks (кілька підзадач для одного завдання). Ключовим полем для зв'язку є зовнішній ключ task_id у таблиці subtasks, який посилається на первинний ключ id таблиці tasks.

Схема бази даних зображена на рисунку 3.3. На схемі наведено ER-діаграму структури бази даних, що відображає дві основні сутності tasks і subtasks та зв'язок «один-до-багатьох» між ними.

У таблиці 3.1 наведено структуру таблиці tasks, включно з основними полями, їх типами та призначенням.

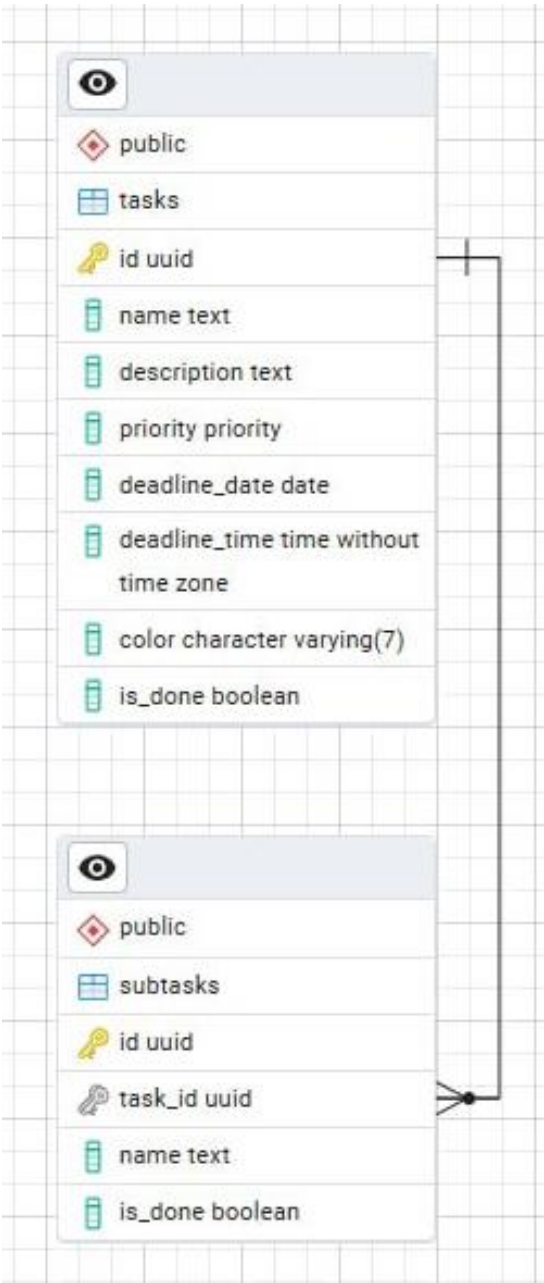


Рисунок 3.3 – Схема бази даних

Таблиця 3.1 – Структура таблиці tasks (завдання)

Поле	Тип даних	Опис і призначення
id	UUID (PRIMARY)	Унікальний ідентифікатор завдання (первинний ключ).
name	TEXT (NOT NULL)	Назва завдання. Короткий текст, що описує задачу.
description	TEXT	Опис завдання. Довільний текст із деталями (необов'язковий).

Продовження таблиці 3.1

Поле	Тип даних	Опис і призначення
priority	priority (ENUM)	Пріоритет завдання: наприклад, High, Medium, Low, Neutral. Зберігається як значення перелічуваного типу priority.
deadline_date	DATE	Дата дедлайну (гранична дата виконання завдання, може бути відсутня).
deadline_time	TIME	Час дедлайну (граничний час в зазначену дату, може бути відсутній).
is_done	BOOLEAN	Статус виконання завдання: FALSE (невиконано, за замовчуванням) або TRUE (виконано).

У таблиці 3.2 наведено структуру таблиці subtasks з описом її полів.

Таблиця 3.2 – Структура таблиці subtasks (підзадачі)

Поле	Тип даних	Опис і призначення
id	UUID (PRIMARY)	Унікальний ідентифікатор підзадачі (первинний ключ).
task_id	UUID	Зовнішній ключ, що посилається на tasks.id. Визначає, до якого завдання належить підзадача. При видаленні завдання всі підзадачі з цим task_id видаляються автоматично (ON DELETE CASCADE).
name	TEXT (NOT NULL)	Назва підзадачі. Короткий опис конкретної дії чи кроку.
is_done	BOOLEAN	Статус виконання підзадачі: виконано (TRUE) або ні (FALSE, значення за замовчуванням).

Звертає на себе увагу поле `priority` у таблиці `tasks` – воно має користувацький тип `priority`. Цей тип даних визначено в базі як перелік допустимих значень (`ENUM('High','Medium','Low','Neutral')`) для зберігання пріоритетів. Таке рішення забезпечує цілісність даних (пріоритет може бути лише одним із заздалегідь визначених рівнів). Нижче наведено фрагмент коду, рисунок 3.4 та рисунок 3.5, який виконує ініціалізацію схеми бази даних – створює тип `priority` та таблиці `tasks` і `subtasks` з потрібними атрибутами та зв'язками:

```
await sql`CREATE TYPE priority AS ENUM ('High', 'Medium', 'Low', 'Neutral');`
await sql`
CREATE TABLE IF NOT EXISTS tasks (
  id UUID PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT DEFAULT NULL,
  priority priority,
  deadline_date DATE DEFAULT NULL,
  deadline_time TIME DEFAULT NULL,
  color VARCHAR(7) DEFAULT NULL,
  is_done BOOLEAN DEFAULT FALSE
);
`;
await sql`
CREATE TABLE IF NOT EXISTS subtasks (
  id UUID PRIMARY KEY,
  task_id UUID REFERENCES tasks(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  is_done BOOLEAN DEFAULT FALSE
);
`;
```

Рисунок 3.4 – SQL-скрипт створення структури бази даних для задач (Перша частина коду)

Як видно з наведених, при створенні таблиці `subtasks` використано конструкцію `REFERENCES tasks(id) ON DELETE CASCADE`. Це гарантує збереження цілісності даних: якщо завдання видаляється з таблиці `tasks`, усі пов'язані з ним підзадачі з таблиці `subtasks` видаляються автоматично на рівні СУБД.

Отже, спроектована база даних ефективно моделює необхідні зв'язки. Вибір окремої таблиці для підзадач забезпечує гнучкість – кількість підзадач не обмежена і вони швидко вибираються за допомогою зовнішнього ключа (з можливістю індексації поля `task_id` для оптимізації запитів). Така схема легко розширюється для підтримки додаткових сутностей (наприклад, категорій задач або користувачів) у разі подальшого розвитку системи.

3.4 Структура програмного забезпечення

Розроблений застосунок має чітко організовану файлову структуру проєкту, що розділяє фронтенд і бекенд частини. Це дозволяє різним компонентам системи розвиватися незалежно і полегшує навігацію в коді.

Back-end

Back-end (папка `todo_backend/`): містить вихідний код серверної частини.

а) `src/` – вихідні файли сервера:

1) `controllers/` – контролери (обробники HTTP-запитів; зокрема, `task.controller.ts` для задач);

2) `routes/` – визначення маршрутів API (наприклад, `task.routes.ts` з описом ендпойнтів для задач);

3) `services/` – бізнес-логіка і доступ до БД (наприклад, `task.service.ts` містить функції роботи із задачами в базі даних);

4) `middlewares/` – проміжні обробники (наприклад, `validation.middleware.ts` для перевірки вхідних даних);

5) `schemas/` – схеми валідації та моделі даних (файли з описом Zod-схем, такі як `task.schema.ts`);

6) `db/` – модулі роботи з базою даних:

– `index.ts` – конфігурація підключення до БД;

– `migrations/` – скрипти ініціалізації/міграції БД (наприклад, `init.ts` створює таблиці);

7) `db/` – модулі роботи з базою даних;

8) `ws/` – модуль WebSocket (файл `socket.ts` налаштовує WebSocket-сервер і його взаємодію з HTTP-сервером);

9) інші утиліти та типи (`utils/`, `types/`) – допоміжні функції, оголошення TypeScript-типів тощо.

б) файл `app.ts` – головний файл сервера, який запускає Express-додаток, налаштовує middleware (CORS, JSON body parser), підключає маршрути `/v1/tasks`, інтегрує Swagger-документацію API (`/docs`) та запускає HTTP-сервер разом із WebSocket.

Front-end

Front-end (папка `ToDoList-master`): містить код інтерфейсу.

а) `public/` – статичні файли (HTML-шаблон, іконки, маніфест PWA, service worker тощо);

б) `src/` – вихідні файли React-застосунку:

1) `index.js` та `App.js` – точка входу та кореневий компонент застосунку. В `App.js` описано основну логіку роботи з задачами на клієнті;

2) `api/axiosInstance.js` – налаштування Axios для HTTP-запитів до бекенду (базовий URL API тощо);

3) `components/` – папка з React-компонентами інтерфейсу. Кожен компонент розміщено в окремій директорії:

– `Header/` (`Header.jsx`, `Header.module.css`) – шапка застосунку (заголовок, можливо, кнопка додати задачу тощо);

- Controls/ (Controls.jsx) – компонент з елементами керування списком задач (фільтри: переключення між “всі/виконані/невиконані”, пошуковий рядок, можливо фільтр по пріоритету);

- TaskList/ (TaskList.jsx) – компонент, що відображає список задач. Він отримує масив задач та рендерить для кожної об’єкт TaskListItem;

- TaskListItem/ (TaskListItem.jsx) – компонент для відображення однієї задачі у списку. Містить логіку відображення назви, статусу (галочка виконано/невиконано), кольорової мітки, списку підзадач (якщо є) та кнопок редагування/видалення;

- ModalWindow/ (ModalWindow.jsx) – модальне вікно для створення або редагування завдання. Містить форму, поля для вводу назви, опису, вибору пріоритету, дати, часу, введення підзадач тощо, а також кнопку збереження;

- ConfirmDialog/ (ConfirmDialog.jsx) – діалог підтвердження дії (наприклад, підтвердження видалення завдання);

4) стовпові файли стилів (CSS-модулі) для кожного компоненту поруч із файлами компонентів, що забезпечує ізоляцію стилів.

На рисунку 3.6 показано приклад файлової структури проекту, де видно поділ на фронтенд та бекенд папки, а також ключові файли і директорії в їх складі. Чітка структура полегшує командну роботу та підтримку коду: розробники інтерфейсу та серверної логіки можуть працювати паралельно, не заважаючи один одному, а новий учасник команди легко зорієнтується де шукати відповідний код.

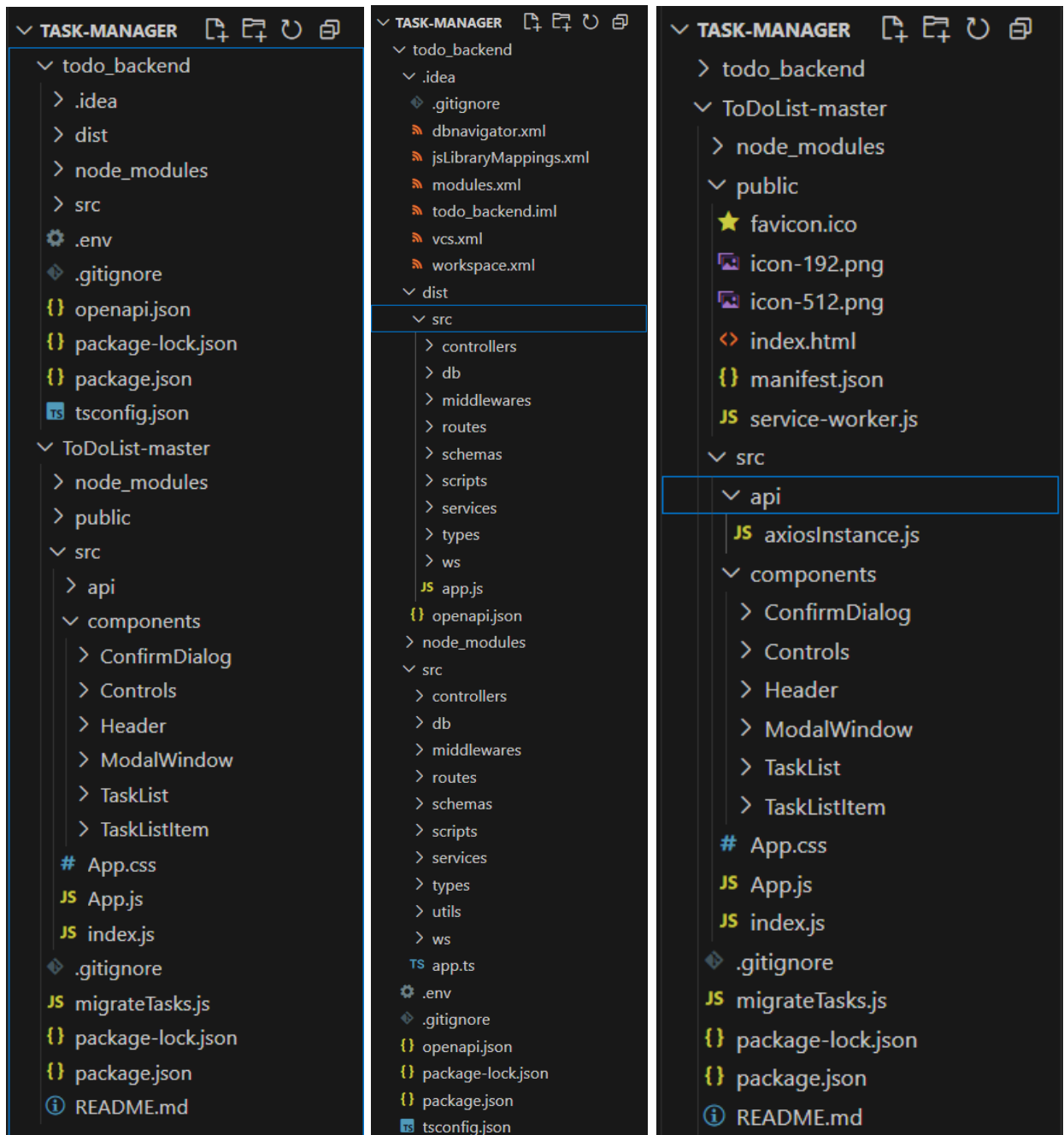


Рисунок 3.6 – Файлова структура фронтенд і бекенд частин вебзастосунку

Важливо, що така організація відповідає принципам розподілу обов'язків: фронтенд частина повністю відповідає за відображення та взаємодію з користувачем, не містить бізнес-логіки з управління даними, а бекенд – навпаки, зосереджений на бізнес-правилах, зберіганні та видачі даних, і не залежить від деталей реалізації інтерфейсу. Комунікація між ними відбувається через чітко визначений інтерфейс (HTTP API та події

WebSocket), що також робить можливим у майбутньому заміну клієнтської частини (наприклад, на мобільний додаток) без значних змін у серверній логіці.

4 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

4.1 Реалізація серверної частини (API)

Серверна частина вебзастосунку реалізована на Node.js/Express і надає RESTful API для управління задачами. Основні принципи роботи API можна розглянути на прикладі модуля задач, що складається з трьох взаємопов'язаних компонентів:

- маршрути (routes) – визначають URI та HTTP-методи для операцій над задачами;
- контролер (controller) – містить методи-обробники для кожного маршруту (логіка отримання запиту та формування відповіді);
- сервіс (service) – реалізує бізнес-логіку: безпосередньо працює з базою даних та іншими сервісами, виконує необхідні операції над даними.

Для модуля задач ці компоненти представлені файлами `task.routes.ts`, `task.controller.ts` і `task.service.ts` відповідно.

Взаємодія між ними відбувається таким чином: коли надходить HTTP-запит (наприклад, POST запит на створення задачі), спершу Express співставляє його з маршрутом в `task.routes.ts`, де вказано, що для шляху `/v1/tasks/` з методом POST треба викликати метод `TaskController.createTask`. Контролер у свою чергу викликає метод сервісу `TaskService.createTask` для виконання бізнес-логіки (збереження даних), а отримавши результат – формує HTTP-відповідь (наприклад, повертає успішний статус і тіло з створеним ресурсом або його ID).

Нижче наведено фрагмент коду, що демонструє налаштування маршрутів для задач та використання проміжного шару для валідації даних:

Як видно з рисунку 4.1, для деяких маршрутів підключено middleware `validateData(...)`. Ця функція перевіряє вхідні дані запиту за допомогою `Zod-schem`:

- `validateData(getTaskSchema, 'query')` – перевіряє параметр `id`, переданий у рядку запиту (очікується UUID завдання) перед тим, як виконати пошук завдання за ID;
- `validateData(createTaskSchema)` – перевіряє JSON-тіло POST-запиту при створенні нового завдання (усі обов'язкові поля, типи даних, довжина текстів тощо);
- `validateData(updateTaskSchema)` – перевіряє дані для оновлення (PUT-запиту), що можуть містити як сам об'єкт завдання, так і вкладені підзадачі.

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

// Отримати всі задачі
router.get('/', TaskController.getAllTasks);

// Отримати задачу за ID (валідація ID через query-параметр)
router.get('/:id', validateData(getTaskSchema, 'query'), TaskController.getTaskById);

// Створити нову задачу (валідація тіла запиту)
router.post('/', validateData(createTaskSchema), TaskController.createTask);

// Оновити існуючу задачу
router.put('/:id', validateData(updateTaskSchema), TaskController.updateTask);

// Видалити задачу
router.delete('/:id', TaskController.deleteTask);

export default router;
```

Рисунок 4.1 – Маршрути API для задач (файл `task.routes.ts`)

Використання бібліотеки `Zod` для валідації дозволяє централізовано описати форму та типи даних наших сутностей. Схеми `taskSchema`,

`createTaskSchema`, `updateTaskSchema` визначені у файлі `task.schema.ts`. Наприклад, схема задачі містить поля: `id` (тип `UUID`), `name` (рядок 1–24 символи), `description` (необов'язковий рядок до 240 символів), `priority` (перелічення з допустимих значень), `deadlineDate` (рядок у форматі дати), `deadlineTime` (рядок у форматі часу та `isDone` (булеве значення), а також масив підзадач `subtasks` (кожна підзадача – об'єкт із полями `id`, `text`, `isDone`).

Ці ж схеми використовуються для автоматичної генерації документації `Swagger` (`OpenAPI`) та забезпечують, що сервер отримає правильно структуровані дані. Якщо перевірка не проходить, `middleware` відхиляє запит і повертає клієнту помилку `400 Bad Request` із описом, які поля невірні заповнені – це допомагає забезпечити надійність API.

Логіка збереження та обробки задач

В `TaskService` зосереджено основну логіку роботи з базою даних для задач. Розглянемо, як реалізовані основні операції API на рівні сервісу:

а) створення завдання: Метод `TaskService.createTask` приймає об'єкт завдання (що пройшов валідацію) та зберігає його в базі даних. Спочатку генерується унікальний ідентифікатор (`UUID`) для нового завдання. Далі виконується SQL-вставка запису в таблицю `tasks` з відповідними полями. Після цього, якщо в створюваного завдання є підзадачі, для кожної з них генерується `UUID` та виконується вставка в таблицю `subtasks` із прив'язкою до `task_id` новоствореного завдання. Після успішного збереження даних метод повертає клієнту щойно згенерований `id` завдання. Важлива особливість – після створення завдання сервіс викликає функцію `broadcastUpdate` (через модуль `WebSocket`), щоб повідомити всіх підключених клієнтів про появу нового завдання. Це робиться шляхом надсилання повідомлення типу `task:create` з даними завдання.

б) оновлення завдання: Метод `TaskService.updateTask(id, task)` відповідає за зміну існуючого завдання. Він виконує оновлення полів у таблиці `tasks` для запису з заданим `id` (назва, опис, пріоритет, дедлайн дата/час, статус виконання). Також метод обробляє вкладені підзадачі:

проходить по списку підзадач, що надійшли від клієнта, та порівнює їх з наявними в базі:

1) якщо підзадача вже існує (має свій id і знайдена в БД), виконується SQL-запит на UPDATE (оновлення назви або статусу);

2) якщо є нові підзадачі (без id), для них генеруються нові id та вставляються рядки в таблицю subtasks (INSERT);

3) якщо якісь підзадачі були видалені користувачем (тобто присутні в БД, але відсутні у переданому списку), то такі записи видаляються з таблиці subtasks (DELETE).

в) видалення завдання: Метод `TaskService.deleteTask(id)` видаляє завдання з таблиці tasks за заданим ідентифікатором. Завдяки обмеженню цілісності ON DELETE CASCADE в схемі БД, всі підзадачі цього завдання автоматично видаляються з таблиці subtasks. Метод додатково може явно виконувати видалення підзадач для надійності, після чого викликає `broadcastUpdate` з повідомленням типу `task:delete` (в payload передається лише ID видаленого завдання). Це сигналізує клієнтським застосункам, що завдання треба прибрати зі списку на екрані.

Таким чином забезпечується синхронізація стану підзадач між клієнтом і базою. Після успішного оновлення записів метод `updateTask` викликає `broadcastUpdate` з типом `task:update` і відправляє оновлені дані завдання всім клієнтам.

У показаному нижче коді, фрагмент реалізації методу створення завдання у сервісному шарі, що ілюструє описані кроки, рисунок 4.2, після вставки даних у таблиці викликається `broadcastUpdate(...)`. Ця функція (визначена у модулі WebSocket `socket.ts`) формує JSON із переданими даними та розсилає його всім активним клієнтським вебсокетам, які підключені до сервера. Аналогічні виклики `broadcastUpdate` із типом `task:update` та `task:delete` присутні у методах оновлення та видалення завдання відповідно. Таким чином, бекенд повідомляє про кожну зміну, і клієнти можуть оперативно оновити інтерфейс.

```

static async createTask(task: Task) {
  const id = uuidv4();
  // Вставка запису задачі в БД
  await sql`
    INSERT INTO tasks (id, name, description, priority, deadline_date, deadline_time,
color, is_done)
    VALUES (${id}, ${task.name}, ${task.description ?? null}, ${task.priority},
    ${task.deadlineDate ?? null}, ${task.deadlineTime ?? null}, ${task.color ??
null}, ${task.isDone})
  `;
  // Вставка підзадач (якщо передані)
  for (const subtask of task.subtasks ?? []) {
    await sql`
      INSERT INTO subtasks (id, task_id, name, is_done)
      VALUES (${uuidv4()}, ${id}, ${subtask.text}, ${subtask.isDone})
    `;
  }
  // Сповіщення всіх клієнтів про створення нового завдання
  broadcastUpdate({
    type: 'task:create',
    payload: {
      id,
      ...task,
    },
  });
  return { id };
}

```

Рисунок 4.2 – Метод створення нового завдання (Service, файл task.service.ts)

Основні операції API для ресурсу завдань можна підсумувати так:

- створення завдання: HTTP POST /v1/tasks/ – створює новий запис задачі (та підзадач, якщо задані) у базі даних, повертає ID створеного завдання. Всім клієнтам надсилається повідомлення task:create;

- отримання списку завдань: HTTP GET /v1/tasks/ – повертає JSON-масив усіх завдань із вкладеними підзадачами. Цей запит використовується при завантаженні застосунку для відображення актуального списку;
- отримання конкретного завдання: HTTP GET /v1/tasks/{id} – повертає деталі завдання з заданим ID (включно зі списком підзадач). Якщо завдання не знайдено – повертається статус 404 (Not Found);
- оновлення завдання: HTTP PUT /v1/tasks/{id} – оновлює поля існуючого завдання та його підзадач. Клієнт надсилає JSON з новим станом завдання; сервер зберігає зміни. У разі успіху повертається статус 204 (No Content). Клієнти отримують повідомлення task:update для синхронізації;
- видалення завдання: HTTP DELETE /v1/tasks/{id} – видаляє завдання з бази даних. Сервер повертає статус 204 при успішному видаленні. Після цього розсилається повідомлення task:delete з ID видаленого завдання, щоб клієнти могли видалити його зі свого відображення.

Кожна з цих операцій проходить через перевірку правильності даних (валідацію схемами Zod) та виконується відповідним методом контролера і сервісу, забезпечуючи чистий розподіл обов'язків. Завдяки такій структурі, код легше підтримувати та тестувати – можна окремо перевіряти логіку сервісу (роботу з БД), не залежно від контролерів і маршрутизації, і навпаки.

4.2 Реалізація клієнтської частини (інтерфейс)

Front-end застосунку побудований на бібліотеці React та взаємодіє із сервером через REST API і WebSocket, забезпечуючи зручний інтерфейс для користувача. Основний код клієнта зосереджений у компоненті App (файл App.js), який є кореневим компонентом і відповідає за стан додатку та високорівневу логіку.

У стані (useState) компонента App зберігаються:

- taskList – масив поточних задач (кожна з яких включає свої підзадачі);

- пов'язані з відображенням змінні: фільтр статусу (button – наприклад, значення "all" / "active" / "completed" для відображення всіх або тільки виконаних/невиконаних задач), рядок пошуку value для фільтрації за назвою, фільтр пріоритету priorityFilter;
- стан модального вікна modalOpen та вибране завдання для редагування editingTask (якщо користувач відкрив існуючу задачу в модальному вікні);
- об'єкт WebSocket-з'єднання ws (або null, якщо з'єднання ще не встановлено).

При монтуванні компонента App спрацьовує useEffect, який:

а) завантажує початкові дані завдань: викликає функцію synchronizeTasks(). Ця функція намагається взяти збережений список задач із локального сховища браузера (LocalStorage) та паралельно отримує актуальний список із сервера (HTTP GET /tasks). Далі вона порівнює локальний і віддалений списки і виконує синхронізацію – при необхідності відправляє на сервер відсутні локальні задачі або оновлення (HTTP POST/PUT), а також видаляє зайві задачі з бази (HTTP DELETE), щоб обидва списки збігалися. Після синхронізації taskList у стані додатку оновлюється до узгодженого списку. Такий механізм дозволяє працювати офлайн з локальним збереженням і при відновленні зв'язку синхронізувати зміни з сервером;

б) встановлює WebSocket-з'єднання: створює новий WebSocket, підключаючись до URL сервера (наприклад, ws://localhost:3033/ws). Визначаються обробники подій:

1) onopen: при успішному підключенні до WebSocket – в консолі з'являється повідомлення, у майбутньому можна виконати додаткові дії (наприклад, повторна синхронізація, якщо треба);

2) onmessage: при отриманні повідомлення від сервера – викликається функція для обробки цього повідомлення;

3) onclose: при закритті з'єднання – виводиться інформація про розрив зв'язку; при неочікуваному обриві з'єднання (наприклад, мережевий збій) компонент намагається підключитися повторно через певний інтервал (використовується механізм установки ws в null і повторного виклику useEffect, який ініціює reconnection).

Обробка вхідних повідомлень по WebSocket – ключовий аспект реалізації реального часу. Кожне повідомлення містить JSON із полями type (тип події) та payload (дані). В нашому випадку можливі типи: task:create, task:update, task:delete. У коді фронтенду реалізовано наступну логіку (рисунок 4.3).

Цей код проходить через всі отримані від сервера події і відповідно модифікує локальний стан taskList. Завдяки цьому, якщо інший користувач (або інша вкладка браузера) додав чи змінив завдання, наш користувач побачить ці зміни миттєво в своєму списку. Наприклад, коли приходить подія task:create, нове завдання додається до масиву (якщо його ще немає). При task:update – знаходиться завдання за ID та оновлюються його поля. При task:delete – виключаються зі списку всі завдання з переданим ID (тобто одне видалене завдання). Таким чином, забезпечено узгодженість даних між клієнтами у режимі реального часу.

Компонент App також містить функції для відкриття/закриття модального вікна, вибору завдання для редагування, застосування фільтрів. Самі фільтри реалізовані у компоненті Controls: користувач може натиснути кнопку “виконані” чи “активні” щоб обмежити список, або ввести текст для пошуку по назві – цей стан передається в App, який фільтрує taskList та передає відфільтрований масив у компонент TaskList для відображення.

Відображення списку задач: Компонент TaskList отримує масив задач (з урахуванням вибраних фільтрів) і рендерить список компонентів TaskListItem. Кожен TaskListItem відображає інформацію про одну задачу:

```

newWs.onmessage = (event) => {
  const data = JSON.parse(event.data);
  if (data && data.type && data.payload) {
    setTaskList(prevTaskList => {
      let updatedTaskList = [...prevTaskList];
      switch (data.type) {
        case 'task:create':
          // Додати нове завдання, якщо його ще немає у списку
          if (!updatedTaskList.some(task => task.id === data.payload.id)) {
            updatedTaskList.push(data.payload);
          }
          break;
        case 'task:update':
          // Оновити існуюче завдання в списку, знайшовши за ID
          updatedTaskList = updatedTaskList.map(task =>
            task.id === data.payload.id ? { ...task, ...data.payload } : task
          );
          break;
        case 'task:delete':
          // Видалити завдання з списку за ID
          updatedTaskList = updatedTaskList.filter(task => task.id !== data.payload.id);
          break;
      }
      return updatedTaskList;
    });
  }
};

```

Рисунок 4.3 – Обробка вхідних повідомлень по WebSocket

— поруч із назвою задачі розміщується чекбокс або індикатор статусу, на який можна натиснути, щоб змінити стан `isDone`. При натисканні викликається обробник, який оновлює статус у локальному стані (відзначає завдання виконаним або невиконаним) і ініціює синхронізацію або

безпосередньо викликає API для оновлення (в нашій реалізації після кліку зміна статусу буде виявлена на етапі синхронізації і відправлена на сервер);

- індикатор пріоритету (можливо, іконка чи текст) відображає значення пріоритету (наприклад, висока, середня важливість);
- список підзадач: під основним завданням можуть бути вкладені елементи – якщо у `task.subtasks` є елементи, `TaskListItem` може відображати їх (наприклад, у вигляді відступу під назвою задачі). Підзадачі також мають чекбокси виконання. У коді компонента `TaskListItem` використовується внутрішній стан для підзадач, щоб відстежувати їх стан (наприклад, яке підзавдання активне для редагування);
- кнопки дій: “Редагувати” (значок олівця) та “Видалити” (значок кошика). Кнопка редагування відкриває модальне вікно `ModalWindow` з формою, заповненою поточними даними завдання. Кнопка видалення відкриває `ConfirmDialog` для підтвердження.

Модальне вікно для створення/редагування завдання: Компонент `ModalWindow` містить форму, реалізовану засобами `React` (контрольовані компоненти для полів вводу):

- поле назви задачі (обов’язкове);
- текстове поле/`textarea` для опису (необов’язкове).;
- випадаючий список або радіо-кнопки для вибору пріоритету (High, Medium, Low, Neutral);
- поля вибору дати та часу дедлайну (можуть бути відсутні);
- блок для підзадач: користувач може додати кілька підзадач – для цього у формі передбачено, поле вводу підзадачі, яка додає підзадачу до локального стану модального вікна. Існуючі підзадачі відображаються списком, кожен можна видалити або відзначити виконаною прямо у формі;
- кнопка “Зберегти” викликає колбек, який залежно від ситуації або створює нове завдання, або зберігає зміни до існуючого.

При підтвердженні форми (натисканні “Зберегти”) компонент `ModalWindow` передає зібрані дані до `App` (через пропси, наприклад, `onSave`

колбек). Якщо це нове завдання, App додасть його до локального `taskList` (з тимчасовим ID або без ID) і збереже також у `LocalStorage`, а потім синхронізує із сервером (HTTP POST запит відправиться у `synchronizeTasks`). Якщо це редагування, App оновить завдання в стані і так само ініціює синхронізацію (HTTP PUT). Завдяки негайному оновленню локального стану, користувач бачить зміну одразу, не чекаючи відповіді сервера, а у фоновому режимі відбувається передача даних на сервер і обробка відповіді/помилки.

Видалення завдання: Коли користувач натискає “видалити” і підтверджує дію у `ConfirmDialog`, додаток одразу видаляє завдання зі списку `taskList` у стані (щоб прибрати з UI) і викликає API DELETE для сервера. Якщо з якоїсь причини видалення на сервері не вдалось, можна обробити помилку (наприклад, повторно завантажити список або показати повідомлення).

У нашому випадку після видалення сервер розішле повідомлення `task:delete`, але до того моменту завдання вже видалене з локального стану (щоб інтерфейс був швидким). Якщо інші клієнти відкриті, вони отримають `task:delete` і також видалять цей елемент. Таким чином, фронтенд реалізований таким чином, щоб максимально швидко реагувати на дії користувача, оновлюючи інтерфейс негайно, і асинхронно синхронізуватися із сервером. Вбудована підтримка реального часу через `WebSocket` гарантує, що всі учасники роботи бачать актуальні дані.

Наприклад, якщо два користувачі працюють одночасно: один додає нове завдання через форму (рис. 4.4), то у другого це завдання автоматично з’явиться у списку (рис. 4.5) без необхідності оновлювати сторінку.

Візуально інтерфейс забезпечує зручність: нові завдання додаються за певною логікою сортування; виконані задачі відображаються зі спеціальною позначкою; підзадачі візуально відокремлені від основних задач.

Створення таску

Новий таск... 0/24

Підтаски 0/0

☐ Підтаск...

Додати деталі...

Рисунок 4.4 – Форма створення нового завдання (модальне вікно в інтерфейсі)

To Do List

Пошук...

+ Додати завдання

Високий

- ☒ Виконати дом. завд
 - ☐ матем
 - ☐ укр мов

Середній

- ☐ Купити зхдію

Нейтральний

Виконані

- ☒ Купити морозиво

13.06.25 23:01

11.06.25 23:03

Рисунок 4.5 – Інтерфейс списку задач із відображенням задач та підзадач

Кольорове маркування задач допомагає швидко ідентифікувати важливі елементи (наприклад, червоний колір – високий пріоритет). Фільтри зверху дозволяють переключатися між всіма задачами і лише невиконаними, що актуально для щоденного перегляду.

4.3 Оновлення у реальному часі через WebSocket

Як вже зазначалося, однією з важливих особливостей застосунку є миттєве оновлення даних у реальному часі. Це досягається за допомогою технології WebSocket, яка забезпечує двостороннє постійне з'єднання між клієнтом і сервером.

На сервері налаштовано WebSocket-сервер (на базі бібліотеки ws). Він інтегрований з HTTP-сервером: при запуску застосунку (app.ts) викликається функція `setupWebSocket(server)`, що відкриває WebSocket на тому ж порту (рисунок 4.6).

Код ініціалізації виглядає так:

```
// Ініціалізація WebSocket-сервера на базі існуючого HTTP-сервера
const wss = new WebSocketServer({ server });
wss.on('connection', (ws: Client) => {
  clients.add(ws);
  console.log('🔗 WebSocket client connected');
  ws.on('close', () => {
    clients.delete(ws);
    console.log('✖ WebSocket client disconnected');
  });
});
```

Рисунок 4.6 – Ініціалізація WebSocket на базі сервера

Цей код створює `WebSocketServer` і зберігає кожне нове підключення у множині `clients`. При відключенні клієнта – він забирається зі списку.

Змінна `clients` використовується, щоб розіслати повідомлення всім підключеним клієнтам. В модулі `socket.ts` визначена функція `broadcastUpdate(data: any)`, яка робить `JSON.stringify(data)` і викликає `client.send(...)` для кожного активного клієнта у `clients`. Таким чином, будь-яка частина серверного коду, викликавши `broadcastUpdate` з певним об'єктом, надішле його всім під'єднаним фронтендам.

У нашому застосунку, як ми бачили, `broadcastUpdate` викликається у трьох місцях – після створення, оновлення та видалення завдання. Відповідно, фронтенд отримує три типи повідомлень. Такий підхід масштабовано дозволяє розширювати систему: можна додати нотифікації про зміну окремих підзадач, про зміну статусів, або підключити авторизацію для окремих каналів повідомлень тощо.

На клієнтській стороні `WebSocket`-з'єднання відкривається на початку роботи програми (як описано у підрозділі 4.2). При отриманні повідомлення, фронтенд здійснює мінімально необхідну зміну інтерфейсу – додає, оновлює або видаляє елемент зі списку. Це дуже ефективно, оскільки передається тільки інформація про зміну, а не весь список. В результаті навантаження на мережу мінімальне, а користувач завжди бачить актуальний список задач.

Даний механізм був протестований шляхом відкриття застосунку у двох різних браузерних вікнах: при додаванні нового завдання в першому вікні – воно майже миттєво з'являється у другому, при редагуванні назви задачі – зміни відображаються на іншому клієнті без перезавантаження сторінки, при видаленні – елемент зникає скрізь. Це підтверджує коректність реалізації режиму реального часу.

4.4 Тестування функціональності застосунку

Після реалізації основних модулів була проведена серія функціональних тестувань вебзастосунку в ручному режимі через користувацький інтерфейс. Метою тестування було переконатися, що всі

заявлені можливості працюють відповідно до вимог, а також виявити та виправити можливі помилки в логіці як на клієнті, так і на сервері.

Основні кроки тестування включали наступні сценарії:

а) створення нового завдання. В інтерфейсі відкривали форму створення (рис. 4.1), заповнювали всі поля (назва, опис, пріоритет, дедлайн дату і час, декілька підзадач) та натискали “Зберегти”. Перевіряли:

1) завдання з’явилося у списку задач (рис. 4.5) одразу після збереження;

2) поля відображаються правильно (назва, опис, пріоритет – наприклад, іконка або текст; дедлайн – дата і час; колір позначки відповідає вибраному);

3) підзадачі відображаються під завданням;

4) на сервері (через інтерфейс Swagger чи безпосередньо в БД) перевіряли, що створився запис у таблиці tasks і відповідні записи у subtasks;

5) в іншому браузерному вікні (підключеному до того ж сервера) новостворене завдання також з’явилося автоматично (тест роботи WebSocket при створенні);

б) редагування завдання. Обирали існуюче завдання, відкривали його в модальному вікні (натискали кнопку редагування – і дані завантажувалися у форму). Змінювали кілька полів (наприклад, назву, опис, статус виконання, додавали нову підзадачу та змінювали статус однієї з наявних підзадач) і зберігали. Перевіряли:

1) у списку задач відповідний елемент оновився: відобразилась нова назва, статус (якщо позначили “виконано” – відображається, наприклад, галочка), змінилась кількість підзадач і їхній статус;

2) в локальній пам’яті (LocalStorage) і на сервері оновлені поля також змінилися (наприклад, відкривши додаток заново або через Swagger GET /tasks/{id}, отримували вже оновлені дані);

3) на другому клієнті ці зміни застосувалися через WebSocket (той самий завданняний елемент оновився без перезавантаження);

4) граничні випадки: спробували ввести недопустимі значення (наприклад, дуже довгу назву > 24 символи, некоректний формат дати) – перевіряли, що форма/застосунок не дозволяє відправити некоректні дані (спрацьовує валідація на фронтенді або помилка від сервера, яка відображається користувачу);

в) відмітка виконання задач та підзадач. Тестували натискання на чекбокс завдання та підзадачі:

1) при встановленні позначки “виконано” для задачі – вона переміщується (за потреби) до секції виконаних або просто змінює свій стиль (наприклад, стає перекресленою);

2) якщо ввімкнено фільтр “активні задачі”, після відмітки виконання задача зникає зі списку (тому що більше не відповідає фільтру);

3) зняття позначки (повернення у невиконаний стан) – елемент повертається у відповідний список/відображає незавершений статус;

4) перевірено, що статус змінюється і на сервері (тобто після перезавантаження сторінки завдання має оновлений статус);

5) всі ці дії також перевірялися на миттєве відображення на іншому клієнті через WebSocket (особливо важливо для статусів підзадач, щоб двоє користувачів бачили актуальний стан списку підзадач);

г) видалення завдання. Обирали завдання, натискали кнопку видалення, у діалозі підтверджували. Переконувалися:

1) завдання зникло зі списку в поточного користувача;

2) на іншому клієнті отримано подію `task:delete` і задача також прибрана зі списку;

3) пов’язані підзадачі теж зникли (на сервері їх більше немає; зокрема, виконали запит `GET /tasks/{id}` щоб впевнитися, що

повертається 404 Not Found, та запит до таблиці subtasks на предмет записів з цим task_id – переконалися, що таких немає);

4) якщо видалити задачу, поки відкрита форма її редагування на іншій вкладці, – перевіряли коректність поведінки (наприклад, при спробі зберегти зміни після видалення сервер поверне помилку, а інтерфейс виведе повідомлення, що задачі не існує).

г) завантаження та збереження даних. Перезапускали застосунок у браузері (або відкривали нову вкладку) щоб впевнитися, що:

1) завантажуються останні зміни (всі попередньо додані або відредаговані задачі відображаються правильно, тобто зчитування з сервера працює);

2) локальний список задач зберігається у LocalStorage і коректно зливається з серверним при старті (це перевіряли, наприклад, створивши задачу офлайн, а потім підключившись – задача успішно синхронізувалася на сервер);

3) ніякі дані не дублюються і не втрачаються при повторному підключенні або перезавантаженні.

д) інші функції інтерфейсу. Переконались, що пошук працює (при введенні тексту value список фільтрується по назвах задач або підзадач), що фільтр пріоритету коректно відображає тільки задачі з обраним пріоритетом. Перевірили адаптивність інтерфейсу (при різних розмірах вікна чи на мобільному пристрої основні елементи залишаються зручними для взаємодії). Також протестували обробку помилок: наприклад, вимикали сервер і пробували виконати дію – інтерфейс повідомив про неможливість з'єднання; або викликали некоректний URL – отримали відповідь 404 та відповідне відображення.

Результати тестування показали, що застосунок працює відповідно до специфікації: створення, редагування, видалення задач відбуваються правильно, дані зберігаються і передаються, синхронізація у реальному часі функціонує. Були виявлені та виправлені дрібні недоліки, наприклад:

помилка валідації ID при отриманні задачі (в маршруті використовувався невірний параметр – це було виправлено, щоб перевіряти req.params замість req.query), скориговано довжину поля назви підзадач (щоб довгі назви коректно відображались), додано скидання форми після створення нового завдання та інші поліпшення UX.

Загалом, після проведених тестів можна стверджувати, що вебзастосунок “Планувальник задач з підтримкою підзадач” досяг поставленої мети. Система успішно надає користувачу інструмент для керування завданнями, забезпечуючи зручний інтерфейс, надійну серверну логіку та актуалізацію даних у реальному часі. Користувач може ефективно планувати свої справи, деталізувати їх через підзадачі та бути впевненим, що всі пристрої або учасники бачать єдину актуальну картину завдань.

ВИСНОВКИ

У результаті виконання дипломної роботи досягнуто поставленої мети — створено зручний, адаптивний вебзастосунок-PWA для створення, планування та контролю виконання завдань, що забезпечує повноцінну роботу користувача як онлайн, так і в автономному режимі. На підставі аналізу існуючих аналогів сформульовано функціональні та нефункціональні вимоги до системи, що дозволило обґрунтувати вибір оптимального технологічного стеку: React на клієнтській частині, Node.js / Express на серверній, PostgreSQL для зберігання даних і Service Worker разом з локальними сховищами IndexedDB та localStorage для підтримки офлайн-сценаріїв.

Спроектовано й реалізовано багаторівневу архітектуру із REST-API та каналом WebSocket, що забезпечує синхронізацію завдань у реальному часі та цілісність даних після відновлення мережевого з'єднання. Розроблений застосунок дозволяє створювати й редагувати завдання з гнучким пріоритетуванням, автоматично розміщуючи нові елементи у відповідні блоки, а також надає користувачеві мінімалістичний інтерфейс із функціями drag-and-drop і прогресивної анімації. Проведене тестування підтвердило працездатність функціоналу, стійкість до втрати мережі та швидке відновлення синхронності після підключення, що свідчить про відповідність отриманого рішення заданим вимогам.

Таким чином, предмет дослідження — програмна реалізація вебзастосунку для управління завданнями з автономним режимом — розкрито повністю; усі визначені завдання, а саме аналіз аналогів, вибір технологій, проєктування архітектури, реалізація основних можливостей і перевірка працездатності, виконано в повному обсязі. Створений інструмент має практичну цінність як для індивідуального, так і для командного використання в умовах нестабільного інтернет-з'єднання, підвищує

продуктивність планування та забезпечує високий рівень приватності даних користувачів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. SignalDB. Offline First Approach [Electronic resource]. – Access mode: <https://signaldb.js.org/offline-first/>.
2. MDN Web Docs. Progressive Web Apps (PWA) [Electronic resource]. – Access mode: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps.
3. Todoist. Features Overview [Electronic resource]. – Access mode: <https://todoist.com/features>.
4. Todoist Review - Pros, Cons, and Features [Electronic resource]. – Access mode: <https://www.digitalproductsdp.com/blog-featured-reviews/todoist-review>.
5. Trello. Kanban Boards [Electronic resource]. – Access mode: <https://trello.com>.
6. Atlassian. Privacy in Trello [Electronic resource]. – Access mode: <https://support.atlassian.com/trello/docs/privacy/>
7. Trello Mobile. Offline Functionality [Electronic resource]. – Access mode: <https://www.atlassian.com/blog/trello/trello-mobile-offline>.
8. HostIQ. Що таке Trello? [Електрон. ресурс]. – Режим доступу: <https://hostiq.ua/blog/ukr/what-is-trello-2/>.
9. TheTodo. Менеджер завдань онлайн [Електрон. ресурс]. – Режим доступу: <https://theodo.net/uk/>.
10. Usemotion. Google Tasks vs Todoist [Electronic resource]. – Access mode: <https://www.usemotion.com/blog/google-tasks-vs-todoist>.
11. Google Tasks Help. FAQ [Electronic resource]. – Access mode: <https://support.google.com/tasks/answer/10431521>.
12. Microsoft To Do. Task List App [Електрон. ресурс]. – Режим доступу: <https://www.microsoft.com/uk-ua/microsoft-365/microsoft-to-do-list-app>.
13. Timeneye Blog. Microsoft To Do Tutorial [Electronic resource]. –

Access mode: <https://www.timeneye.com/blog/microsoft-to-do-tutorial>.

14. Cloudwards. Microsoft To Do Review [Electronic resource]. – Access mode: <https://www.cloudwards.net/microsoft-to-do-review/>.

15. Habr. React vs Vue vs Angular [Electronic resource]. – Access mode: <https://habr.com/en/companies/auriga/articles/703836/>.

16. ZeroToMastery. Angular vs React vs Vue [Electronic resource]. – Access mode: <https://zerotomastery.io/blog/angular-vs-react-vs-vue/>.

17. BrowserStack. Angular vs React vs Vue [Electronic resource]. – Access mode: <https://www.browserstack.com/guide/angular-vs-react-vs-vue>.

18. State of JS 2024. Front-end Frameworks [Electronic resource]. – Access mode: <https://2024.stateofjs.com/en-US/libraries/front-end-frameworks/>.

19. Peerbits. Node.js vs Django vs Laravel [Electronic resource]. – Access mode: <https://www.peerbits.com/blog/nodejs-vs-django-vs-laravel-back-end-web-framework.html>.

20. Flatirons. Node.js vs ASP.NET [Electronic resource]. – Access mode: <https://flatirons.com/blog/nodejs-vs-aspnet-which-is-better-2024/>.

21. SecureHack. Node.js vs Django vs Laravel [Electronic resource]. – Access mode: <https://securehack.net/blogs/web-development/node-js-vs-django-vs-laravel/>.

22. PostgreSQL. About [Electronic resource]. – Access mode: <https://www.postgresql.org/about/>.

23. ByteScout. MySQL vs PostgreSQL vs Oracle [Electronic resource]. – Access mode: <https://bytescout.com/blog/mysql-vs-postgresql-vs-oracle.html>.

24. Visual Studio Magazine. TypeScript Tops JetBrains Index [Electronic resource]. – Access mode: <https://visualstudiomagazine.com/articles/2024/12/11/typescript-tops-new-jetbrains-language-promise-index.aspx>.

ДОДАТОК А
Приклади коду програми

A.1 Zod-схема для валідації даних задачі на стороні сервера

```
import { z } from 'zod';

export const createTaskSchema = z.object({
  body: z.object({
    name: z.string().min(1).max(24),
    description: z.string().max(240).optional(),
    priority: z.enum(['High', 'Medium', 'Low', 'Neutral']),
    deadlineDate: z.string().optional(),
    deadlineTime: z.string().optional(),
    color: z.string().regex(/^#[0-9A-Fa-f]{6}$/).optional(),
    subtasks: z.array(z.object({
      text: z.string().min(1).max(100),
      isDone: z.boolean().optional(),
    })).optional(),
    isDone: z.boolean().optional()
  })
});

export const getTaskSchema = z.object({
  query: z.object({ id: z.string().uuid() })
});

export const updateTaskSchema = z.object({
  body: createTaskSchema.shape.body.partial()
});
```

A.2 Відображення однієї задачі та її підзадач у списку

```
function TaskListItem({ task, onToggleTaskDone, onToggleSubtaskDone, onEdit,
onDelete }) {
  return (
    <div className="task-item">
      /* Заголовок задачі з чекбоксом статусу */
      <div className="task-header">
        <input
          type="checkbox"
          checked={task.isDone}
          onChange={() => onToggleTaskDone(task.id, !task.isDone)}
        />
        <span className="task-name">{task.name}</span>
      </div>
    </div>
  );
}
```

```

    <button onClick={() => onEdit(task)}>✎</button>
    <button onClick={() => onDelete(task.id)}>🗑️</button>
  </div>
  { /* Список підзадач (якщо наявні) */ }
  { task.subtasks && task.subtasks.length > 0 && (
    <ul className="subtask-list">
      { task.subtasks.map(subtask => (
        <li key={subtask.id} className="subtask-item">
          <input
            type="checkbox"
            checked={subtask.isDone}
            onChange={() => onToggleSubtaskDone(task.id, subtask.id,
!subtask.isDone)}
          />
          <span>{subtask.text}</span>
        </li>
      ))}
    </ul>
  )}
</div>
);
}

```

А.3 Обробка повідомлень, що надходять по WebSocket, для синхронного оновлення списку задач на клієнті

```

newWs.onmessage = (event) => {
  const data = JSON.parse(event.data);
  if (data && data.type && data.payload) {
    setTaskList(prevTasks => {
      let updatedTasks = [...prevTasks];
      switch (data.type) {
        case 'task:create':
          if (!updatedTasks.some(t => t.id === data.payload.id)) {
            updatedTasks.push(data.payload);
          }
          break;
        case 'task:update':
          updatedTasks = updatedTasks.map(t =>
            t.id === data.payload.id ? { ...t, ...data.payload } : t
          );
          break;
      }
    });
  }
}

```

```

    case 'task:delete':
      updatedTasks = updatedTasks.filter(t => t.id !== data.payload.id);
      break;
    }
    return updatedTasks;
  });
}
};

```

A.4 Функція синхронізації задач між локальним сховищем та сервером

```

async function synchronizeTasks() {
  const localTasks = JSON.parse(localStorage.getItem('tasks') || '[]');
  setTaskList(localTasks);
  try {
    const { data: serverTasks } = await axiosInstance.get('/tasks');
    for (const task of localTasks) {
      const existsOnServer = serverTasks.find(t => t.id === task.id);
      if (!existsOnServer) {
        await axiosInstance.post('/tasks', task);
      } else if (JSON.stringify(existsOnServer) !== JSON.stringify(task)) {
        await axiosInstance.put(`/tasks/${task.id}`, task);
      }
    }
    for (const task of serverTasks) {
      if (!localTasks.find(t => t.id === task.id)) {
        await axiosInstance.delete(`/tasks/${task.id}`);
      }
    }
    const { data: updatedTasks } = await axiosInstance.get('/tasks');
    localStorage.setItem('tasks', JSON.stringify(updatedTasks));
    setTaskList(updatedTasks);
  } catch (error) {
    console.error('Error synchronizing tasks:', error);
  }
}

```

A.5 Фільтрація списку задач на клієнті за статусом виконання

```
const [filter, setFilter] = useState('all');

const filteredTasks = taskList.filter(task => {
  if (filter === 'active') return !task.isDone;
  if (filter === 'completed') return task.isDone;
  return true; // 'all' – всі задачі
});

<button onClick={() => setFilter('all')}>Всі</button>
<button onClick={() => setFilter('active')}>Невиконані</button>
<button onClick={() => setFilter('completed')}>Виконані</button>

<TaskList tasks={filteredTasks} />
```

A.6 Збереження задач у LocalStorage офлайн

```
function handleSaveTask(taskData) {
  // Генеруємо тимчасовий ID для нового завдання (якщо нема)
  if (!taskData.id) taskData.id = Date.now();
  setTaskList(prevList => {
    const updatedList = [...prevList, taskData];
    localStorage.setItem('tasks', JSON.stringify(updatedList));
    return updatedList;
  });
  synchronizeTasks();
}
```

A.7 Фільтрація списку задач на клієнті за статусом виконання

```
useEffect(() => {
  // Отримати збережені офлайн задачі з LocalStorage
  const savedTasks = JSON.parse(localStorage.getItem('tasks') || '[]');
  if (savedTasks.length) {
    setTaskList(savedTasks);
  }
  // Паралельно спробувати завантажити актуальні задачі з сервера
  fetch('/api/tasks')
    .then(res => res.json())
    .then(serverTasks => {
      // Об'єднати або синхронізувати списки (локальні та серверні)
```

```

    synchronizeTasks(serverTasks);
  })
  .catch(err => console.warn('Не вдалося отримати задачі з сервера:', err));
}, []);

```

А.8 Фільтрація списку задач на клієнті за статусом виконання

```

async function synchronizeTasks(serverTasks = []) {
  const localTasks = JSON.parse(localStorage.getItem('tasks') || '[]');
  // Якщо не отримали список з сервера, спробувати запросити
  if (serverTasks.length === 0) {
    try {
      const res = await fetch('/api/tasks');
      serverTasks = await res.json();
    } catch {
      return;
    }
  }
  // Надіслати на сервер локальні задачі, яких там немає
  for (const task of localTasks) {
    if (!serverTasks.find(t => t.id === task.id)) {
      await fetch('/api/tasks', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(task)
      });
    }
  }
  // Видалити на сервері задачі, яких немає локально (якщо були видалені офлайн)
  for (const task of serverTasks) {
    if (!localTasks.find(t => t.id === task.id)) {
      await fetch(`/api/tasks/${task.id}`, { method: 'DELETE' });
    }
  }
  // Оновити локальний стан узгодженим списком
  const mergedList =
    setTaskList(mergedList);
  localStorage.setItem('tasks', JSON.stringify(mergedList));
}

```

A.9 Service Worker – кешування та обробка запитів

```

const CACHE_NAME = 'task-manager-cache-v1';
const URLS_TO_CACHE = ['/', '/index.html', '/app.js', /* інші файли */];

self.addEventListener('install', event => {
  // Встановлення: кешуємо потрібні файли для роботи офлайн
  event.waitUntil(
    caches.open(CACHE_NAME).then(cache => {
      console.log('Service Worker: Caching app shell');
      return cache.addAll(URLS_TO_CACHE);
    })
  );
});

self.addEventListener('activate', event => {
  // Активація: очищуємо старі кеші, якщо такі є
  event.waitUntil(
    caches.keys().then(keys => Promise.all(
      keys.filter(name => name !== CACHE_NAME).map(name =>
        caches.delete(name)
      ))
    );
  console.log('Service Worker: Activated');
});

self.addEventListener('fetch', event => {
  // Обробка запитів: повертаємо кешований контент, якщо є, або робимо
  // запит до мережі
  event.respondWith(
    caches.match(event.request).then(response => {
      return response || fetch(event.request).catch(() => {
        // Мережа недоступна: можна повернути офлайн-сторінку або пусту
        // відповідь
        return caches.match('/index.html');
      });
    })
  );
});

```

А.10 Модальне вікно створення/редагування задач (офлайн)

```
function ModalWindow({ task, onSave, onClose }) {
  // Локальний стан для полів форми задачі
  const [title, setTitle] = useState(task?.title || "");
  const [description, setDescription] = useState(task?.description || "");
  const [subtasks, setSubtasks] = useState(task?.subtasks || []);
  const [newSubtaskText, setNewSubtaskText] = useState("");

  // Додати підзадачу до локального списку
  const handleAddSubtask = () => {
    if (newSubtaskText.trim() === "") return;
    const newSubtask = { id: Date.now(), text: newSubtaskText, isDone: false };
    setSubtasks([...subtasks, newSubtask]);
    setNewSubtaskText("");
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    // Формуємо об'єкт задачі з полів форми
    const taskData = { ...task, title, description, subtasks };
    onSave(taskData);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input value={title} onChange={e => setTitle(e.target.value)} required />
      <textarea value={description} onChange={e => setDescription(e.target.value)} />
      { /* Поля вибору пріоритету, дати дедлайн тощо... */ }
      <div className="subtasks">
        {subtasks.map(st => (
          <div key={st.id}>
            <input type="checkbox" checked={st.isDone} /* оновлення isDone */ />
            <span>{st.text}</span>
          </div>
        ))}
        <input
          value={newSubtaskText}
          onChange={e => setNewSubtaskText(e.target.value)}
          placeholder="Нова підзадача..."
        />
        <button type="button" onClick={handleAddSubtask}>Додати
        підзадачу</button>
      </div>
    </form>
  );
}
```

```

    </div>
    <button type="submit">Зберегти</button>
    <button type="button" onClick={onClose}>Скасувати</button>
  </form>
);
}

```

A.11 Метод створення нового завдання

```

static async createTask(task: Task) {
  const id = uuidv4();
  // Вставка запису задачі в БД
  await sql`
    INSERT INTO tasks (id, name, description, priority, deadline_date,
    deadline_time, color, is_done)
    VALUES (${id}, ${task.name}, ${task.description ?? null}, ${task.priority},
    ${task.deadlineDate ?? null}, ${task.deadlineTime ?? null}, ${task.color ??
    null}, ${task.isDone})
  `;
  // Вставка підзадач (якщо передані)
  for (const subtask of task.subtasks ?? []) {
    await sql`
      INSERT INTO subtasks (id, task_id, name, is_done)
      VALUES (${uuidv4()}, ${id}, ${subtask.text}, ${subtask.isDone})
    `;
  }
  // Сповіщення всіх клієнтів про створення нового завдання
  broadcastUpdate({
    type: 'task:create',
    payload: {
      id,
      ...task,
    },
  });
  return { id };
}

```

A.12 Маршрути API для задач

```

const router = express.Router();

// Отримати всі задачі
router.get('/', TaskController.getAllTasks);

```

```
// Отримати задачу за ID (валідація ID через query-параметр)
router.get('/:id', validateData(getTaskSchema, 'query'),
TaskController.getTaskById);

// Створити нову задачу (валідація тіла запиту)
router.post('/', validateData(createTaskSchema), TaskController.createTask);

// Оновити існуючу задачу
router.put('/:id', validateData(updateTaskSchema), TaskController.updateTask);

// Видалити задачу
router.delete('/:id', TaskController.deleteTask);

export default router;
```

A. 13 SQL-скрипт створення структури бази даних для задач

```
// Ініціалізація типу "priority" та таблиці "tasks"
await sql`CREATE TYPE priority AS ENUM ('High', 'Medium', 'Low', 'Neutral');`
await sql`

CREATE TABLE IF NOT EXISTS tasks (
  id UUID PRIMARY KEY,
  name TEXT NOT NULL,
  description TEXT DEFAULT NULL,
  priority priority,
  deadline_date DATE DEFAULT NULL,
  deadline_time TIME DEFAULT NULL,
  color VARCHAR(7) DEFAULT NULL,
  is_done BOOLEAN DEFAULT FALSE
);
`;

// Створення таблиці "subtasks" із зовнішнім ключем на tasks.id

await sql`
CREATE TABLE IF NOT EXISTS subtasks (
  id UUID PRIMARY KEY,
  task_id UUID REFERENCES tasks(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  is_done BOOLEAN DEFAULT FALSE
);
`;
```

ДОДАТОК Б
Слайди презентації

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»
кафедра програмних засобів

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА
НА ТЕМУ: «ПРОЄКТУВАННЯ ТА РОЗРОБКА ВЕБЗАСТОСУНКУ ДЛЯ ПЛАНУВАННЯ
ЗАВДАНЬ ІЗ ЗАБЕЗПЕЧЕННЯМ ОФЛАЙН РОБОТИ»

Виконав: студент групи КНТ-141
Володимир МІЩЕНКО
Керівник: доцент
Олена ШИТІКОВА

2025

Рисунок Б.1 – Слайд 1

ОБ'ЄКТ, ПРЕДМЕТ, МЕТА ТА ЗАВДАННЯ ДОСЛІДЖЕННЯ

- Об'єктом дослідження цієї роботи є процес організації управління завданнями у вебсередовищі.
- Предметом дослідження виступає програмна реалізація вебзастосунку з підтримкою автономного режиму роботи.
- Метою роботи є розробити зручний та адаптивний вебзастосунок для створення, редагування, планування і контролю виконання завдань, здатний підвищити ступінь автономності завдяки технології PWA.
- Для досягнення поставленої мети передбачено виконання наступних завдань:
 - провести аналіз існуючих аналогів;
 - визначити оптимальний технологічний стек;
 - спроектувати архітектуру застосунку;
 - реалізувати основний функціонал ;
 - перевірити працездатність створеного рішення.

2

Рисунок Б.2 – Слайд 2

ПОРІВНЯЛЬНА ТАБЛИЦЯ АНАЛОГІВ

Характеристика	<u>Todoist</u>	<u>Trello</u>	<u>Google Tasks</u>	<u>Microsoft To Do</u>	<u>Власний застосунок</u>
Офлайн-режим (веб)	✓	✗	✗	✗	✓
Простота інтерфейсу	✗	✓	✓	✓	✓
<u>Швидкість роботи</u>	✗	✓	✓	✓	✓
Конфіденційність	✗	✗	✗	✗	✓
<u>Можливість розширення</u>	✓	✗	✓	✓	✓

3

Рисунок Б.3 – Слайд 3

ПОРІВНЯННЯ ПОПУЛЯРНИХ ФРОНТЕНД-ТЕХНОЛОГІЙ

Характеристика	Архітектурна модель	Підтримка PWA	Масштабованість	Мова/технології	Рівень складності навчання	Стан спільноти
React	Компоненти, одностороннє зв'язування	Плагіни	<u>Гнучка</u> масштабованість для будь-яких проєктів	JavaScript (+JSX), підтримка TS	середній (JSX)	Дуже широка
Angular	MVVM (двостороннє зв'язування)	Вбудована підтримка	<u>Відмінна</u> для великих, але не малих проєктів	TypeScript, HTML, CSS	<u>високий</u> (TypeScript, шаблони)	Широка
Vue.js	Компоненти, гнучке зв'язування	Плагіни	Добра для малих-середніх проєктів	JavaScript, HTML-подібний синтаксис	низько-середній (інтуїтивна структура)	Зростаюча

4

Рисунок Б.4 – Слайд 4

ПОРІВНЯННЯ ПОПУЛЯРНИХ БЕКЕНД-ТЕХНОЛОГІЙ						
Характеристика	Мова	Модель паралельності	Кривизна навчання	Продуктивність	Екосистема/бібліотеки	Архітектурні особливості
Node.js (Express, Nest)	JavaScript/TypeScript	Однопоточна з event-loop	Низька	Висока	Дуже широка	Легкий сервер без жорсткої структури, <u>неблокуюча I/O</u>
.NET (ASP.NET Core)	C# (та інші .NET-мови)	Багатопоточна (JVM/CLR)	Висока	Висока	Дуже широка	Зі строгим поділом на контролери, моделі, в'ю. Статична типізація
Django	Python	Багатопоточна, асинхр. бібліотеки	Низько-середня	Середня	Широка	Багато автогенерації коду та конфігурацій

5

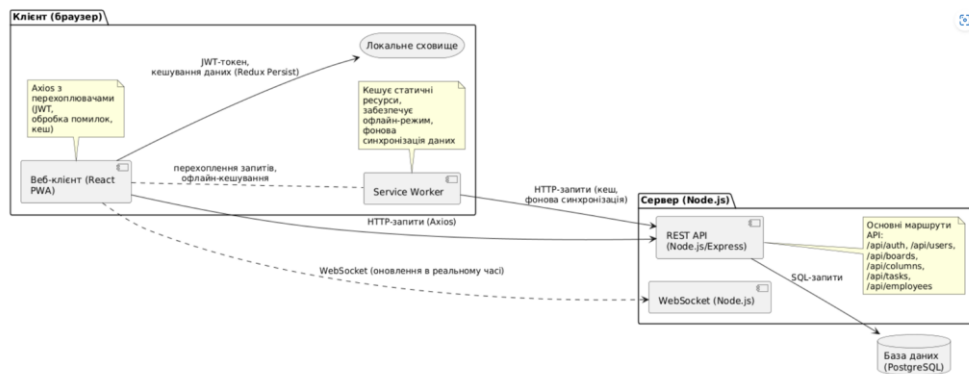
Рисунок Б.5 – Слайд 5



6

Рисунок Б.6 – Слайд 6

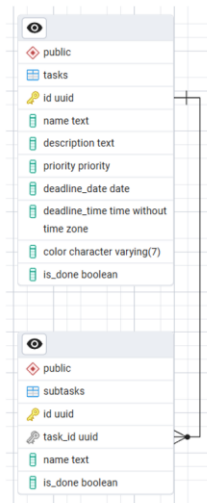
ДІАГРАМА КОМПОНЕНТІВ СИСТЕМИ (СХЕМА КЛІЄНТ-СЕРВЕР)



7

Рисунок Б.7 – Слайд 7

СХЕМА БАЗИ ДАНИХ



8

Рисунок Б.8 – Слайд 8

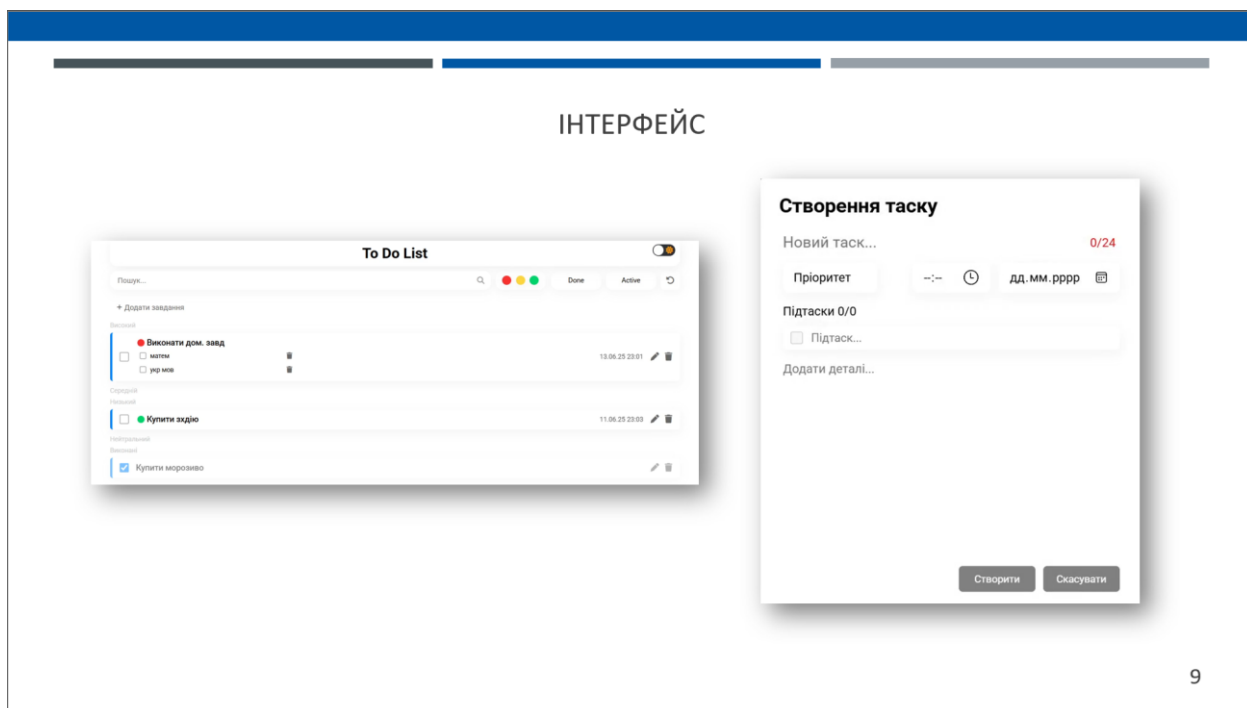


Рисунок Б.9 – Слайд 9

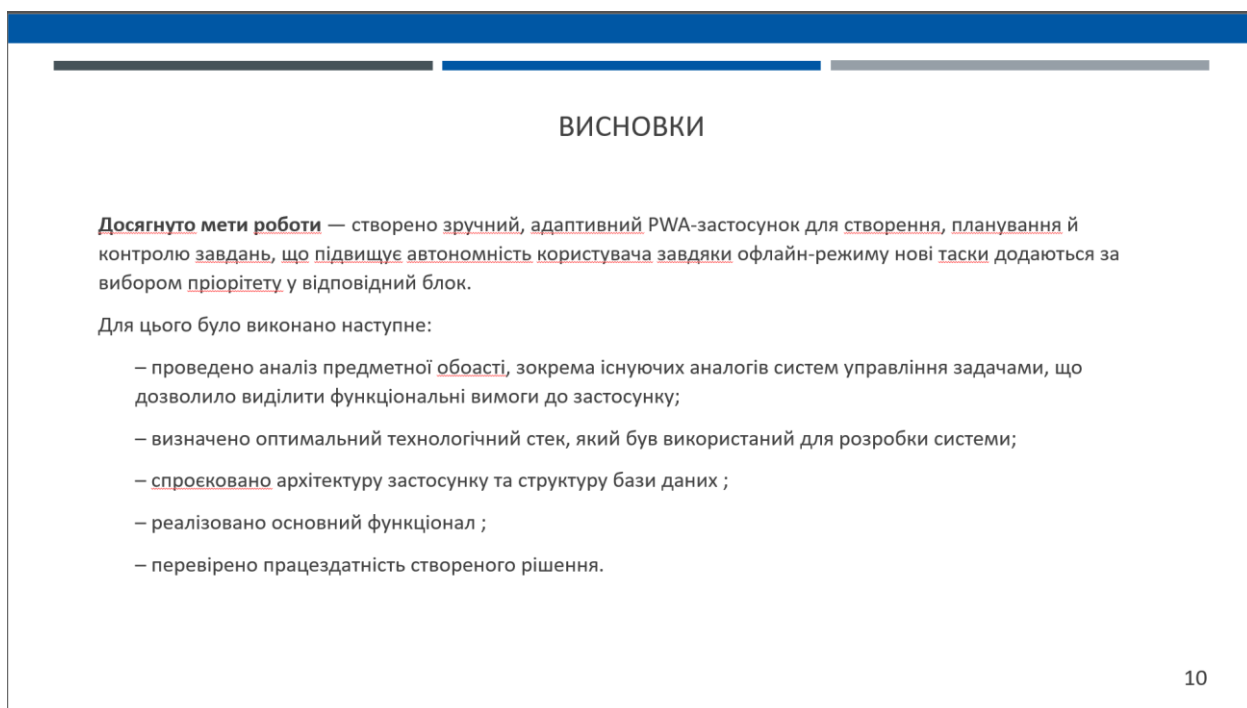


Рисунок Б.10 – Слайд 10