МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ВЕБСЕРВІСУ ДЛЯ УПРАВЛІННЯ ЗАВДАННЯМИ З АДАПТИВНИМ ІНТЕРФЕЙСОМ НА ОСНОВІ NODE.JS І REACT.JS»**

Виконав: студент 2 курсу, групи 8.1214-іпз-1
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
К.А. Кошель
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, к.ф.-м.н. Мильцев О.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кошелю Кірілу Андрійовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебсервісу для управління завданнями з адаптивним інтерфейсом на основі Node.js і React.js

керівник роботи Мильцев Олександр Михайлович, к.ф.-м.н.
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 04.12.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Основні теоретичні відомості.
2. Проєктування системи.
3. Розробка вебсервісу для управління завданнями.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	28.03.2025	
2.	Збір вихідних даних.	25.04.2025	
3.	Обробка методичних та теоретичних	04.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	20.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	18.12.2025	

Студент _____ К.А. Кошель
(підпис) (ініціали та прізвище)

Керівник роботи _____ О.М. Мильцев
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка вебсервісу для управління завданнями з адаптивним інтерфейсом на основі Node.js і React.js»: 97 с., 22 рис., 5 табл., 11 джерел, 5 додатків.

АДАПТИВНИЙ ІНТЕРФЕЙС, БАЗА ДАНИХ, ВЕБЗАСТОСУНОК, ВЕБСЕРВІС, ПРОГРАМНА ІНЖЕНЕРІЯ, УПРАВЛІННЯ ЗАВДАННЯМИ, API, NODE.JS, REACT.JS.

Об’єкт дослідження – процес організації та підтримки роботи користувачів у вебсередовищі для створення, редагування, планування та контролю виконання завдань.

Мета роботи: створення вебсервісу для управління завданнями, який забезпечує зручний і адаптивний інтерфейс, стабільну серверну логіку, а також можливість ефективної взаємодії користувачів із даними щодо планування та контролю робочих процесів.

Метод дослідження – методи аналізу вимог до програмного забезпечення, методи структурного й об’єктно-орієнтованого моделювання, методи проєктування вебзастосунків, методи розробки клієнтської та серверної частин програмних систем, а також методи тестування функціональних можливостей.

У результаті виконання кваліфікаційної роботи отримано функціональний вебсервіс для управління завданнями, який надає користувачам можливість створювати, редагувати, упорядковувати та контролювати виконання завдань у зручному адаптивному інтерфейсі. Результати роботи можуть бути використані як основа для подальшого розширення функціональності або інтеграції з іншими інформаційними системами.

SUMMARY

Master’s qualifying paper “Development of a Web Service for Tasks Managing with an Adaptive Interface Based on Node.js and React.js”: 97 pages, 22 figures, 5 tables, 11 references, 5 supplements.

ADAPTIVE INTERFACE, DATABASE, WEB APPLICATION, WEBSERVICE, SOFTWARE ENGINEERING, TASK MANAGEMENT, API, NODE.JS, REACT.JS.

The object of the study is the process of organizing and supporting user work in a web environment for creating, editing, planning and controlling task execution.

The aim of the study is to create a web service for task management, which provides a convenient and adaptive interface, stable server logic, as well as the possibility of effective user interaction with data for planning and controlling workflows.

The methods of research are analyzing software requirements, methods of structural and object-oriented modeling, methods of designing web applications, methods of developing client and server parts of software systems, as well as methods of testing functionality.

As a result of the qualification work, a functional web service for task management was obtained, which provides users with the ability to create, edit, organize and monitor the execution of tasks in a convenient adaptive interface. The results of the work can be used as a basis for further expansion of functionality or integration with other information systems.

ЗМІСТ

Завдання на кваліфікаційну роботу……..……..2
Реферат 4
Summary 5
Вступ..... 8
1 Теоретичні основи вебсервісів..... 11
1.1 Базовий огляд предметної області..... 11
1.2 REST API: принципи, методи, структура та стандартизація..... 14
1.3 Node.js як серверна платформа..... 19
1.3.1 Загальна інформація 19
1.3.2 Подієва модель та неблокуюче введення/виведення 20
1.3.3 Express.js як основа побудови вебсервісів 22
1.3.4 Робота з базами даних у Node.js..... 23
1.4 Технології фронтенду 24
1.5 Адаптивний дизайн та UX-практики 30
1.6 Аналіз існуючих систем управління завданнями 32
2 Проєктування вебсервісу..... 38
2.1 Формування вимог до системи 38
2.2 Архітектура системи..... 42
2.2.1 Проєктування архітектури клієнт-серверної взаємодії та організаціъ даних 42
2.2.2 Технологічний стек..... 48
2.3 Проєктування бази даних..... 53
2.4 Проєктування фронтенд-інтерфейсу..... 57
2.5 Моделі бізнес-процесів..... 62
3 Програмна реалізація вебсервісу..... 73
3.1 Середовище розробки..... 73
3.2 Реалізація серверної частини 75

3.3 Реалізація клієнтської частини 78
Висновки 85
Перелік посилань 87
Додаток А Файл docker-compose.yml 89
Додаток Б Файл Prisma Migrate 90
Додаток В Контролер авторизації 91
Додаток Г Контролер задач 93
Додаток Д Компонент TaskCard 96

ВСТУП

У сучасних умовах цифрової трансформації особливе значення набувають веборієнтовані системи, які забезпечують ефективну організацію діяльності користувачів, автоматизацію рутинних операцій та підвищення продуктивності у різних сферах. Одним із ключових напрямів розвитку інформаційних технологій є створення інструментів для управління завданнями, адже сучасні користувачі – як індивідуальні, так і корпоративні – постійно стикаються з необхідністю структурування робочих процесів, планування активностей і контролю виконання задач. Висока динамічність робочого середовища та потреба у швидкій адаптації до змін зумовлюють актуальність розробки гнучких, адаптивних та інтуїтивно зрозумілих вебсервісів.

Сучасні вебтехнології дозволяють створювати комплексні рішення, здатні забезпечити якісно новий рівень взаємодії користувача з інтерфейсом і можливість безперервної роботи з даними у реальному часі. Особливої уваги потребують інструменти управління завданнями, оскільки вони є важливою складовою організації ефективної діяльності як окремої особи, так і групи користувачів. Наявні рішення далеко не завжди відповідають сучасним вимогам щодо адаптивності інтерфейсу, швидкодії та зручності використання, що формує потребу у створенні нового вебсервісу, орієнтованого на доступність, масштабованість та гнучкість.

Метою кваліфікаційної роботи є розробка вебсервісу для управління завданнями з адаптивним інтерфейсом на основі технологій Node.js і React.js, який забезпечуватиме ефективну організацію завдань, їх класифікацію, планування та контроль виконання.

Досягнення поставленої мети передбачає реалізацію таких завдань:

– провести аналіз предметної області та визначити специфіку процесів управління завданнями;

- сформувати перелік функціональних і нефункціональних вимог до вебсервісу;
- розробити архітектуру системи, обґрунтувавши вибір технологій Node.js та React.js;
- спроєктувати інтерфейс користувача з урахуванням принципів адаптивності та UX-орієнтованості;
- реалізувати серверну та клієнтську частини застосунку;
- виконати тестування функціональних компонентів і проаналізувати результати роботи системи.

Об’єктом дослідження є процес організації та управління завданнями в умовах використання веборієнтованих інформаційних систем.

Предметом дослідження є методи, підходи та програмні рішення для проєктування та розробки вебсервісу управління завданнями з адаптивним інтерфейсом на базі Node.js і React.js.

Методи дослідження включають аналіз та формалізацію вимог до системи, методи структурного та об’єктно-орієнтованого моделювання, порівняльний аналіз вебтехнологій, методи проєктування архітектури програмних систем, а також методи тестування і верифікації функціональності.

Вибір Node.js як основної технології серверної частини зумовлений його високою продуктивністю, неблокуючою моделлю вводу-виводу та розвиненою екосистемою модулів. Використання React.js для клієнтської частини забезпечує побудову динамічного інтерфейсу, можливість ефективного управління станом та створення компонентної структури, що сприяє масштабованості й повторному використанню коду. Для зберігання даних передбачається застосування реляційної або документно-орієнтованої бази даних (залежно від моделі зберігання), що забезпечує надійність та швидкий доступ до інформації.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків, переліку використаних джерел та додатків.

У першому розділі подано огляд предметної області, визначено актуальність створення вебсервісу, охарактеризовано основні властивості процесів управління завданнями та здійснено теоретичне обґрунтування необхідності розробки адаптивного інтерфейсу.

У другому розділі розглянуто процес проєктування системи: сформульовано вимоги, розроблено архітектуру, описано структурні та поведінкові діаграми, спроєктовано інтерфейс користувача.

Третій розділ присвячено реалізації вебсервісу. У ньому подано опис серверної частини, клієнтського інтерфейсу, механізмів взаємодії між компонентами та проведено тестування основних модулів системи.

Результатом виконаної роботи є створення вебсервісу для управління завданнями з адаптивним інтерфейсом, який забезпечує зручну організацію робочого процесу, підвищує ефективність користувачів та демонструє можливості сучасних вебтехнологій у побудові продуктивних інформаційних систем.

1 ТЕОРЕТИЧНІ ОСНОВИ ВЕБСЕРВІСІВ

1.1 Базовий огляд предметної області

У сучасному швидкоплинному цифровому світі бізнес процвітає не лише завдяки традиційним операціям. Він розвивається, адаптуючи, модернізуючи та впроваджуючи технології, які допомагають йому функціонувати розумніше та швидше. Цей зсув спонукав організації до впровадження надійних послуг з розробки вебдодатків, які підтримують як внутрішні операції, так і діяльність, орієнтовану на клієнтів.

Сучасні компанії значною мірою покладаються на цифрові платформи для комунікації, адаптації, співпраці, аналітики, продажів та щоденних робочих процесів. Саме тут професійно розроблені вебсервіси мають величезне значення. Вони дозволяють компаніям створювати безпечні, гнучкі та швидко реагуючі цифрові системи, на які користувачі можуть покластися будь-коли. Незалежно від галузі – роздрібна торгівля, фінанси, логістика, охорона здоров’я чи освіта – зараз кожен залежить від надійних онлайн-платформ для ефективної роботи [1].

Управління завданнями є важливою складовою сучасних інформаційних систем, які забезпечують організацію індивідуальної та колективної діяльності в умовах швидкого розвитку цифрових технологій. Зі зростанням обсягів інформації, що обробляється користувачем, виникає потреба у структурованих інструментах, здатних оптимізувати планування, пріоритизацію та контроль виконання різноманітних робіт. У минулому такі процеси здійснювалися переважно вручну за допомогою паперових записників, календарів або простих текстових файлів. Проте сучасні умови функціонування як бізнесу, так і освітніх та адміністративних структур вимагають використання цифрових платформ, які надають користувачеві універсальний доступ до даних, забезпечують спільну роботу над завданнями, дозволяють

контролювати прогрес та зберігають історію змін. Саме тому вебсервіси для управління завданнями стали важливими інструментами в діяльності команд та окремих користувачів.

Предметна область управління завданнями охоплює широкий спектр понять і механізмів, спрямованих на організацію робочого процесу. В основі такої діяльності лежить поділ роботи на окремі задачі, для яких визначаються статуси, дедлайни, виконавці, пріоритети та інші параметри. Подібна модель дозволяє користувачам систематизувати свою роботу, контролювати навантаження, вчасно реагувати на зміни та координувати дії всередині команди. У межах вебзастосунку це реалізується через відповідні функції: створення, редагування та видалення завдань, керування їхніми атрибутами, формування категорій, відстеження прогресу та інше. Важливою рисою сучасних систем є наявність можливості спільної роботи, коли зміни одного користувача повинні бути негайно доступні іншим учасникам процесу.

Широке використання вебтехнологій зумовило появу великої кількості інструментів для роботи із завданнями, серед яких особливу цінність мають ті, що працюють онлайн та не потребують інсталяції на пристрої користувача. Вебсервіси такого типу дають змогу взаємодіяти з системою з будь-якої точки світу, використовуючи будь-який сучасний браузер, що значно розширює їхній потенціал і робить їх незамінними у віддаленій роботі. Важливим аспектом є також універсальність, яка забезпечується за допомогою адаптивних інтерфейсів. Значна частина користувачів сьогодні працює зі смартфонів, тому інтерфейс вебзастосунку повинен коректно масштабуватися та забезпечувати комфортну взаємодію з елементами управління на різних екранах. Наявність адаптивного дизайну перетворюється на обов’язкову характеристику якісного вебсервісу, адже саме він визначає доступність та зручність використання системи в ширшому контексті.

Ідея створення вебсервісу для управління завданнями ґрунтується на прагненні надати користувачеві універсальний інструмент для планування, організації та аналізу своєї діяльності. В основі проєкту лежить концепція

централізованого простору, де користувач може зберігати всі робочі чи особисті завдання, впорядковувати їх за категоріями, встановлювати часові межі, контролювати прогрес та підтримувати логічну структуру своєї роботи. Такий підхід сприяє підвищенню продуктивності, дозволяє уникнути хаотичності у виконанні справ і знижує ризики втрати важливої інформації.

Потреба у подібному сервісі визначається тим, що сучасний користувач цінує не лише функціональність, а й зручність взаємодії. Інтерфейс повинен швидко адаптуватися до поведінки користувача, контексту його роботи та різних розмірів екрану. Важливо, щоб робота із завданнями була інтуїтивною, логічною та ненавантаженою зайвими елементами. Саме тому концепція адаптивності та орієнтації на користувача є центральною в розробці цього проєкту. Проєкт прагне створити комфортний цифровий простір для тих, хто хоче ефективно структурувати свої справи й мати доступ до них у будь-який момент.

Системи управління завданнями розв’язують низку проблем, пов’язаних із хаотичністю інформації, відсутністю централізованого доступу до даних, складністю координації роботи між учасниками команди та недостатньою структурованістю процесів. Вебсервіс дає змогу об’єднати всі необхідні інструменти в одній платформі, забезпечуючи прозорий і логічний робочий процес. Він дозволяє користувачу швидко створювати завдання, призначати відповідальних, контролювати терміни виконання, фіксувати зміни та аналізувати результати діяльності. Таким чином, предметна область включає як технологічні аспекти – вибір архітектури, інструментів розробки, методів взаємодії з базою даних та організації інтерфейсу, – так і організаційні, пов’язані з оптимізацією робочого процесу і взаємодією між користувачами.

Предметна область управління завданнями є багатогранною та охоплює широкий спектр теоретичних і практичних аспектів. В умовах цифровізації вона відіграє ключову роль у забезпеченні ефективної організації діяльності як окремих осіб, так і колективів. Сучасні вебтехнології, зокрема Node.js і React.js, відкривають широкі можливості для створення функціональних,

адаптивних, масштабованих і доступних систем. Розуміння особливостей предметної області формує базу для проєктування вебсервісу управління завданнями, що буде детально розглянуто у наступних розділах роботи.

Підсумовуючи, можна зазначити, що предметна область управління завданнями є багатогранною та охоплює широкий спектр теоретичних і практичних аспектів. В умовах цифровізації вона відіграє ключову роль у забезпеченні ефективної організації діяльності як окремих осіб, так і колективів. Сучасні вебтехнології відкривають широкі можливості для створення функціональних, адаптивних, масштабованих і доступних систем. Розуміння особливостей предметної області формує базу для проєктування вебсервісу управління завданнями.

1.2 REST API: принципи, методи, структура та стандартизація

Архітектурний стиль REST (Representational State Transfer) посідає ключове місце в сучасних підходах до побудови вебсервісів, що забезпечують можливість масштабованого, гнучкого та стандартизованого обміну даними між різними компонентами програмних систем. REST був сформульований Р. Філдінгом як результат дослідження архітектури вебу, і з часом перетворився на один із найпоширеніших механізмів організації інтеракції між клієнтськими та серверними частинами програмних продуктів. Основна цінність REST полягає у спрощеному підході до структурування API, де кожний ресурс системи може бути представлений у вигляді унікального ідентифікатора, а взаємодія з ним здійснюється через стандартизовані операції протоколу HTTP.

Чудові продукти розробляються з чіткою орієнтацією на клієнта, глибоким розумінням клієнтів, їхніх потреб та бажань. І з цими складовими можна створювати чудовий клієнтський досвід навколо продукту. Щоб розробити чудові API, спочатку потрібно усвідомити, що API насправді є

продуктами. API – це самі по собі продукти! API ніколи не надаються безпосередньо кінцевим користувачам, але вони все одно мають клієнтів. Клієнти – це розробники споживача API. Тому, створюючи чудовий клієнтський досвід, насправді потрібно створити чудовий досвід розробника. Саме тут на допомогу приходить REST. REST дозволяє легко створити чудовий досвід розробника. REST накладає кілька архітектурних обмежень на дизайн API. Обмеження часто розглядаються як обмеження, але насправді це не так – вони допомагають розробнику. Треба розглядати обмеження REST не як обмеження, а радше як підтримку для створення чудових API, тобто API, які є простими, зрозумілими, чистими та доступними. Звичайно, потрібно знати про ці обмеження REST та як їх використовувати [2].

Одним із важливих аспектів REST є акцент на представленні ресурсів. Ресурс у цьому контексті розуміється як будь-яка сутність, яка може бути ідентифікована та оброблена системою: документ, масив даних, користувач, окреме завдання або колекція об’єктів. Ресурси описуються у вигляді URL-адрес і можуть мати різні представлення – наприклад, JSON або XML. Завдяки цьому REST-орієнтовані системи залишаються гнучкими та здатними до адаптації під різні вимоги клієнтів.

REST базується на шести фундаментальних обмеженнях, дотримання яких визначає відповідність сервісу архітектурному стилю. Першим є клієнт-серверна архітектура, яка забезпечує чіткий розподіл відповідальності між компонентами: клієнт відповідає за інтерфейс користувача та ініціювання запитів, тоді як сервер зосереджується на бізнес-логіці та управлінні даними. Такий поділ сприяє незалежному розвиткові клієнтів і серверів, підвищує масштабованість і зменшує рівень зв’язності.

Другим принципом є відсутність стану (statelessness). Кожний HTTP-запит має містити всю необхідну інформацію, яка дозволяє обробити його без урахування попередніх запитів. Сервер не зберігає інформацію про стан сесії клієнта між окремими зверненнями. Це спрощує масштабування, оскільки будь-який сервер у кластері може обробити будь-який запит, не потребуючи збережених контекстів.

Третім є кешування, що дозволяє зменшити навантаження на сервер та підвищити швидкість взаємодії, коли відповідь може бути використана повторно. HTTP підтримує низку механізмів кешування, які REST рекомендує застосовувати з метою зниження затримок та підвищення продуктивності.

Четвертим принципом є уніфікований інтерфейс, що передбачає одноманітний спосіб взаємодії з ресурсами. Уніфікованість досягається через стандартизацію URI-адрес, використання фіксованого набору HTTP-методів, застосування стандартних кодів відповідей та передбачуваної структури запитів і відповідей.

П’ятим принципом є шарова система, у межах якої клієнт може взаємодіяти з проміжними серверами, не маючи інформації про їхню кількість або функції. Це дозволяє реалізовувати балансувачі навантаження, кешування та інші проміжні механізми.

Шостим є код за запитом (code-on-demand), який є опціональним і дозволяє серверу передавати клієнту виконуваний код (наприклад, скрипти). Хоча цей механізм використовується рідше, він є частиною повного набору REST-принципів.

REST-архітектура активно використовує стандартні можливості протоколу HTTP, зокрема його методи. Кожен HTTP-метод відображає одну з CRUD-операцій над ресурсами: створення, отримання, оновлення або видалення.

Метод GET використовується для отримання інформації про ресурс або колекцію ресурсів без внесення змін у їхній стан. Оскільки GET не впливає на дані, його вважають безпечним (safe) і ідемпатентним (idempotent). Методи цього типу можуть бути кешовані, що важливо для підвищення ефективності роботи клієнтів.

Метод POST призначений для створення нових ресурсів або виконання операцій, які не є ідемпатентними. Він передає на сервер дані, що мають бути оброблені або збережені. У контексті REST POST найчастіше використовується для додавання нових об’єктів у колекцію.

Метод PUT є ідемпатентним і застосовується для повного оновлення ресурсу. Кожен повторний PUT-запит повинен призводити до однакового результату, що є важливим для передбачуваності поведінки API.

Метод PATCH використовується для часткового оновлення ресурсу. На відміну від PUT, PATCH дозволяє передавати лише ті поля, що мають бути змінені, що оптимізує передачу даних у випадках великих об’єктів.

Метод DELETE видаляє зазначений ресурс. Оскільки результат операції після успішного видалення не змінюється при повторі запиту, DELETE також є ідемпатентним.

Правильне застосування цих методів забезпечує прозору й передбачувану логіку API, що полегшує роботу розробників та інтеграцію систем.

Раціональна структура URL-адрес є однією з найважливіших складових REST API. REST рекомендує використовувати іменники в множині для опису колекцій ресурсів та уникати дієслів, оскільки дії визначаються HTTP-методами. Наприклад, адреса /tasks може представляти колекцію завдань, а /tasks/15 – конкретний елемент цієї колекції.

Ієрархічність структури дозволяє організовувати вкладені ресурси. Якщо, наприклад, завдання належать певному користувачеві, можливим є використання маршруту /users/10/tasks, що вказує на зв’язок між сутностями.

REST також передбачає можливість використання фільтрації, сортування й пагінації через параметри запиту: /tasks?status=completed&sort=created_at.

Такі механізми забезпечують гнучкість при отриманні великих обсягів даних та оптимізують роботу клієнтських застосунків.

Стандартизована структура повідомлень є ключовою для взаємодії між клієнтом і сервером. У межах REST зазвичай використовується формат JSON, однак можлива робота з XML, YAML або іншими форматами.

REST не визначає обов’язкових полів структури відповіді, але внаслідок широкої практики сформувалися загальні рекомендації.

Типова відповідь містить такі компоненти:

– тіло повідомлення (body), що містить дані ресурсу;
– заголовки HTTP (headers), які несуть метаінформацію;
– код стану (status code), що інформує про результат операції.

У таблиці 1.1 представлені коди відповідей стандартизації REST API.

Таблиця 1.1 – Коди відповідей стандартизації REST API

200 OK	Успішне отримання інформації
201 Created	Створення нового ресурсу
400 Bad Request	Некоректні дані або запит
401 Unauthorized	Відсутність доступу без автентифікації
404 Not Found	Ресурс не знайдено
500 Internal Server Error	Загальна помилка сервера

Застосування стандартних кодів дозволяє клієнтам швидше розуміти результат операції та реагувати відповідно до її типу.

REST API часто працює в умовах, де необхідний контроль доступу до ресурсів. Оскільки базовий протокол HTTP є безстанним, автентифікація і авторизація реалізуються через окремі механізми.

Найпоширенішими є токен-орієнтовані підходи, зокрема:

– передання токенів у заголовку Authorization;
– використання короткострокових токенів доступу;
– застосування додаткових механізмів шифрування інформації.

Безпека REST API також залежить від правильного використання HTTPS, обмеження можливостей CORS, валідації даних та захисту від типових вразливостей вебдодатків.

Для забезпечення коректної інтеграції з REST API важливе місце посідає документування. Популярними підходами є використання специфікацій OpenAPI або Swagger. Такі інструменти дозволяють автоматично генерувати документацію, створювати інтерактивні середовища для тестування запитів, а також застосовувати опис API як контракт між клієнтами та сервером.

1.3 Node.js як серверна платформа

1.3.1 Загальна інформація

Стрімкий розвиток вебтехнологій та зростання потреб у високопродуктивних серверних рішеннях спричинили появу широкого спектра інструментів, орієнтованих на обробку великої кількості мережевих запитів. Одним із ключових серед них стала платформа Node.js.

Node.js – це захоплива нова платформа для розробки вебзастосунків, серверів застосунків, будь-якого мережевого сервера чи клієнта, а також програмування загального призначення. Вона розроблена для надзвичайної масштабованості в мережевих застосунках завдяки геніальному поєднанню серверного JavaScript, асинхронного вводу/виводу, асинхронного програмування, побудованого на основі анонімних функцій JavaScript, та архітектури з одним потоком виконання, керованої подіями. Хоча Node.js існує лише кілька років, він швидко набув популярності та відіграє значну роль. Компанії, малі та великі, використовують його для великомасштабних проєктів. Наприклад, PayPal перевів багато сервісів з Java на Node.js.

Модель Node.js дуже відрізняється від поширених платформ серверів додатків, що використовують потоки. Стверджується, що завдяки однопотоковій подієво-керованій архітектурі обсяг пам’яті низький, пропускна здатність висока, профіль затримки під навантаженням кращий, а модель програмування простіша. Платформа Node.js перебуває у фазі швидкого зростання, і багато хто розглядає її як переконливу альтернативу традиційним підходам Java, PHP, Python, Ruby on Rails тощо до створення вебдодатків.

По суті, це окремий рушій JavaScript з розширеннями, що робить його придатним для програмування загального призначення та з чітким акцентом на розробку серверів додатків. Node.js – це середовище виконання програмування, подібне до Python, Go або Java SE. Однак існують фреймворки

вебдодатків та сервери додатків, написані на Node.js. За кілька років доступності Node.js швидко набув значної ролі, виправдовуючи прогноз про те, що він потенційно може витіснити інші стеки вебдодатків.

Він реалізований навколо неблокуючого циклу подій вводу/виводу та шару файлових і мережевих бібліотек вводу/виводу, все це побудовано на основі рушія JavaScript V8 (з веббраузера Chrome) [3].

1.3.2 Подієва модель та неблокуюче введення/виведення

Основою роботи Node.js є неблокуюча, або асинхронна, модель обробки операцій, що забезпечується за допомогою подієвого циклу (event loop). На відміну від традиційних багатопотокових серверних систем, де кожному запиту може відповідати окремий потік, у Node.js застосовується однопотокова модель виконання, яка здатна ефективно обробляти значні навантаження за рахунок відсутності витрат на перемикання контекстів.

Подієвий цикл є механізмом, що постійно відстежує запити та події, які надходять до системи, і делегує їх обробку відповідним функціям зворотного виклику. Його робота базується на чергах завдань, де кожна подія ставиться в чергу та очікує завершення попередніх операцій.

Загальна логіка виконання виглядає таким чином:

– запит або подія надходить до системи;
– node.js реєструє завдання у черзі подій;
– подієвий цикл перевіряє чергу й виконує функції, з якими пов’язана подія;
– операції, які потребують взаємодії з файловою системою, мережею або базою даних, делегуються на рівень системних потоків, не блокуючи основний процес;
– після завершення таких операцій відповідні колбеки знову повертаються до черги й виконуються у встановленому порядку.

Цей механізм забезпечує високу продуктивність у сценаріях, де значна частина операцій пов’язана з I/O-взаємодією. Однак нерідко виникає плутанина щодо багатопоточності, оскільки хоча обробка JavaScript дійсно виконується в одному потоці, сама платформа використовує пул системних потоків для операцій, що можуть виконуватися паралельно.

Неблокуючий I/O означає, що виконання коду не зупиняється до завершення операції. Наприклад, при зверненні до файлової системи сервер не очікує завершення читання і не блокує подієвий цикл, а передає обробку до відповідного системного інтерфейсу. Це дає змогу обробляти тисячі запитів одночасно з мінімальними затримками.

Сучасні високонавантажені вебсервіси часто працюють з великою кількістю конкурентних запитів, де кожен із них може потребувати доступу до бази даних, зовнішнього API чи файлової системи. Використання неблокуючої моделі дозволяє зменшити час простою процесора та забезпечити стабільність обробки запитів під час пікового навантаження.

Подієво-орієнтована модель Node.js забезпечує:

– швидку реакцію сервера на множину паралельних запитів;
– мінімізацію ресурсного навантаження, оскільки не створюються нові потоки;
– оптимальне використання процесора за рахунок підготовлених системних механізмів для обробки I/O;
– зручність організації асинхронної логіки, що особливо важливо у вебсередовищі.

Водночас така модель має й обмеження. Наприклад, виконання складних обчислювальних задач у межах одного потоку може блокувати роботу сервера. Для уникнення цього використовуються воркери, розподіл навантаження або перенесення складних операцій на зовнішні служби. Це дозволяє зберегти високу чутливість системи до нових запитів, не перевантажуючи основний потік виконання.

1.3.3 Express.js як основа побудови вебсервісів

Однією з найпоширеніших бібліотек для створення вебсервісів на платформі Node.js є Express.js.

Express.js базується на базових можливостях Node, пропонуючи фреймворк сервера вебзастосунків. Цей фреймворк забезпечує обгортку навколо інтерфейсу Node нижчого рівня: надаючи розробнику зручний засіб для обробки маршрутизації та HTTP-операцій (таких як GET та POST). Express.js пропонує спрощене та елегантніше рішення, ніж (повторна) реалізація цих сервісів безпосередньо за допомогою Node [4].

Популярність Express.js пояснюється тим, що він поєднує простоту та достатню гнучкість для створення як невеликих API, так і складних вебплатформ з розгалуженою логікою. Фреймворк не нав’язує архітектурних обмежень, однак надає інструменти, які дають змогу розробнику самостійно будувати структуру застосунку.

Маршрутизація у Express.js базується на понятті маршруту (route), що описує адресу та метод HTTP-запиту. Кожний маршрут відповідає певному обробнику, який виконується при надходженні запиту за зазначеним URL. Використання маршрутизаторів (Router) сприяє організації чіткої структури сервісу та дотриманню принципів REST.

Однією з ключових можливостей Express.js є middleware – функції, що виконуються послідовно при обробці запитів. Вони можуть виконувати такі завдання, як: логування, перевірка та обробка тіла запиту, автентифікація та авторизація, обробка помилок, виконання додаткових дій перед передачею запиту до контролера. Концепція middleware забезпечує гнучку та розширювану модель обробки запитів, у якій кожен модуль виконує лише свою частину роботи.

Функціональність Express.js може бути масштабована шляхом підключення безлічі додаткових бібліотек. Фреймворк не містить вбудованих компонентів для роботи з базами даних, тестування чи валідації, але

забезпечує можливість інтеграції будь-яких зовнішніх рішень. Серед основних характеристик Express.js виділяють: легкість інтеграції з REST-архітектурою, можливість побудови багаторівневих структур API, підтримку різних форматів відповідей, високий рівень конфігурованості, сумісність з іншими інструментами Node.js. Ці властивості роблять Express.js доцільним вибором для розробки вебсервісів, орієнтованих на обробку завдань, управління ресурсами та реалізацію API.

1.3.4 Робота з базами даних у Node.js

Побудова вебсервісу зазвичай передбачає виконання значної кількості операцій над даними: збереження, модифікація, отримання, фільтрація тощо. Тому питання взаємодії з базами даних є невід'ємною частиною функціонування серверної платформи Node.js. Платформа підтримує роботу як з реляційними, так і з нереляційними базами даних, що дозволяє обирати оптимальну інфраструктуру залежно від вимог конкретного сервісу.

Реляційні бази даних (наприклад, PostgreSQL, MySQL) організовані у вигляді таблиць із фіксованими схемами, де дані пов'язані між собою за допомогою ключів. Їх використання доцільне у ситуаціях, де важливими є цілісність даних, транзакційність та строгість структур.

У сервісах, що використовують Node.js, взаємодія з реляційними СУБД зазвичай реалізується через:

– драйвери для прямого виконання SQL-запитів;
– ORM – або query builder-рішення, що абстрагують SQL-інструкції у вигляді об'єктних моделей.

Підтримка транзакцій є важливою частиною взаємодії з реляційними базами. Транзакції дозволяють виконувати групи операцій як єдине ціле, забезпечуючи узгодженість даних у випадку одночасної роботи багатьох клієнтів. Реляційні СУБД часто використовуються у системах управління

завданнями, де є потреба у чітко структурованих зв’язках між користувачами, проєктами, статусами та іншими сутностями.

Нереляційні бази даних (наприклад, MongoDB) орієнтуються на гнучкі схеми та документно-орієнтовану структуру даних. Вони підходять для систем, де дані можуть мати змінну структуру, або для високонавантажених сервісів із вимогами до горизонтального масштабування. У таких СУБД дані зберігаються у вигляді документів (найчастіше формату JSON-подібних структурувань), що забезпечує: відсутність суворої схеми, можливість швидкої модифікації структури даних, зручність у зберіганні вкладених об’єктів, масштабування за допомогою шардінгу.

Для Node.js характерним є зручне поєднання документно-орієнтованих баз даних з JavaScript-орієнтованими структурами. Це спрощує роботу з даними та дає можливість використовувати однакові моделі об’єктів на клієнті й на сервері.

Незалежно від вибраної СУБД, у Node.js застосовуються такі загальні архітектурні принципи:

– створення окремого шару доступу до даних (Data Access Layer), що розмежовує бізнес-логіку та логіку роботи з БД;
– використання пулів з’єднань для оптимізації одночасних запитів;
– реалізація асинхронної взаємодії з базою даних для уникнення блокування подієвого циклу;
– структурування моделей даних та їх централізація;
– застосування механізмів валідації для забезпечення цілісності.

Такі підходи дають змогу підвищити якість архітектури й забезпечити стабільну роботу сервісу за умов високого навантаження.

1.4 Технології фронтенду

Розроблення фронтенд-частини вебзастосунку виступає ключовим елементом створення сучасних інформаційних систем, оскільки цей

компонент визначає інтерфейсну взаємодію кінцевого користувача із функціональністю продукту. У межах побудови інтерфейсу вебсервісу для управління завданнями виникає необхідність у застосуванні ефективних технологій, здатних забезпечити високу продуктивність, швидку реакцію інтерфейсу, оптимізацію передачі даних та стабільне відображення динамічного контенту. Вибір інструментів фронтенд-розробки має базуватися на досягненні балансу між гнучкістю, масштабованістю та простотою підтримки, що є критично важливим для систем, що оперують великою кількістю взаємопов’язаних об’єктів (завдань, проєктів, користувачів, коментарів тощо).

Одним із фундаментальних підходів до побудови клієнтської частини сучасних вебзастосунків є компонентна архітектура. Така архітектура передбачає створення інтерфейсу у вигляді сукупності окремих модулів, кожен з яких має власну локальну логіку, стилі та стан. Компонентний підхід дозволяє розробляти масштабовані системи з чіткою структурою та мінімізує ризик виникнення неузгодженості між елементами інтерфейсу. Модульність забезпечує підвищення повторного використання коду та спрощує процес модифікації інтерфейсу під час супроводу або розширення системи, що важливо при подальшому розвитку вебсервісу управління завданнями.

Для реалізації фронтенд-частини системи моніторингу доцільним є застосування фреймворку React.js. У фронтенд-розробці функція, яку виконує react.js, стає дедалі важливішою, надаючи розробникам нові можливості для створення нових додатків.

React – це проєкт JavaScript, який останнім часом домінує в екосистемі фронтенду. Поверхня React API менша, API мало для розуміння, і вони рідко змінюються. Спільнота React є однією з найбільших, і вона має потужну екосистему інструментів, компонентів інтерфейсу з відкритим кодом та чудових онлайн-ресурсів для вивчення React. Багато інструментів розробки React покращують досвід розробника. Інструменти розробника React дозволяють перевіряти, переглядати та змінювати властивості та стани компонентів у браузері. Гаряче перезавантаження за допомогою webpack

дозволяє бачити зміни коду в нашому браузері без оновлення. Розробка фронтенду вимагає редагування коду, збереження та відновлення браузера. Гаряче перезавантаження допомагає, усуваючи останній крок [5].

React.js використовує концепцію віртуального DOM, що забезпечує швидке оновлення інтерфейсу без зайвої перерендеризації елементів. Завдяки цьому вдається досягти високої продуктивності при роботі з даними, які динамічно змінюються. Фреймворк підтримує використання функціональних та класових компонентів, однак у сучасних реалізаціях перевага найчастіше надається функціональному підходу в поєднанні з хуками, які дозволяють управляти станом і життєвим циклом компонентів без застосування класів.

Для реалізації фронтенд-частини вебсервісу управління завданнями доцільним є застосування фреймворку React.js, який базується на принципах компонентності, декларативного опису інтерфейсу та управління станом. React використовує концепцію віртуального DOM, що забезпечує ефективне оновлення інтерфейсу без зайвої перерендеризації елементів. Це дозволяє підтримувати високу продуктивність при роботі з динамічними наборами даних, такими як списки завдань, фільтри, сортування та табличні відображення, де зміни можуть відбуватися часто і для великої кількості елементів. Фреймворк дозволяє застосовувати як функціональні, так і класові компоненти, але в сучасних практиках перевага надається функціональному підходу з використанням хуків, які забезпечують зручне управління станом та побічними ефектами у компонентах.

Актуальність застосування React обумовлюється його здатністю ефективно взаємодіяти з API бекенд-частини при побудові RESTful або GraphQL інтерфейсів, що забезпечує гнучке отримання й оновлення даних. У контексті вебсервісу для управління завданнями це дозволяє реалізувати такі функціональні можливості, як асинхронне завантаження списків завдань, оновлення статусів у режимі реального часу або після запитів, а також оптимізоване оновлення тільки тих частин інтерфейсу, що змінилися. Широка екосистема бібліотек для React пришвидшує розробку складних інтерфейсів та

дозволяє інтегрувати додаткові компоненти, наприклад для побудови календарів, drag-and-drop інтерфейсів, таблиць із пагінацією чи графіків прогресу.

Важливим аспектом є управління станом застосунку. Оскільки вебсервіс для управління завданнями передбачає роботу з різними об’єктами даних – задачі, проєкти, користувачі, теги, коментарі, статуси та інші – виникає потреба в централізованому зберіганні станів і в механізмах, що дозволяють узгоджено оновлювати інтерфейс.

Для управління станом розглядаються різні підходи: Context API, Redux, Zustand та інші. Обрання конкретної архітектури управління станом має ґрунтуватися на вимогах до складності системи та передбачуваної інтенсивності змін даних. Обрання конкретної архітектури управління станом має ґрунтуватися на вимогах до складності системи та передбачуваної інтенсивності змін даних. Використання паттерну централізованого сховища (Store) дозволяє уніфікувати доступ до даних та мінімізувати ймовірність розсинхронізації компонентів.

У таблиці 1.2 наведені порівняння основних інструментів управління станом.

Таблиця 1.2 – Основні інструменти управління станом

Метод управління станом	**Особливості**	**Переваги**	**Недоліки**
Context API	Вбудований механізм React; призначений для передачі глобальних значень безпосередньо в компоненти.	Простота використання, відсутність зовнішніх залежностей.	Неоптимальний для великих обсягів стану, призводить до зайвих перерендерів.
Redux	Централізований стор, строгі правила оновлення стану.	Прогнозованість, масштабованість, велика екосистема.	Відносна складність конфігурації.
Zustand	Легка бібліотека з мінімалістичним API.	Простота, висока продуктивність, селектори стану	Менша поширеність порівняно з Redux.

Вибір конкретного підходу до управління станом має визначатися характером навантаження на інтерфейс та складністю бізнес-логіки. У контексті системи управління завданнями доцільно віддавати перевагу рішенням, що мінімізують перерендери компонентів і дозволяють ефективно працювати з великими колекціями об’єктів у режимі фільтрації та сортування. Окремо має бути передбачена можливість локального кешування даних і застосування селекторів для вибіркового підключення компонентів до частин стану.

Ще одним важливим аспектом є побудова односторінкового застосунку (SPA). SPA-підхід передбачає завантаження основного пакета клієнтського коду один раз і подальше динамічне оновлення вмісту без повного перезавантаження сторінки. Така архітектура позитивно впливає на швидкість взаємодії користувача із системою, що є критично важливим для інструментів, де потрібно часто переглядати, змінювати та сортувати завдання. SPA дозволяє організувати плавну навігацію між розділами (список завдань, деталі завдання, панель керування проєктом, звіти), зменшуючи затримки, пов’язані з повторним завантаженням ресурсів, і оптимізуючи кількість запитів до сервера.

Побудова SPA зазвичай включає такі елементи:

– маршрутизацію на клієнтському боці для забезпечення переходів між логічними сторінками;
– асинхронне завантаження даних;
– зберігання стану застосунку у централізованій структурі;
– використання компонентів, що реагують на зміни стану;
– реалізацію механізмів кешування даних для покращення продуктивності.

До ключових елементів SPA-підходу належать: клієнтська маршрутизація для забезпечення переходів між логічними сторінками, асинхронне завантаження даних (lazy loading), збереження стану застосунку у централізованому сховищі, компоненти, що реагують на зміни стану, та механізми кешування для підвищення продуктивності. Впровадження цих

елементів у вебсервісі управління завданнями забезпечує можливість відображення великих масивів інформації (наприклад списків завдань із фільтрами та пагінацією), а також підтримує інтерактивні дії, такі як перетягування завдань між колонками канбан-дошки чи моментальне оновлення статусів.

Особливу увагу слід приділити інтерфейсним патернам, що підвищують зручність користування і забезпечують адаптивність. До таких патернів належать: чітка типографіка та ієрархія візуальних елементів, консистентні елементи керування (кнопки, форми, селектори), зрозумілі повідомлення про результати дій користувача (сповіщення, підказки) та механізми перевірки введених даних. Важливим елементом є також підтримка доступності (accessibility) – забезпечення можливості використання інтерфейсу людьми з обмеженими можливостями через клавіатурну навігацію, коректні ARIA-атрибути та контрастність.

Необхідним компонентом є реалізація адаптивного дизайну, що гарантує коректне відображення інтерфейсу на різних пристроях: смартфонах, планшетах, ноутбуках та десктопах. Підхід mobile-first часто використовується для забезпечення оптимізації інтерфейсу під мобільні пристрої, а також для поступового додавання складніших елементів при переході до більших екранів. Адаптивність реалізується за допомогою відносних одиниць виміру, медіа-запитів CSS, гнучких макетів та компонентних бібліотек із підтримкою респонсивності.

Серед додаткових аспектів слід зазначити інтеграцію інструментів для організації взаємодії з користувачем у реальному часі – сповіщення про оновлення завдань, зміни статусів, коментарі або запрошення до проєктів. Для реалізації таких механізмів можуть використовуватися підходи на основі періодичних опитувань (polling), довгих підключень (long polling) або вебсокетів, залежно від вимог до своєчасності оновлень і масштабованості системи. Вважається, що протокол WebSocket є найбільш ефективним для встановлення постійного двостороннього з’єднання, яке мінімізує затримки передачі даних. Оптимальним рішенням може бути застосування гібридного

підходу, де вебсокети використовуються для критично важливих подій, а менш пріоритетні оновлення обробляються через регулярні опитування.

1.5 Адаптивний дизайн та UX-практики

Термін «адаптивний вебдизайн» був введений Ітаном Маркоттом. У своїй основоположній статті List Apart [6] він об’єднав три існуючі методи (гнучке макетування сітки, гнучкі зображення, а також медіа та медіа-запити) в один єдиний підхід і назвав його «адаптивний вебдизайн». Цей термін часто використовується для позначення того ж значення, що й низка інших описів, таких як флюїдний дизайн, еластичне макетування, гумове макетування, рідкий дизайн, адаптивне макетування, крос-пристроєвий дизайн та гнучкий дизайн [7].

Адаптивний дизайн є однією з ключових складових сучасної розробки вебсервісів, оскільки забезпечує коректне відображення інтерфейсу на різних розширеннях екрана та сприяє підвищенню продуктивності роботи користувача. У межах створення вебсервісу для управління завданнями, реалізованого на основі Node.js і React.js, адаптивний підхід дозволяє сформувати інтерфейс, який коректно масштабується на екранах із різною щільністю пікселів, орієнтацією та доступною площею відображення, а також забезпечує зручну взаємодію користувача з функціональними елементами системи. Основна увага приділяється оптимізації інтерфейсу під середні й великі розміри екранів, характерні для настільних комп’ютерів та ноутбуків.

Адаптивна побудова інтерфейсу базується на використанні сучасних підходів і технологій фронтенд-розробки, зокрема TailwindCSS. Завдяки цій технології легко забезпечити створення інтерфейсу, здатного гнучко реагувати на зміну розміру вікна браузера та перерозподіляти елементи таким чином, щоб користувач отримував чітку, логічно структуровану та зручну для читання інформацію. TailwindCSS, у свою чергу, пропонує систему utility-

класів, яка спрощує реалізацію адаптивності через кастомізовані breakpoints та стилі, що змінюються залежно від ширини екрана.

У процесі побудови інтерфейсу особливе значення приділяється принципам User Experience (UX). UX-практики визначають структуру інтерфейсу, логіку переходів, послідовність виконання операцій та організацію візуальних елементів. Використання цих принципів дає змогу сформувати зручне й інтуїтивне середовище, що оптимізує процес виконання користувацьких дій, мінімізує когнітивне навантаження та забезпечує продуктивну роботу із системою.

У межах дослідження були проаналізовані ключові UX-підходи, що застосовуються в сучасних вебдодатках. Перевага надається мінімалістичній структурі інтерфейсу, чіткому контрасту між елементами, логічному групуванню функцій, наочній візуальній ієрархії та використанню підказок, спрямованих на спрощення взаємодії. Забезпечення високої юзабіліті розглядається як критичний фактор для підвищення рівня задоволеності користувачів та ефективності виконання ними завдань. Особливу увагу приділено принципам консистентності, доступності, прогнозованості та швидкої взаємодії з елементами інтерфейсу. Для досягнення цих цілей рекомендується застосовувати загальновизнані гайдлайни, що стосуються розміщення елементів та їхньої інтерактивної поведінки. Нижче наведено таблицю, у якій подано основні UX-практики.

У таблиці 1.3 наведені UX-практики та їхній вплив на інтерфейс вебсервісу. При створенні інтерфейсу адаптивність може бути реалізована на основі системи брейкпоінтів TailwindCSS, що надає можливість оптимально структурувати елементи робочої області. Основний підхід передбачає, що інтерфейс змінюється за шириною, зберігаючи при цьому логіку розташування блоків. Наприклад, при великих розширеннях екранів інформація подається у двох-трьох колонках, тоді як при зменшенні ширини відображення система автоматично трансформує макет у двоколонковий або одноколонковий режим. Особливий акцент робиться на зручності роботи із дошками завдань, списками задач та панелями навігації.

Таблиця 1.3 – UX-практики та їхній вплив на інтерфейс вебсервісу

UX-практика	**Характеристика**	**Очікуваний вплив**
Консистентність	Єдині шаблони компонентів, повторювані елементи інтерфейсу.	Зменшення когнітивного навантаження, швидке навчання користувача.
Візуальна ієрархія	Чітке розмежування головних і другорядних елементів.	Легке сприйняття інформації, швидке знаходження важливих дій.
Мінімалізм	Скорочення зайвих елементів, спрощення інтерфейсу.	Оптимізація взаємодії, зниження ризику помилок і перевантаження.
Інтерактивність	Реалізація drag-and-drop, анімації та активні стани елементів.	Підвищення ефективності управління завданнями.
Адаптивність	Підлаштування макета під різні ширини екрана.	Комфортна взаємодія незалежно від розміру робочої області.
Доступність	Контрастність, відповідні шрифти, текстові підказки.	Підвищення доступності для широкого кола користувачів.

Таким чином, адаптивний дизайн та UX-практики є невід’ємною частиною процесу розробки вебсервісів. Завдяки застосуванню сучасних інструментів та методологій можна сформувати інтерфейс, який забезпечує комфортну, передбачувану та ефективну роботу користувачів у середовищі з різними розмірами екранів. Реалізований підхід підвищує продуктивність взаємодії, мінімізує можливість помилок, оптимізує навігацію та створює основу для подальшого розширення функціональності системи.

1.6 Аналіз існуючих систем управління завданнями

У сучасних умовах цифрової трансформації процеси планування та координації роботи мають ключове значення для ефективного функціонування організацій, проєктних команд і окремих користувачів. Інструменти управління завданнями поступово стали важливими елементами інформаційної інфраструктури, забезпечуючи структурування робочих

процесів, контроль виконання, комунікацію та аналіз продуктивності. У зв'язку з високим попитом на подібні сервіси на ринку існує значна кількість платформ, орієнтованих на різні категорії користувачів, масштаби використання та сценарії взаємодії. Аналіз цих рішень є необхідним етапом для розуміння актуальних тенденцій, визначення ключових вимог до функціональності, а також ідентифікації недоліків, що можуть бути усунені в межах розроблюваного вебсервісу.

Однією з найвідоміших систем у сфері управління завданнями є Trello, що пропонує канбан-орієнтований підхід до організації робочих процесів. На рисунку 1.1 представлено відкритий проєкт у Trello.

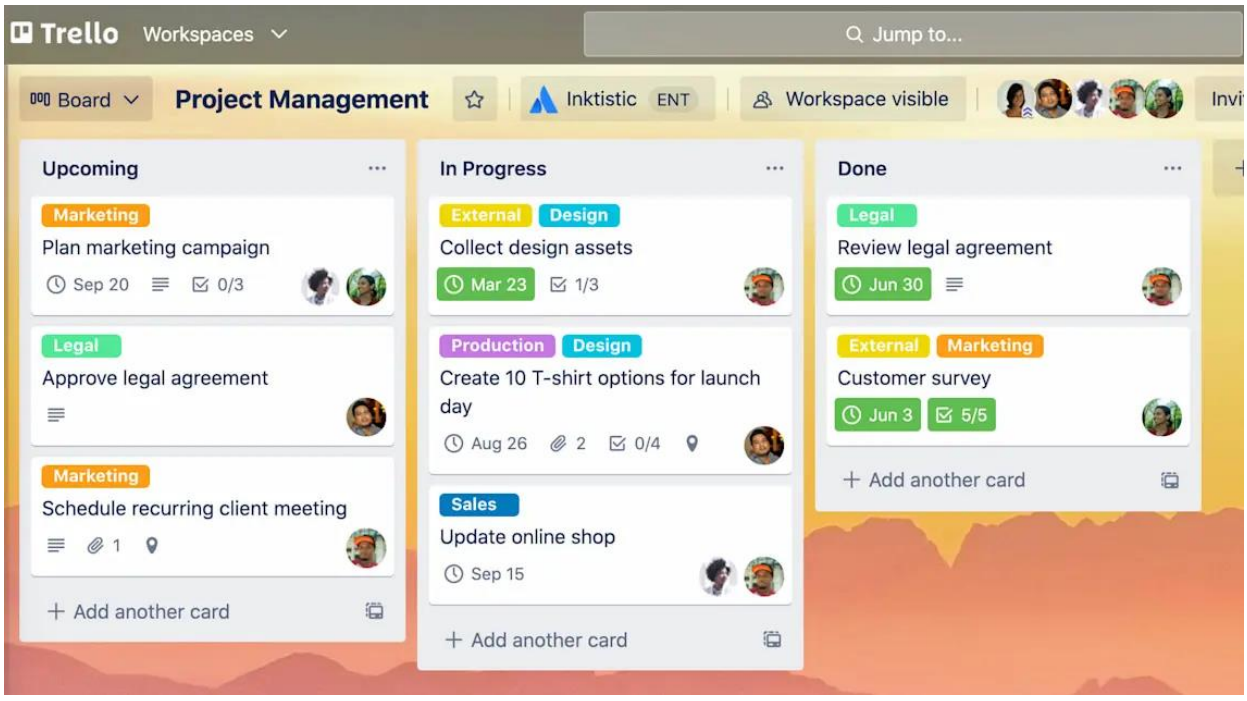


Рисунок 1.1 – Відкритий проєкт у Trello

Основна концепція полягає у представленні проєктів у вигляді дошок, які містять списки та картки. Така модель є інтуїтивно зрозумілою, забезпечує швидке переміщення завдань між статусами та підтримує гнучкість у налаштуванні структури. Перевагою Trello є простота використання, можливість адаптації під невеликі команди та доступність базових функцій без додаткових витрат. Разом із тим, у міру збільшення масштабів проєкту виникають певні обмеження, пов'язані зі складністю відстеження великої

кількості карток, обмеженою кількістю інструментів для аналітики та меншою функціональною глибинною порівняно з професійними системами.

Іншим широко використовуваним рішенням є Jira, орієнтована насамперед на команди, що працюють за гнучкими методологіями розробки. На рисунку 1.2 представлено демонстрацію роботи у Jira.

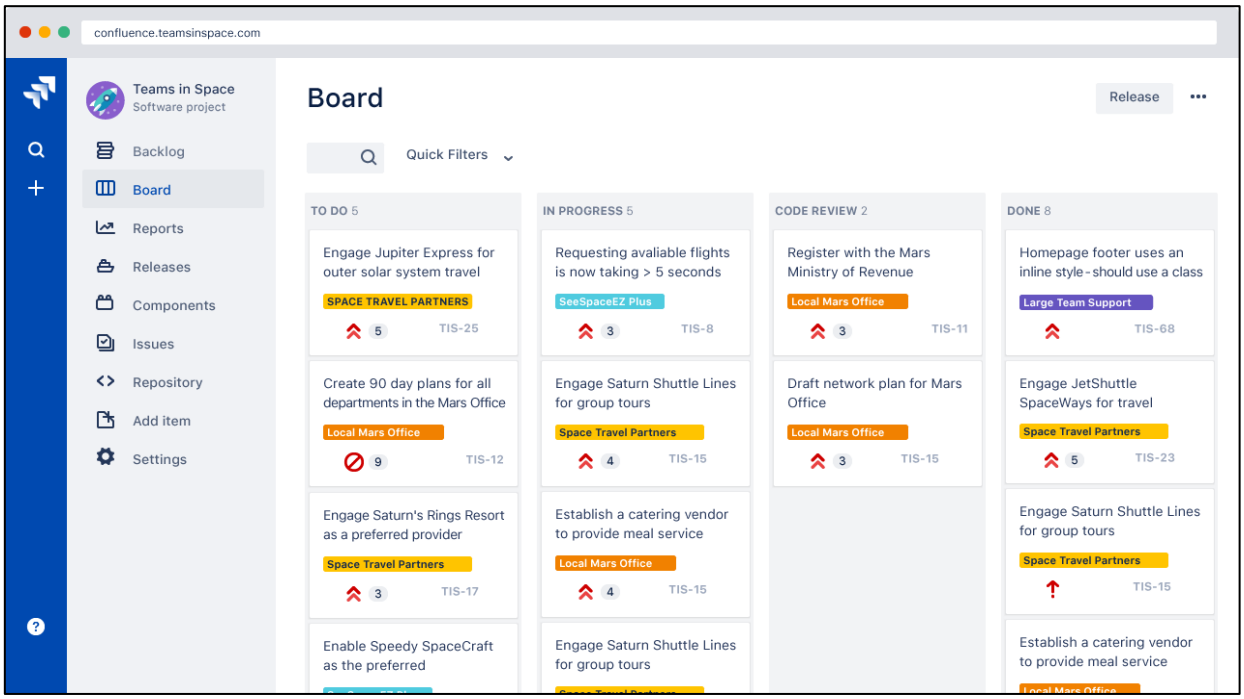


Рисунок 1.2 – Демонстрація роботи у Jira

Система пропонує широкі можливості конфігурації, розвинені інструменти управління спринтами, автоматизацію робочих процесів, а також інтеграцію з великою кількістю сторонніх сервісів. Важливим аспектом Jira є наявність розширених механізмів звітності, які дозволяють проводити аналіз ефективності роботи команди, прогнозувати завантаженість і приймати технічно обґрунтовані рішення. Проте висока функціональність супроводжується підвищеною складністю навігації та адміністрування. Для нових користувачів система є менш інтуїтивною, а її впровадження потребує додаткового навчання або спеціальної підготовки.

Ще одним популярним рішенням у сфері управління повсякденними завданнями є Todoist, орієнтований на індивідуальних користувачів та невеликі робочі групи. На рисунку 1.3 представлено відкриту дошку у Todoist.

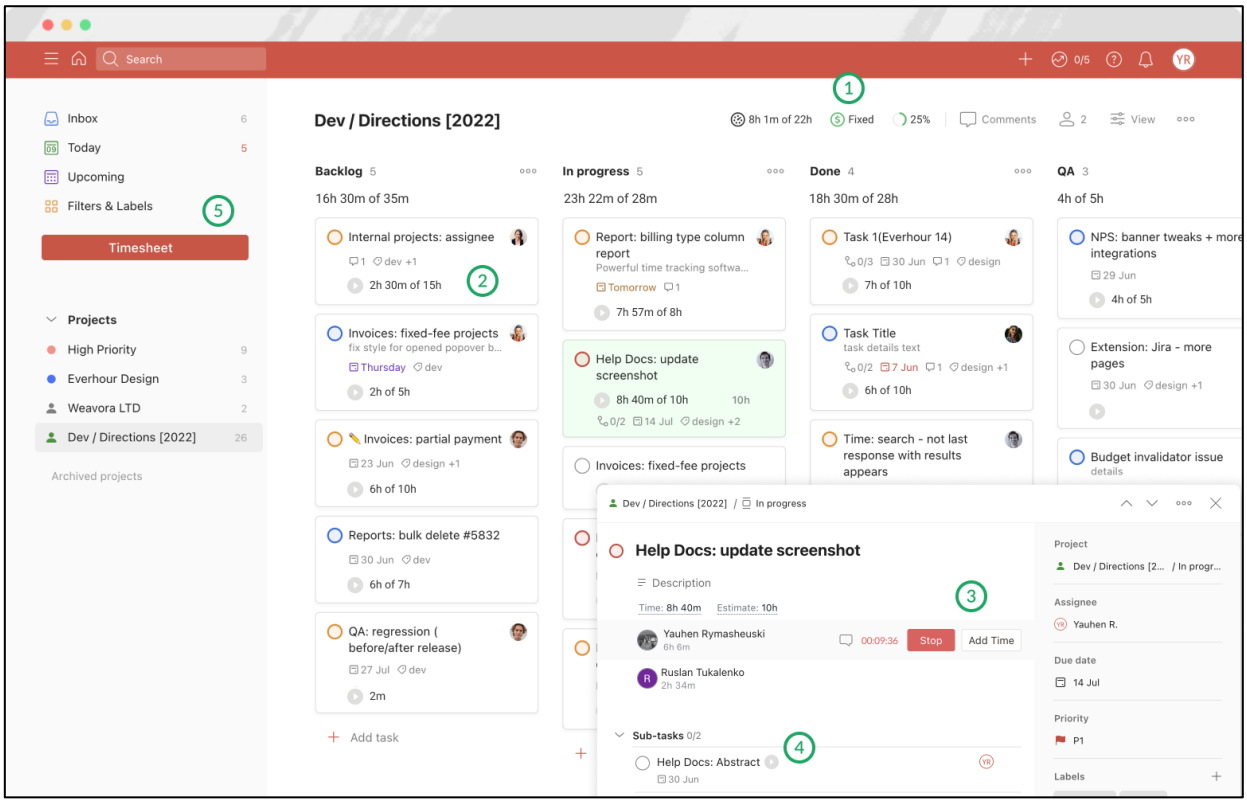


Рисунок 1.3 – Відкрита дошка у Todoist

Система пропонує класичну модель списків справ із підтримкою тегів, пріоритетів і планування. Перевагою Todoist є його мінімалістичний інтерфейс, висока швидкодія та можливість використання на різних платформах. Водночас система не орієнтована на великі команди та складні робочі процеси, що обмежує її застосування в проєктному управлінні. Недостатність інструментів для колективної взаємодії, звітності та гнучкої адаптації структури завдань робить її менш ефективною для корпоративних середовищ.

Система Asana пропонує більш універсальний підхід, поєднуючи можливості для командної роботи із підтримкою декількох моделей представлення завдань: списки, дошки, часові лінії, календарі. Завдяки цьому Asana є гнучким рішенням і може застосовуватися для команд різного масштабу. Платформа має розвинені засоби створення залежностей між завданнями, визначення виконавців, відстеження прогресу та автоматизації. Серед її недоліків користувачі відзначають складність навігації при великій кількості проєктів, обмеження безкоштовного тарифного плану та порівняно

високі вимоги до продуктивності клієнтського пристрою при роботі з великими даними.

Порівняльний аналіз зазначених систем дає змогу виявити найбільш поширені функціональні характеристики сучасних інструментів управління завданнями. До таких особливостей належать: використання канбан-дощок або гнучких моделей відображення завдань, підтримка колективної роботи, можливість інтеграції з іншими програмними продуктами, наявність аналітичних інструментів. При цьому суттєвою проблемою є складність адаптації користувацького інтерфейсу під конкретні потреби, обмеження безкоштовних тарифів, а також відсутність інструментів глибокої персоналізації без додаткового програмування.

Спостерігається тенденція до надмірного ускладнення інтерфейсів, що знижує загальну юзабіліті для нових користувачів.

Для невеликих команд часто є надмірними функції, орієнтовані на масштабні проєкти, тоді як для великих компаній не вистачає продуктивності або деталізації базових планувальників.

Таким чином, виявлено потребу в розробці рішення, яке пропонує модульну архітектуру для адаптації функціоналу. Саме тому розробка нового вебсервісу повинна враховувати необхідність поєднання простоти, гнучкості та можливості масштабування. Створення такого універсального інструменту вимагає ретельного балансу між базовою функціональністю та архітектурною гнучкістю.

Особлива увага має приділятися проєктуванню API, що забезпечує легку розширюваність та кастомізацію зовнішніми розробниками. Це забезпечить ефективне управління завданнями для команд різного розміру та спеціалізації. Впровадження гнучких налаштувань відображення та фільтрації даних вважається ключовим фактором для підвищення продуктивності кінцевого користувача.

У таблиці 1.4 наведено порівняльний аналіз існуючих систем управління завданнями.

Таблиця 1.4 – Порівняльний аналіз

Система	Основна модель роботи	Переваги	Недоліки	Цільова аудиторія
Trello	Канбан-дошка.	Простота, візуальність, швидке переміщення завдань.	Обмежені аналітичні інструменти, складність роботи з великими проєктами.	Невеликі команди, індивідуальні користувачі.
Jira	Agile-орієнтована система.	Розвинена аналітика, інтеграції, автоматизація.	Висока складність, потреба у навчанні.	Великі команди, IT-компанії.
Todoist	Списки завдань.	Мінімалізм, швидкодія, універсальність платформи.	Недостатня підтримка командної роботи, слабка аналітика.	Індивідуальні користувачі.
Asana	Списки, дошки, часові лінії.	Універсальність, залежності завдань, гнучкість.	Обмежений безкоштовний тариф, складність навігації.	Команди різного масштабу.

З огляду на це створення власного вебсервісу для управління завданнями дозволяє забезпечити точне відтворення вимог, адаптованих під конкретні сценарії використання. Можливість застосування сучасних технологій дає змогу забезпечити швидкодію, модульність і легку кастомізацію інтерфейсу. Це створює передумови для формування конкурентоспроможного рішення, яке поєднає у собі функціональну повноту, адаптивність та простоту взаємодії, що є особливо важливим у сучасному цифровому середовищі.

2 ПРОЄКТУВАННЯ ВЕБСЕРВІСУ

2.1 Формування вимог до системи

Формування вимог до програмної системи є одним із ключових етапів її проєктування, оскільки саме на цьому етапі визначається необхідний функціонал, архітектурні особливості, умови використання, а також обмеження, які впливають на подальшу реалізацію вебсервісу. Для вебзастосунку управління завданнями з адаптивним інтерфейсом процес визначення вимог охоплює аналіз цілей проєкту, визначення очікуваних сценаріїв використання, окреслення функціональних можливостей та встановлення якісних характеристик системи, що забезпечують її продуктивність, зручність та безпеку. Основною метою системи є створення повноцінного вебсервісу для керування завданнями, який містить серверну частину на Node.js з REST API, клієнтський інтерфейс на React.js, адаптивний до різних розмірів моніторів, а також можливість взаємодії з постійним сховищем даних у вигляді PostgreSQL. Під час формування вимог особливу увагу приділено підтримці сучасних принципів побудови вебзастосунків, таких як модульність, розширюваність, безпека, стійкість до некоректних даних, а також можливість подальшого масштабування.

У межах проєкту визначено низку цілей, які мають бути досягнуті в рамках розробки програмного забезпечення. Загальна мета передбачає створення системи, що забезпечує користувачеві можливість авторизації, керування завданнями, їх категоризації, редагування та відстеження прогресу виконання. Для реалізації цих можливостей необхідним є впровадження структурованого REST API, компонентної архітектури фронтенду, інтеграції із системою баз даних та реалізації інтуїтивного інтерфейсу з підтримкою drag-and-drop взаємодії. Додатковою вимогою є забезпечення різних розмірів екранів на рівні десктопних пристроїв, що передбачає застосування адаптивної

верстки, CSS-фреймворку TailwindCSS та грамотної побудови компонування елементів.

Важливою частиною визначення вимог є опис функціональних можливостей системи, які забезпечують базові сценарії користувацької взаємодії.

До функціональних вимог віднесено:

– автентифікацію користувачів;
– управління сутністю «завдання»;
– drag-and-drop взаємодію між колонками;
– синхронізацію дій користувача з сервером;
– обробку помилок і повідомлення про некоректні дії.

Функціональні вимоги в рамках системи сформульовано таким чином.

Автентифікація та авторизація:

– можливість реєстрації користувача;
– можливість авторизації з використанням JWT-токенів;
– перевірка прав доступу до ресурсів;
– обробка протермінованих або некоректних токенів.

Управління завданнями:

– створення нового завдання через POST-запит;
– редагування параметрів завдання (назви, опису, тегів, статусу);
– видалення завдань;
– можливість встановлення тегів;
– відображення завдань на дошці за колонками “Upcoming”, “In Progress”, “Done”;
– можливість перетягування завдань між колонками з використанням react-dnd;
– оновлення статусу на сервері після завершення перетягування;
– відкат стану при помилках мережі (rollback).

Інтерфейс і взаємодія:

– чітке візуальне виділення області, куди може бути переміщене завдання;

– адаптивна верстка для різних розмірів моніторів (FullHD, 2K, 4K);
– повідомлення про успішні операції або помилки;
– підтримка окремих сторінок: реєстрація, авторизація, список завдань, редактор завдання.

Серверна логіка:

– структура REST API з маршрутами /login, /register;
– валідація вхідних даних через Zod;
– інтеграція з базою даних через Prisma;
– логування критичних подій.

Інтеграція фронтенду з бекендом:

– виконання HTTP-запитів через Axios;
– автоматичне додавання токена до заголовків;
– синхронізація позицій завдань після drag-and-drop;
– відображення актуального стану колонок після відповіді від сервера.

Інфраструктурні вимоги:

– наявність Dockerfile для бекенду;
– наявність Dockerfile для фронтенду;
– використання docker-compose для підняття повного стеку;
– наявність файлу .env;
– локальний репозиторій з базовим README.md.

Окрім функціональних вимог, сформульовано набір нефункціональних вимог, що визначають експлуатаційні характеристики системи. Їх виконання гарантує стабільність і якість роботи вебзастосунку під різними навантаженнями та у різних середовищах.

Нефункціональні вимоги системи охоплюють такі категорії.

Продуктивність:

– система повинна забезпечувати швидку обробку запитів;
– оптимізація запитів до бази даних через ORM;
– швидке переміщення завдань без відчутної затримки для користувача.

Доступність:

– застосунок має забезпечувати стабільну роботу;
– система повинна коректно функціонувати в межах Docker-середовища;
– користувач повинен мати можливість використовувати сервіс з будь-якого сучасного браузера;
– інтерфейс має коректно відображатися на різних роздільних здатностях.

Безпека:

– застосування JWT для захищеної автентифікації;
– обмеження доступу до приватних ресурсів;
– валідація всіх вхідних даних.

Масштабованість і підтримуваність:

– модульна структура проєкту з поділом на бізнес-логіку, контролери та маршрути;
– можливість додавання нових функцій без суттєвої перебудови застосунку;
– використання TypeScript для підвищення надійності коду.

Юзабіліті:

– інтерфейс має бути інтуїтивно зрозумілим;
– візуальні реакції інтерфейсу на взаємодії користувача;
– простота навігації.

Визначення цих вимог забезпечує структурований підхід до проєктування системи та гарантує, що розроблений вебсервіс відповідатиме потребам користувачів, буде ефективним, безпечним і придатним для подальшого розвитку. Чітка деталізація функціональних можливостей дозволяє уникнути двозначності в розумінні завдань. Це стає фундаментом для точної оцінки часових рамок та необхідних ресурсів проєкту. Функціональні вимоги детально описують сценарії використання, такі як реєстрація, авторизація та відновлення доступу. Окремої уваги заслуговує логіка

управління життєвим циклом завдань: від їх створення та редагування до архівування. Сформовані вимоги створюють основу для побудови архітектури системи, розробки інтерфейсу, серверної логіки та інфраструктурних рішень, необхідних для якісної реалізації повноцінного вебзастосунку керування завданнями.

2.2 Архітектура системи

2.2.1 Проєктування архітектури клієнт-серверної взаємодії та організаціъ даних

Розробка вебсервісу базується на сучасних принципах побудови розподілених систем, використовуючи клієнт-серверну архітектуру з чітким розмежуванням зон відповідальності між фронтендом, бекендом та базою даних. Наведемо візуальну структуру проєкта на рисунку 2.1.

```
task-management/
├── backend/           # Backend API (Node.js + Express + Prisma)
│   ├── src/
│   │   ├── modules/   # Auth and Tasks modules
│   │   ├── config/    # Configuration
│   │   ├── middleware/# Express middleware
│   │   └── utils/     # Utilities
│   ├── prisma/        # Database schema and migrations
│   └── Dockerfile
├── frontend/          # Frontend app (React + Vite)
│   ├── src/
│   │   ├── components/# React components
│   │   ├── pages/     # Page components
│   │   ├── api/       # API client
│   │   ├── hooks/     # Custom hooks
│   │   └── context/   # React Context
│   └── Dockerfile
└── docker-compose.yml # Full stack orchestration
```

Рисунок 2.1 – Project Structure

Структурна організація системи реалізована з використанням мікросервісного підходу в частині ізоляції середовищ виконання, що забезпечується технологією контейнеризації Docker. Використання Docker Compose дозволяє оркеструвати роботу трьох основних контейнерів системи: клієнтського застосунку, серверного API та системи управління базами даних, що гарантує ідентичність середовищ розробки та розгортання.

На верхньому рівні абстракції система поділена на три логічні блоки, які взаємодіють між собою через мережеві протоколи. Першим блоком є клієнтська частина (Frontend), розроблена на базі бібліотеки React з використанням збирача Vite. Цей компонент відповідає за візуалізацію інтерфейсу та взаємодію з кінцевим користувачем. Другим блоком є серверна частина (Backend), реалізована на платформі Node.js із використанням фреймворку Express, яка виконує роль RESTful API, обробляючи запити клієнта та реалізуючи бізнес-логіку застосунку. Вибір Node.js та Express забезпечує високу продуктивність завдяки неблокуючій моделі вводу-виводу, що є оптимальним для обробки численних паралельних запитів. Третім блоком є реляційна база даних PostgreSQL, яка забезпечує надійне зберігання та цілісність даних.

Архітектура клієнтської частини спроєктована за компонентним принципом, де інтерфейс будується з ізольованих та перевикористовуваних елементів. Головним структурним елементом є сторінки, такі як BoardPage, RegisterPage та LoginPage. Сторінка BoardPage, яка є центральним елементом функціонала, компонує в собі менші компоненти, зокрема TaskColumn для відображення колонок статусів, TaskCard для візуалізації окремих завдань та TaskForm для створення або редагування записів. Такий підхід суттєво підвищує ремонтопридатність коду та спрощує процес тестування окремих елементів інтерфейсу. Управління станом застосунку реалізовано через механізм контекстів, що дозволяє уникнути надмірної передачі пропсів через ієрархію компонентів. Зокрема, виділено AuthContext для зберігання даних сесії користувача та ToastContext для відображення спливаючих повідомлень.

Застосування контексту (Context API) вважається доцільним для управління глобальними станами, які вимагають доступу з різних, віддалених частин DOM-дерева.

Для інкапсуляції логіки роботи із завданнями розроблено кастомний хук useTasks, який виступає проміжною ланкою між компонентами відображення та шаром API. Кастомні хуки ефективно забезпечують повторне використання логіки станів між різними функціональними компонентами.

Взаємодія з сервером здійснюється через окремий шар, побудований на базі бібліотеки Axios, де налаштовано окремі екземпляри для роботи з завданнями та авторизацією. Формалізація цього шару взаємодії мінімізує залежність компонентів від деталей мережевих запитів та спрощує перехід до інших API-протоколів у майбутньому.

Архітектура клієнтської частини спроєктована за компонентним принципом, де інтерфейс будується з ізольованих та перевикористовуваних елементів.

На рисунку 2.2 представлено загальну діаграму архітектури системи.

Головним структурним елементом є сторінки, такі як BoardPage, RegisterPage та LoginPage. Сторінка BoardPage, яка є центральним елементом функціонала, компонує в собі менші компоненти, зокрема TaskColumn для відображення колонок статусів, TaskCard для візуалізації окремих завдань та TaskForm для створення або редагування записів. Управління станом застосунку реалізовано через механізм контекстів, що дозволяє уникнути надмірної передачі пропсів через ієрархію компонентів. Зокрема, виділено AuthContext для зберігання даних сесії користувача та ToastContext для відображення спливаючих повідомлень.

Для інкапсуляції логіки роботи із завданнями розроблено кастомний хук useTasks, який виступає проміжною ланкою між компонентами відображення та шаром API. Взаємодія з сервером здійснюється через окремий шар, побудований на базі бібліотеки Axios, де налаштовано окремі екземпляри для роботи з завданнями та авторизацією.

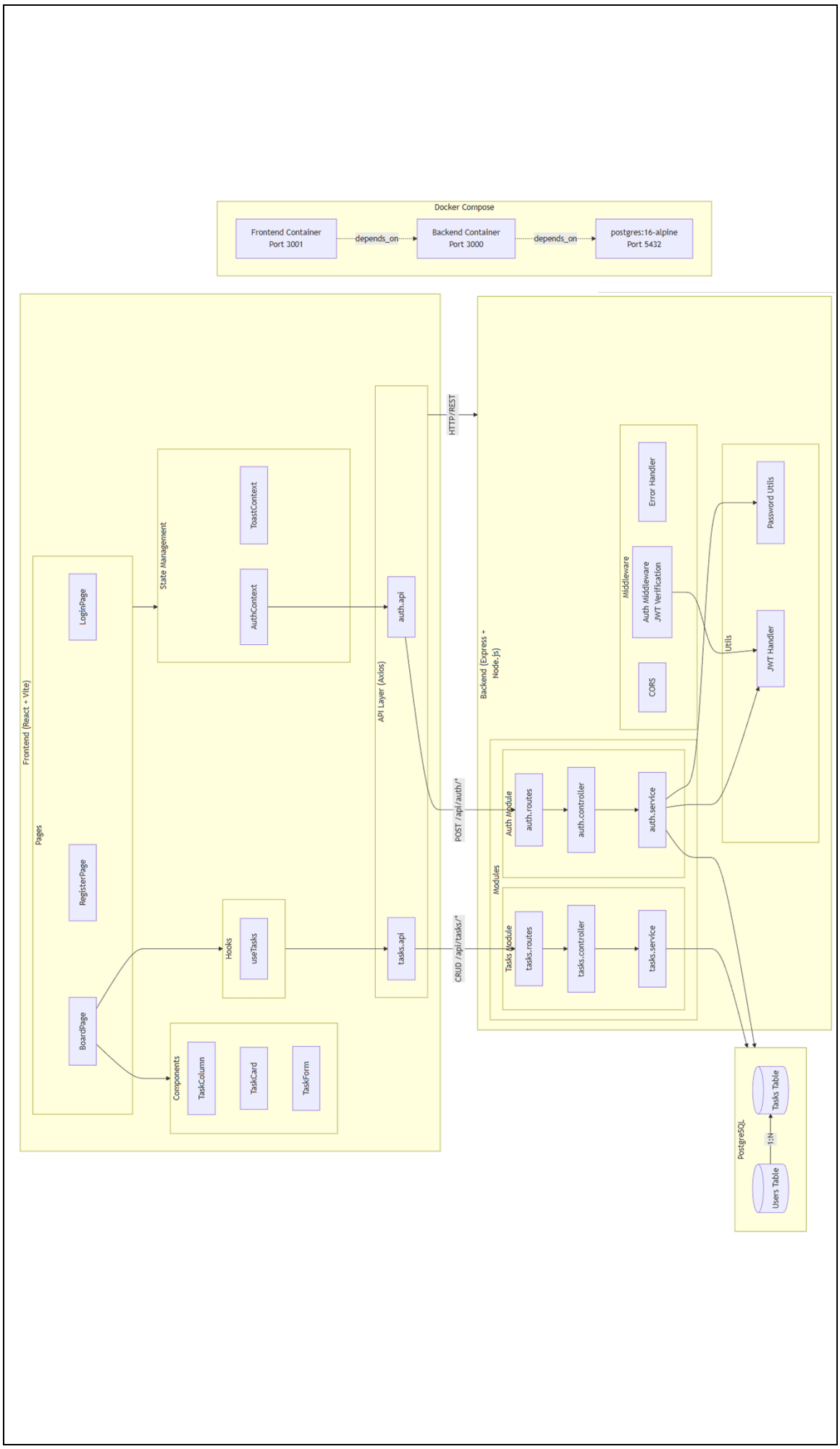


Рисунок 2.2 – Architecture Diagram

Серверна частина системи побудована за модульною, шаруватою архітектурою, що забезпечує гнучкість та легкість підтримки коду. Вхідною точкою для запитів є HTTP-сервер Express, який працює на порту 3000. Всі вхідні запити проходять через ланцюжок проміжного програмного забезпечення (Middleware). До нього входять модуль CORS для налаштування політики доступу з інших доменів, Auth Middleware для верифікації JSON Web Token JWT та Error Handler для централізованої обробки помилок. Бізнес-логіка системи розділена на функціональні модулі: модуль завдань та модуль авторизації.

Внутрішня структура кожного модуля реалізує патерн «Маршрут-Контролер-Сервіс» (Route-Controller-Service). Рівень маршрутизації відповідає за визначення ендпоінтів та перенаправлення запитів до відповідних контролерів. Контролери виконують обробку вхідних даних HTTP-запиту, їх валідацію та формування відповіді клієнту. Основна бізнес-логіка зосереджена на рівні сервісів, які виконують обчислювальні операції та взаємодіють з базою даних. Така сепарація дозволяє тестувати бізнес-логіку ізольовано від транспортного рівня HTTP. Додатково в архітектурі бекенду виділено шар утиліт, який включає JWT Handler для генерації та перевірки токенів доступу, а також Password Utils для хешування паролів, що підвищує рівень безпеки системи.

Персистентність даних забезпечується системою управління базами даних PostgreSQL, версія 16-alpine, що розгорнута в окремому контейнері та доступна через порт 5432. Схема бази даних спроєктована з урахуванням реляційної моделі та включає дві основні таблиці: Users Table для зберігання інформації про користувачів та Tasks Table для зберігання завдань. Між цими таблицями встановлено відношення «один-до-багатьох» 1:N, що означає, що один користувач може мати безліч завдань, але кожне завдання належить лише одному конкретному користувачеві.

Комунікація між клієнтською та серверною частинами здійснюється за протоколом HTTP з використанням архітектурного стилю REST. Обмін

даними відбувається у форматі JSON. Безпека взаємодії забезпечується механізмом авторизації на основі токенів. При успішній авторизації через POST, сервер генерує токен, який клієнт зберігає та додає до заголовків усіх наступних запитів до захищених ресурсів.

Для перевірки валідності цих токенів на стороні сервера реалізовано спеціальне проміжне програмне забезпечення (middleware), яке перехоплює вхідні запити. Воно автоматично блокує доступ до приватних маршрутів API та повертає відповідну помилку у разі відсутності або закінчення терміну дії ключа доступу. Також для коректної взаємодії з клієнтським застосунком налаштовано політику CORS, що дозволяє безпечний обмін ресурсами між різними джерелами.

Такий комплексний підхід до побудови забезпечує високу продуктивність системи, її масштабованість та надійний захист даних користувачів. На рисунку 2.3 представлено загальну схему взаємодії на стороні клієнта.

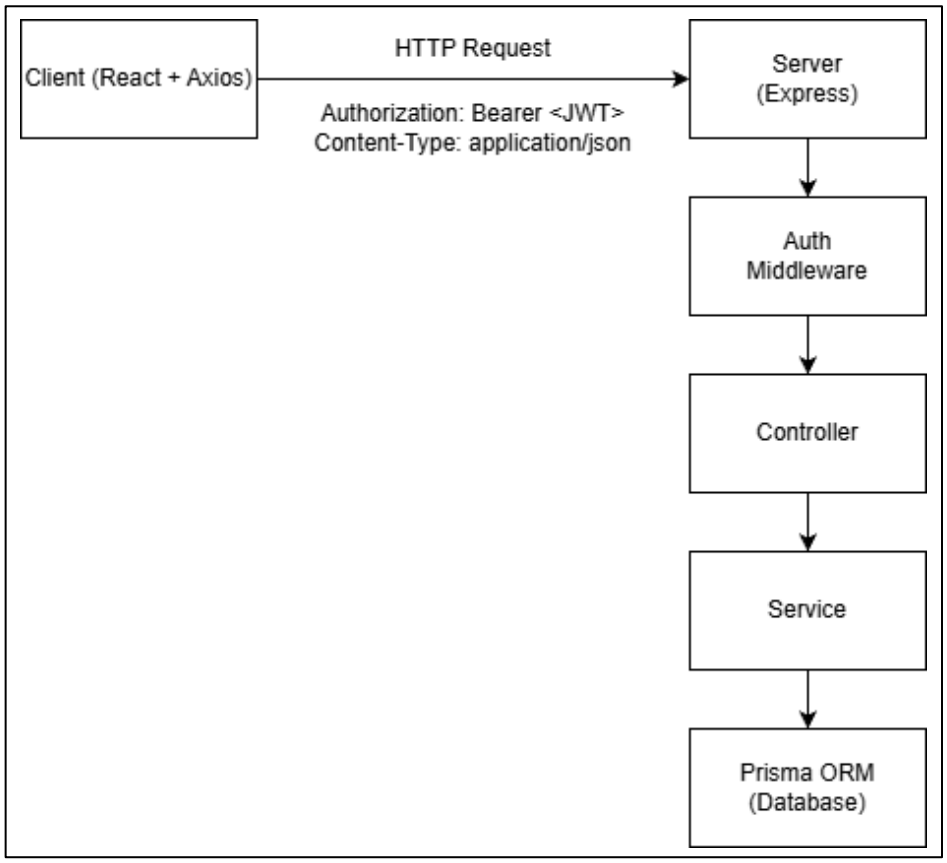


Рисунок 2.3 – Загальна схема взаємодії на стороні клієнта

Схема взаємодії ініціюється на стороні клієнта, реалізованого на базі бібліотеки React із застосуванням HTTP-клієнта Axios, шляхом формування та надсилання запиту до серверної програми.

Приклад повного циклу запиту:

– клієнт відправляє запит HTTP через Axios;
– interceptor додає JWT токен в заголовок Authorization;
– сервер отримує запит;
– auth Middleware перевіряє та декодує токен;
– controller обробляє бізнес-логіку;
– service взаємодіє з базою даних через Prisma;
– відповідь повертається клієнту у форматі JSON.

Даний підхід забезпечує цілісність процесу обробки інформації, починаючи від ініціалізації запиту користувачем і закінчуючи збереженням даних у системі. Це створює стійке підґрунтя для безпечного функціонування вебсервісу та спрощує його подальшу підтримку.

2.2.2 Технологічний стек

Розробка сучасного вебсервісу з адаптивним інтерфейсом вимагає ретельного підбору технологій, які забезпечують високу продуктивність, масштабованість та зручність розробки. Технологічний стек розробленої системи базується на використанні JavaScript/TypeScript екосистеми як для серверної, так і для клієнтської частини застосунку.

Архітектура системи побудована за принципом розділення відповідальності, де серверна частина відповідає за бізнес-логіку, роботу з базою даних та аутентифікацію користувачів, тоді як клієнтська частина забезпечує інтерактивний користувацький інтерфейс. Контейнеризація застосунку за допомогою Docker забезпечує однаковість середовища виконання на різних платформах та спрощує процес розгортання системи.

Серверна частина застосунку реалізована на платформі Node.js, яка є середовищем виконання JavaScript на стороні сервера. Node.js використовує подієво-орієнтовану модель з неблокуючим введенням-виведенням, що забезпечує високу продуктивність при обробці великої кількості одночасних з'єднань. Ця характеристика є особливо важливою для систем управління завданнями, де користувачі можуть одночасно виконувати операції перетягування задач та отримувати оновлення у реальному часі.

Фреймворк Express.js використовується як основа для побудови REST API серверної частини. Express є мінімалістичним та гнучким фреймворком для Node.js. Модульна архітектура Express дозволяє легко розширювати функціональність за допомогою проміжного програмного забезпечення middleware, що забезпечує обробку запитів, аутентифікацію, логування та інші аспекти роботи сервера. Маршрутизація в Express організована за модульним принципом, де кожен функціональний модуль системи має власний набір маршрутів та обробників.

Для постійного зберігання даних обрано реляційну систему управління базами даних PostgreSQL. PostgreSQL є потужною об'єктно-реляційною СУБД, яка відзначається надійністю, цілісністю даних та підтримкою складних запитів. Вибір реляційної бази даних обумовлений структурованим характером даних системи управління завданнями, де існують чіткі зв'язки між користувачами та їх задачами.

Взаємодія з базою даних здійснюється через Prisma ORM, який є сучасним інструментом для роботи з базами даних у Node.js та TypeScript застосунках. Prisma надає типобезпечний клієнт для виконання запитів до бази даних, систему міграцій для управління схемою бази даних та графічний інтерфейс Prisma Studio для перегляду та редагування даних. Схема бази даних визначається у декларативному форматі у файлі schema.prisma, на основі якого Prisma автоматично генерує TypeScript типи та клієнт для виконання запитів.

Система аутентифікації побудована на основі JSON Web Tokens (JWT), що є стандартизованим підходом до передачі інформації про автентифікацію

між сторонами у вигляді компактного токена. Бібліотека jsonwebtoken використовується для генерації та верифікації токенів, що містять ідентифікатор користувача та термін дії. Токени підписуються секретним ключем, що зберігається у змінних середовища, та мають обмежений термін дії для підвищення безпеки.

Паролі користувачів зберігаються у хешованому вигляді з використанням бібліотеки bcryptjs, яка реалізує алгоритм bcrypt для криптографічного хешування. Bcrypt є адаптивною функцією хешування, яка використовує сіль налаштовуваний коефіцієнт складності, що робить атаки перебором практично неможливими. При реєстрації пароль хешується з використанням випадкової солі, а при вході в систему введений пароль порівнюється з збереженим хешем.

Валідація вхідних даних здійснюється за допомогою бібліотеки Zod, яка надає декларативний спосіб опису схем валідації з повною підтримкою TypeScript.

Клієнтська частина застосунку розроблена з використанням бібліотеки React. React використовує компонентний підхід до розробки, де інтерфейс розбивається на незалежні багаторазово використовувані компоненти, кожен з яких управляє власним станом та логікою відображення.

Як інструмент збірки та розробницький сервер використовується Vite, який є сучасною альтернативою традиційним збирачам на кшталт Webpack.

TypeScript використовується як основна мова програмування для клієнтської частини, забезпечуючи статичну типізацію та покращене виявлення помилок на етапі розробки. Строга типізація особливо важлива при роботі з даними API та станом компонентів, оскільки дозволяє уникнути цілого класу помилок, пов’язаних з некоректними типами даних.

Стилізація інтерфейсу реалізована з використанням фреймворку TailwindCSS, який представляє утилітарний підхід до написання CSS. На відміну від традиційних CSS фреймворків, що надають готові компоненти, Tailwind надає низькорівневі утилітарні класи, які дозволяють створювати

унікальний дизайн безпосередньо у HTML розмітці. Такий підхід забезпечує високу гнучкість стилізації, уникнення конфліктів імен класів та значне зменшення розміру кінцевого CSS файлу завдяки видаленню невикористаних стилів. TailwindCSS надає вбудовану підтримку адаптивного дизайну через систему брейкпоінтів, що дозволяє застосовувати різні стилі залежно від ширини екрану. Префікси sm, md, lg, xl та 2xl відповідають стандартним розмірам екранів та дозволяють створювати інтерфейси, що коректно відображаються на настільних комп'ютерах. Система Flexbox доступна через відповідні утилітарні класи, що спрощує створення складних макетів.

Функціональність перетягування задач між колонками реалізована з використанням бібліотеки react-dnd, яка надає гнучкий та потужний API для імплементації drag-and-drop взаємодій у React застосунках. Бібліотека базується на концепції джерел перетягування та цілей скидання, що з'єднуються через єдиний менеджер контексту. React DnD абстрагує складність роботи з DOM подіями перетягування та надає декларативний спосіб визначення поведінки компонентів.

Для роботи у браузерному середовищі використовується HTML5 бекенд react-dnd-html5-backend, який використовує нативний HTML5 Drag and Drop API. Цей бекенд забезпечує підтримку перетягування у всіх сучасних браузерах та надає доступ до стандартних візуальних ефектів перетягування.

HTTP клієнт Axios використовується для виконання запитів до серверного API. Axios надає зручний інтерфейс для виконання асинхронних HTTP запитів, підтримує автоматичне перетворення JSON даних та надає можливість налаштування інтерцепторів для модифікації запитів та відповідей.

Управління глобальним станом застосунку реалізовано через React Context API, який надає механізм передачі даних через дерево компонентів без необхідності передавати пропси на кожному рівні. AuthContext зберігає інформацію про поточного аутентифікованого користувача та надає методи для входу, виходу та реєстрації. ToastContext забезпечує централізоване управління сповіщеннями користувача про результати операцій.

Кастомний хук useTasks інкапсулює логіку роботи з задачами, включаючи завантаження списку задач, створення, оновлення та видалення задач, а також оптимістичне оновлення інтерфейсу при перетягуванні. Оптимістичні оновлення забезпечують миттєвий візуальний відгук на дії користувача, тоді як у фоновому режимі виконується запит до сервера. У разі помилки сервера стан автоматично відкочується до попереднього значення.

Контейнеризація застосунку реалізована за допомогою Docker, який забезпечує ізольоване та відтворюване середовище виконання для кожного компонента системи. Docker Compose використовується для оркестрації багатоконтейнерного застосунку, що включає контейнери для бази даних PostgreSQL, серверної частини та клієнтської частини. Файл docker-compose.yml визначає конфігурацію сервісів, мережеві з’єднання між ними та залежності запуску.

Dockerfile для серверної частини використовує багатоетапну збірку, де перший етап виконує встановлення залежностей та компіляцію TypeScript коду, а другий етап створює мінімальний образ для продакшену. Такий підхід зменшує розмір кінцевого образу та підвищує безпеку за рахунок виключення інструментів розробки.

Конфігурація застосунку здійснюється через змінні середовища, що зберігаються у файлах .env для локальної розробки та передаються як змінні середовища у контейнерах для продакшену. Такий підхід забезпечує розділення конфігурації від коду та дозволяє легко змінювати налаштування для різних середовищ виконання.

Статичний аналіз коду виконується за допомогою ESLint, який перевіряє код на відповідність визначеним правилам стилю та виявляє потенційні помилки. Використання лінтера є обов’язковим етапом життєвого циклу розробки програмного забезпечення для забезпечення високої якості та підтримки кодової бази. Конфігурація ESLint налаштована для роботи з TypeScript та React, включаючи правила для перевірки хуків React та забезпечення консистентності коду. Інтеграція ESLint з редактором коду

забезпечує виявлення проблем у реальному часі під час написання коду. Це дозволяє оперативно вносити корективи та запобігати появі дефектів на ранніх стадіях розробки.

Обраний технологічний стек забезпечує створення сучасного, масштабованого та підтримуваного вебсервісу для управління завданнями. Використання єдиної мови програмування для клієнтської та серверної частин спрощує розробку та підтримку системи, тоді як контейнеризація забезпечує однаковість середовища виконання на різних платформах. Вибір зрілих та широко використовуваних технологій гарантує наявність документації, підтримки спільноти та довгострокову життєздатність проєкту.

2.3 Проєктування бази даних

Проєктування бази даних є одним із ключових етапів створення вебсервісу, оскільки саме структура збереження даних визначає цілісність інформації, ефективність доступу до неї та можливості подальшого масштабування системи. На цьому етапі формується логічна модель даних, визначаються сутності предметної області, їх атрибути та взаємозв’язки. Коректно спроєктована база даних забезпечує надійне зберігання користувацьких даних, підтримує виконання основних бізнес-операцій і мінімізує ризик виникнення аномалій під час вставки, оновлення або видалення інформації.

У межах вебсервісу управління завданнями база даних повинна зберігати інформацію про користувачів системи, їхні завдання, стан виконання, часові характеристики та допоміжні метадані. Модель даних має підтримувати сценарії багатокористувацької роботи, при цьому забезпечуючи чітке розмежування доступу до даних різних користувачів. Це досягається шляхом встановлення відповідних зв’язків між сутностями та використання механізмів ідентифікації власника кожного запису.

Структура бази даних є фундаментальною частиною будь-якої інформаційної системи, що забезпечує зберігання, керування та доступ до даних. У цьому проєкті використовується реляційна база даних, спроєктована за допомогою ER-діаграми (діаграми «сутність-зв’язок»), що ілюструє логічні зв’язки між різними компонентами системи. У контексті реляційних баз даних зв’язок між двома таблицями має дві цікаві потужності: потужність – кількість елементів – пов’язаних з першою таблицею та потужність – як ви здогадалися, кількість елементів – пов’язаних з другою таблицею [8].

Найбільш поширеним типом відношення для реалізації логіки «користувач-завдання» є зв’язок «один-до-багатьох», який дозволяє одному акаунту володіти необмеженою кількістю створених карток.

Для технічної реалізації такої структури використовуються зовнішні ключі, що гарантують посилальну цілісність і унеможливлюють існування завдань без прив’язки до конкретного власника.

Важливим аспектом проєктування також є нормалізація таблиць, яка допомагає уникнути дублювання даних та зменшує ймовірність логічних помилок при їх модифікації.

Окрім структуризації даних, схема бази передбачає створення індексів для ключових полів, що критично необхідно для швидкого пошуку та сортування інформації у великих масивах.

На рисунку 2.4 представлено ER-діаграму бази даних системи для управління завданнями, яка демонструє взаємозв’язки між сутностями та їх атрибутами.

На основі ER-діаграми сформовано структуру таблиць реляційної бази даних. Кожна таблиця відповідає окремій сутності та містить первинний ключ для унікальної ідентифікації записів. Для встановлення зв’язку між користувачами та завданнями використовується зовнішній ключ.

У таблиці 2.1 наведено узагальнений опис основних таблиць бази даних та їхніх ключових атрибутів.

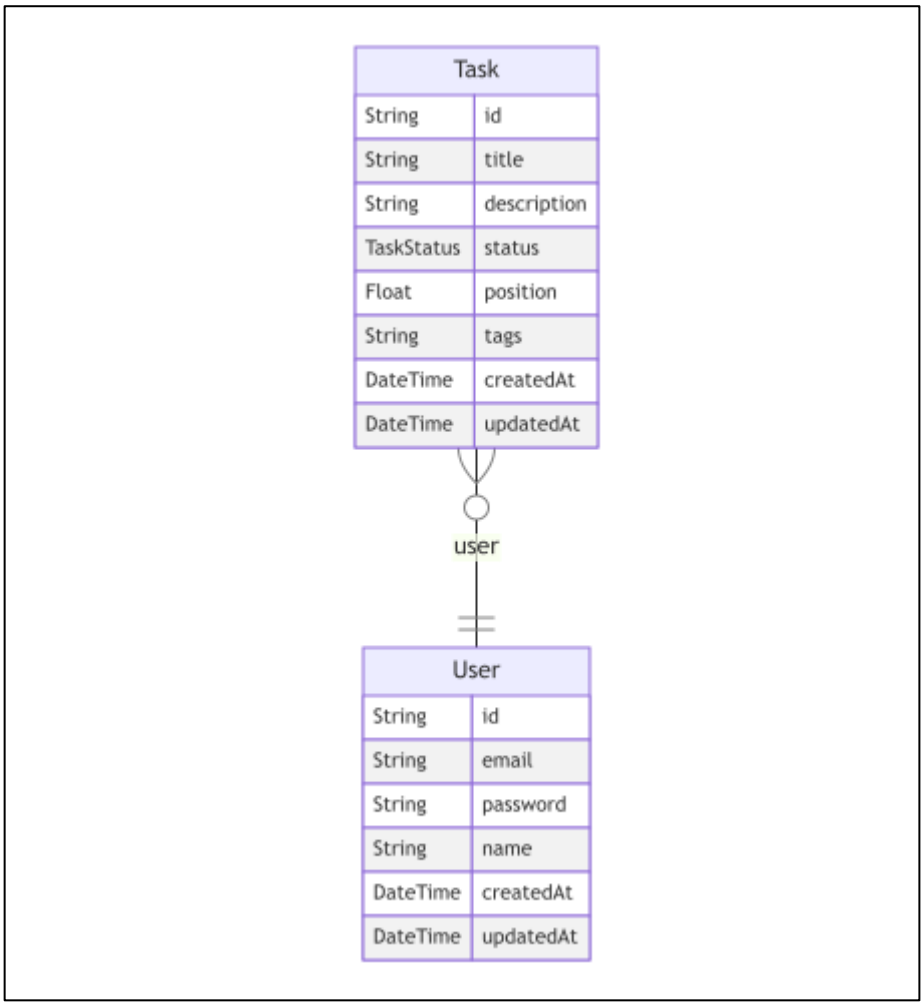


Рисунок 2.4 – ER-діаграма БД

Таблиця 2.1 – Основні таблиці

Таблиця	**Основні поля**	**Призначення**
User	id, email, password, name, createdAt, updatedAt.	Зберігання облікових даних користувачів та інформації для автентифікації.
Task	id, title, description, status, position, tags, createdAt, updatedAt, userId.	Зберігання інформації про завдання користувачів, їхній стан та порядок відображення.

Поле id у кожній таблиці виконує роль первинного ключа та має тип рядка, що дозволяє використовувати UUID або аналогічні механізми генерації унікальних ідентифікаторів. Атрибут userId у таблиці Task є зовнішнім

ключем, який забезпечує зв’язок із таблицею User та дозволяє визначити власника завдання.

Логічна модель бази даних визначає структуру сутностей, типи атрибутів та характер взаємозв’язків без прив’язки до конкретної системи управління базами даних. У межах логічної моделі вебсервісу управління завданнями враховано вимоги до підтримки канбан-подібного робочого процесу, що реалізується за допомогою поля status у сутності Task. Значення цього поля визначає, у якій колонці відображається завдання (наприклад, заплановане, в процесі виконання або завершене).

Поле position використовується для збереження порядку відображення завдань усередині однієї колонки. Це дозволяє підтримувати операції перетягування завдань і коректно відновлювати їх порядок після оновлення сторінки або повторного завантаження даних з сервера. Поле tags забезпечує можливість логічної класифікації завдань і подальшої реалізації фільтрації або пошуку.

До ключових характеристик логічної моделі бази даних належать:

– чітке розмежування даних користувачів;
– відсутність надлишкового дублювання інформації;
– підтримка масштабування за рахунок розширення кількості завдань;
– можливість подальшого розширення моделі (додавання коментарів, історії змін, ролей користувачів).

Фізична модель бази даних описує спосіб реалізації логічної структури у конкретній СУБД. У контексті вебсервісу управління завданнями фізична модель орієнтована на використання сучасних реляційних або документно-орієнтованих систем з підтримкою транзакцій та індексації. Атрибути часових міток createdAt та updatedAt використовуються для відстеження змін даних і можуть бути автоматично оновлюваними на рівні прикладної логіки або СУБД.

Для підвищення продуктивності доступу до даних доцільним є створення індексів для полів, що часто використовуються у фільтрації або сортуванні, зокрема userId, status та position. Це дозволяє оптимізувати

виконання запитів під час завантаження списків завдань та зменшити час відповіді серверної частини.

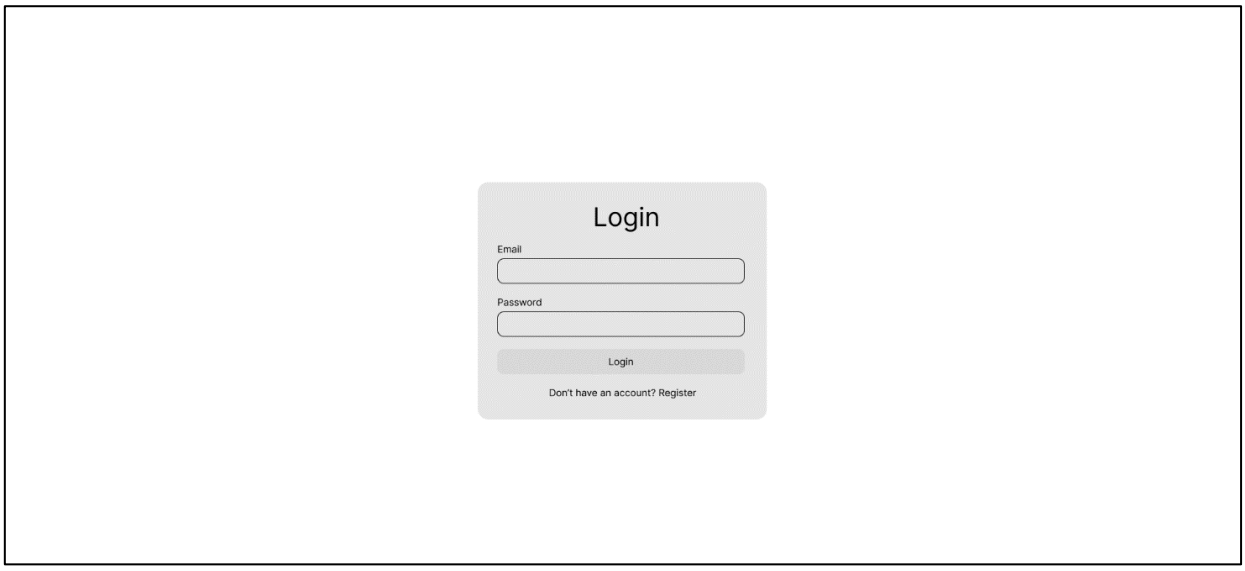
Спроєктована база даних відповідає вимогам вебсервісу для управління завданнями, забезпечує цілісність і узгодженість даних, а також створює надійну основу для реалізації бізнес-логіки системи. Обрана структура є достатньо гнучкою для подальшого розширення функціональності та інтеграції додаткових можливостей без суттєвих змін існуючої моделі.

2.4 Проєктування фронтенд-інтерфейсу

Розробка клієнтської частини вебсервісу для управління завданнями базується на принципах побудови Single Page Application, SPA, що забезпечує високу швидкість реакції інтерфейсу та плавний досвід взаємодії без необхідності повного перезавантаження сторінки. При проєктуванні візуальної складової системи ключову роль відіграло використання бібліотеки React.js, яка дозволяє реалізувати компонентний підхід до побудови інтерфейсу. Це означає, що кожен елемент сторінки – від кнопки до складної форми введення даних – є ізольованим програмним модулем, що має власний стан та логіку відображення. Такий підхід не лише спрощує процес розробки та тестування, але й забезпечує уніфікацію стилю у всьому додатку.

Є три найбільш поширені методи, які матеріалізуються в таких результатах, як каркаси, карти сайту та персони. Знання цих методів та правильний час для використання кожного з них – одна з найбільш затребуваних навичок серед UX-дизайнерів [9].

На рисунку 2.5 представлено макет сторінки форми входу та реєстрації. Взаємодія користувача з системою розпочинається з процедури автентифікації, для якої спроєктовано окремі екранні форми входу та реєстрації. Інтерфейс цих сторінок реалізовано за принципом максимальної концентрації уваги. Форма реєстрації містить поля для введення імені користувача, електронної пошти та пароля.



а) Login page



б) Resigter page

Рисунок 2.5 – Макет сторінок форми входу та реєстрації

Варто зазначити, що поле імені позначено як опціональне, що знижує поріг входу для нових користувачів, дозволяючи їм швидше отримати доступ до функціонала. При цьому для забезпечення безпеки та цілісності даних передбачається валідація введеної інформації на стороні клієнта і сервера. Форма входу є ще більш лаконічною та вимагає лише ідентифікатор email та пароль. Обидві форми містять чітко виділені кнопки дії Register та Login, та посилання для перемикання між режимами, що забезпечує логічну навігацію.

Після успішної авторизації користувач потрапляє на головний екран системи – Task Board, яка є центральним елементом управління робочим процесом. На рисунку 2.6 представлено макет сторінки дошки завдань.

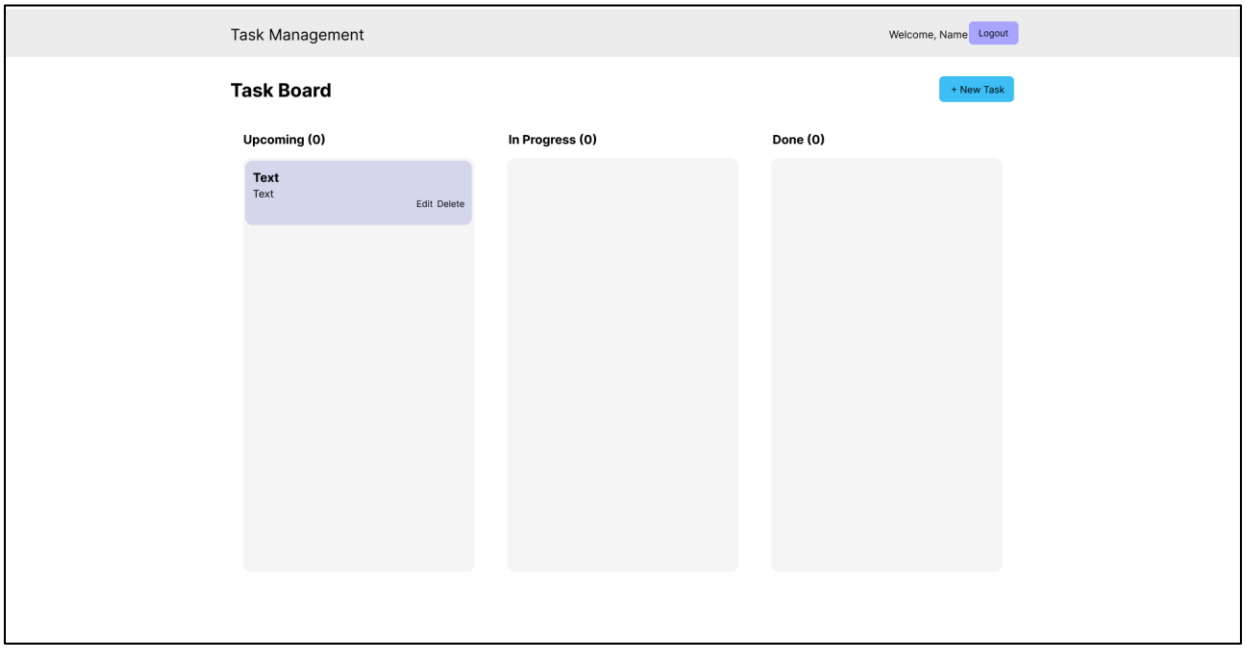


Рисунок 2.6 – Task Board

Структура цієї сторінки розроблена з урахуванням ієрархії інформації. Верхня частина вікна зайнята навігаційною панеллю, яка виконує інформаційну та керуючу функції. У лівій частині хедера розміщено назву системи “Task Management”, що слугує якірним елементом брендингу. У правій частині реалізовано блок персоналізації, де відображається вітання з ім’ям поточного користувача та кнопка “Logout” для завершення сеансу. Виділення кнопки виходу дозволяє чітко розмежувати зони управління контентом та управління сесією.

Основний робочий простір сторінки організовано за методологією Kanban, що передбачає візуалізацію етапів виконання завдань у вигляді вертикальних колонок. Інтерфейс поділено на три основні секції: Upcoming, In Progress та Done. Кожна колонка має заголовок, що чітко ідентифікує статус завдань, розміщених у ній, а також лічильник, який у дужках відображає поточну кількість карток у відповідному стовпці. Таке рішення дозволяє

користувачеві миттєво оцінити загальний обсяг роботи та її розподіл по стадіях, не вдаючись до детального перегляду кожної картки.

Візуальне представлення окремого завдання реалізовано через компонент картки завдання. Картка натякаючи на можливість інтерактивної взаємодії, перетягування. Внутрішня структура картки включає заголовок завдання, виділений жирним шрифтом, текстовий опис та блок керування з посиланнями “Edit” та “Delete”.

Важливим аспектом проєктування інтерфейсу є реалізація механізму створення нових завдань. Для цієї мети передбачено кнопку + New Task, розташовану у верхній правій частині робочої області, безпосередньо над колонками. Розміщення цієї кнопки відповідає патерну F-подібного сканування сторінки, згідно з яким погляд користувача часто фіксується у правому верхньому куті при пошуку ключових дій. Натискання на цю кнопку ініціює відкриття модального вікна Create New Task. Використання модального вікна замість переходу на нову сторінку є обґрунтованим рішенням з точки зору User Experience, оскільки це дозволяє зберегти контекст роботи: користувач бачить дошку на задньому плані й не втрачає орієнтації у просторі додатку. На рисунку 2.7 представлено макет модального вікна Create New Task.

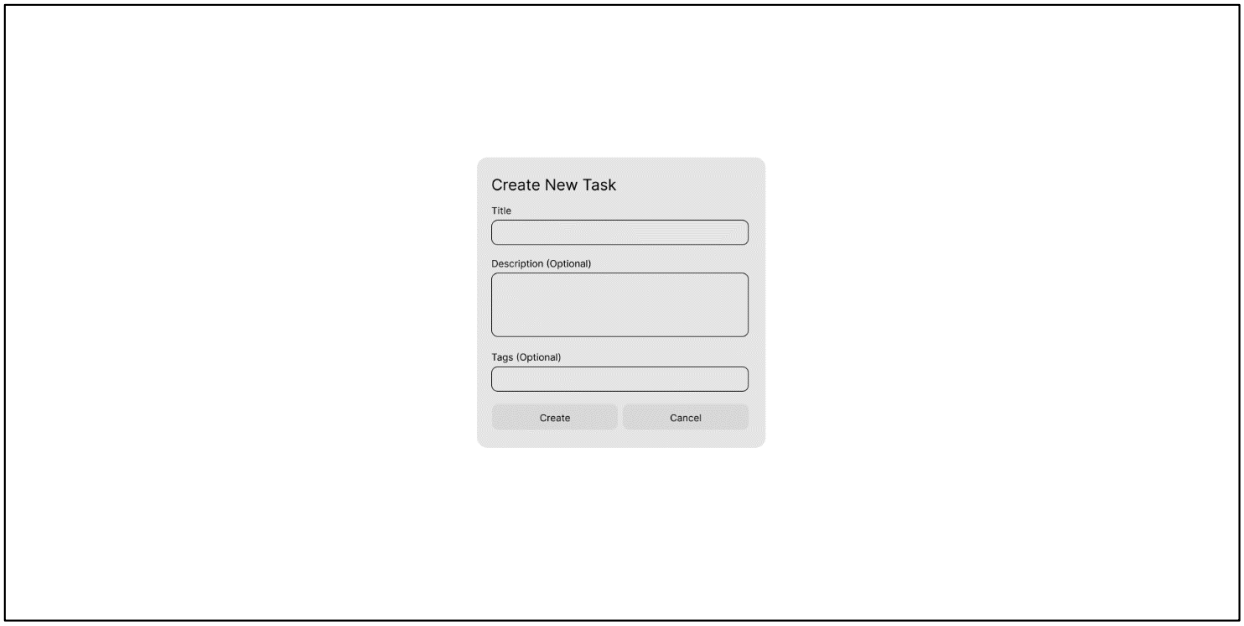


Рисунок 2.7 – Модальне вікно Create New Task

Форма створення завдання у модальному вікні спроєктована з урахуванням балансу між повнотою даних та швидкістю заповнення. Вона містить поле для заголовка Title, яке є обов’язковим, а також опціональні поля для опису Description та тегів. Поля введення мають чіткі межі та достатній внутрішній відступ для комфортного сприйняття тексту. Особливу увагу приділено полю опису, яке виконано у вигляді багаторядкового текстового блоку, що дозволяє вводити деталізовану інформацію про завдання. У нижній частині модального вікна розташовано кнопки управління процесом: Create для підтвердження створення та Cancel для скасування дії. Кнопки візуально розрізняються. Затемнення фону (backdrop) при відкритті модального вікна допомагає сфокусувати увагу користувача виключно на процесі введення даних, блокуючи взаємодію з іншими елементами інтерфейсу до моменту завершення операції.

Адаптивність розробленого інтерфейсу забезпечується використанням гнучкої сітки Flexbox, реалізованої за допомогою допоміжних класів CSS Tailwind, яка дозволяє автоматично змінювати розташування колонок та розміри карток залежно від ширини екрана пристрою. На великих екранах колонки розташовуються горизонтально в ряд, повністю використовуючи доступну ширину. Шрифтове оформлення інтерфейсу базується на використанні гротескних шрифтів без засічок, що забезпечує високу читабельність тексту різного розміру та є стандартом для сучасних вебдодатків.

Підсумовуючи, можна стверджувати, що спроєктований фронтенд-інтерфейс вебсервісу для управління завданнями відповідає сучасним вимогам до ергономіки та юзабіліті. Використання компонентної моделі React дозволило створити модульну структуру, легку для підтримки та масштабування. Візуальні рішення, такі як Kanban-дошка, модальні вікна для введення даних та чітка ієрархія навігаційних елементів, сприяють ефективній роботі користувача, мінімізуючи час на навчання та виконання рутинних операцій. Лаконічний дизайн та продумана логіка взаємодії формують

цілісний та завершений продукт, готовий до використання у реальних сценаріях управління проєктами.

2.5 Моделі бізнес-процесів

Розробка сучасного вебсервісу для управління завданнями вимагає ретельного підходу до проєктування архітектури взаємодії між клієнтською та серверною частинами. Основним завданням цього етапу є визначення чітких протоколів обміну даними, розмежування зон відповідальності компонентів системи та забезпечення цілісності даних при виконанні користувацьких сценаріїв. Для реалізації поставлених завдань було обрано архітектурний патерн клієнт-сервер. При цьому клієнтська частина відповідає за презентацію даних та обробку користувацького інтерфейсу, тоді як серверна частина забезпечує бізнес-логіку та доступ до сховища даних. Вибір такого патерну мінімізує залежність між компонентами та сприяє масштабованості системи. Для формалізації вимог та візуалізації логічних процесів було розроблено діаграму прецедентів Use Case Diagram та набір діаграм послідовності Sequence Diagrams, які детально описують динаміку системи. На підставі цих діаграм здійснюється верифікація відповідності системи початковим функціональним вимогам. Вважається, що такий підхід забезпечує структурну прозорість та спрощує подальше впровадження змін.

Уніфікована мова моделювання UML дозволяє описувати системи словами та зображеннями. Примітними є різні графічні діаграми, особливо така як діаграма варіантів використання [10].

Проєктування системи є ключовим етапом розробки, на якому визначаються функціональні можливості системи, її архітектура та основні компоненти. Така візуалізація дозволяє чітко розмежувати зони відповідальності між акторами та визначити повний набір системних функцій. На рисунку 2.8 зображено діаграма прецедентів, яка ілюструє основні дії, які виконуються різними типами користувачів у системі.

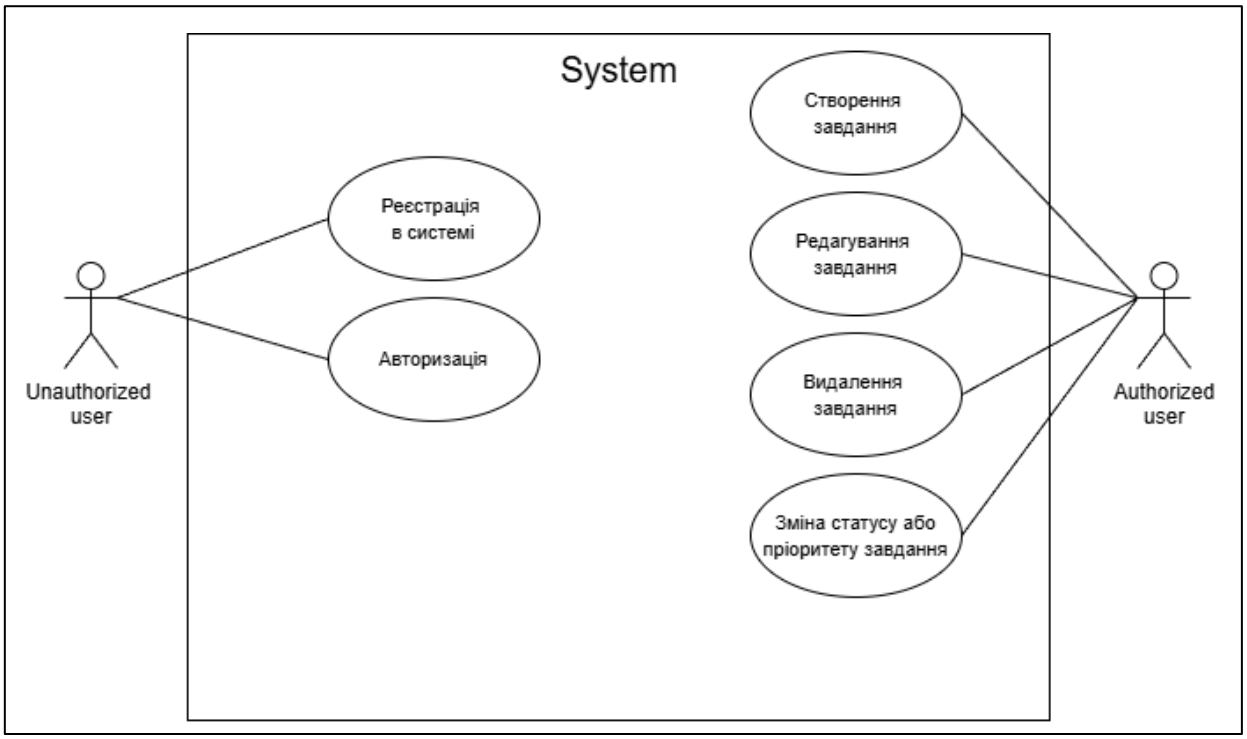


Рисунок 2.8 – Діаграма прецедентів

В межах спроєктованої системи виділено дві основні категорії користувачів: «Неавторизований користувач» Unauthorized user та «Авторизований користувач» Authorized user. Даний поділ є фундаментальним для забезпечення безпеки інформаційної системи, оскільки доступ до бізнес-логіки управління завданнями повинен надаватися виключно ідентифікованим суб’єктам.

Для неавторизованого користувача система надає обмежений набір функціональних можливостей, спрямованих на отримання доступу до основного функціонала. До таких прецедентів належать «Реєстрація в системі» та «Авторизація». Процес реєстрації передбачає створення нового облікового запису користувача зі збереженням облікових даних у базі даних, тоді як процес авторизації забезпечує перевірку цих даних та видачу токена доступу JWT, що переводить користувача у статус авторизованого.

Після успішного проходження процедури автентифікації актор отримує статус «Авторизований користувач» і доступ до повного спектра функцій управління завданнями. Як відображено на діаграмі прецедентів, до основних сценаріїв використання належать: «Створення завдання», «Редагування

завдання», «Видалення завдання» та «Зміна статусу або пріоритету завдання». Всі ці дії відбуваються в контексті захищеної сесії, де кожен запит до сервера супроводжується перевіркою прав доступу. Система проєктується таким чином, щоб забезпечити інтуїтивно зрозумілий інтерфейс для виконання цих дій, приховуючи складність внутрішніх процесів обробки даних.

Детальний розгляд динаміки системи доцільно розпочати з процесу створення нового завдання, який є точкою входу для наповнення системи даними.

Діаграми послідовностей UML – це діаграми взаємодії, які детально описують, як виконуються операції. Вони фіксують взаємодію між об’єктами в контексті співпраці. Діаграми послідовностей орієнтовані на час і візуально показують порядок взаємодії за допомогою вертикальної осі діаграми для представлення часу, часу надсилання повідомлень і часу їх надсилання [11].

Процес ініціюється на боці клієнта в компоненті BoardPage, який відповідає за відображення дошки завдань. Користувач взаємодіє з елементом інтерфейсу, ініціюючи подію створення нового завдання. У відповідь система змінює внутрішній стан інтерфейсу, встановлюючи прапорець видимості форми setShowForm(true), що призводить до рендерингу модального вікна TaskForm. Передбачається, що всі введені дані проходять первинну валідацію на стороні клієнта для забезпечення коректності формату. Цей компонент працює в режимі створення mode: create, надаючи поля для введення заголовка, опису та тегів завдання.

Після заповнення форми та підтвердження дії користувачем, керування передається спеціалізованому хуку useTasks, який інкапсулює логіку взаємодії з API. На рисунку 2.9 зображена діаграма послідовності, яка ілюструє повний цикл обробки запиту: від взаємодії користувача з графічним інтерфейсом до фіксації даних у персистентному сховищі.

На цьому етапі вступає в дію шар комунікації з сервером, реалізований через API Client – екземпляр бібліотеки Axios. Критично важливим аспектом безпеки є використання механізму перехоплювачів запитів interceptors. Перед

відправленням HTTP-запиту POST на ендпоінт /api/tasks, перехоплювач автоматично додає заголовок авторизації Authorization: Bearer token. Це забезпечує передачу ідентифікатора сесії без необхідності явного дублювання коду в кожному виклику API.

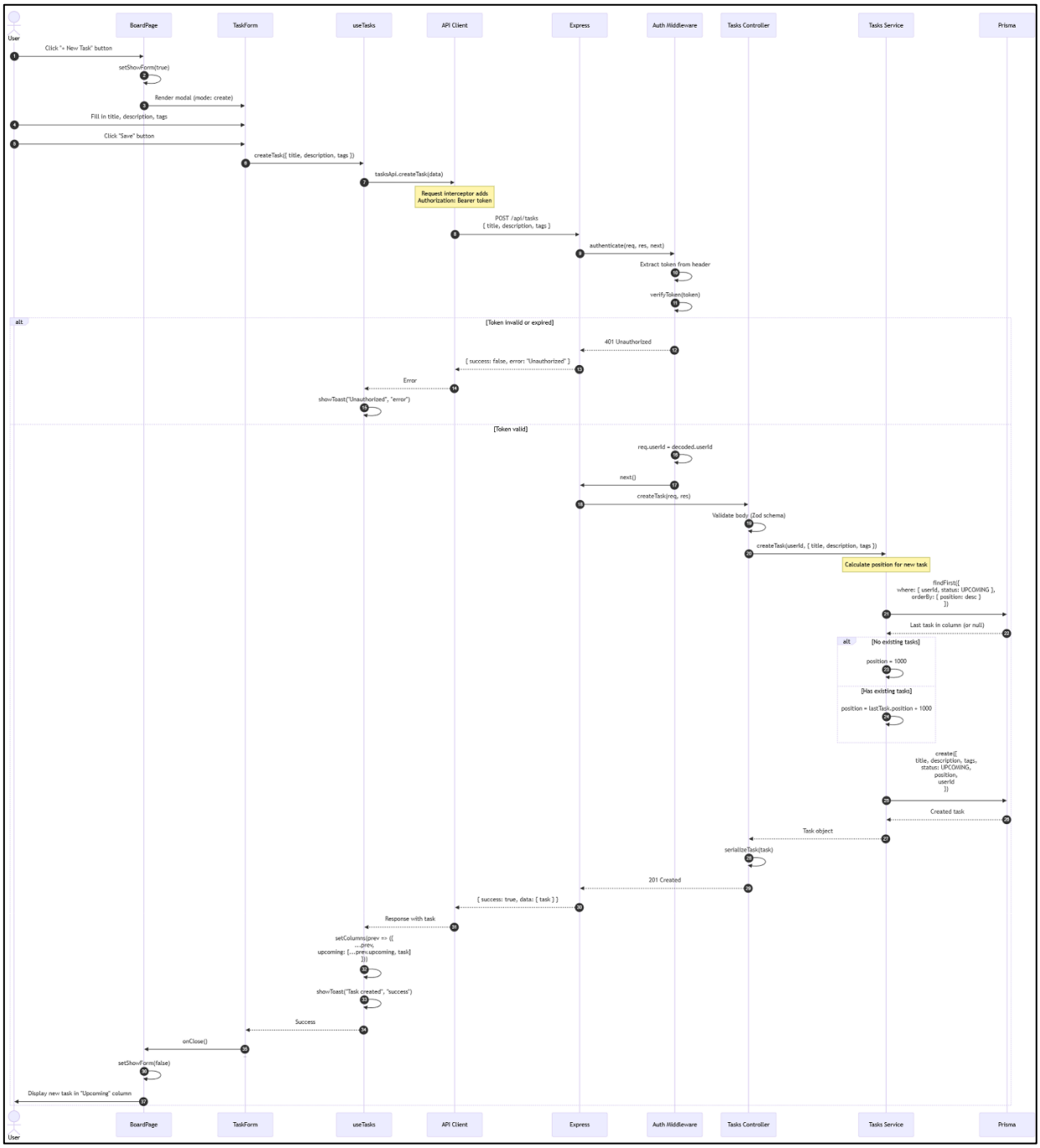


Рисунок 2.9 – Creation Sequence Diagram

Серверна частина, побудована на базі Express, приймає запит і пропускає його через ланцюжок проміжного програмного забезпечення

middleware. Першим бар'єром виступає AuthMiddleware. Його завданням є екстракція токена із заголовка запиту та його верифікація. У випадку, якщо токен відсутній, недійсний або його термін дії закінчився, middleware перериває обробку запиту, повертаючи статус помилки 401 Unauthorized. Клієнтська частина, отримавши таку відповідь, обробляє її, відображаючи відповідне повідомлення користувачеві toast message. Якщо ж токен валідний, middleware розшифровує його decode, витягує ідентифікатор користувача userId та додає його до об'єкта запиту, передаючи керування наступному обробнику – контролеру.

Контролер TasksController виконує роль оркестратора. Він отримує тіло запиту та проводить його первинну валідацію на відповідність схемі даних (наприклад, перевіряє наявність обов'язкового заголовка). Після успішної валідації контролер викликає метод сервісного шару TasksService. Саме тут зосереджена основна бізнес-логіка створення завдання, зокрема алгоритм визначення позиції нового елемента.

Оскільки завдання на дошці повинні мати певний порядок сортування, система повинна автоматично обчислити позицію для нового запису. Сервіс звертається до бази даних через ORM Prisma, виконуючи запит findFirst для пошуку останнього завдання в конкретній колонці Upcoming. Якщо в колонці вже існують завдання, нове завдання отримує значення позиції, збільшене на фіксований крок lastTask.position + 1000. Такий підхід із використанням великого кроку дозволяє в майбутньому вставляти завдання між існуючими без необхідності перерахунку позицій усіх елементів. Якщо колонка порожня, завданню присвоюється початкове значення – 1000.

Сформований об'єкт завдання, що включає згенерований ID, розраховану позицію, прив'язку до користувача та введені дані, зберігається в базі даних методом create. Успішно створений запис повертається через весь ланцюжок викликів назад до контролера, який серіалізує дані та відправляє HTTP-відповідь зі статусом 201 Created. На клієнтській стороні хук useTasks отримує відповідь, оновлює локальний стан колонок, додаючи нове завдання

до списку, закриває модальне вікно та сповіщає користувача про успіх операції.

Наступним критично важливим сценарієм є редагування. Цей процес має схожу структуру з процесом створення, але включає додаткові кроки перевірки прав доступу та цілісності даних. Користувач ініціює редагування, натискаючи кнопку на картці завдання TaskCard. Це викликає відкриття модального вікна TaskForm, яке цього разу ініціалізується даними обраного завдання. Важливою особливістю є те, що система повинна відстежувати зміни: при натисканні кнопки збереження відбувається збір змінених полів: title, description, tags, status. Це дозволяє оптимізувати мережевий трафік, відправляючи на сервер лише ті дані, що дійсно зазнали змін, використовуючи семантику часткового оновлення.

Запит на оновлення PATCH або PUT так само проходить через AuthMiddleware для підтвердження особи користувача. Однак, на рівні TasksService додається додаткова логіка авторизації. Перед внесенням змін у базу даних система повинна переконатися, що завдання з вказаним ID дійсно існує і, що найважливіше, належить користувачеві, який ініціював запит. Для цього виконується пошук завдання за ID. Якщо завдання не знайдено, сервіс генерує виняток NotFoundError – повертається статус 404. Якщо завдання існує, але ID власника не збігається з ID користувача з токена, генерується виняток ForbiddenError – статус 403. Така багаторівнева перевірка гарантує ізоляцію даних користувачів.

При успішному проходженні перевірок та валідації вхідних даних, сервіс виконує оновлення запису в базі даних. Особливої уваги потребує ситуація зміни статусу завдання. Якщо користувач змінив колонку завдання через форму редагування. На рисунку 2.10 зображена діаграма послідовності, яка ілюструє логіку редагування існуючого завдання.

Алгоритм розрахунку позиції аналогічний до процесу створення: завдання додається в кінець нової колонки. Після оновлення в базі даних сервер повертає оновлений об’єкт.

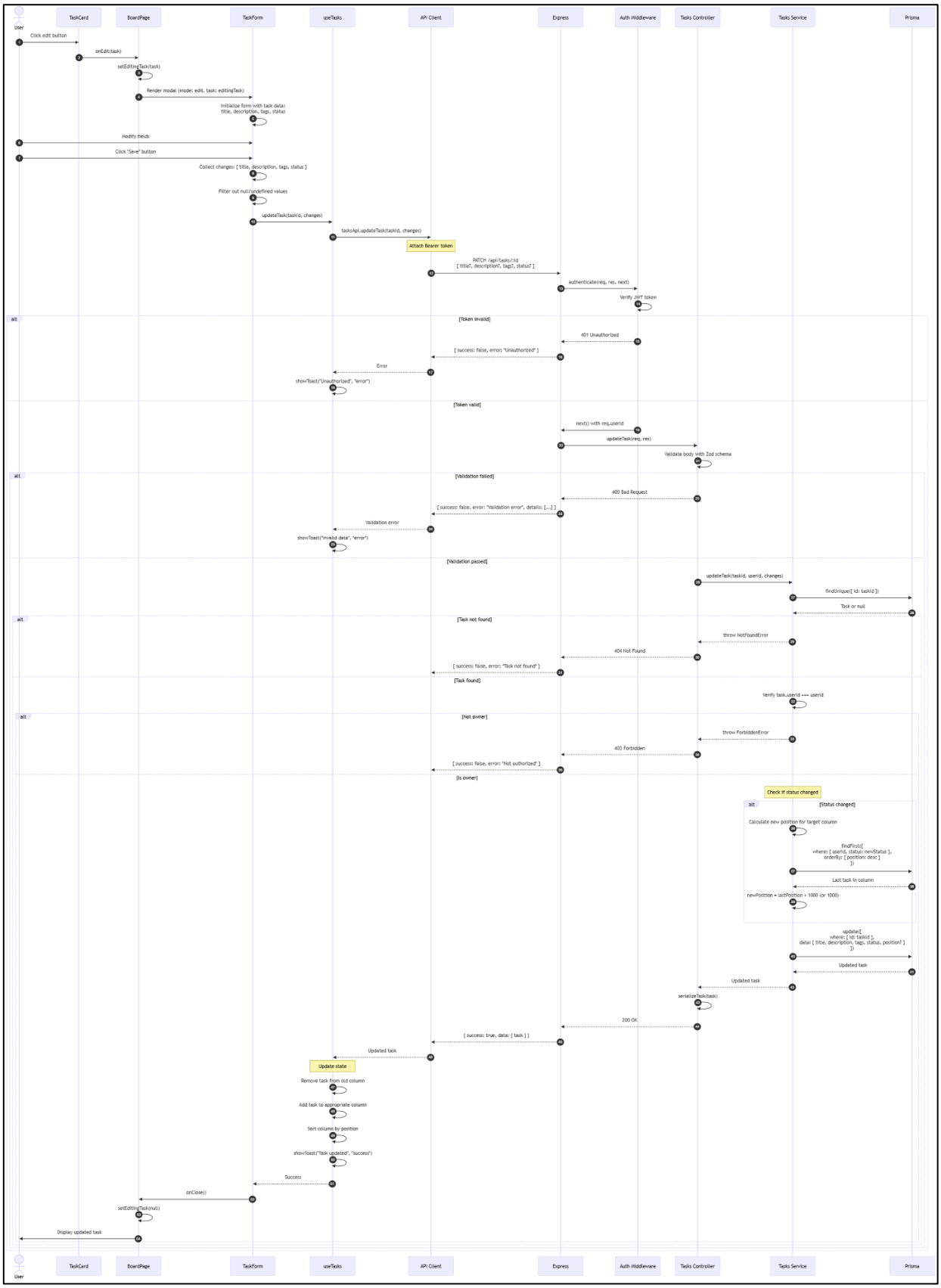


Рисунок 2.10 – Update Sequence Diagram

Клієнтська програма обробляє відповідь, оновлюючи локальний стан. Це передбачає видалення завдання зі старої колонки та додавання його до

нової, або ж просте оновлення полів, якщо статус не змінювався. У разі виникнення помилок користувачеві відображається відповідне повідомлення, а модальне вікно залишається відкритим, надаючи можливість виправити дані.

Якщо ж операція проходить успішно, модальне вікно автоматично закривається, повертаючи користувача до робочого простору. Одночасно з цим система генерує коротке спливаюче сповіщення, яке підтверджує збереження внесених змін.

Для забезпечення миттєвого відгуку інтерфейсу використовуються оптимізовані алгоритми рендерингу, що перемальовують лише змінені елементи списку. Це дозволяє уникнути зайвого навантаження на браузер клієнта навіть при роботі з великою кількістю активних завдань.

Важливим аспектом є також збереження цілісності даних при одночасному редагуванні різними учасниками команди. Тому після локального оновлення клієнт може виконувати фонову перевірку актуальності версії завдання.

Усі зміни статусу або змісту картки логуються на сервері для формування історії активності проєкту. Такий комплексний підхід до обробки редагування гарантує надійність системи та зручність її використання в реальних умовах. Завдяки цьому користувачі отримують безшовний досвід взаємодії, де складні технічні процеси залишаються непомітними для кінцевого споживача.

На рисунку 2.11 зображена діаграма послідовності, яка ілюструє сценарій переміщення завдань методом Drag & Drop, найбільш складним із технічної точки зору. Цей процес вимагає високої реактивності інтерфейсу та складної логіки синхронізації стану.

Сценарій починається, коли користувач захоплює картку завдання і починає перетягування.

Бібліотека, що відповідає за Drag & Drop – react-dnd, відстежує координати та події миші. Коли користувач «кидає» завдання в цільову зону – TaskColumn, спрацьовує подія onDrop.

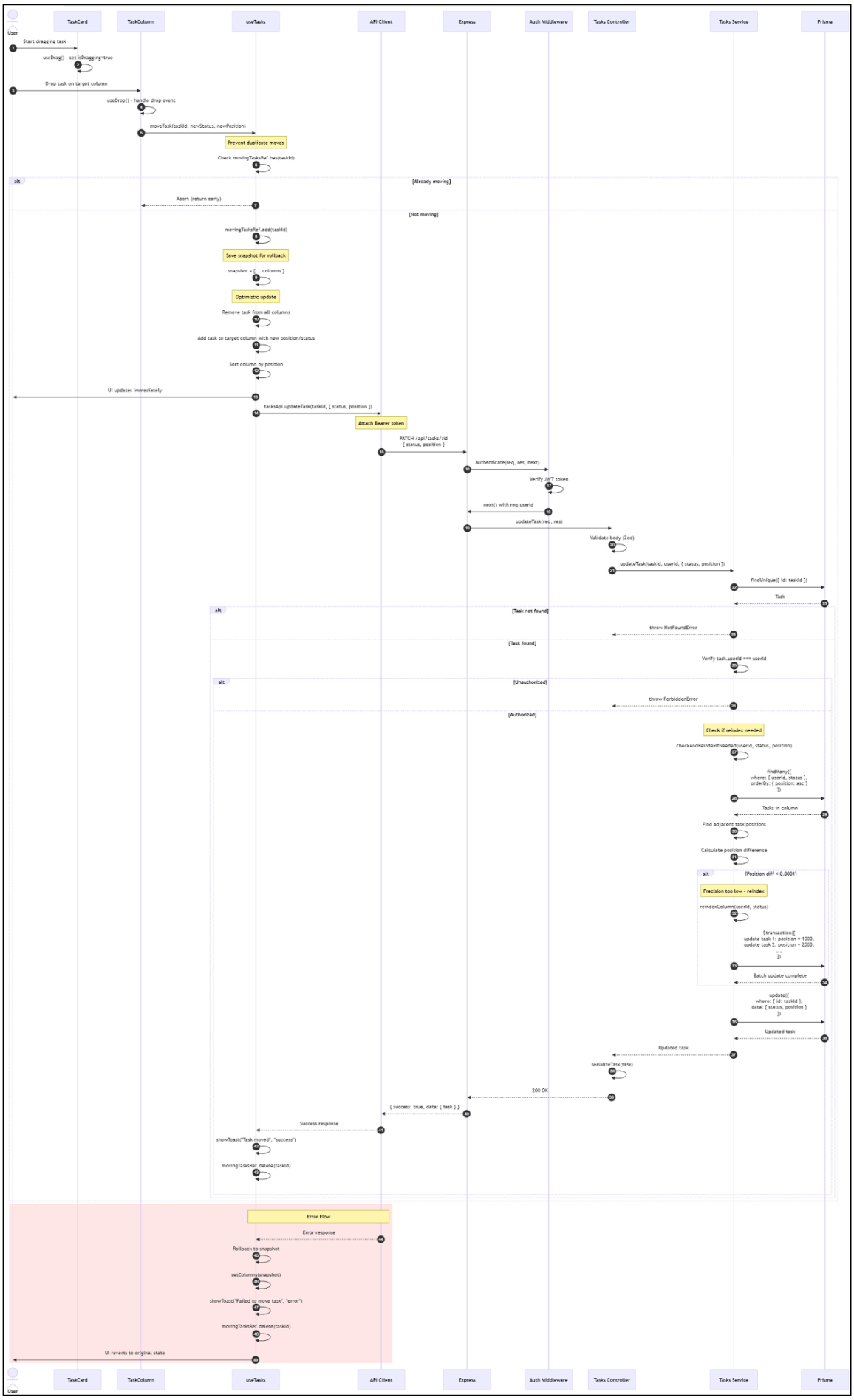


Рисунок 2.11 – Drag & Drop Sequence Diagram

Для забезпечення миттєвого візуального відгуку застосовується стратегія Optimistic UI Update. Клієнтська програма не чекає відповіді від сервера, щоб відобразити зміни. Хук useTasks негайно видаляє завдання зі старої колонки та вставляє його в нову позицію в цільовій колонці. Це створює відчуття швидкодії та плавності роботи додатку. Паралельно з цим, у фоновому режимі формується та відправляється запит до API з інформацією про новий статус та індекс завдання.

На серверній стороні обробка такого запиту вимагає специфічної логіки в TasksService. Сервер отримує ID завдання та його цільові параметри. Ключовим моментом є обчислення точного числового значення позиції. Оскільки завдання може бути вставлене між двома існуючими завданнями, нова позиція розраховується як середнє арифметичне позицій сусідніх елементів. Наприклад, якщо завдання вставляється між елементами з позиціями 1000 і 2000, нове значення буде 1500. Це дозволяє змінювати порядок без необхідності перезаписувати позиції всіх інших завдань у колонці. Система знаходить «сусідів» у базі даних, виконує математичні операції та оновлює запис цільового завдання.

Важливим аспектом діаграми Drag & Drop є обробка помилок та механізм відкату. Оскільки візуальне оновлення відбулося до підтвердження сервером, існує ризик розсинхронізації даних у разі збою запиту (наприклад, втрата інтернет-з’єднання або помилка сервера). Діаграма ілюструє блок Error Flow. У випадку, якщо API-клієнт отримує помилку у відповідь на запит переміщення, хук useTasks перехоплює цю подію. Він ініціює процедуру відновлення стану: завдання примусово повертається у вихідну колонку на попередню позицію, а користувачеві відображається повідомлення про неможливість збереження змін. Це гарантує, що те, що користувач бачить на екрані, завжди зрештою відповідатиме реальному стану даних у системі.

Таким чином, розроблена архітектура взаємодії компонентів забезпечує надійне та безпечне функціонування вебсервісу. Використання діаграм послідовності дозволило деталізувати відповідальність кожного шару системи

– від презентаційного React-компоненту до рівня даних Prisma/PostgreSQL, а також чітко визначити протоколи обробки як успішних сценаріїв, так і виняткових ситуацій. Застосування сучасних підходів, таких як JWT-авторизація, оптимістичні оновлення інтерфейсу та багаторівнева валідація, створює підґрунтя для побудови масштабованої та зручної для користувача системи управління завданнями.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБСЕРВІСУ

3.1 Середовище розробки

Розробка вебсервісу для управління завданнями з адаптивним інтерфейсом на основі Node.js і React.js здійснювалася у середовищі, побудованому на сучасному стеку вебтехнологій із чітким розмежуванням ролей бекенд і фронтенд-підсистем, контейнеризацією сервісів та формалізованими контрактами між компонентами. Середовище розробки організовано на базі Visual Studio Code із застосуванням інтегрованого терміналу, системи керування залежностями npm, інструментів типобезпечності TypeScript, генератора клієнта БД Prisma та збірника Vite. Конфігурації середовища задаються через змінні оточення у файлах .env, який продемонстрован на рисунку 3.1.

```
# Database
DATABASE_URL="postgresql://postgres:password@localhost:5432/taskmanagement?schema=public"

# JWT
JWT_SECRET=
JWT_EXPIRES_IN="24h"

# Server
PORT=3000
NODE_ENV="development"

# CORS
CORS_ORIGIN="http://localhost:5173,http://localhost:3001"
```

Рисунок 3.1 – Файл .env

А також через docker-compose.yml з **Додаток А**, який забезпечує консистентні умови виконання та мережеву взаємодію між сервісами. Для забезпечення відтворюваності стану БД використано Prisma Migrate з **Додаток Б** із декларативною схемою даних у файлі schema.prisma, а мережеві контракти API визначено у коді маршрутизаторів Express та їх валідаційних специфікацій на основі Zod. Такий підхід дозволив формалізувати інтерфейси, типи та

поведінку системи, зменшити кількість помилок конфігурації та прискорити цикл розробки завдяки гарячому перезавантаженню і швидкій інкрементальній компіляції.

На рівні серверної частини середовище розробки визначається сукупністю конфігураційних артефактів та інструментів, що забезпечують життєвий цикл застосунку. У файлі package.json визначено ключові сценарії: для розробки з автоматичним перезапуском серверного процесу, для компіляції TypeScript у артефакти продакшн-збірки, для запуску зібраного серверу, а також та для управління схемою даних та генерації типобезпечного клієнта доступу до БД. Використання tsx дозволяє запускати TypeScript без окремої стадії трансляції у режимі розробки, що зменшує латентність змін і сприяє швидкій ітерації. Набір залежностей включає express як серверний фреймворк, cors для налаштування політик міждоменного доступу, dotenv для завантаження конфігурацій, bcryptjs і jsonwebtoken для реалізації автентифікації, а також zod для декларативної валідації даних запитів. Типи для середовища Node, Express, JWT та інших бібліотек підключені через devDependencies, що забезпечує коректність взаємодії модулів у компіляторі TypeScript та автодоповнення в IDE.

Ключові параметри середовища бекенду задаються у файлі .env. Файл .env встановлює фактичні значення, зокрема рядок підключення до PostgreSQLDATABASE_URL="postgresql://postgres:password@localhost:5432/taskmanagement?schema=public", секрет для підписання JWT та CORS-джерело для дев-середовища CORS_ORIGIN="http://localhost:5173". Ініціалізація цих параметрів відбувається у модулі src/config/env.ts, де зчитуються змінні оточення, виконуються перевірки наявності обов’язкових параметрів, а потім їх значення експортуються як централізований конфігураційний об’єкт. Це забезпечує детерміновану ініціалізацію налаштувань сервера і спрощує модульне тестування компонентів.

Джерелом даних виступає СУБД PostgreSQL, на яку орієнтована декларативна схема у файлі schema.prisma з **Додаток Б**. Запуск середовища у

контейнерах виконується за допомогою docker-compose.yml з **Додаток А**, де визначено три сервіси: postgres, backend і frontend. Сервіс postgres базується на образі postgres:16-alpine, налаштовує користувача, пароль, БД та том для персистентності, а також має healthcheck із командою pg_isready. Сервіс backend збирається з Dockerfile, отримує змінні оточення, прокидає порт 3000 і виконує команду sh -c "npx prisma migrate deploy && npm start", що гарантує застосування міграцій перед стартом програми. Сервіс frontend збирається на базі Dockerfile, публікує порт 3001:80 і залежить від готовності бекенду. Така оркестрація забезпечує відтворюваність середовища, ізоляцію залежностей та спрощену доставку на інші хости.

Загалом середовище розробки поєднує типобезпечність TypeScript і Prisma, контрактну валідацію Zod, стандартизацію помилок, керування станом інтерфейсу через React Context, адаптивну верстку Tailwind CSS, відтворюваність виконання через Docker Compose та узгодженість мережевих конфігурацій CORS. Ключові частини коду включають декларативну схему даних у schema.prisma з **Додаток А**, сценарії життєвого циклу у package.json, централізовану конфігурацію у src/config/env.ts, маршрути і валідації модулів auth та tasks, глобальні middleware, клієнтські модулі API, сторінки, хуки та утиліти позиціонування. Взаємодія цих компонентів забезпечує стабільний, масштабований процес розробки вебсервісу з чіткою трасованістю від конфігурації до поведінки застосунку.

3.2 Реалізація серверної частини

Сервер організовано модульно. Точка входу src/server.ts піднімає HTTP-сервер, імпортуючи застосунок з src/app.ts, де виконується ініціалізація middleware (CORS, JSON-парсер), підключення маршрутів та обробник помилок. Конфігураційні параметри завантажуються через dotenv у src/config/env.ts, а підключення до PostgreSQL керується Prisma-клієнтом у src/config/database.ts, рядок підключення задається в .env.

Маршрути згруповано у модулі auth та tasks. Файл src/modules/auth/auth.routes.ts продемонстровано на рисунку 3.2, який реєструє ендпоїнти /api/auth/register, /api/auth/login, /api/auth/me, делегуючи обробку до контролера auth.controller.ts.

```
1   import { Router } from 'express';
2   import { register, login, getCurrentUser } from './auth.controller';
3   import { validateRequest } from '../../middleware/validateRequest';
4   import { registerSchema, loginSchema } from './auth.validation';
5   import { authenticate } from './auth.middleware';
6
7   const router = Router();
8
9   router.post('/register', validateRequest(registerSchema), register);
10  router.post('/login', validateRequest(loginSchema), login);
11  router.get('/me', authenticate, getCurrentUser);
12
13  export default router;
```

Рисунок 3.2 – Файл auth.routes.ts

Контролер використовує сервіс auth.service.ts з **Додаток В**, що інкапсулює логіку створення користувача, хешування пароля utils/password.ts з bcrypt і випуску JWT utils/jwt.ts з jsonwebtoken. Валідація вхідних даних виконується через Zod-схеми у auth.validation.ts, а універсальний middleware middleware/validateRequest.ts застосовує ці схеми до тіла запиту. Перевірка автентифікації на захищених маршрутах реалізована в auth.middleware.ts: токен зчитується з заголовка Authorization, верифікується jwt.verify.

Модуль завдань експонує маршрути у tasks.routes.ts для CRUD-операцій:

– GET /api/tasks;
– POST /api/tasks;
– PATCH /api/tasks/:id;
– DELETE /api/tasks/:id.

Вони всі захищені middleware автентифікації. Контролер tasks.controller.ts викликає сервіс tasks.service.ts з **Додаток Г**, де сконцентрована бізнес-логіка: створення задач з початковою позицією, зміна

статусу/позиції, видалення, отримання списку задач користувача. Серіалізація відповіді для фронтенду винесена в формування DTO. Валідацію вхідних полів забезпечують схеми в tasks.validation.ts.

Обробка помилок централізована в middleware/errorHandler.ts: будь-які необроблені виключення логуються та повертається JSON із повідомленням і кодом 500. Для явних бізнес-помилок застосовується утиліта utils/ApiError.ts. Асинхронні контролери обгорнуті у utils/asyncHandler.ts продемонстровано на рисунку 3.3, що передає помилки в єдиний обробник.

```
1   import { Request, Response, NextFunction } from 'express';
2
3   type AsyncRequestHandler = (
4     req: Request,
5     res: Response,
6     next: NextFunction
7   ) => Promise<any>;
8
9   export const asyncHandler = (fn: AsyncRequestHandler) => {
10    return (req: Request, res: Response, next: NextFunction) => {
11      Promise.resolve(fn(req, res, next)).catch(next);
12    };
13  };
```

Рисунок 3.3 – Файл asyncHandler.ts

Схема БД визначена у prisma/schema.prisma з **Додаток Б**:

– моделі User та Task з полями status і position для drag-and-drop впорядкування;
– створені індекси за userId, status, position для швидкого вибіркового доступу;
– міграції й генерація клієнта Prisma виконуються командами npm run prisma:migrate та npm run prisma:generate;
– у продакшн-контейнері це робить команда npx prisma migrate deploy && npm start (див. docker-compose.yml, сервіс backend).

JWT-конфігурація задається змінними JWT_SECRET і JWT_EXPIRES_IN у .env. У сервісі автентифікації створення токена

відбувається через signJwt, а перевірка – у middleware через verifyJwt, що гарантує безпеку доступу до маршрутів завдань. CORS налаштовано через CORS_ORIGIN із .env, що дозволяє фронтенду на http://localhost:5173/3001 звертатися до API.

Таким чином, серверна частина реалізує чітке розділення відповідальностей: маршрути лише делегують, контролери координують, сервіси інкапсулюють бізнес-правила, middleware забезпечують крос-секційні вимоги: валідація, автентифікація, обробка помилок. Prisma надає типобезпечний доступ до PostgreSQL. Це створює надійний шар API для адаптивного інтерфейсу керування завданнями на React.js.

3.3 Реалізація клієнтської частини

Клієнтська частина вебсервісу розроблена з використанням сучасного фреймворку React у комбінації з інструментом збірки Vite, що забезпечує швидку розробку та оптимальну продуктивність. Архітектура клієнтської частини побудована на модульному принципі з чітким поділом відповідальності між компонентами, контекстом керування станом та API клієнтом, що дозволяє досягти високої масштабованості та зручності супроводження коду.

Архітектура клієнтської частини базується на трьох основних рівнях: шар компонентів користувацького інтерфейсу, шар керування станом та комунікаційний шар для взаємодії з серверним API. Структура папок відображає цей поділ з директоріями для компонентів, сторінок, контекстів, хуків, типів та утиліт. Основний вхідний файл App.tsx визначає маршрути застосунку з використанням React Router та встановлює глобальні провайдери для автентифікації та системи сповіщень. Додатково впроваджено механізм захищених маршрутів через компонент ProtectedRoute продемонстровано на рисунку 3.4, який перевіряє аутентифікацію користувача та перенаправляє

неавторизованих осіб на сторінку входу, забезпечуючи безпеку доступу до ресурсів.

```tsx
import React from 'react';
import { Navigate } from 'react-router-dom';
import { useAuth } from '../../context/AuthContext';

interface ProtectedRouteProps {
  children: React.ReactNode;
}

export const ProtectedRoute: React.FC<ProtectedRouteProps> = ({ children }) => {
  const { isAuthenticated } = useAuth();

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  return <>{children}</>;
};
```

Рисунок 3.4 – Компонент ProtectedRoute

Керування станом у застосунку реалізовано через контекст React, а саме AuthContext та ToastContext. AuthContext забезпечує централізоване управління даними користувача, токеном доступу та методами автентифікації. Цей контекст зберігає інформацію про поточного користувача, стан завантаження та методи входу, реєстрації та виходу з системи. Значний елемент реалізації полягає у збереженні даних автентифікації у локальному сховищі браузера через функції утиліти storage, що дозволяє зберігати сеанс користувача при перезавантаженні сторінки. Статус автентифікації перевіряється через властивість isAuthenticated, яка визначається наявністю токена. Компонент AuthProvider обгортає весь застосунок, забезпечуючи доступ до контексту через кастомний хук useAuth для всіх дочірніх компонентів. Контекст ToastContext управляє системою сповіщень користувачу, дозволяючи відображати повідомлення про успіх, помилки та інформаційні повідомлення у реальному часі.

Взаємодія з серверним API реалізована через конфігурований клієнт Axios, визначений у файлі client.ts, який продемонстровано на рисунку 3.5.

```
import axios from 'axios';

const API_BASE_URL = import.meta.env.VITE_API_URL || 'http://localhost:3000/api';

export const apiClient = axios.create({
  baseURL: API_BASE_URL,
  headers: {
    'Content-Type': 'application/json',
  },
});

// Request interceptor to add auth token
apiClient.interceptors.request.use(
  (config) => {
    const token = localStorage.getItem('token');
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

// Response interceptor for error handling
apiClient.interceptors.response.use(
  (response) => response,
  (error) => {
    if (error.response?.status === 401) {
      // Unauthorized - clear token and redirect to login
      localStorage.removeItem('token');
      localStorage.removeItem('user');
      window.location.href = '/login';
    }
    return Promise.reject(error);
  }
```

Рисунок 3.5 – Конфігурований клієнт Axios

Клієнт автоматично додає базову URL-адресу API з змінної середовища та встановлює необхідні заголовки. Критичним аспектом безпеки є впровадження interceptor для запитів, який автоматично додає JWT токен до заголовка Authorization для кожного запиту. При отриманні відповіді зі статусом 401 Unauthorized, interceptor автоматично очищує збережені дані автентифікації та перенаправляє користувача на сторінку входу, забезпечуючи автоматичну обробку закінчення сеансу. Специфічні методи для роботи з завданнями оформлені у модулі tasks.api.ts, який містить функції для

отримання списку завдань, створення, оновлення та видалення окремих завдань. Структура компонентів розроблена за принципом композиції та одиничної відповідальності. На рисунку 3.6 продемонстровано сторінку BoardPage.

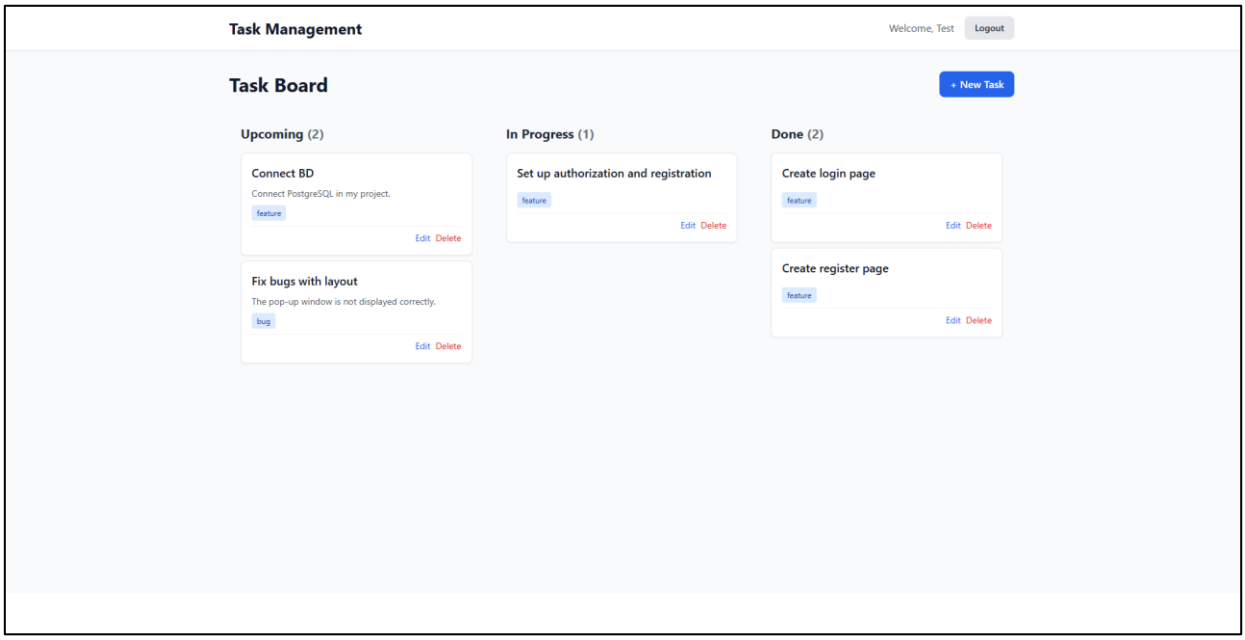


Рисунок 3.6 – Основна сторінка

Основна сторінка BoardPage інтегрує всі необхідні компоненти та управляє основним потоком користувача. Компонент обгортає дошку завдань у провайдер DndProvider з react-dnd бібліотеки, що забезпечує функціональність перетягування.

На сторінці відображаються три колонки через компонент TaskColumn, кожна з яких представляє один зі статусів завдання. Компонент TaskColumn встановлює drop zone за допомогою хука useDrop з react-dnd, дозволяючи перетягувати завдання в межах колонки.

Для кожного завдання в колонці використовується компонент TaskCard з **Додаток Д**, який реалізує логіку drag через хук useDrag. Компонент TaskCard відображає інформацію про завдання включаючи назву, описання та теги, а також надає кнопки для редагування та видалення.

На рисунку 3.7 зображена форма для створення та редагування завдань.

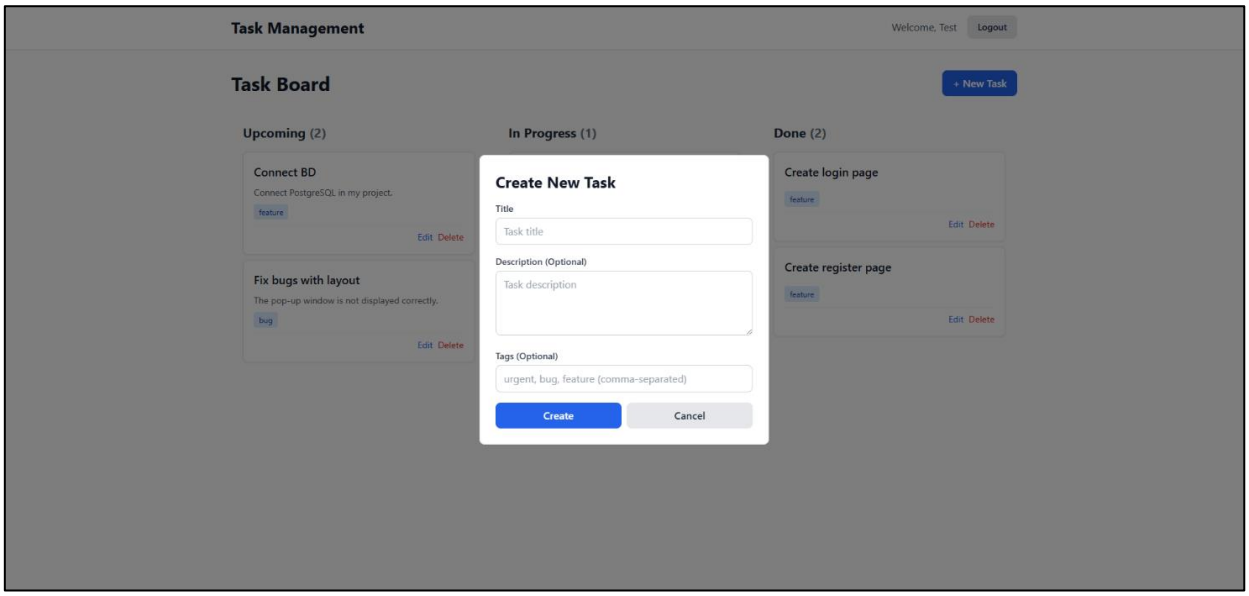


Рисунок 3.7 – Форма для створення та редагування завдань

Форма для створення та редагування завдань реалізована в компоненті TaskForm, який використовує контрольовані компоненти input для введення даних.

Валідація даних здійснюється на клієнтській стороні з використанням утиліт, що перевіряють наявність та формат введених значень. Компоненти користувацького інтерфейсу, такі як Button, Input, Card, оформлені як переозначуємі примітивні компоненти, що дозволяє встановлювати консистентні стилі та поведінку по всьому застосунку. Заголовок Header відображає привітання користувачу та кнопку для виходу з системи, забезпечуючи навігацію та управління сеансом.

Адаптивність інтерфейсу забезпечена фреймворком TailwindCSS. Конфігурація TailwindCSS розширює стандартні можливості з визначенням кастомної палітри кольорів, зокрема primary кольори для узгодження з дизайном застосунку.

Макет доски завдань використовує flexbox для гнучкого розташування колонок з горизонтальним прокручуванням на вузьких екранах, забезпечуючи зручність роботи на мобільних пристроях.

Компоненти застосовують умовні CSS класи для різних розмірів екрана, наприклад через утиліти типу sm:px-6 lg:px-8 у компоненті Header, що

дозволяє налаштовувати відступи залежно від ширини екрану. Сторінки входу та реєстрації розроблені з урахуванням мобільності з адаптивною шириною та зручним розташуванням елементів форми.

Авторизація на фронтенді реалізована через JWT токени, що передаються в заголовках запитів. При вході користувача або реєстрації сервер повертає JWT токен, який зберігається у локальному сховищі браузера. Компонент AuthContext управляє цим токеном та забезпечує його передачу всім API запитам через interceptor клієнта Axios.

При кожному запуску застосунку контекст перевіряє наявність збереженого токена та користувача в локальному сховищі, відновлюючи сеанс без необхідності повторного входу. Механізм автоматичного перенаправлення при закінченні сеансу гарантує, що користувач не зможе отримати доступ до захищених ресурсів після закінчення терміну дії токена.

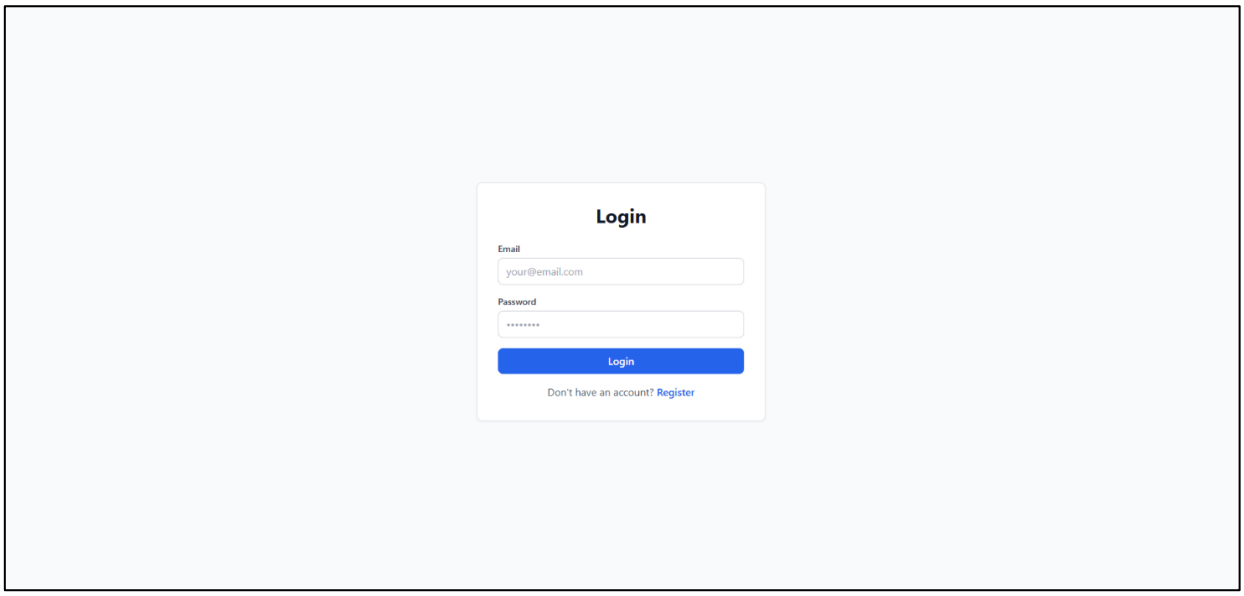
Реалізована також система сповіщень про помилки автентифікації, що надає користувачу зрозумілу інформацію при невдалих спробах входу або реєстрації.

Для розмежування доступу до публічних та приватних сторінок впроваджено компонент захищеного маршруту, який аналізує стан контексту перед відображенням контенту. У випадку відсутності дійсного токена цей механізм автоматично блокує перехід до внутрішніх розділів та миттєво переадресовує користувача на сторінку авторизації.

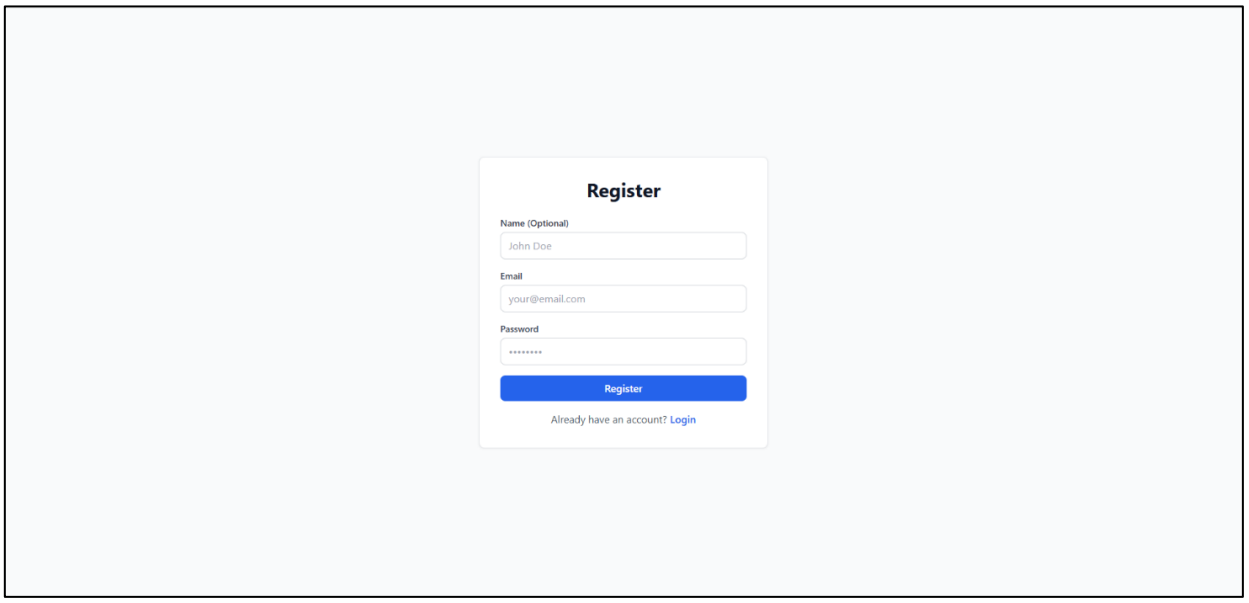
Окремо передбачено функціонал безпечного виходу із системи, який ініціює повне видалення конфіденційних даних із пам’яті браузера та скидання глобального стану застосунку. Така архітектура клієнтської частини дозволяє поєднати високий рівень захисту даних.

На рисунку 3.8 представлено реалізовані сторінки форми входу та реєстрації.

Загалом, архітектура клієнтської частини демонструє застосування сучасних практик веброзробки, включаючи типизацію через TypeScript, управління станом через React Context API, компонентно-орієнтований дизайн та адаптивність интерфейсу.



а) реалізована Login page



б) реалізована Register page

Рисунок 3.8 – Сторінки входу та реєстрації

Модульна структура коду, відокремлення логіки комунікації з серверим від логіки компонентів та використання кастомних хуків забезпечують чистоту коду та його легкість у розробці та тестуванні. Впровадження безпечної авторизації, системи сповіщень та реактивного управління станом забезпечує надійну та зручну взаємодію користувача з вебсервісом для управління завданнями.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було реалізовано повний цикл розробки вебсервісу для управління завданнями з адаптивним інтерфейсом на основі технологій Node.js та React.js. Проведене дослідження охопило декілька ключових етапів, серед яких аналіз предметної області, формування вимог до системи, проєктування архітектури, розробка користувацького інтерфейсу та створення програмної реалізації.

На етапі дослідження предметної області було проаналізовано специфіку сучасних інструментів управління завданнями та виявлено потребу у створенні вебсервісу, здатного забезпечити гнучкість, швидкодію, зручність у використанні та адаптивність до різних пристроїв. Результати аналізу визначили основні функціональні можливості майбутньої системи та слугували підґрунтям для подальшого проєктування.

Проєктування системи включало розробку структурної та логічної архітектури, моделювання прецедентів використання, визначення ключових компонентів серверної та клієнтської частин, а також обґрунтування вибору технологій. Застосування Node.js дозволило забезпечити продуктивне опрацювання запитів і масштабованість серверної частини, а використання React.js дало можливість реалізувати динамічний, адаптивний та зручний інтерфейс користувача. Це забезпечило узгодженість програмних модулів та ефективність роботи всієї системи.

У процесі розробки користувацького інтерфейсу значна увага приділялася принципам адаптивності та юзабіліті. Інтерфейс було побудовано таким чином, щоб забезпечити зручний доступ до функціоналу системи з різних пристроїв, включно з мобільними. Реалізація компонентної структури сприяла простоті підтримки та розширюваності проєкту.

Етап програмної реалізації продемонстрував ефективність обраних технологій та методів. Серверну частину було реалізовано з урахуванням

принципів модульності, безпеки та можливості масштабування. Клієнтська частина забезпечила інтуїтивну роботу з завданнями, включно зі створенням, редагуванням, впорядкуванням та контролем виконання. Була досягнута узгодженість між фронтендом і бекендом, що забезпечило стабільну роботу програмного продукту.

У результаті проведеної роботи досягнуто поставлену мету – створено вебсервіс для управління завданнями, який характеризується адаптивним інтерфейсом, високою зручністю використання та гнучкістю масштабування. Система дозволяє оптимізувати робочі процеси, підвищує ефективність управління завданнями та може використовуватися як у невеликих командах, так і індивідуальними користувачами.

Розроблений вебсервіс має потенціал для подальшого вдосконалення та розширення. Перспективними напрямами розвитку можуть бути інтеграція з зовнішніми сервісами, розширення функціоналу аналітики, впровадження системи ролей та прав доступу, додавання можливостей командної співпраці та розширення адаптивних можливостей інтерфейсу. Виконана робота демонструє актуальність комплексного підходу до створення сучасних вебзастосунків та підтверджує ефективність використання Node.js і React.js для реалізації динамічних і масштабованих систем.

Таким чином, кваліфікаційна робота реалізувала повноцінний сучасний вебсервіс, який відповідає вимогам актуальних підходів до веброзробки та може слугувати основою для подальших досліджень і практичних впроваджень у сфері систем управління завданнями.

ПЕРЕЛІК ПОСИЛАНЬ

1. Appvin Web Application Development Services for Modern Businesses. *Medium*. URL: https://medium.com/@appvintechnologies/appvin-web-application-development-services-for-modern-businesses-fea862fbda9b (дата звернення: 19.04.2025).

2. Biehl M. RESTful API Design – Best Practices in API Design with REST: API-University Series, #3. API-University Press, 2016. 475 p. URL: https://www.everand.com/book/407678610/RESTful-API-Design-Best-Practices-in-API-Design-with-REST-API-University-Series-3 (дата звернення: 20.04.2025).

3. Herron D. Node.js Web Development : 3rd ed. Packt Publishing, 2016. 809 p. URL: https://www.everand.com/book/365187139/Node-js-Web-Development-Third-Edition (дата звернення: 25.04.2025).

4. Poulter A. J., Johnston S. J., Cox S. J. Using the MEAN stack to implement a RESTful service for an Internet of Things application. *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. Milan, Italy, 2015. P. 280-285. URL: https://doi.org/10.1109/WF-IoT.2015.7389066 (дата звернення: 25.04.2025).

5. Chen S., Thaduri U. R., Ballamudi V. K. R. Front-End Development in React: An Overview. *Engineering International*. 2019. Vol. 7, No. 2. P. 117-120.

6. Responsive Web Design. URL: https://alistapart.com/article/responsive-web-design/ (дата звернення: 11.06.2025).

7. Frain B. Responsive Web Design with HTML5 and CSS3 : 2nd ed. Packt Publishing, 2015. 312 p.

8. Taylor A. G. SQL All-in-One For Dummies. Wiley, 2011. 1196 p. URL: https://www.everand.com/book/220195238/SQL-All-in-One-For-Dummies (дата звернення: 15.08.2025).

9. Teixeira F. Introdução e boas práticas em UX Design. Casa do Código, 2014. 308 p. URL: https://www.everand.com/book/405708544/Introducao-e-boas-praticas-em-UX-Design (дата звернення: 27.08.2025).

10. Grassle P., Baumann H., Baumann P. UML 2.0 in Action: A project-based tutorial. Packt Publishing, 2005. 551 p. URL: https://www.everand.com/book/272083393/UML-2-0-in-Action-A-project-based-tutorial (дата звернення: 06.09.2025).

11. What is Sequence Diagram? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/ (дата звернення: 10.09.2025).

ДОДАТОК А

Файл docker-compose.yml

```
services:
  postgres:
    image: postgres:16-alpine
    container_name: taskmanagement-postgres
    restart: unless-stopped
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: password
      POSTGRES_DB: taskmanagement
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U postgres"]
      interval: 10s
      timeout: 5s
      retries: 5

  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: taskmanagement-backend
    restart: unless-stopped
    depends_on:
      postgres:
        condition: service_healthy
    environment:
      DATABASE_URL: postgresql://postgres:password@postgres:5432/taskmanagement?schema=public
      JWT_SECRET:
      JWT_EXPIRES_IN: 24h
      PORT: 3000
      NODE_ENV: production
      CORS_ORIGIN: http://localhost:3001,http://localhost:5173
    ports:
      - "3000:3000"
    command: sh -c "npx prisma migrate deploy && npm start"

  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: taskmanagement-frontend
    restart: unless-stopped
    depends_on:
      - backend
    ports:
      - "3001:80"

volumes:
  postgres_data:
```

ДОДАТОК Б

Файл Prisma Migrate

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id        String   @id @default(uuid())
  email     String   @unique
  password  String
  name      String?
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  tasks     Task[]
}

model Task {
  id          String     @id @default(uuid())
  title       String
  description String?
  status      TaskStatus @default(UPCOMING)
  position    Float
  tags        String[]
  userId      String
  user        User       @relation(fields: [userId], references: [id], onDelete: Cascade)
  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt

  @@index([userId, status, position])
  @@index([userId, createdAt])
}

enum TaskStatus {
  UPCOMING
  IN_PROGRESS
  DONE
}
```

ДОДАТОК В

Контролер авторизації

```
import prisma from '../../config/database';
import { hashPassword, comparePassword } from '../../utils/password';
import { generateToken } from '../../utils/jwt';
import { UnauthorizedError, ConflictError, NotFoundError } from '../../utils/ApiError';
import { RegisterInput, LoginInput } from './auth.validation';

export class AuthService {
  async register(data: RegisterInput) {
    // Check if user already exists
    const existingUser = await prisma.user.findUnique({
      where: { email: data.email },
    });

    if (existingUser) {
      throw new ConflictError('Email already registered');
    }

    // Hash password
    const hashedPassword = await hashPassword(data.password);

    // Create user
    const user = await prisma.user.create({
      data: {
        email: data.email,
        password: hashedPassword,
        name: data.name,
      },
      select: {
        id: true,
        email: true,
        name: true,
        createdAt: true,
        updatedAt: true,
      },
    });

    // Generate JWT token
    const token = generateToken({
      userId: user.id,
      email: user.email,
    });

    return { user, token };
  }

  async login(data: LoginInput) {
    // Find user
    const user = await prisma.user.findUnique({
      where: { email: data.email },
    });

    if (!user) {
```

```
      throw new UnauthorizedError('Invalid email or password');
    }

    // Verify password
    const isValidPassword = await comparePassword(data.password, user.password);

    if (!isValidPassword) {
      throw new UnauthorizedError('Invalid email or password');
    }

    // Generate JWT token
    const token = generateToken({
      userId: user.id,
      email: user.email,
    });

    // Return user without password
    const { password, ...userWithoutPassword } = user;

    return { user: userWithoutPassword, token };
  }

  async getCurrentUser(userId: string) {
    const user = await prisma.user.findUnique({
      where: { id: userId },
      select: {
        id: true,
        email: true,
        name: true,
        createdAt: true,
        updatedAt: true,
      },
    });

    if (!user) {
      throw new NotFoundError('User');
    }

    return user;
  }
}
```

ДОДАТОК Г

Контролер задач

```
import prisma from '../../config/database';
import { TaskStatus } from '@prisma/client';
import { NotFoundError, ForbiddenError } from '../../utils/ApiError';
import { CreateTaskInput, UpdateTaskInput, GetTasksQuery } from './tasks.validation';

export class TasksService {
  async getTasksByUser(userId: string, query: GetTasksQuery) {
    const { status, search, tags, page = 1, limit = 50 } = query;
    const skip = (page - 1) * limit;

    const where: any = { userId };

    if (status) {
      where.status = status;
    }

    if (search) {
      where.OR = [
        { title: { contains: search, mode: 'insensitive' } },
        { description: { contains: search, mode: 'insensitive' } },
      ];
    }

    if (tags) {
      const tagsArray = tags.split(',').map((t) => t.trim());
      where.tags = { hasSome: tagsArray };
    }

    const [tasks, total] = await Promise.all([
      prisma.task.findMany({
        where,
        skip,
        take: limit,
        orderBy: [{ status: 'asc' }, { position: 'asc' }],
      }),
      prisma.task.count({ where }),
    ]);

    return {
      tasks,
      pagination: {
        total,
        page,
        limit,
        totalPages: Math.ceil(total / limit),
      },
    };
  }

  async getTaskById(taskId: string, userId: string) {
    const task = await prisma.task.findUnique({
      where: { id: taskId },
```

```
  });

  if (!task) {
    throw new NotFoundError('Task');
  }

  if (task.userId !== userId) {
    throw new ForbiddenError('You do not have access to this task');
  }

  return task;
}

async createTask(userId: string, data: CreateTaskInput) {
  // Get max position in UPCOMING column for this user
  const lastTask = await prisma.task.findFirst({
    where: { userId, status: 'UPCOMING' },
    orderBy: { position: 'desc' },
  });

  const position = lastTask ? lastTask.position + 1000 : 1000;

  const task = await prisma.task.create({
    data: {
      title: data.title,
      description: data.description,
      tags: data.tags || [],
      userId,
      position,
      status: 'UPCOMING',
    },
  });

  return task;
}

async updateTask(taskId: string, userId: string, data: UpdateTaskInput) {
  // Verify ownership
  const existingTask = await this.getTaskById(taskId, userId);

  // If status or position is being updated, check for re-indexing
  if (data.status !== undefined || data.position !== undefined) {
    const targetStatus = data.status || existingTask.status;
    const targetPosition = data.position !== undefined ? data.position : existingTask.position;

    await this.checkAndReindexIfNeeded(userId, targetStatus, targetPosition);
  }

  const updatedTask = await prisma.task.update({
    where: { id: taskId },
    data: {
      title: data.title,
      description: data.description,
      status: data.status,
      position: data.position,
      tags: data.tags,
    },
  });

  return updatedTask;
```

```
  }

  async deleteTask(taskId: string, userId: string) {
    // Verify ownership
    await this.getTaskById(taskId, userId);

    await prisma.task.delete({
      where: { id: taskId },
    });
  }

  private async checkAndReindexIfNeeded(
    userId: string,
    status: TaskStatus,
    newPosition: number
  ) {
    const tasksInColumn = await prisma.task.findMany({
      where: { userId, status },
      orderBy: { position: 'asc' },
    });

    // Check if any two tasks are too close
    let needsReindex = false;
    for (let i = 0; i < tasksInColumn.length - 1; i++) {
      const diff = Math.abs(tasksInColumn[i + 1].position - tasksInColumn[i].position);
      if (diff < 0.0001) {
        needsReindex = true;
        break;
      }
    }

    // Also check if the new position is too close to any existing task
    for (const task of tasksInColumn) {
      if (Math.abs(task.position - newPosition) < 0.0001) {
        needsReindex = true;
        break;
      }
    }

    if (needsReindex) {
      await this.reindexColumn(userId, status);
    }
  }

  private async reindexColumn(userId: string, status: TaskStatus) {
    const tasks = await prisma.task.findMany({
      where: { userId, status },
      orderBy: { position: 'asc' },
    });

    const updates = tasks.map((task, index) =>
      prisma.task.update({
        where: { id: task.id },
        data: { position: (index + 1) * 1000 },
      })
    );

    await prisma.$transaction(updates);
  }
}
```

ДОДАТОК Д

Компонент TaskCard

```
import React from 'react';
import { useDrag } from 'react-dnd';
import { Task } from '../../types/task.types';
import { Card } from '../ui/Card';

interface TaskCardProps {
  task: Task;
  onEdit?: (task: Task) => void;
  onDelete?: (id: string) => void;
}

export const TaskCard: React.FC<TaskCardProps> = ({ task, onEdit, onDelete }) => {
  const [{ isDragging }, drag] = useDrag(() => ({
    type: 'TASK',
    item: { id: task.id, status: task.status, position: task.position },
    collect: (monitor) => ({
      isDragging: monitor.isDragging(),
    }),
  }), [task.id, task.status, task.position]);

  return (
    <div ref={drag} style={{ opacity: isDragging ? 0.5 : 1 }}>
      <Card className="p-4 mb-2 cursor-move hover:shadow-md transition-shadow">
        <h3 className="font-semibold text-lg mb-2">{task.title}</h3>
        {task.description && (
          <p className="text-gray-600 text-sm mb-2 line-clamp-2">{task.description}</p>
        )}
        {task.tags && task.tags.length > 0 && (
          <div className="flex flex-wrap gap-1 mb-2">
            {task.tags.map((tag, index) => (
              <span
                key={index}
                className="px-2 py-1 bg-blue-100 text-blue-800 text-xs rounded"
              >
                {tag}
              </span>
            ))}
          </div>
        )}
        {(onEdit || onDelete) && (
          <div className="flex justify-end gap-2 mt-2 pt-2 border-t border-gray-100">
            {onEdit && (
              <button
                onClick={(e) => {
                  e.stopPropagation();
                  onEdit(task);
                }}
                className="text-blue-600 text-sm hover:text-blue-700"
              >
                Edit
              </button>
            )}
            {onDelete && (
              <button
```

```
                onClick={(e) => {
                  e.stopPropagation();
                  onDelete(task.id);
                }}
                className="text-red-600 text-sm hover:text-red-700"
              >
                Delete
              </button>
            )}
          </div>
        )}
      </Card>
    </div>
  );
};
```