БАКАЛАВРСЬКА

РОБОТА

БР.ІП – 03.00.00.000 ПЗ

Група ІП-22-1

Дутчак Юлія

Івано-Франківський національний технічний університет нафти і газу

Факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Дутчак Юлія Василівна

(прізвище, ім'я, по батькові)

УДК 004.942
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розроблення веб-застосунку для управління проєктами

(назва роботи)

Інженерія програмного забезпечення

(назва освітньої програми)

121– Інженерія програмного забезпечення

(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Дутчак Ю.В.
(підпис, ініціали та прізвище здобувача)

Науковий керівник Григорчук Любомир Іванович, к.п.н., доцент
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
Завідувач кафедри

доц. Бандура В.В.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу

Інститут, факультет інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Спеціальність 121 – Інженерія програмного забезпечення

***ЗАТВЕРДЖУЮ*:**
Зав. кафедрою *ІПЗ*
доц. В.В. Бандура

“_____” _____2026 р

ЗАВДАННЯ

НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Дутчак Юлія Василівна

(прізвище, ім’я, по-батькові)

1. ***Тема проекту (роботи)*** **“*Розроблення веб-застосунку для управління проєктами*”**

керівник проекту (роботи) Григорчук Любомир Іванович, к.п.н., доцент

затверджені наказом закладу вищої освіти від **“** 21 ” квітня 2026 р. № 179/7

2. ***Строкподання студентом проекту (роботи)*** червня 2026 р.

3. ***Вихідні дані до проекту (роботи)***

4. ***Зміст розрахунково - пояснювальної записки (перелік питань, які потрібно розробити)***

1. Аналіз предметної області та існуючих рішень для управління проєктами.

2. Формування вимог і моделювання сценаріїв використання веб-застосунку.

3. Проєктування архітектури системи та бази даних.

4. Реалізація функціональних модулів веб-застосунку.

5. Тестування програмного продукту та представлення результатів роботи.

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

1.Діаграма варіантів використання веб-застосунку для управління проєктами (рис. 2.1, ст.29)

2. Діаграма активності процесу роботи із завданнями (рис. 2.2, ст.30)

3. Діаграма компонентів веб-застосунку для управління проєктами (рис. 3.1, ст. 43)

4. ER-діаграма бази даних веб-застосунку (рис. 3.2, ст. 46)

5. Головне вікно веб-застосунку та інтерфейс керування завданнями (рис. 5.5, ст. 72)

1. ***Консультанти розділів проекту (роботи)***

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1-5	Григорчук Л.І.		

2. ***Дата видачі завдання*** _____

Керівник _____ _____
(підпис) (розшифровка підпису)

Завдання прийняв до виконання _____ _____
(підпис) (розшифровка підпису)

КАЛЕНДАРНИЙ ПЛАН

№ з/п	*Назва етапів дипломного проекту (роботи)*	*Строк виконання етапів проекту*	*Примітка*
1	Визначення та обґрунтування теми роботи	15.01.2026	Виконано
2	Огляд існуючих концепцій, рішень та сервісів в даній області	25.02.2026	Виконано
3	Побудова моделі або алгоритму власного рішення	15.03.2026	Виконано
4	Документування реалізації власного оригінального рішення вибраними засобами	25.04.2026	Виконано
5	Оформлення пояснювальної записки бакалаврської роботи	10.06.2026	Виконано

Студент – дипломник _____ Дутчак Ю.В.
(підпис) (розшифровка підпису)

Керівник роботи _____ Григорчук Л.І.
(підпис) (розшифровка підпису)

АНОТАЦІЯ

Бакалаврська робота складається з 84 сторінок, містить 20 рисунків, 3 таблиці, список використаних джерел із 30 найменувань.

Метою роботи є розроблення веб-застосунку для управління проєктами, який оптимізує процеси командного планування, розподілу завдань та контролю термінів виконання робіт.

Об'єктом дослідження є процес організації, координації та контролю виконання робіт у командах, та створення програмного забезпечення із зручним та зрозумілим інтерфейсом.

Предмет дослідження: архітектурні рішення, методи проєктування, технологічні інструменти та алгоритми реалізації веб-застосунку.

Результати дослідження: розроблено веб-застосунок TaskFlow

У першому розділі проаналізовано предметну область, платформи Jira, Trello та Asana й обґрунтовано елементи удосконалення TaskFlow.

У другому розділі визначено потреби користувачів, сформовано функціональні та нефункціональні вимоги та побудовано UML-діаграми сценаріїв взаємодії.

У третьому розділі спроєктовано клієнт-серверну архітектуру, REST API, модель доступу та реляційну базу даних.

У четвертому розділі обґрунтовано технологічний стек і реалізовано основні функціональні модулі та удосконалений механізм контролю робочого процесу.

У п’ятому розділі налаштовано тестове середовище, виконано 45 автоматизованих інтеграційних перевірок і проведено ручне функціональне тестування.

Висновок: Спроєктований веб-застосунок забезпечує гнучке налаштування робочих процесів, відповідає критеріям надійності й швидкодії та є готовим інструментом для управління проєктами.

КЛЮЧОВІ СЛОВА: ВЕБ-ЗАСТОСУНОК, УПРАВЛІННЯ ПРОЄКТАМИ, ПРОЄКТУВАННЯ СИСТЕМИ, UML-ДІАГРАМА, БАЗА ДАНИХ.

ABSTRACT

The bachelor’s thesis consists of 84 pages and contains 20 figures, 3 tables, and a list of 30 references.

The purpose of the thesis is to develop a project management web application that improves team planning, task allocation, and deadline monitoring.

The object of the research is the process of organizing, coordinating, and monitoring work within teams, as well as the development of software with a clear and user-friendly interface.

The subject of the research comprises architectural solutions, design methods, technological tools, and algorithms used to implement a project management web application.

The main result of the research is the developed TaskFlow web application.

The first chapter analyzes the subject area and the Jira, Trello, and Asana platforms and substantiates the proposed improvements introduced in TaskFlow.

The second chapter identifies user needs, defines the functional and non-functional requirements, and presents UML diagrams describing the main user interaction scenarios.

The third chapter describes the design of the client-server architecture, REST API, access control model, and relational database.

The fourth chapter substantiates the selected technology stack and describes the implementation of the main functional modules and the improved workflow monitoring mechanism.

The fifth chapter presents the configuration of the testing environment, the results of 45 automated integration tests, and the manual functional testing of the web application.

The developed web application supports project planning, role-based access control, task and sprint management, workflow monitoring, and project analytics. The testing results confirmed the reliability and correct operation of the main functional modules. TaskFlow can be used as a practical tool for improving project management efficiency in small teams.

KEYWORDS: WEB APPLICATION, PROJECT MANAGEMENT, SYSTEM DESIGN, UML DIAGRAM, DATABASE.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ІТ — інформаційні технології.

ПЗ — програмне забезпечення.

СУБД — система управління базами даних.

AAA (Arrange, Act, Assert) — підхід до організації тестів, що передбачає підготовку даних, виконання дії та перевірку результату.

API (Application Programming Interface) — програмний інтерфейс застосунку.

CORS (Cross-Origin Resource Sharing) — механізм спільного використання ресурсів між різними джерелами.

ER (Entity–Relationship) — модель даних «сутність–зв’язок».

HTTP (Hypertext Transfer Protocol) — протокол передавання гіпертексту.

JSON (JavaScript Object Notation) — текстовий формат представлення та обміну структурованими даними.

JWT (JSON Web Token) — вебтокен у форматі JSON, що використовується для автентифікації й передавання даних про користувача.

MIME (Multipurpose Internet Mail Extensions) — стандарт визначення типу вмісту файла або повідомлення.

ORM (Object-Relational Mapping) — технологія об’єктно-реляційного відображення даних.

OWASP (Open Worldwide Application Security Project) — міжнародний відкритий проєкт, присвячений безпеці веб-застосунків.

REST (Representational State Transfer) — архітектурний стиль взаємодії компонентів розподіленої системи.

RFC (Request for Comments) — серія технічних документів, що містять специфікації та стандарти Інтернету.

SQL (Structured Query Language) — мова структурованих запитів до реляційних баз даних.

UML (Unified Modeling Language) — уніфікована мова моделювання.

ВСТУП

Актуальність теми зумовлена тим, що у нинішній ІТ-індустрії успіх релізу чи продукту залежить від взаємовідносин людей в середині компанії. Проблема координації суттєво загострилася внаслідок масового переходу ІТ-фахівців на віддалений формат роботи та географічного розподілу учасників команд розробників. В наслідок чого проводити комунікацію між співробітниками а також координувати процеси на відстані стало важче. Невеликі команди потребують середовище, яке поєднує наочне відображення/представлення завдань, підтримку спринтів і розмежування доступу без надмірної складності корпоративних платформ. Розроблення та впровадження спеціалізованого програмного забезпечення, з інтерактивними дошками та наочними графіками розв'язує цю проблему без зайвих витрат.

Обґрунтування вибору теми пов’язане зі зростаючою потребою у сучасних системах управління проєктами, які поєднують простоту використання з широкими функціональними можливостями. На ринку існує чіткий попит на зручні середовища для координації та управління проєктами, що забезпечують гнучкий розподіл прав доступу, зручний контроль виконання завдань та можливість інтеграції з іншими інформаційними системами за допомогою REST API.

Метою роботи є розробка веб-застосунку TaskFlow для централізованого планування проєктів, розподілу завдань між учасниками команди, контролю їх виконання та формування аналітичних даних про поточний стан проєкту.

Для досягнення поставленої мети необхідно виконати такі завдання:

1. проаналізувати предметну область і функціональні можливості сучасних систем управління проєктами;

2. порівняти платформи Jira, Trello та Asana й визначити їхні переваги та обмеження;

3. визначити категорії користувачів TaskFlow і сформувати функціональні та нефункціональні вимоги;

4. змоделювати основні сценарії взаємодії користувачів із системою за допомогою UML-діаграм;

5. спроєктувати клієнт-серверну архітектуру, реляційну модель даних, REST API та механізм розмежування доступу;

6. реалізувати серверну й клієнтську частини веб-застосунку та основні функціональні модулі;

7. реалізувати механізми автоматичного ведення історії змін, формування стрічки активності та контролю стану спринтів;

8. налаштувати тестове середовище, провести автоматизоване інтеграційне й ручне функціональне тестування та проаналізувати отримані результати.

Об'єктом дослідження є процес організації, координації та контролю виконання робіт у командах, та створення програмного забезпечення із зручним та зрозумілим інтерфейсом.

Предметом дослідження є методи, інструменти та програмні засоби побудови веб-застосунку для управління проєктами на основі технологічного стеку React, платформа Node.js та СУБД PostgreSQL.

Елементом новизни роботи є узгоджений механізм контролю робочого процесу, який поєднує дворівневе розмежування доступу, автоматичну фіксацію змін з журналюванням в TaskHistory, серверний контроль бізнес-правила «один активний спринт» і формування аналітичних показників на основі агрегації актуальних проєктних даних.

Практичне значення результатів полягає у готовому до розгортання веб-застосунку, який може використовуватися малими командами для організації Agile-процесів без необхідності адміністрування складних корпоративних платформ. Вихідний код структуровано для подальшого розширення.

Перспективами подальшого розвитку TaskFlow є: сповіщення в реальному часі на основі WebSocket, інтеграція з системами контролю версій (GitHub / GitLab), розширення аналітики (burn-down chart, velocity), мобільна адаптація інтерфейсу та навантажувальне тестування для визначення межових характеристик продуктивності.

Бакалаврська робота містить 84 сторінок,20 рисунків та 3 таблиці, список літератури із 30 найменувань, та 1 додаток.

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1. Поняття систем управління проєктами в сучасній інженерії програмного забезпечення

Покращення якості проекту та ефективність організації робочих процесів є критичними для успіху проєкту в сучасній індустрії розробки програмного забезпечення. Специфіка інженерії програмного забезпечення полягає у високій динаміці вимог, змінах пріоритетів, які відбуваються швидко з часом; цикл, що характеризується постійною взаємодією між проєктними менеджерами, розробниками, тестувальниками, бізнес-аналітиками та замовниками. За цих умов системи управління проєктами є необхідним засобом організації роботи для управління та контролю ходу розробки програмного забезпечення. Система управління проєктами - це набір інструментів для планування, організації та моніторингу масиву завдань, спрямованих на досягнення конкретних цілей у межах, визначених часом, бюджетом та якістю. Ці системи виконують низку важливих функцій в інженерії програмного забезпечення:

Керування функціями та завданнями включає відстеження процесу виконання робіт та розбиття великих цілей, таких як епіки або функціональні модулі, на менші та зрозумілі завдання для розробників. До таких елементів можуть належати окремі задачі та підзадачі, які полегшують планування, розподіл відповідальності й контроль виконання роботи. Кожне завдання має унікальний ідентифікатор, опис, пріоритет, дедлайн та відповідального виконавця.

Візуалізація робочого процесу. За допомогою засобів візуального моделювання, таких як дошки Kanban чи діаграми Ганта, команда може бачити стан кожного завдання в реальному часі («До виконання», «В роботі», «Тестування», «Завершено»). Координація співпраці. Створення

централізованої інформаційної зони для учасників проєкту, де можна обмінюватися коментарями, публікувати технічні документи, посилання на фрагменти коду або ескізи дизайну.

Відстеження часу та розподіл ресурсів. Ведення журналу годин, витрачених на реалізацію конкретної функціональності. Це також сприяє оцінці продуктивності команди, що дозволяє надавати більш точні прогнози термінів для майбутніх етапів (спринтів).

Збір аналітики та звітності. Побудова діаграм згоряння, звітів про швидкість команди, аналізу метрик та розподілу навантаження. Такі показники важливі для менеджерів, щоб вимірювати та аналізувати процеси для оцінки результативності та управління ризиками.

Сучасні системи управління проєктами адаптовані до вимог гнучких методологій, зокрема Agile, Scrum та Kanban. Для ефективної координації процесів засоби автоматизації повинні підтримувати ітераційну модель розробки. Це передбачає наявність функціоналу для формування та пріоритезації беклогу, планування й моніторингу спринтів, а також фіксації результатів ретроспективних оглядів. Відповідно до Scrum Guide, робота команди організовується в межах спринтів, протягом яких створюється завершений інкремент продукту [26]. Kanban орієнтований на визначення, візуалізацію та вдосконалення потоку робіт, що дає змогу контролювати проходження завдань між етапами виконання [27].

Таким чином, у межах інженерії програмного забезпечення системи управління проєктами перейшли від простих переліків завдань до комплексних веб-орієнтованих платформ, які автоматизують супровід життєвого циклу програмного забезпечення, забезпечують планування, розподіл і контроль виконання робіт, а також сприяють прозорій взаємодії між учасниками команди. Такі системи дозволяють мінімізувати вплив людського фактора на організаційному рівні, стандартизувати процеси постановки, відстеження задач, контролювати пріоритети та терміни виконання

1.2. Аналіз існуючих рішень для координації командної роботи

Для обґрунтування створення веб-застосунку TaskFlow слід здійснити аналіз ринку сучасних систем управління проектами. Відомі розробники інструментів для відслідковування роботи команд, такі як Atlassian Jira, Asana та Trello, у своїй офіційній документації та технічних специфікаціях акцентують увагу на різних підходах до організації робочого середовища - від гнучких інженерних платформ до візуальних карт завдань.

Одним із найбільш поширених інструментів у сфері розроблення програмного забезпечення є Jira. Згідно з офіційними матеріалами Atlassian, платформа надає засоби для планування, відстеження та контролю виконання робіт [1]. Система підтримує гнучкі методології Scrum і Kanban. Однією з її основних переваг є можливість детального налаштування життєвого циклу завдання відповідно до реальних етапів розроблення програмного забезпечення [2]. Команди можуть створювати власні статуси, керувати беклогом, планувати спринти та формувати звіти щодо виконання завдань. Для великих компаній Jira є ефективним інструментом, однак для невеликих значна кількість налаштувань може ускладнювати початкове освоєння та адміністрування системи.

Простішим рішенням для організації роботи є Trello. В офіційному керівництві платформа описується як універсальний інструмент для координації різних типів проєктів [3]. Основу її структури становлять дошки, списки та картки. Дошка відповідає певному проєкту або процесу, списки відображають етапи виконання, а картки використовуються для представлення окремих завдань. Такий підхід є інтуїтивно зрозумілим і добре підходить для особистого планування, навчальних проєктів та нескладних бізнес-процесів.

Trello підтримує автоматизацію повторюваних дій за допомогою вбудованого інструменту Butler [4]. Користувачі можуть створювати правила та кнопки, які автоматично переміщують картки, призначають виконавців або

змінюють строки виконання. Додаткові функціональні можливості та інтеграції підключаються за допомогою розширень Power-Ups [5]. Водночас базові можливості Trello є недостатніми для складного управління процесами розроблення програмного забезпечення, оскільки система не має повноцінної підтримки спринтів, розвиненої ієрархії завдань та детальної Agile-аналітики.

Asana займає проміжне положення між функціонально насиченою Jira та спрощеним підходом Trello. Відповідно до офіційної документації, один проєкт в Asana можна переглядати у вигляді списку, Kanban-дошки, календаря або часової шкали [6]. Використання різних способів представлення дозволяє менеджерам контролювати строки виконання, планувати роботу та аналізувати загальний стан проєкту.

Asana також підтримує встановлення залежностей між завданнями, завдяки чому можна визначити, що виконання одного завдання залежить від завершення іншого [7]. Це допомагає контролювати послідовність робіт і виявляти можливі затримки. Платформа є зручною для загального управління командною діяльністю.

Для узагальнення результатів проведеного аналізу та наочного порівняння функціональних можливостей платформ Jira, Trello та Asana основні характеристики розглянутих систем наведено в таблиці 1.1.

Таблиця 1.1

Порівняння функціональних можливостей Jira, Trello та Asana

Критерій	Jira	Trello	Asana
Основний підхід	Управління процесами розроблення програмного забезпечення	Візуальне керування картками на дошках	Планування та координація командної роботи
Kanban-дошка	Підтримується	Є основним способом організації роботи	Підтримується як один зі способів представлення
Scrum і спринти	Повноцінна підтримка	У базовій конфігурації відсутні	Обмежена підтримка
Керування беклогом	Підтримується	Спеціалізований беклог відсутній	Не є основною функцією

Продовження таблиці 1.1

Залежності між завданнями	Підтримуються	Обмежені	Підтримуються
Аналітика та звітність	Розвинені Agile-звіти	Базові можливості	Звіти щодо стану робіт і строків
Гнучкість налаштування	Висока	Обмежена	Середня
Складність освоєння	Висока	Низька	Середня
Основна цільова аудиторія	Команди розроблення програмного забезпечення	Невеликі команди та прості проєкти	Бізнес-команди та міжфункціональні проєкти

Порівняльний аналіз розглянутих систем показує, що всі вони забезпечують базові можливості для координації командної роботи, однак відрізняються рівнем складності, гнучкістю налаштувань і цільовою аудиторією. Jira найбільше орієнтована на команди розробників програмного забезпечення та підтримує розвинені процеси Agile-управління. Trello є простішим інструментом для візуального керування завданнями, а Asana забезпечує зручне планування роботи та підтримує декілька способів представлення проєкту.

Отже, аналіз існуючих рішень показує, що сучасні системи координації командної роботи мають широкий набір функцій для планування, контролю та аналізу виконання завдань. Найбільш функціонально наближеною до потреб команд розробників є Jira, оскільки вона підтримує завдання, спринти, беклог, Kanban- і Scrum-дошки, звітність та налаштування робочих процесів. Водночас значна кількість функцій і складність налаштування можуть бути надмірними для невеликих команд або навчального використання. Тому доцільним є розроблення власного веб-застосунку, який реалізує ключові можливості подібних систем, зокрема створення проєктів, керування завданнями, Kanban-дошку, ролі користувачів, коментарі, спринти та базову аналітику, але має простішу структуру й орієнтований на конкретні потреби користувачів.

1.3. Обґрунтування доцільності розроблення веб-застосунку TaskFlow

Проведений аналіз сучасних систем управління проєктами показав, що наявні програмні рішення пропонують широкий набір засобів для планування, розподілу та контролю виконання завдань. Платформи Jira, Trello та Asana реалізують різні підходи до організації командної роботи, починаючи від простого візуального представлення завдань на дошці й завершуючи підтримкою складних робочих процесів, спринтів, залежностей і аналітичних звітів. Водночас жодна з розглянутих систем не забезпечує оптимального співвідношення між функціональністю, простотою використання та можливістю адаптації до потреб невеликих команд розробників.

Jira містить розвинені інструменти для управління процесами створення програмного забезпечення, однак її використання передбачає налаштування значної кількості параметрів, статусів, типів завдань і правил переходу між етапами роботи. Для великих організацій така гнучкість є перевагою, проте для невеликих команд, навчальних проєктів або короткотривалих розробок надмірна функціональна насиченість може ускладнювати початок роботи із системою. Trello, навпаки, вирізняється простотою та наочністю, але має обмежені можливості для планування спринтів, формування ієрархії завдань, розмежування ролей і детального аналізу результатів роботи. Asana забезпечує зручне планування та підтримує декілька способів відображення проєктної інформації, однак меншою мірою орієнтована на специфічні процеси розроблення програмного забезпечення.

Виявлені особливості розглянутих платформ свідчать про доцільність створення власного веб-застосунку, який поєднуватиме найбільш корисні можливості сучасних систем управління проєктами та водночас матиме простішу структуру. Веб-застосунок TaskFlow повинен забезпечувати централізоване зберігання даних про робочі процеси, учасників команди та заплановані роботи, наочне відображення їх поточного стану, а також зручну

взаємодію між користувачами. Функціональність системи доцільно зосередити на засобах планування, розподілу відповідальності та контролю виконання робіт без перевантаження інтерфейсу другорядними можливостями й надмірною кількістю налаштувань.

Отже, розроблення TaskFlow є доцільним з огляду на потребу невеликих команд у доступному та зрозумілому середовищі для координації спільної роботи. Результати проведеного аналізу дають змогу перейти до визначення потреб користувачів, формулювання вимог і постановки задачі проєктування системи.

1.4. Висновки по розділу

У першому розділі виконано аналіз предметної області систем управління проєктами та визначено їхню роль у плануванні, координації й контролі командної роботи. Розглянуто основні принципи застосування методологій Scrum і Kanban у процесі розроблення програмного забезпечення.

Виконано порівняльний аналіз платформ Jira, Trello та Asana за критеріями функціональності, підтримки Kanban-дошок і спринтів, наявності аналітики, гнучкості налаштування та складності освоєння. Результати порівняння наведено в таблиці 1.1.

Обґрунтовано доцільність розроблення веб-застосунку TaskFlow для невеликих команд. Визначено основні елементи удосконалення запропонованого рішення: дворівневе розмежування доступу, автоматичне ведення історії змін, формування стрічки активності, контроль станів спринтів і побудову аналітичних показників на основі актуальних даних проєкту.

РОЗДІЛ 2. АНАЛІЗ ВИМОГ ТА ПОСТАНОВКА ЗАДАЧІ

2.1. Аналіз потреб користувачів веб-застосунку TaskFlow

Веб-застосунок TaskFlow призначений для невеликих команд, які спільно працюють над програмними проєктами та потребують єдиного інформаційного середовища для планування, координації й контролю виконання робіт. У процесі командної розробки учасники постійно взаємодіють між собою, обмінюються інформацією, уточнюють вимоги, розподіляють відповідальність і контролюють дотримання визначених строків. За відсутності централізованої системи ці дії можуть виконуватися за допомогою різних засобів комунікації, електронних таблиць або текстових документів, що ускладнює пошук актуальної інформації та підвищує ризик її втрати або дублювання.

Веб-застосунок TaskFlow призначений для невеликих команд, які спільно працюють над програмними проєктами та потребують єдиного середовища для планування, координації й контролю виконання робіт. Використання централізованої системи дозволяє зберігати актуальну інформацію про проєкти, завдання, строки та відповідальних осіб в одному місці, що особливо важливо для команд із дистанційним форматом роботи.

Основними користувачами системи є адміністратор, власник або менеджер проєкту, розробник і гість. Їхні потреби відрізняються залежно від рівня відповідальності та участі в робочому процесі. Тому система повинна забезпечувати не лише доступ до необхідних функцій, а й розмежування повноважень відповідно до ролі користувача.

Адміністратор потребує засобів для керування обліковими записами, перегляду списку користувачів і призначення системних ролей. Власники та менеджери проєктів повинні мати можливість створювати проєкти, формувати склад команди, призначати виконавців, планувати строки та контролювати поточний стан робіт.

Для керівних ролей важливим є узагальнене представлення даних про проєкт. Аналітичний модуль повинен відображати розподіл завдань за статусами, пріоритетами, типами й виконавцями, а також прогрес активного спринту. Це дозволяє своєчасно виявляти затримки та нерівномірне навантаження між учасниками.

Розробникам необхідний швидкий доступ до призначених завдань, їхніх описів, строків, пріоритетів і додаткових матеріалів. Для наочного відображення робочого процесу доцільно використовувати Kanban-дошку, де завдання представлені у вигляді карток і розподілені між колонками відповідно до поточного статусу.

Важливою потребою учасників команди є можливість взаємодії в межах завдання. Коментарі дозволяють обговорювати деталі виконання, а файлові вкладення — зберігати пов’язані документи та матеріали. Збереження історії змін і стрічки активності забезпечує прозорість робочого процесу та дозволяє відстежувати дії користувачів.

Гості повинні мати переважно доступ до перегляду інформації без можливості внесення змін. Розмежування прав доступу зменшує ризик несанкціонованого редагування даних і дозволяє чітко визначити відповідальність кожного учасника.

Отже, користувачі TaskFlow потребують зрозумілого та централізованого середовища для планування робіт, командної взаємодії, контролю строків і перегляду стану проєкту. Визначені потреби є основою для формування функціональних і нефункціональних вимог до системи.

2.2. Функціональні та нефункціональні вимоги до системи

Формування вимог є одним із основних етапів проєктування програмного забезпечення, оскільки дозволяє визначити призначення системи, перелік операцій, доступних користувачам, обмеження доступу та якісні характеристики програмного продукту. Вимоги створюють основу для

проєктування архітектури, структури бази даних, інтерфейсу користувача та подальшого тестування системи.

Веб-застосунок TaskFlow призначений для організації командної роботи над проєктами. Система забезпечує централізоване зберігання інформації про користувачів, проєкти, учасників, завдання, спринти, коментарі та історію змін. Основними завданнями застосунку є планування робіт, розподіл відповідальності між учасниками команди, відстеження стану виконання завдань і надання узагальненої інформації про перебіг проєкту.

До користувачів TaskFlow належать адміністратор системи, керівник проєкту, розробник і гість. Доступні операції залежать як від системної ролі користувача, так і від його ролі в конкретному проєкті. Такий підхід дозволяє відокремити загальні адміністративні повноваження від прав, які користувач отримує в межах окремої команди.

Функціональні вимоги визначають можливості системи та операції, які можуть виконувати її користувачі.

Вимоги до автентифікації та керування обліковим записом**.** TaskFlow має забезпечувати реєстрацію нових користувачів, вхід до системи та завершення користувацького сеансу. Для входу користувач повинен вказати електронну пошту та пароль. Після успішної автентифікації система має формувати токен доступу, який використовується для звернення до захищених ресурсів.

Авторизований користувач повинен мати можливість переглядати й редагувати дані власного профілю, зокрема ім’я, адресу електронної пошти та зображення профілю. Також необхідно передбачити зміну пароля з обов’язковим введенням поточного пароля та перевіркою мінімальної довжини нового значення. Система повинна підтримувати такі системні ролі: адміністратор, менеджер проєктів, розробник та гість.

Адміністратор повинен мати можливість переглядати список користувачів, створювати нові облікові записи, редагувати дані користувачів і змінювати їхні системні ролі.

Вимоги до керування проєктами. Авторизований користувач із відповідними повноваженнями повинен мати можливість створити проєкт, указавши його назву, унікальний ключ і, за необхідності, опис. Користувач, який створив проєкт, автоматично отримує роль його власника. Система повинна забезпечувати перегляд доступних користувачу проєктів, відображення детальної інформації про вибраний проєкт, редагування його назви та опису, видалення проєкту з бази даних, а також перегляд кількості пов’язаних завдань та учасників.

Власник або менеджер проєкту повинен мати можливість переглядати склад команди, знаходити зареєстрованих у системі користувачів, додавати їх до проєкту та призначати відповідні проєктні ролі.

Також необхідно передбачити видалення учасника зі складу команди. Після вилучення користувач не повинен мати доступу до даних проєкту, якщо він не володіє іншими правами, що дозволяють такий доступ.

Керування завданнями. Центральним функціональним елементом TaskFlow є створення завдання. Користувачі з відповідними правами повинні мати можливість створювати, переглядати, редагувати та видаляти завдання в межах проєкту. Кожне завдання має обов'язково містити його назву, опис, тип, поточний статус, пріоритет, автора, відповідального виконавця, термін виконання, оцінку трудомісткості, а також належність до конкретного проєкту та спринту.

У системі повинні підтримуватися типи «завдання/Task», «помилка/Bug», «користувацька історія/User Story» та «епік/Epic». Для визначення важливості роботи мають використовуватися низький, середній, високий і критичний рівні пріоритету.Життєвий цикл завдання є послідовним і повинен містити статуси: заплановано (To Do), у роботі (In Progress), перевірка коду (Code Review), тестування (Testing) та завершено (Done). Для зручності навігації система повинна дозволяти фільтрувати завдання за їхнім статусом, пріоритетом, типом, виконавцем і спринтом, а також підтримувати швидкий пошук за назвою або унікальним ключем завдання

Kanban-дошка необхідна для візуального відображення робочого процесу TaskFlow. Завдання на ній повинні бути представлені у вигляді карток, розміщених у колонках відповідно до поточного статусу.

Користувач повинен мати можливість переміщувати картку між колонками методом перетягування. Після такого переміщення система повинна автоматично оновити статус завдання в базі даних і відобразити зміну в інтерфейсі.

Вимоги до керування спринтами. TaskFlow має підтримувати планування робіт у межах спринтів. Функціонал системи має дозволяти додавати конкретні завдання до сформованого спринту, запускати його в роботу та завершувати активний спринт після закінчення встановлених термінів. Окрім цього, необхідно забезпечити можливість зручного перегляду запланованих, активних та вже завершених спринтів.

Коментарі та командна взаємодія. Учасники проєкту повинні мати можливість переглядати обговорення завдання та додавати власні коментарі. Для кожного коментаря система має зберігати його текст, автора й дату створення. Користувач повинен мати можливість видаляти власні коментарі. Адміністратору системи може надаватися право видалення коментарів інших користувачів у випадках, передбачених внутрішніми правилами використання системи.

Історія змін. Веб-застосунок TaskFlow повинен автоматично фіксувати зміни основних властивостей завдання. До історії необхідно записувати зміну статусу, пріоритету, відповідального виконавця, терміну виконання та інших поточних параметрів.

Аналітика. Система повинна надавати узагальнену інформацію про стан виконання проєкту. Аналітичний модуль має містити:

- розподіл завдань за статусами;
- розподіл за пріоритетами;
- розподіл за типами;
- кількість завдань для кожного виконавця;

- кількість виконаних і прострочених завдань;
- прогрес активного спринту;
- відсоток завершення робіт у межах активної ітерації.

Аналітична інформація повинна подаватися у вигляді числових показників, таблиць і графічних діаграм, що дозволяють швидко оцінити стан проєкту.

Нефункціональні вимоги визначають якісні характеристики TaskFlow, зокрема безпеку, надійність, зручність використання, продуктивність і придатність до супроводу. Визначені характеристики узгоджуються з моделлю якості програмного продукту ISO/IEC 25010:2023, яка використовується для специфікації, вимірювання та оцінювання якості програмних систем [31].

Архітектурні вимоги. TaskFlow повинен бути реалізований відповідно до клієнт-серверної архітектури. Клієнтська частина має функціонувати як односторінковий веб-застосунок, а серверна частина повинна надавати REST API для роботи з даними.

Обмін між клієнтом і сервером має виконуватися через HTTP-запити у форматі JSON. Дані про користувачів, проєкти, завдання, спринти, коментарі та історію повинні зберігатися в реляційній базі даних PostgreSQL.

Вимоги до безпеки. Усі маршрути, крім реєстрації та входу, повинні бути захищені механізмом автентифікації. Кожен запит до приватного ресурсу має містити дійсний JWT-токен. Серед основних ризиків для HTTP API OWASP виділяє порушення авторизації на рівні об’єктів і недоліки механізмів автентифікації, тому право доступу необхідно перевіряти для кожного запиту до проєктного ресурсу [28].

Користувач не повинен отримувати доступ до чужих проєктів, завдань, спринтів, коментарів або аналітичних даних. Для некоректних ідентифікаторів, відсутніх ресурсів і заборонених операцій система має повертати відповідні коди помилок.

Паролі не повинні зберігатися у відкритому вигляді. Новий пароль має складатися щонайменше з восьми символів. Конфіденційні параметри, зокрема адреса підключення до бази даних і секретний ключ JWT, повинні зберігатися у змінних середовища та перевірятися під час запуску серверної частини. Для захисту паролів необхідно застосовувати спеціалізовані алгоритми хешування з налаштованим фактором обчислювальної складності [29].

Надійність і цілісність даних. Система повинна забезпечувати узгоджене зберігання даних і збереження зв'язків між користувачами, проєктами, учасниками, завданнями, спринтами та коментарями.

Після успішного створення або оновлення об'єкта зміни мають зберігатися в PostgreSQL і відображатися після повторного завантаження сторінки. У разі помилки система не повинна показувати користувачу хибне повідомлення про успішне виконання операції.

Вимоги до продуктивності. Типові операції, такі як авторизація, відкриття проєкту, завантаження дошки, створення завдання, зміна статусу та додавання коментаря, повинні виконуватися без тривалого блокування інтерфейсу.

Сумісність. Клієнтська частина повинна працювати у сучасному веббраузері без необхідності встановлення додаткового програмного забезпечення. Взаємодія із серверною частиною має здійснюватися через стандартизований REST API.

Frontend і backend повинні мати окремі конфігурації запуску та підтримувати одночасну роботу на різних мережевих портах.

Супроводжуваність. Серверна частина повинна бути розділена на маршрути, контролери, middleware, утиліти й окремий модуль взаємодії з базою даних. Такий поділ спрощує внесення змін і додавання нових API-операцій.

Функції взаємодії клієнтської частини з API повинні бути винесені в окремі сервіси, а моделі даних - описані за допомогою TypeScript-інтерфейсів.

Після зміни структури бази даних повинна бути передбачена можливість застосування міграцій і повторної генерації Prisma-клієнта.

Отже, сформовані функціональні та нефункціональні вимоги визначають основний склад можливостей TaskFlow і характеристики його роботи. Функціональні вимоги охоплюють керування користувачами, проєктами, учасниками, завданнями, спринтами, коментарями, історією змін і аналітикою. Нефункціональні вимоги регламентують архітектуру, безпеку, надійність, продуктивність, зручність використання та придатність системи до подальшого супроводу. На основі цих вимог здійснюється подальше проєктування архітектури, бази даних, API та користувацького інтерфейсу.

2.3. Постановка задачі проєктування та моделювання сценаріїв взаємодії користувачів із системою

На основі проведеного аналізу потреб користувачів та сформульованих функціональних і нефункціональних вимог визначено задачу проєктування веб-застосунку TaskFlow. Необхідно створити програмну систему, призначену для організації командної роботи над проєктами, централізованого зберігання проєктної інформації, розподілу відповідальності між учасниками та контролю стану виконання робіт.

Результатом проєктування має бути клієнт-серверний веб-застосунок, доступний користувачам через веббраузер. Клієнтська частина повинна забезпечувати відображення даних і взаємодію користувача з функціональними модулями системи, а серверна - виконання бізнес-логіки, перевірку прав доступу, обробку запитів і взаємодію з базою даних. Обмін інформацією між компонентами системи має здійснюватися через REST API.

У межах поставленої задачі необхідно передбачити створення та ведення облікових записів користувачів, автентифікацію, зміну даних профілю й керування системними ролями. Для захисту проєктної інформації система має використовувати дворівневу модель доступу.

Основними акторами системи є адміністратор, власник або менеджер проєкту, розробник і гість. Адміністратор відповідає за керування обліковими записами та системними ролями. Власник і менеджер організовують роботу проєкту, формують склад команди, створюють завдання та керують спринтами. Розробник працює з призначеними завданнями, змінює їхній стан, додає коментарі й пов’язані матеріали. Гість отримує доступ переважно до перегляду інформації про перебіг робіт.

Проєктований застосунок повинен забезпечувати створення окремих проєктів із власною командою та набором завдань. Для кожного завдання необхідно зберігати назву, опис, тип, статус, пріоритет, автора, відповідального виконавця, строк виконання, оцінку трудомісткості та належність до проєкту або спринту. У системі мають використовуватися типи «завдання», «помилка», «користувацька історія» та «епік».

Для відображення життєвого циклу завдання необхідно реалізувати Kanban-дошку зі статусами «Заплановано», «У роботі», «Перевірка коду», «Тестування» та «Завершено». Зміна положення картки між колонками повинна супроводжуватися оновленням відповідного статусу та формуванням запису в історії змін.

Для підтримки ітераційного планування система має забезпечувати створення, запуск і завершення спринтів, додавання до них завдань та перегляд прогресу. У межах одного проєкту одночасно може бути активним лише один спринт, а завдання, не включені до поточного спринта, повинні залишатися доступними в беклозі.

Важливою складовою проєктованої системи є підтримка взаємодії учасників команди. Користувачі повинні мати можливість додавати коментарі й вкладення до завдань, переглядати пов’язані матеріали та історію внесених змін. Для керівних ролей необхідно передбачити аналітичний модуль, який відображатиме розподіл завдань за статусами, пріоритетами, типами й виконавцями, а також рівень виконання активного спринту.

Для формалізації взаємодії користувачів із TaskFlow використано UML-моделювання. Діаграма варіантів використання відображає основних акторів системи та доступні їм функції. Вона дозволяє наочно представити розмежування повноважень між адміністратором, менеджером проєкту, розробником і гостем (рис.2.1)

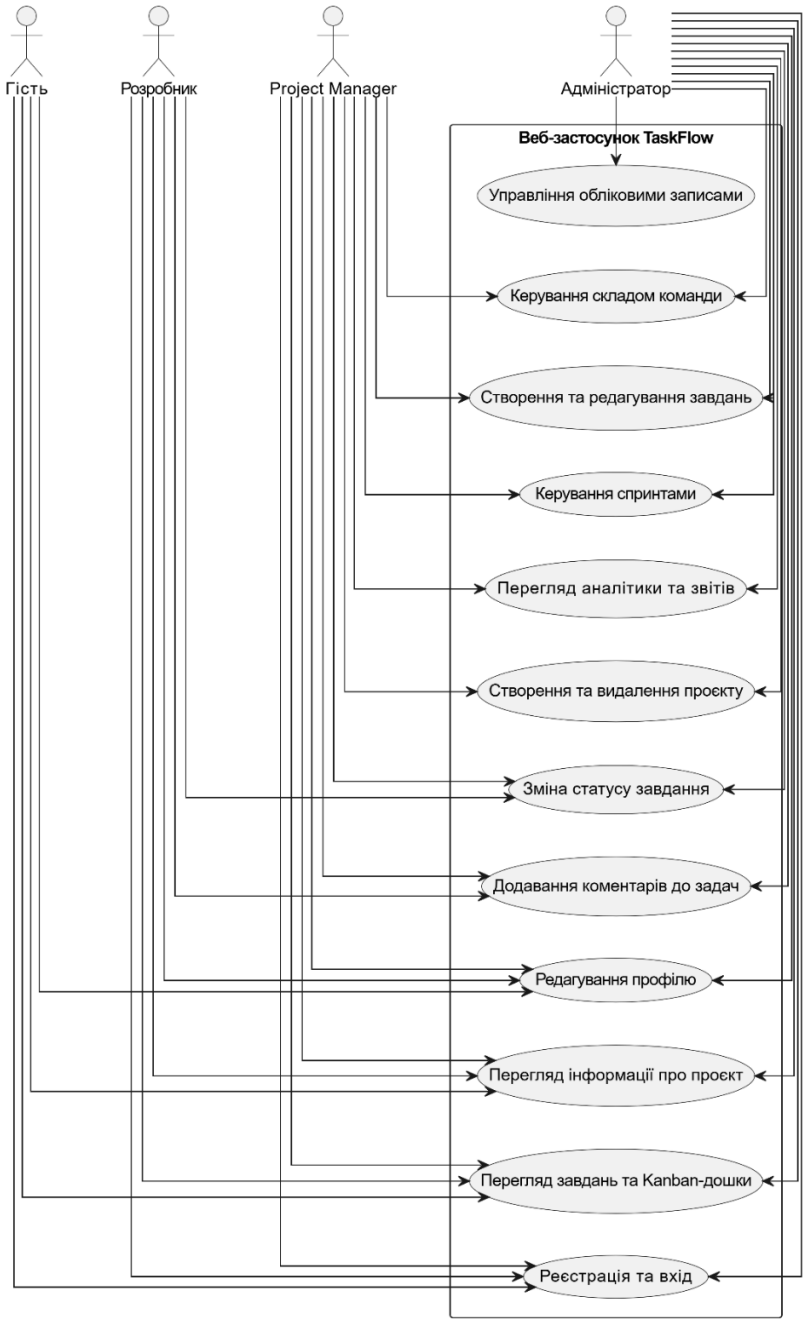


Рисунок 2.1 – Діаграма варіантів використання веб-застосунку для управління проєктами

Діаграма демонструє, що функції керування користувачами та системними ролями належать адміністратору. Власник або менеджер проєкту виконує операції з проєктами, учасниками, завданнями та спринтами. Розробник взаємодіє із завданнями, Kanban-дошкою, коментарями, вкладеннями та історією змін, тоді як гість має обмежений доступ до перегляду стану проєкту.

Окрім визначення доступних функцій, необхідно змоделювати послідовність виконання основного робочого процесу. Центральним об’єктом TaskFlow є завдання, яке проходить декілька етапів від моменту створення до завершення. Для відображення цього процесу побудовано діаграму активності.

На рисунку 2.2 наведено діаграму активності процесу роботи із завданнями.

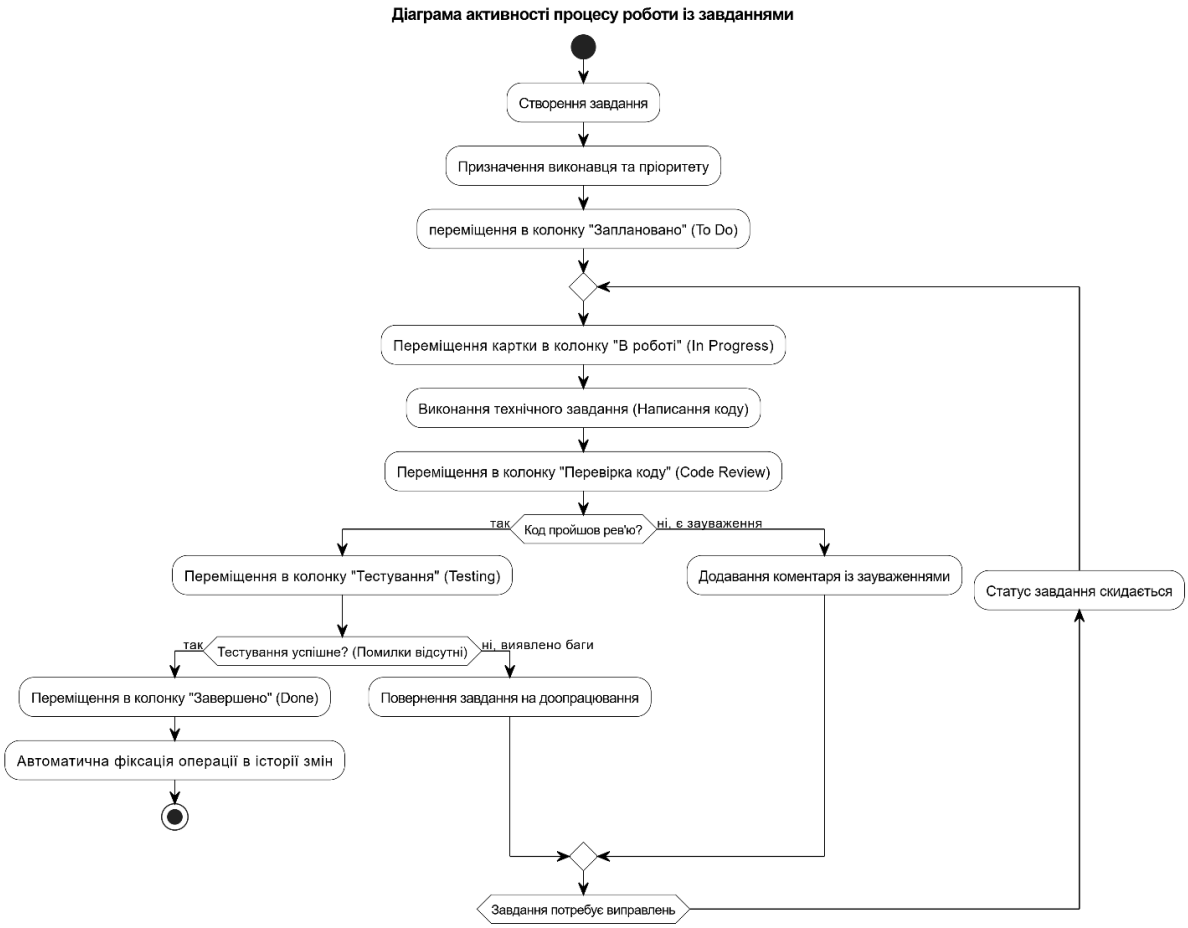


Рисунок 2.2 – Діаграма активності процесу роботи із завданнями

Відповідно до наведеної діаграми, менеджер створює завдання, визначає його пріоритет і призначає відповідального виконавця. Після збереження завдання розміщується в колонці «Заплановано». На початку виконання розробник переводить його до стану «У роботі», після завершення реалізації — до стану «Перевірка коду», а потім — «Тестування». Якщо під час перевірки або тестування виявлено недоліки, завдання повертається на доопрацювання. Після успішної перевірки результату воно переходить до стану «Завершено». Кожна зміна статусу повинна зберігатися в базі даних і фіксуватися в історії завдання.

Отже, задача проєктування полягає у створенні клієнт-серверного веб-застосунку TaskFlow, який забезпечуватиме керування користувачами, проєктами, командами, завданнями та спринтами, підтримуватиме комунікацію між учасниками й надаватиме засоби контролю результатів роботи. Побудовані UML-діаграми формалізують основні способи взаємодії користувачів із системою та життєвий цикл завдання. Отримані результати є основою для подальшого проєктування архітектури, структури бази даних і програмних компонентів веб-застосунку.

2.4. Висновки по розділу

У другому розділі виконано аналіз потреб основних категорій користувачів веб-застосунку TaskFlow. Визначено функції адміністратора, власника або менеджера проєкту, розробника та гостя.

Сформовано функціональні вимоги до автентифікації, керування користувачами, проєктами, учасниками, завданнями, Kanban-дошкою, спринтами, коментарями, вкладеннями, історією змін та аналітичними показниками. Визначено нефункціональні вимоги до архітектури, безпеки, надійності, продуктивності, сумісності та супроводжуваності системи.

Отримані результати використано як основу для подальшого проєктування структури системи.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ ВЕБ-ЗАСТОСУНКУ

3.1. Обґрунтування вибору технологій та інструментів розробки

Вибір технологій для реалізації веб-застосунку TaskFlow здійснювався з урахуванням його функціональних і нефункціональних вимог. Основними критеріями були підтримка клієнт-серверної архітектури, можливість створення інтерактивного користувацького інтерфейсу, зручна реалізація REST API, надійне зберігання реляційних даних, підтримка статичної типізації, безпечної автентифікації та подальшого розширення системи. Також враховувалися доступність інструментів, якість документації та можливість локального розгортання програмного продукту.

Мова програмування TypeScript

Основною мовою програмування для клієнтської та серверної частин TaskFlow обрано TypeScript. Вона розширює можливості JavaScript, додаючи статичну перевірку типів, інтерфейси, перелічувані типи та інші засоби контролю структури даних. TypeScript виконує перевірку типів до запуску програмного коду, що дозволяє виявляти частину потенційних помилок на етапі розроблення [8].

Використання TypeScript дозволяє виявляти частину помилок ще на етапі компіляції. Це особливо важливо для системи управління проєктами, у якій використовуються взаємопов’язані моделі користувачів, проєктів, завдань, спринтів, коментарів і вкладень. Типізація допомагає контролювати правильність параметрів функцій, структуру відповідей API та відповідність значень установленим статусам і ролям.

Спільне використання TypeScript на frontend і backend також спрощує узгодження моделей даних між клієнтською та серверною частинами. Сумісність із JavaScript дає змогу використовувати наявну екосистему бібліотек і поступово посилювати типізацію програмного коду.

Серверна платформа Node.js та фреймворк Express

Серверну частину TaskFlow реалізовано на платформі Node.js. Вона використовує подієву модель і неблокувальне виконання операцій введення-виведення. Цикл подій Node.js дозволяє виконувати асинхронні операції, не блокуючи обробку інших запитів [9]. Такий підхід відповідає характеру веб-застосунку, більшість серверних операцій якого пов’язана з обробкою HTTP-запитів, зверненням до бази даних і роботою з файлами.

Важливою перевагою Node.js є можливість застосовувати TypeScript як на клієнтській, так і на серверній стороні. Це зменшує відмінності між середовищами розроблення та спрощує супровід проєкту.

Для створення REST API обрано фреймворк Express. Він надає засоби для визначення HTTP-маршрутів, підключення middleware, обробки запитів і формування відповідей сервера [10]. На його основі серверну частину TaskFlow поділено на маршрути, контролери, middleware автентифікації, утиліти перевірки доступу та модуль взаємодії з базою даних.

Express не нав’язує жорсткої структури застосунку, тому його можливостей достатньо для реалізації обраної архітектури. Альтернативою міг бути NestJS, який має більш формалізовану модульну структуру та механізм впровадження залежностей. Однак для поточного масштабу TaskFlow використання Express дозволило уникнути додаткового рівня абстракції.

Для опрацювання файлових вкладень використано middleware Multer. Він призначений для обробки запитів у форматі multipart/form-data, який застосовується під час завантаження файлів [11]. У TaskFlow Multer забезпечує приймання вкладень, обмеження їхнього розміру, перевірку допустимих MIME-типів і збереження у визначеному файловому сховищі.

ORM Prisma 7

Для взаємодії серверної частини з базою даних використано Prisma ORM. Центральним елементом її конфігурації є файл schema.prisma, у якому описано моделі даних, їхні поля, обмеження та зв’язки. Prisma Schema є основним засобом конфігурації ORM і містить модель даних застосунку [12].

На основі схеми Prisma генерує типізований клієнт для виконання операцій із базою даних. Це забезпечує автодоповнення назв моделей і полів, а також перевірку структури запитів засобами TypeScript. Такий підхід зменшує ризик використання неіснуючих полів або передавання даних невідповідного типу.

Prisma також підтримує механізм міграцій, який дає змогу послідовно змінювати структуру бази даних і зберігати історію її розвитку. Prisma Migrate синхронізує структуру бази даних зі схемою Prisma та створює історію SQL-міграцій [13]. Декларативний API спрощує формування запитів для отримання проєктів, завдань, учасників, спринтів і пов’язаних сутностей.

Порівняно з прямим використанням SQL Prisma зменшує обсяг допоміжного коду та краще узгоджується зі статичною типізацією TypeScript. Інші ORM, зокрема Sequelize або TypeORM, також могли бути використані, однак Prisma забезпечує зручну генерацію типізованого клієнта та централізований опис схеми даних.

Система керування базами даних PostgreSQL 16

Для зберігання даних обрано PostgreSQL 16. Предметна область TaskFlow має виражену реляційну структуру: користувачі беруть участь у проєктах, проєкти містять завдання та спринти, а завдання пов’язані з виконавцями, коментарями, вкладеннями й історією змін.

PostgreSQL підтримує первинні та зовнішні ключі, обмеження унікальності, перевірки й інші механізми контролю цілісності даних [14]. Це дозволяє забезпечити узгодженість інформації між пов’язаними сутностями. Система також має засоби фільтрації, групування й агрегування, необхідні для формування аналітичних показників проєкту.

Підтримка ACID-властивостей є важливою для операцій, під час яких необхідно узгоджено змінювати декілька пов’язаних записів. ACID-властивості транзакцій спрямовані на забезпечення коректності даних під час паралельного виконання операцій і виникнення помилок [15]. Крім

стандартних реляційних типів, PostgreSQL підтримує JSON та інші розширені типи даних, що залишає можливість подальшого розвитку структури системи.

Документоорієнтована база даних, наприклад MongoDB, також могла б використовуватися для зберігання частини інформації. Однак наявність значної кількості чітко визначених зв'язків між сутностями робить реляційну модель природнішою для TaskFlow.

Бібліотека React

Клієнтську частину веб-застосунку реалізовано на основі React. Бібліотека дозволяє створювати користувацький інтерфейс з окремих компонентів, які можуть поєднуватися у сторінки та функціональні екрани [16].

Для TaskFlow це важливо під час роботи з Kanban-дошкою, формами, модальними вікнами, списками завдань, спринтами та аналітичними показниками. Після створення завдання, зміни статусу або додавання коментаря інтерфейс може оновити відповідні дані без повторного завантаження всієї сторінки.

Компонентна модель React дає змогу представляти інтерфейс у вигляді окремих логічних елементів, наприклад картки завдання, картки спринту, модального вікна або панелі навігації. Це створює основу для повторного використання компонентів і подальшого поділу клієнтської частини на функціональні модулі.

Для керування локальним станом і виконання побічних операцій використовуються React Hooks, зокрема useState, useEffect та useRef. Hooks надають компонентам доступ до стану, посилань на елементи та інших можливостей React [17]. У TaskFlow вони застосовуються для збереження даних інтерфейсу, завантаження інформації з API та взаємодії з елементами сторінки.

Альтернативами React могли бути Angular або Vue.js. Angular надає більш комплексну структуру, але містить значно більше вбудованих механізмів, ніж потрібно для поточного масштабу системи. Vue.js також

підходить для створення односторінкових застосунків, однак React було обрано через наявний досвід використання та широку екосистему сумісних бібліотек.

Інструмент складання Vite

Для організації середовища розроблення та складання клієнтської частини використано Vite. Він забезпечує запуск локального сервера, підтримує TypeScript і React та має механізм гарячого оновлення модулів. Vite також надає команду складання, яка формує оптимізовані статичні ресурси для виробничого середовища [18].

Hot Module Replacement дозволяє застосовувати зміни в компонентах під час розроблення без повного перезавантаження сторінки [19]. Це прискорює створення та перевірку користувацького інтерфейсу.

Порівняно з ручним налаштуванням Webpack Vite потребує меншого обсягу початкової конфігурації. Create React App також міг використовуватися для створення React-застосунку, однак Vite є зручнішим для сучасного середовища розроблення та активніше підтримується.

HTTP-клієнт Axios

Для обміну даними між клієнтською та серверною частинами використано бібліотеку Axios. Вона забезпечує виконання HTTP-запитів, автоматичне опрацювання JSON-відповідей і єдиний механізм обробки помилок.

Важливою перевагою Axios для TaskFlow є підтримка перехоплювачів запитів і відповідей. Interceptors дозволяють виконувати додаткові дії до надсилання запиту або після отримання відповіді [20]. За допомогою interceptor JWT-токен автоматично додається до заголовка Authorization під час звернення до захищених API-маршрутів. Це дозволяє централізувати логіку автентифікації та не дублювати її в кожному компоненті.

API-функції для роботи з користувачами, проєктами, завданнями, спринтами, коментарями й аналітикою винесено в окремі сервіси. Такий підхід відокремлює мережеву взаємодію від логіки відображення інтерфейсу.

Автентифікація на основі JWT

Для автентифікації користувачів використано JSON Web Token. JWT є компактним і придатним для передавання через HTTP способом представлення набору тверджень між сторонами взаємодії [21]. Після успішного входу сервер формує токен, що містить ідентифікатор користувача та його системну роль. Токен передається клієнтом у заголовку кожного запиту до захищеного ресурсу.

JWT дозволяє серверу перевірити справжність токена без зберігання окремої серверної сесії. Водночас для доступу до конкретного проєкту сервер додатково перевіряє належність користувача до його команди та відповідну проєктну роль у базі даних. Під час реалізації JWT необхідно перевіряти алгоритм підпису, строк дії та інші критичні параметри токена відповідно до рекомендацій із безпечного використання цього формату [22].

Термін дії токена становить сім днів. У поточній реалізації він зберігається в localStorage, а його автоматичне додавання до запитів виконує Axios interceptor.

Використання JWT добре узгоджується з REST API та дає змогу централізовано захищати серверні маршрути за допомогою middleware автентифікації.

Засоби тестування Jest і Supertest

Для автоматизованої перевірки серверної частини використано Jest і Supertest. Jest забезпечує організацію та запуск тестів, а також перевірку очікуваних результатів [23].

Supertest є бібліотекою для тестування HTTP-серверів Node.js за допомогою програмного інтерфейсу формування запитів і перевірки відповідей [24]. Він дозволяє надсилати HTTP-запити безпосередньо до Express-застосунку без запуску окремого сервера на мережевому порту.

Для цього конфігурацію Express винесено в окремий модуль app.ts, а запуск прослуховування порту виконується у файлі index.ts. Інтеграційні тести охоплюють автентифікацію, перевірку прав доступу, операції із завданнями,

спринтами, коментарями та учасниками проєктів. Такий підхід дозволяє перевірити спільну роботу маршрутів, контролерів, middleware та бази даних.

Контейнеризація за допомогою Docker

PostgreSQL розгорнуто в Docker-контейнері. Контейнер є ізольованим процесом із власною файловою системою, який створюється на основі визначеного образу [25]. Контейнеризація дозволяє використовувати однакову версію PostgreSQL у різних середовищах і зменшує залежність проєкту від конфігурації операційної системи.

Конфігурація контейнера визначає мережевий порт, параметри підключення й постійний том для збереження даних. Завдяки цьому розробнику не потрібно встановлювати PostgreSQL безпосередньо в операційну систему.

Docker також спрощує повторне створення середовища, перенесення проєкту на інший комп’ютер і скидання тестового стану бази даних.

Отже обраний технологічний стек відповідає функціональним і нефункціональним вимогам веб-застосунку TaskFlow. TypeScript забезпечує типізацію програмного коду, React і Vite використовуються для створення інтерактивної клієнтської частини, а Node.js та Express — для реалізації серверної логіки й REST API. PostgreSQL у поєднанні з Prisma забезпечує зберігання та опрацювання взаємопов’язаних даних.

Використання Axios і JWT забезпечує взаємодію клієнта із захищеними серверними маршрутами, Jest і Supertest дозволяють виконувати інтеграційне тестування API, а Docker спрощує розгортання бази даних. Сукупність обраних технологій забезпечує достатню продуктивність, безпеку, супроводжуваність і можливість подальшого розвитку програмного продукту.

3.2. Архітектура системи та організація зберігання даних

Архітектура веб-застосунку. Структура TaskFlow базується на трирівневій клієнт-серверній архітектурі, яка забезпечує чітке розділення

відповідальності між компонентами системи. Архітектурна модель застосунку включає клієнтський рівень, призначений для формування користувацького інтерфейсу та обробки локальних подій, серверний рівень, який реалізує ключову бізнес-логіку й обробку запитів, а також рівень зберігання даних, відповідальний за надійну ізоляцію та структурування. Така декомпозиція дозволяє підвищити масштабованість системи, спрощує процес тестування окремих модулів та оптимізує навантаження на обчислювальні ресурси.

Клієнтська й серверна частини функціонують як окремі застосунки. Клієнтську частину реалізовано на основі React і TypeScript у вигляді односторінкового веб-застосунку. Перехід між функціональними екранами, оновлення списків і зміна стану Kanban-дошки виконуються без повного перезавантаження вебсторінки.

Для передавання запитів до серверної частини використовується бібліотека Axios. Функції взаємодії з API винесено в окремі модулі, що відокремлює мережеві операції від компонентів користувацького інтерфейсу. Під час звернення до захищених ресурсів перехоплювач Axios автоматично додає JWT-токен до заголовка Authorization.

Клієнтська частина не має прямого доступу до бази даних. Кожна операція проходить через серверний API.

Послідовність опрацювання запиту:

- Користувач виконує дію в інтерфейсі TaskFlow.
- React-компонент викликає відповідну функцію API-сервісу.
- Axios формує HTTP-запит і додає JWT-токен.
- Express визначає необхідний маршрут.
- Middleware автентифікації перевіряє токен.
- Контролер перевіряє повноваження користувача.
- Prisma виконує операцію з PostgreSQL.
- Сервер повертає відповідь клієнтській частині.
- Інтерфейс оновлює відображення даних.

Рівень зберігання даних реалізовано на основі реляційної СУБД PostgreSQL. У базі даних зберігається інформація про користувачів, проєкти, учасників команд, завдання, спринти, коментарі, файлові вкладення та історію змін. Реляційна модель відповідає структурі предметної області, оскільки основні сутності TaskFlow мають чітко визначені зв’язки між собою.

Для взаємодії серверної частини з PostgreSQL використовується Prisma ORM. Структура бази даних описується у файлі schema.prisma, де визначено моделі, типи полів, зв’язки, унікальні обмеження та правила видалення пов’язаних записів. На основі цієї схеми Prisma автоматично генерує типізований клієнт, який використовується контролерами для виконання операцій із даними.

Для зміни структури бази даних застосовується механізм міграцій Prisma. Він дозволяє послідовно створювати таблиці, додавати нові поля й обмеження та синхронізувати структуру PostgreSQL із моделями застосунку.

Файлові вкладення зберігаються окремо від основних реляційних даних. У PostgreSQL записується службова інформація про файл, зокрема його назва, тип, шлях зберігання, автор і зв’язок із завданням. Доступ до вкладень здійснюється через захищений серверний маршрут після перевірки JWT-токена та прав користувача в межах відповідного проєкту. Під час організації файлового сховища враховано рекомендації OWASP щодо перевірки допустимих типів і розміру файлів, обмеження доступу до операцій завантаження та зберігання вкладень поза загальнодоступним вебкаталогом [34].

Таким чином, архітектура TaskFlow забезпечує розділення користувацького інтерфейсу, серверної бізнес-логіки та рівня зберігання даних. Взаємодія між компонентами здійснюється через REST API, а всі операції з проєктними даними проходять перевірку на серверному рівні. Обрана структура спрощує супровід системи, забезпечує контроль доступу та створює основу для подальшого розширення функціональності веб-застосунку.

3.3. Проєктування REST API та моделі доступу

Організація серверної частини. Серверний код веб-застосунку TaskFlow декомпоновано на окремі ізольовані модулі відповідно до їхнього функціонального призначення, що забезпечує високу супроводжуваність та гнучкість розширення системи. Головний файл app.ts відповідає за створення та базову конфігурацію Express-застосунку. У ньому здійснюється підключення глобальних middleware, налаштування політики спільного використання ресурсів різних джерел (CORS), а також реєстрація маршрутів усіх функціональних модулів. Окремий файл index.ts використовується як головна точка входу в програму, яка ініціалізує з'єднання з базою даних та запускає безперервне прослуховування визначеного мережевого порту.

Файл app.ts відповідає за створення й конфігурацію Express-застосунку, підключення CORS, middleware та маршрутів. Точка входу index.ts запускає сервер. Відокремлення конфігурації застосунку від його запуску також дає змогу використовувати Express-застосунок під час інтеграційного тестування без відкриття окремого мережевого порту.

Оскільки клієнтська та серверна частини TaskFlow можуть працювати на різних мережевих портах, для їхньої взаємодії необхідно налаштувати CORS. Цей механізм використовує HTTP-заголовки, за допомогою яких сервер визначає джерела, яким браузер дозволяє доступ до ресурсів [32].

Маршрути REST API розроблюваної системи згруповано відповідно до її основних функціональних ресурсів, що охоплюють модулі автентифікації користувачів, управління обліковими записами, адміністрування проєктів та координації учасників команд. Крім того, архітектура інтерфейсу програмування застосунку включає окремі ендпоінти для безпосередньої роботи із завданнями та Kanban-дошкою, планування спринтів, ведення текстових коментарів, завантаження вкладених файлів і генерації аналітичних даних.

Для отримання даних використовуються запити GET, для створення ресурсів — POST, для часткового оновлення — PATCH, а для видалення — DELETE. Сервер повертає стандартні HTTP-коди, зокрема 200 і 201 для успішних операцій, 400 для некоректних запитів, 401 за відсутності дійсного токена, 403 у разі недостатніх повноважень, 404 за відсутності ресурсу та 409 у разі конфлікту з бізнес-правилами системи. Семантику HTTP-методів і кодів відповідей визначено відповідно до стандарту RFC 9110, що забезпечує однозначне трактування результатів операцій клієнтською та серверною частинами [30].

Автентифікація реалізована за допомогою JWT. Після входу сервер формує токен за допомогою generateToken(userId, role), що містить ідентифікатор користувача та його системну роль. Middleware перевіряє підпис і строк дії токена, після чого додає відомості про користувача до об’єкта запиту. Middleware автентифікації у файлі auth.ts підключається до кожного маршруту через router.use(authenticate) і виконує такі кроки:

- Зчитує заголовок Authorization: Bearer <token>.
- Верифікує підпис токена за допомогою JWT_SECRET.
- Витягує userId і role із payload токена.
- Записує їх у req.userId і req.userRole для використання в контролерах.
- Якщо токен відсутній або недійсний — повертає 401 Unauthorized.

Для взаємодії з PostgreSQL використано Prisma ORM. Структура моделей, зв’язків і перелічуваних типів описана у файлі schema.prisma. На основі цієї схеми генерується типізований Prisma Client, який використовується контролерами для створення, отримання, оновлення та видалення записів.

Загальну структуру програмного рішення та взаємодію його компонентів наведено на рисунку 3.1.

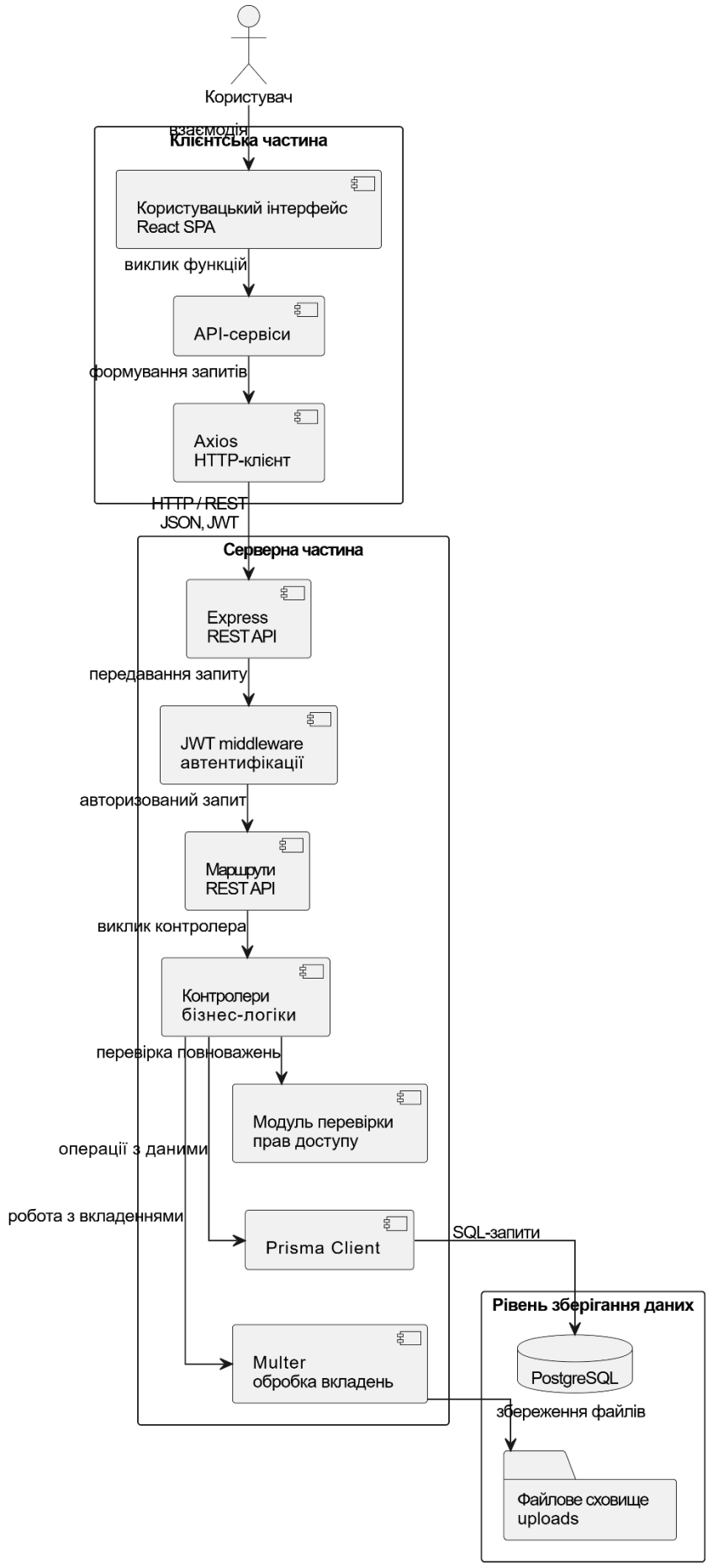


Рисунок 3.1 – Діаграма компонентів веб-застосунку TaskFlow

Діаграма відображає взаємодію між клієнтським застосунком React, Axios, сервером Express, middleware автентифікації, маршрутами, контролерами, Prisma, PostgreSQL та файловим сховищем. Клієнтська частина передає запити через REST API, після чого сервер виконує автентифікацію, перевірку доступу та необхідну бізнес-операцію. Постійні структуровані дані зберігаються в PostgreSQL, тоді як фізичні файли вкладень розміщуються в окремому каталозі серверної частини.

3.4. Проєктування реляційної бази даних

Для постійного зберігання інформації в TaskFlow використовується реляційна база даних PostgreSQL. Вибір реляційної моделі зумовлений наявністю чітких зв’язків між користувачами, проєктами, учасниками, завданнями, спринтами, коментарями, вкладеннями та історією змін.

Основними сутностями бази даних є: User,Project, ProjectMember, Task, Sprint, Comment, Attachment, TaskHistory.

Сутність User зберігає дані облікового запису користувача: ім’я, адресу електронної пошти, хеш пароля, системну роль і зображення профілю. Адреса електронної пошти є унікальною та використовується для входу до системи.

Сутність Project містить назву проєкту, унікальний ключ, опис і посилання на власника. Один користувач може створити декілька проєктів.

Зв’язок між користувачами та проєктами реалізовано через проміжну сутність ProjectMember. Вона містить посилання на користувача, проєкт і роль учасника. Такий підхід реалізує зв’язок багато-до-багатьох і дозволяє одному користувачу брати участь у декількох проєктах із різними повноваженнями.

Сутність Task є центральним елементом предметної області. Для завдання зберігаються назва, опис, тип, статус, пріоритет, автор, виконавець, термін виконання, оцінка трудомісткості та належність до проєкту. Завдання також може бути пов’язане зі спринтом або залишатися в беклозі, якщо значення спринту не визначене.

Для типізації завдань використовуються значення TASK, BUG, STORY та EPIC. Поточний стан роботи визначається статусами TODO, IN_PROGRESS, CODE_REVIEW, TESTING і DONE. Пріоритет завдання може бути низьким, середнім, високим або критичним.

Сутність Sprint належить до певного проєкту та містить назву, мету, дати початку й завершення та поточний статус. Спринт може перебувати в запланованому, активному або завершеному стані. Один проєкт може містити декілька спринтів, однак бізнес-логіка системи дозволяє одночасно мати лише один активний спринт.

Сутність Comment зберігає текст повідомлення та посилання на завдання й автора. Це забезпечує можливість ведення обговорень безпосередньо в контексті конкретної роботи.

Сутність Attachment містить метадані вкладення, зокрема початкову назву файла, шлях до його фізичного розташування, розмір, MIME-тип, автора та пов’язане завдання. Сам файл зберігається у файловій системі серверної частини, а в PostgreSQL розміщуються лише відомості, необхідні для його ідентифікації та контролю доступу.

Сутність TaskHistory використовується для ведення журналу змін завдання. Вона містить назву зміненого поля, попереднє та нове значення, користувача, який виконав операцію, і дату її здійснення. Завдяки цьому можна відстежити зміну статусу, пріоритету, виконавця та інших характеристик завдання.

Для забезпечення цілісності даних у реляційній моделі використовуються первинні та зовнішні ключі, унікальні обмеження й правила зв’язку між сутностями. Це дозволяє запобігти дублюванню критичних даних, зокрема електронних адрес користувачів і ключів проєктів, а також гарантує коректне збереження залежностей між проєктами, завданнями, спринтами, коментарями та історією змін. Така структура бази даних забезпечує узгоджене зберігання інформації та створює основу для стабільної роботи серверної частини веб-застосунку.

Структуру бази даних і зв’язки між сутностями наведено на рисунку 3.2.

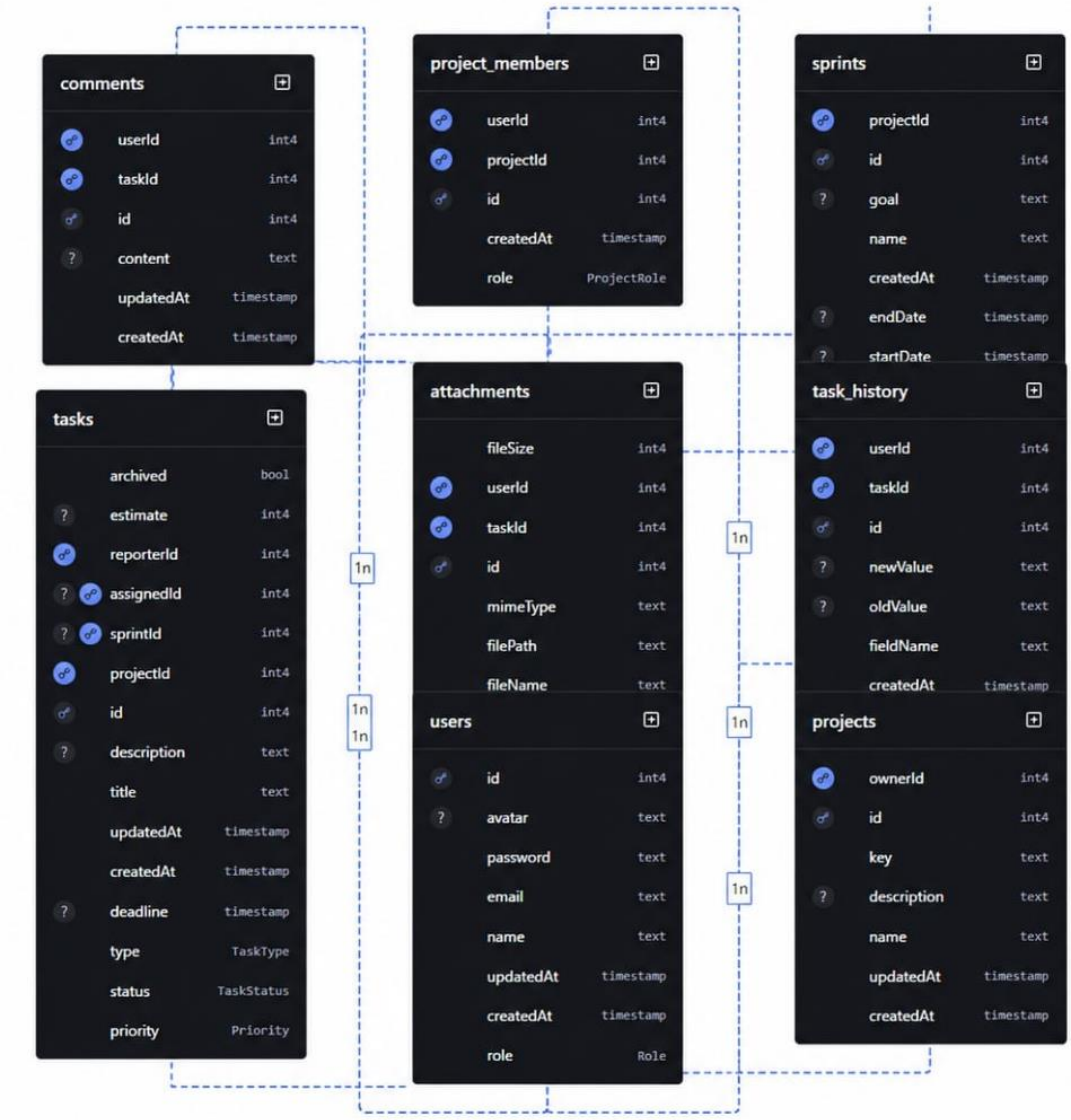


Рисунок 3.2 – ER-діаграма бази даних веб-застосунку TaskFlow

ER-діаграма відображає первинні та зовнішні ключі, обов’язкові й необов’язкові зв’язки, а також відношення між основними таблицями. Проміжна таблиця учасників реалізує зв’язок багато-до-багатьох між користувачами та проєктами. Інші залежності переважно мають тип один-до-багатьох.

Цілісність даних забезпечується первинними й зовнішніми ключами, унікальними обмеженнями та правилами каскадного видалення. Наприклад,

видалення проєкту спричиняє видалення пов’язаних учасників, спринтів і завдань. Під час видалення завдання також видаляються його коментарі, метадані вкладень і записи історії.

Для версіонування структури бази даних використовується Prisma Migrate. Після зміни файлу schema.prisma створюється нова міграція, яка містить SQL-команди для оновлення PostgreSQL. Збереження міграцій у проєкті дозволяє відстежувати розвиток схеми та відтворювати однакову структуру бази даних у різних середовищах.

Після застосування міграції виконується повторна генерація Prisma Client. Завдяки цьому типи, доступні в серверному коді, відповідають актуальній структурі бази даних.

Таким чином, архітектура TaskFlow забезпечує чіткий розподіл відповідальності між клієнтською частиною, серверною логікою та рівнем зберігання даних.

3.5. Висновки по розділу

У третьому розділі виконано обґрунтування вибору технологій та інструментів для розроблення TaskFlow. Для клієнтської частини обрано React, TypeScript, Vite та Axios, для серверної частини — Node.js і Express, а для зберігання та обробки даних — PostgreSQL і Prisma ORM.

Спроєктовано клієнт-серверну архітектуру веб-застосунку з поділом на клієнтський рівень, серверну бізнес-логіку, рівень доступу до даних і файлове сховище. Побудовано UML-діаграму компонентів, яка відображає взаємодію між React, Axios, Express, middleware автентифікації, контролерами, Prisma та PostgreSQL.

Спроєктовано реляційну модель даних, що охоплює користувачів, проєкти, учасників, завдання, спринти, коментарі, вкладення та історію змін. Визначено первинні й зовнішні ключі, типи зв’язків і правила забезпечення цілісності даних.

РОЗДІЛ 4. РЕАЛІЗАЦІЯ ВЕБ-ЗАСТОСУНКУ TASKFLOW

4.1. Реалізація серверної частини

Серверну частину TaskFlow реалізовано мовою TypeScript на платформі Node.js із використанням фреймворку Express. Конфігурацію застосунку відокремлено від запуску мережевого сервера: файл app.ts формує Express-застосунок, підключає middleware та маршрути, тоді як index.ts відповідає за запуск прослуховування порту. Така організація спрощує інтеграційне тестування та повторне використання об’єкта застосунку.

REST API поділено на маршрути автентифікації, користувачів, проєктів, учасників, завдань, спринтів, коментарів, вкладень і аналітики. Маршрути визначають HTTP-метод та адресу ресурсу, після чого передають запит до відповідного контролера. Контролери перевіряють параметри, застосовують бізнес-правила й формують стандартизовану HTTP-відповідь.

Захист приватних маршрутів забезпечено JWT-middleware. Після перевірки підпису й строку дії токена до об’єкта запиту додаються ідентифікатор користувача та його системна роль. Для доступу до проєктних ресурсів додатково перевіряються членство користувача в проєкті та його проєктна роль, що забезпечує дворівневе розмежування повноважень.

Окрему увагу в серверній частині приділено перевірці вхідних даних і контролю доступу до ресурсів. Перед виконанням операцій контролери перевіряють наявність обов’язкових параметрів, коректність ідентифікаторів, належність користувача до відповідного проєкту та допустимість виконання запитаної дії. Такий підхід дозволяє запобігти створенню некоректних записів у базі даних і забезпечує узгоджене виконання бізнес-правил у різних модулях системи.

Взаємодію з PostgreSQL організовано через спільний екземпляр Prisma Client. Запити до бази даних формуються на основі типізованих моделей, а зміни структури таблиць застосовуються за допомогою Prisma Migrate. Для

файлових вкладень використано Multer: фізичні файли зберігаються в окремому каталозі, тоді як їхні метадані записуються до бази даних.

Структура серверної частини побудована за модульним принципом. Окремі групи маршрутів відповідають конкретним функціональним областям системи, зокрема автентифікації, проєктам, завданням, спринтам, коментарям, вкладенням та аналітиці. Такий поділ спрощує навігацію в коді, дозволяє ізольовано змінювати окремі модулі та зменшує ризик впливу однієї частини системи на іншу під час подальшого розширення функціональності.

Бізнес-логіка серверної частини не обмежується простим збереженням і отриманням даних із бази. У контролерах реалізовано перевірку правил роботи із завданнями, спринтами та проєктними ролями. Зокрема, перед зміною завдання перевіряється доступ користувача до відповідного проєкту, під час запуску спринту контролюється відсутність іншого активного спринту, а під час роботи з вкладеннями перевіряється зв’язок файла із завданням і проєктом. Це дозволяє забезпечити коректність операцій незалежно від того, з якого екрана клієнтської частини було надіслано запит.

Для підвищення стабільності роботи серверної частини використано єдиний підхід до формування відповідей. Успішні операції повертають клієнтській частині структуровані дані у форматі JSON, а помилкові ситуації супроводжуються відповідним HTTP-кодом і повідомленням. Завдяки цьому клієнтський застосунок може однаково обробляти результати різних API-запитів, відображати повідомлення користувачу та оновлювати інтерфейс відповідно до фактичного результату операції.

Обробку помилок побудовано на стандартних HTTP-кодах. Сервер повертає коди 400, 401, 403, 404, 409 і 429 залежно від типу порушення, а текст помилки передає в структурованому JSON-об’єкті. Це дозволяє клієнтській частині однаково опрацьовувати помилки різних функціональних модулів.

4.2. Реалізація клієнтської частини

Клієнтську частину TaskFlow реалізовано як односторінковий веб-застосунок на основі React, TypeScript і Vite. Інтерфейс поділено на функціональні екрани та повторно використовувані компоненти, що відповідають за відображення проєктів, завдань, спринтів, учасників, аналітики, форм і модальних вікон.

Взаємодію з сервером винесено до окремих API-сервісів. Спільний екземпляр Axios містить базову адресу REST API та перехоплювач запитів, який автоматично додає JWT-токен до заголовка Authorization. Завдяки цьому компоненти інтерфейсу не дублюють мережеву конфігурацію та працюють із типізованими функціями отримання й зміни даних.

Структура клієнтської частини організована таким чином, щоб відокремити логіку відображення від логіки взаємодії з даними. Компоненти відповідають за побудову інтерфейсу та обробку дій користувача, тоді як API-сервіси виконують запити до серверної частини. Такий підхід спрощує повторне використання компонентів, зменшує дублювання коду та полегшує подальше розширення функціональності веб-застосунку.

Для підвищення зручності роботи користувача інтерфейс TaskFlow побудовано за принципом швидкого доступу до основних модулів системи. Користувач може переходити між проєктами, Kanban-дошкою, спринтами, учасниками команди та аналітикою без повного перезавантаження сторінки. Це забезпечує більш плавну взаємодію із застосунком і дозволяє оперативно переглядати актуальний стан проєкту.

Окрему увагу приділено відображенню стану виконання операцій. Під час завантаження даних або надсилання запитів користувач отримує візуальні індикатори, а після завершення операції система відображає відповідне повідомлення про успіх або помилку. Такий механізм зменшує невизначеність під час роботи із застосунком і дозволяє користувачу зрозуміти результат виконаної дії.

Клієнтська частина також враховує роль користувача під час формування інтерфейсу. Елементи керування, які призначені для створення, редагування або видалення даних, відображаються лише для користувачів із відповідними повноваженнями. Водночас така перевірка на рівні інтерфейсу виконує допоміжну роль, оскільки остаточне рішення щодо доступу до ресурсу приймається серверною частиною.

Для забезпечення узгодженості даних між різними екранами клієнтська частина оновлює відповідні списки та детальні представлення після виконання операцій створення, редагування або видалення. Наприклад, після зміни статусу завдання оновлюється не лише його картка на Kanban-дошці, а й пов’язані дані в деталях завдання, історії змін та аналітичних показниках. Це дозволяє уникнути відображення застарілої інформації та підтримує актуальний стан інтерфейсу без необхідності повторного входу до системи або ручного оновлення сторінки.

Стан інтерфейсу оновлюється після успішного виконання серверних операцій без повного перезавантаження сторінки. Для моделей, формат яких відрізняється на клієнтській і серверній сторонах, застосовано функції адаптації та таблиці відповідності enum-значень.

Форми створення й редагування містять перевірку обов’язкових полів та передають дані лише після успішної валідації. Для інформування користувача реалізовано короткі повідомлення, індикатори виконання запитів, порожні стани та модальні підтвердження операцій видалення. Доступність керівних елементів інтерфейсу залежить від системної та проєктної ролі користувача, однак остаточна перевірка повноважень виконується сервером.

4.3. Реалізація основних функціональних модулів веб-застосунку

Основні функціональні модулі TaskFlow реалізовано як сукупність клієнтських компонентів, API-сервісів, серверних маршрутів, контролерів і

моделей бази даних. Взаємодія між клієнтською та серверною частинами здійснюється через REST API. Для кожного модуля визначено окрему групу маршрутів, а доступ до даних виконується через Prisma Client.

Реалізація автентифікації та відновлення користувацького сеансу

Клієнтський модуль автентифікації реалізовано в компоненті `AuthScreen`, який містить форми входу та реєстрації.

Серверний маршрут передає запит до `auth.controller.ts`. Під час реєстрації контролер перевіряє наявність обов'язкових полів, шукає користувача з такою самою електронною поштою та хешує пароль за допомогою `bcrypt`. Після створення запису в таблиці `users` сервер формує JWT-токен.

Під час входу виконується пошук користувача за електронною поштою. Введений пароль порівнюється зі збереженим хешем за допомогою функції `bcrypt.compare()`. У разі успішної перевірки викликається функція `generateToken(userId, role)`, яка створює токен із терміном дії сім днів. Серверну реалізацію реєстрації та входу наведено в рисунку 4.1.

```
7   export const register = async (req: Request, res: Response) => {
8     const { name, email, password } = req.body;
9     if (!name || !email || !password) {
10      res.status(400).json({ error: 'Name, email and password are required' });
11      return;
12    }
13    const existing = await prisma.user.findUnique({ where: { email } });
14    if (existing) {
15      res.status(409).json({ error: 'Email already in use' });
16      return;
17    }
18    const hashed = await bcrypt.hash(password, 10);
19    const user = await prisma.user.create({
20      data: { name, email, password: hashed },
21      select: { id: true, name: true, email: true, role: true, avatar: true, createdAt: true },
22    });
23    res.status(201).json({ user, token: generateToken(user.id, user.role) });
24  };
25
26  export const login = async (req: Request, res: Response) => {
27    const { email, password } = req.body;
28    const user = await prisma.user.findUnique({ where: { email } });
29    if (!user || !(await bcrypt.compare(password, user.password))) {
30      res.status(401).json({ error: 'Invalid credentials' });
31      return;
32    }
33    const { password: _, ...safeUser } = user;
34    res.json({ user: safeUser, token: generateToken(user.id, user.role) });
35  };
```

Рисунок 4.1 – Контролер автентифікації: реєстрація та вхід (auth.controller.ts)

Формування підписаного JWT-токена винесено в окрему утиліту (рис. 4.2).

Рисунок 4.2 – Генерація JWT-токена (utils/token.ts)

Отриманий токен зберігається клієнтською частиною в `localStorage`. Під час повторного завантаження застосунку виконується запит: GET /api/auth/me

Middleware перевіряє токен, після чого сервер повертає дані поточного користувача. Завдяки цьому застосунок відновлює авторизований стан без повторного введення електронної пошти й пароля.

Для захисту маршрутів входу та реєстрації використовується `express-rate-limit`. Максимальна кількість запитів від одного клієнта становить двадцять протягом п’ятнадцяти хвилин. Після перевищення обмеження сервер повертає код `429 Too Many Requests`. Застосування express-rate-limit дозволяє обмежувати кількість повторних запитів до публічних маршрутів Express і зменшує кількість автоматизованих спроб підбору облікових даних [33].

Реалізація авторизації

Після перевірки JWT middleware записує ідентифікатор та системну роль користувача в об’єкт запиту. Ці дані використовуються контролерами під час перевірки доступу.

Перевірки проєктних повноважень винесено в модуль `src/utils/access.ts`. Функція `canAccessProject()` визначає, чи має користувач право переглядати ресурси проєкту. Для цього виконується пошук запису в

таблиці `project_members` або перевіряється наявність розширеної системної ролі.

Middleware автентифікації зчитує заголовок Authorization, перевіряє підпис токена й додає дані користувача до об'єкта запиту (рис. 4.3).

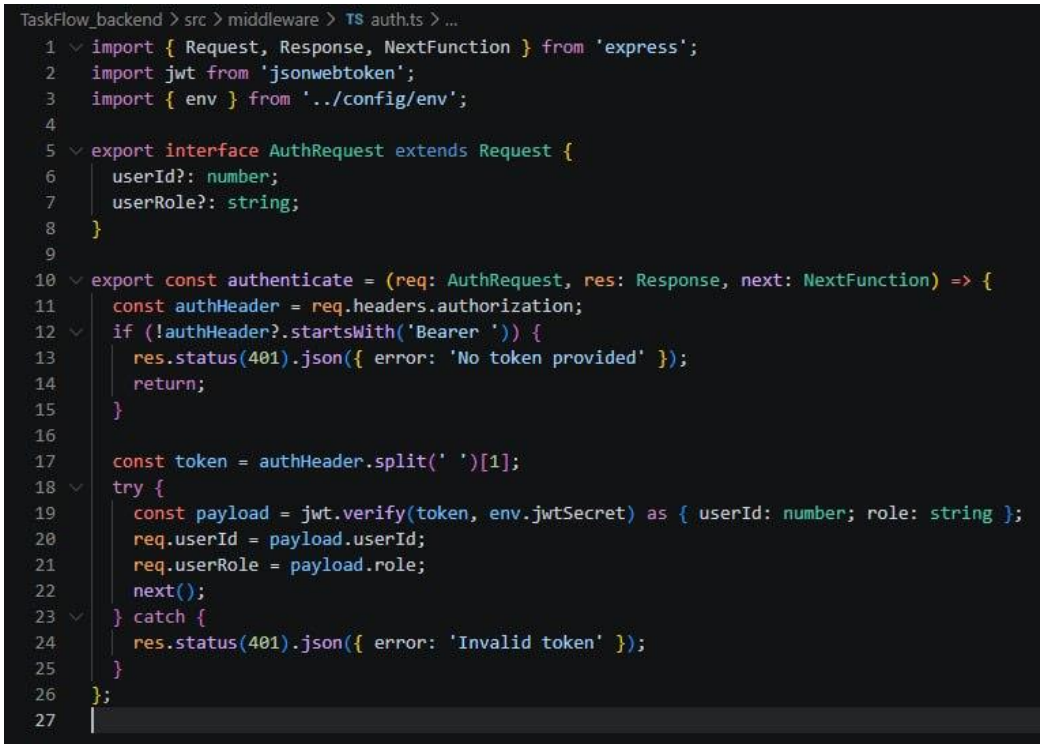


Рисунок 4.3 – Middleware перевірки JWT (middleware/auth.ts)

Функція `canManageProject()` використовується для операцій, що змінюють структуру проєкту. Вона враховує системні ролі `ADMIN` і `PM`, а також проєктні ролі `OWNER` і `MANAGER`.

На frontend додатково обчислюється локальна ознака доступу: canManage = systemRole ∈ {ADMIN, PM} або projectRole ∈ {OWNER, MANAGER}.

Вона використовується для приховування кнопок створення проєктів, завдань і спринтів. Однак клієнтська перевірка виконує лише інтерфейсну функцію. Остаточний контроль завжди здійснюється сервером.

Реалізація модуля проєктів

Під час створення проєкту frontend передає назву, унікальний ключ і опис. Контролер перевіряє заповнення обов'язкових полів та унікальність ключа. Після створення запису `Project` автор автоматично додається до таблиці `project_members` із роллю `OWNER`.

Отримання списку проєктів виконується з урахуванням поточного користувача. Для звичайного користувача Prisma-запит містить умову, за якою він має бути власником або учасником проєкту. Для адміністратора може повертатися розширений набір записів.

Під час видалення проєкту Prisma застосовує правила зв'язків, визначені в `schema.prisma`. Пов'язані учасники, спринти та завдання видаляються відповідно до налаштованих правил каскадного видалення.

Реалізація керування учасниками

Під час додавання учасника контролер перевіряє існування проєкту, користувача та повноваження ініціатора запиту. Після цього створюється запис `ProjectMember`, який містить `projectId`, `userId` і проєктну роль.

Frontend отримує список учасників разом із даними користувачів і використовує його у формах призначення виконавця, керування командою та фільтрації завдань.

Реалізація модуля завдань

Для роботи із завданнями створено такі маршрути:

Під час створення завдання клієнт передає об'єкт із назвою, описом, типом, пріоритетом, виконавцем, терміном виконання, оцінкою та ідентифікатором спринту. Сервер доповнює його ідентифікатором автора й проєкту.

Клієнтську форму створення задачі реалізує компонент CreateTaskModal: Компонент забезпечує введення назви, опису, типу, пріоритету, виконавця, строку виконання та інших параметрів завдання. Після підтвердження форми введені дані передаються до клієнтського API-сервісу, який надсилає запит на створення нового завдання до серверної частини. (рис. 4.4).

```
function CreateTaskModal({ projectId, presetStatus, onClose, onCreate }) {
  const p = project(projectId);
  const [title, setTitle] = useState('');
  const [desc, setDesc] = useState('');
  const [type, setType] = useState('task');
  const [pr, setPr] = useState('med');
  const [status, setStatus] = useState(presetStatus || 'todo');
  const [assignee, setAssignee] = useState(null);
  const [sprintId, setSprintId] = useState(null);
  const [due, setDue] = useState('');
  const availableSprints = SPRINTS.filter(s => s.projectId === projectId && s.status !== 'completed');
```

Рисунок 4.4 – Форма створення задачі CreateTaskModal

Зібрані дані потрапляють в обробник createTask, який перетворює їх у формат API, обгортає запит у стан завантаження та додає створену задачу до локального масиву.

Серверний контролер перевіряє повноваження через canManageProject, генерує людиночитний ключ задачі та створює запис у базі даних (рис. 4.5).

```
const createTask = async (data) => {
  try {
    await runBusy('Створення задачі...', async () => {
      const apiData = {
        title:       data.title,
        description: data.desc || '',
        type:        TYPE_TO_API[data.type]     || 'TASK',
        status:      STATUS_TO_API[data.status] || 'TODO',
        priority:    PRIORITY_TO_API[data.priority] || 'MEDIUM',
        assigneeId:  data.assignee || null,
        sprintId:    data.sprintId || null,
        deadline:    data.due || null,
      };
      const newTask = await tasksApi.create(projectId, apiData);
      const proj = PROJECTS.find(p => p.id === projectId);
      setTasks(ts => [...ts, adaptTask(newTask, proj?.key || 'T')]);
    });
  } catch (e) { notify(e?.response?.data?.error || 'Не вдалося створити задачу'); }
  setCreatingTask(null);
};
```

Рисунок 4.5 – Серверний контролер createTask (tasks.controller.ts)

Для частини полів допускається значення `null`. Наприклад, завдання може не мати виконавця, дедлайну або спринту. Якщо `sprintId` не задано, завдання належить до беклогу.

Отримання списку завдань підтримує параметри фільтрації: status priority, type, assigneeId, sprintId.

Контролер формує об’єкт `where` для Prisma лише з переданих параметрів. Завдяки цьому один маршрут використовується для завантаження всієї дошки, окремого спринту або відфільтрованої частини завдань.

Мапінг enum-значень між frontend і backend

Клієнтська та серверна частини використовують різні формати представлення статусів. На backend значення відповідають enum-типам Prisma: TODO, IN_PROGRESS, CODE_REVIEW, TESTING, DONE.

У клієнтському стані застосовуються скорочені значення: todo, prog, review, test, done.

Для перетворення між форматами використовуються таблиці відповідності `STATUS_TO_API`, `PRIORITY_TO_API`, `TYPE_TO_API` та функція `adaptTask()`.

Таблиці відповідності та функцію адаптації серверної моделі наведено в рисунку 4.6.

```
const STATUS_TO_API   = { todo:'TODO', prog:'IN_PROGRESS', review:'CODE_REVIEW', test:'TESTING', done:'DONE' };
const STATUS_FROM_API = { TODO:'todo', IN_PROGRESS:'prog', CODE_REVIEW:'review', TESTING:'test', DONE:'done' };
const PRIORITY_TO_API   = { low:'LOW', med:'MEDIUM', high:'HIGH', crit:'CRITICAL' };
const PRIORITY_FROM_API = { LOW:'low', MEDIUM:'med', HIGH:'high', CRITICAL:'crit' };
const TYPE_TO_API   = { task:'TASK', bug:'BUG', story:'STORY', epic:'EPIC' };
const TYPE_FROM_API = { TASK:'task', BUG:'bug', STORY:'story', EPIC:'epic' };

function adaptTask(t, projectKey) {
  return {
    id: t.id,
    key: `${projectKey}-${t.id}`,
    projectId: t.projectId,
    sprintId: t.sprintId || null,
    title: t.title,
    desc: t.description || '',
    type: TYPE_FROM_API[t.type] || 'task',
    status: STATUS_FROM_API[t.status] || 'todo',
    priority: PRIORITY_FROM_API[t.priority] || 'med',
    assignee: t.assigneeId || null,
    reporter: t.reporterId,
    due: t.deadline ? t.deadline.split('T')[0] : null,
    points: t.estimate || 3,
    created: t.createdAt?.split('T')[0],
    updated: t.updatedAt?.split('T')[0],
    assigneeObj: t.assignee || null,
    reporterObj: t.reporter || null,
    archived: t.archived || false,
  };
}
```

Рисунок 4.6 – Мапінг enum-значень і функція adaptTask (LegacyApp.tsx)

Під час отримання даних `adaptTask()` перетворює серверну модель у формат, зручний для відображення компонентами. Перед надсиланням змін виконується зворотне перетворення до enum-значень API.

Реалізація Kanban-дошки

Kanban-дошка реалізована компонентом `BoardScreen`. Після відкриття екрана виконується запит до маршруту завдань поточного проєкту. Отримані записи групуються за полем `status` і розміщуються у відповідних колонках.

Під час drag-and-drop клієнтська частина визначає цільову колонку та відповідне enum-значення статусу. Після цього надсилається запит: PATCH /api/tasks/:id/status.

Тіло запиту містить новий статус. Сервер перевіряє доступ, отримує поточне значення, оновлює запис `Task` і додає запис до `TaskHistory`.

Після успішної відповіді frontend оновлює локальний масив завдань. Якщо сервер повертає помилку, стан дошки не повинен остаточно змінюватися, а користувачу відображається повідомлення.

Реалізація історії змін

Під час оновлення завдання контролер порівнює нові значення з поточним записом у базі даних. Для кожного зміненого поля створюється окремий запис `TaskHistory`.

Запис містить: taskId, userId, fieldName, oldValue, newValue, дату створення.

Зміна самого завдання та створення записів історії виконуються в межах однієї серверної операції. Це забезпечує узгодженість між актуальним станом завдання та журналом змін.

Реалізація спринтів

Для роботи зі спринтами використовуються маршрути:

Спринт має статус «PLANNED», «ACTIVE» або «COMPLETED». Під час активації контролер виконує пошук іншого активного спринту в цьому самому проєкті: findFirst({ where: { projectId, status: 'ACTIVE' } }).

Якщо такий запис існує, сервер повертає «409 Conflict». Інакше статус спринту оновлюється.

Зв’язок між завданням і спринтом реалізовано через nullable-поле `sprintId`. Для повернення завдання до беклогу поле встановлюється в `null`.

Прогрес спринту визначається як відношення кількості завдань зі статусом `DONE` до загальної кількості завдань спринту.

Реалізація аналітики

Аналітичні дані формуються маршрутом: GET /api/projects/:id/analytics`.

Окремо отримується активний спринт. Для нього обчислюються: загальна кількість завдань, кількість завершених, відсоток виконання.

Клієнт отримує один об’єкт із підготовленими статистичними даними. Це дозволяє не виконувати агрегацію великого масиву завдань у браузері.

Отже, реалізація функціональних модулів TaskFlow базується на послідовній взаємодії клієнтських компонентів, типізованих API-сервісів, Express-маршрутів, контролерів і Prisma-моделей. Серверна частина централізовано виконує перевірку доступу, бізнес-правил і коректності даних, тоді як клієнтська частина відповідає за формування запитів, адаптацію отриманих моделей і синхронізацію стану інтерфейсу. Такий підхід забезпечує узгоджену роботу основних модулів і відокремлює інтерфейсну логіку від операцій збереження даних.

4.4. Висновки по розділу

У четвертому розділі виконано повну програмну реалізацію веб-застосунку TaskFlow відповідно до архітектурних рішень і вимог, сформованих у попередніх розділах.

Серверну частину реалізовано мовою TypeScript на платформі Node.js з використанням фреймворку Express. Конфігурацію застосунку відокремлено від точки запуску (app.ts / index.ts), що дозволяє використовувати Express-

об'єкт безпосередньо в інтеграційних тестах без відкриття мережевого порту. REST API структуровано за функціональними модулями: автентифікація, користувачі, проєкти, учасники, завдання, спринти, коментарі, вкладення та аналітика — кожен у власній групі маршрутів і контролерів.

Клієнтську частину побудовано на основі React, TypeScript і Vite як односторінковий застосунок. Взаємодію з API винесено до типізованих сервісів з перехоплювачем Axios, що автоматично додає JWT-токен до кожного захищеного запиту. Розбіжність між enum-значеннями серверної (TODO, IN_PROGRESS, CODE_REVIEW, TESTING, DONE) і клієнтської частин усунено через таблиці відповідності STATUS_TO_API та функцію adaptTask(), що забезпечує узгоджену передачу статусів без дублювання логіки в компонентах.

Реалізовано сім ключових функціональних модулів системи. Модуль автентифікації використовує bcrypt для хешування паролів і express-rate-limit для захисту від брутфорсу (20 запитів / 15 хвилин). Модуль завдань підтримує чотири типи (TASK, BUG, STORY, EPIC), чотири рівні пріоритету та п'ять статусів життєвого циклу, з серверною фільтрацією за будь-якою комбінацією параметрів в одному маршруті. Kanban-дошка після drag-and-drop оновлює статус через PATCH /api/tasks/:id/status і одночасно записує зміну до таблиці TaskHistory в межах однієї серверної операції. Модуль спринтів забезпечує серверний контроль бізнес-правила «один активний спринт» з поверненням 409 Conflict. Аналітика формується сервером у вигляді одного готового об'єкта зі зведеними показниками, що усуває потребу в агрегації даних на клієнті. Файлові вкладення обробляються Multer з обмеженням 10 МБ і перевіркою MIME-типу; метадані зберігаються в PostgreSQL, файли — у файловій системі.

Усі реалізовані модулі відповідають функціональним вимогам розділу 2. Відокремлення конфігурації від запуску, декомпозиція маршрутів і типізація через Prisma Client забезпечують супроводжуваність коду та готовність до подальшого розширення системи.

РОЗДІЛ 5. ТЕСТУВАННЯ ТА ДЕМОНСТРАЦІЯ РОБОТИ СИСТЕМИ

5.1. Опис та конфігурація тестового середовища

Тестування веб-застосунку TaskFlow проводилося в локальному середовищі, конфігурація якого відповідає архітектурі розробленої системи. Тестове середовище охоплює клієнтську частину, серверний застосунок, реляційну базу даних, файлове сховище та засоби автоматизованої перевірки REST API.

Основною метою налаштування тестового середовища було забезпечення відтворюваних умов для перевірки функціональних можливостей веб-застосунку, механізмів автентифікації й авторизації, правильності виконання операцій із базою даних, обробки помилок та взаємодії між клієнтською і серверною частинами.

TaskFlow складається з трьох основних програмних компонентів: клієнтського застосунку React, серверного застосунку Node.js та Express, а також бази даних PostgreSQL, розгорнутої в Docker-контейнері. Стан запущеного контейнера бази даних у середовищі Docker наведено на рисунку 5.1.

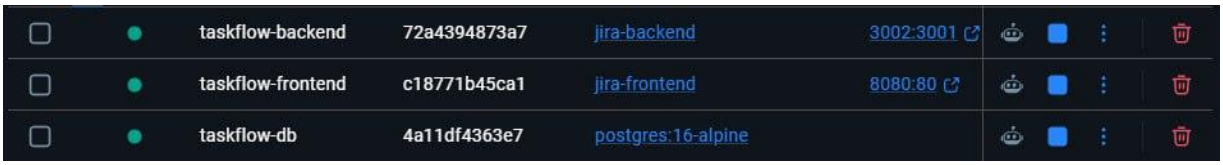


Рисунок 5.1 – Запущені Docker-контейнери компонентів веб-застосунку TaskFlow

Клієнтська й серверна частини запускаються як окремі процеси. Клієнтський застосунок працює через локальний сервер Vite та надсилає HTTP-запити до REST API. Серверна частина опрацьовує запити, перевіряє JWT-токени й повноваження користувачів, після чого виконує операції з PostgreSQL через Prisma ORM.

Основні складові тестового середовища наведено в таблиці 5.1.

Таблиця 5.1

Конфігурація тестового середовища TaskFlow

Компонент	**Використаний засіб**	**Призначення**
Операційна система	Windows/Ubuntu	Запуск клієнтської й серверної частин, Docker та засобів тестування
Клієнтська частина	React, TypeScript, Vite	Відображення інтерфейсу та взаємодія з REST API
Серверна частина	Node.js, Express, TypeScript	Реалізація бізнес-логіки та API
База даних	PostgreSQL 16	Зберігання користувачів, проєктів, завдань, спринтів та інших даних
Контейнеризація	Docker	Ізольований запуск
ORM	Prisma 7	Типізована взаємодія з базою даних і виконання міграцій
HTTP-клієнт	Axios	Передавання запитів між клієнтською і серверною частинами
Автентифікація	JWT	Захист приватних API-маршрутів
Тестовий фреймворк	Jest	Організація та запуск автоматизованих тестів
Тестування API	Supertest	Надсилання HTTP-запитів до Express-застосунку
Веббраузер	Google Chrome	Ручна перевірка користувацького інтерфейсу
Файлове сховище	Каталог uploads	Зберігання вкладень до завдань

Клієнтська частина запускається локально на порту 5173, серверна частина — на порту 3001, а PostgreSQL використовує стандартний порт 5432. Такий розподіл дозволяє одночасно запускати всі компоненти системи та перевіряти їхню взаємодію в умовах, наближених до реального використання.

Змінна DATABASE_URL містить параметри підключення до PostgreSQL. Значення JWT_SECRET використовується для формування та перевірки підпису токенів, а PORT визначає мережевий порт серверного застосунку. Під час запуску backend перевіряє наявність обов'язкових параметрів. Якщо один із них не заданий, застосунок завершує роботу з повідомленням про помилку.

PostgreSQL запускається в Docker-контейнері. Використання контейнеризації дозволяє ізолювати базу даних від основної операційної системи та забезпечити однакову версію СУБД під час розроблення і тестування. Для збереження інформації між перезапусками контейнера використовується постійний Docker-том.

Після запуску PostgreSQL виконується підготовка структури бази даних. Для цього застосовуються міграції Prisma, які створюють таблиці, зв'язки, зовнішні ключі, унікальні обмеження та перелічувані типи.

Після запуску Vite формує локальне середовище, у якому браузер завантажує React-застосунок. Клієнтська частина звертається до серверного API через Axios. JWT-токен автоматично додається до захищених запитів за допомогою перехоплювача Axios.

Для автоматизованої перевірки серверної частини використовуються Jest і Supertest. Конфігурацію Express винесено у файл app.ts, тоді як запуск прослуховування порту виконується у файлі index.ts. Завдяки цьому Supertest може надсилати HTTP-запити безпосередньо до Express-застосунку без запуску окремого мережевого сервера.

Запуск автоматизованих тестів виконується командою: npm test

Після виконання команди Jest запускає набір автоматизованих тестів і виводить у терміналі інформацію про кількість виконаних перевірок, успішні та невдалі результати, а також загальний час тестування.

Інтеграційні тести перевіряють спільну роботу маршрутів, middleware, контролерів, Prisma та PostgreSQL. На відміну від повністю ізольованих

модульних тестів, такі перевірки дозволяють виявити помилки в обміні даними між окремими компонентами серверної частини.

Для проведення перевірок у базі даних створюються тестові користувачі з різними системними та проєктними ролями: адміністратор, менеджер проєкту, розробник, гість, власник проєкту, учасник із роллю менеджера, учасник із роллю розробника та учасник із правами перегляду.

Використання різних ролей необхідне для перевірки не лише успішного виконання операцій, а й правильного блокування заборонених дій. Наприклад, тестове середовище повинно дозволяти перевірити, що гість може переглядати доступні дані, але не може змінювати структуру проєкту, тоді як власник або менеджер має право керувати учасниками, завданнями та спринтами.

До складу тестових даних входять проєкти, учасники, завдання з різними типами, статусами й пріоритетами, заплановані та активні спринти, коментарі, вкладення й записи історії. Це дозволяє відтворювати типові й граничні сценарії роботи системи.

Під час ручного тестування веб-застосунок відкривається у браузері Google Chrome. Перевіряються відображення форм входу та реєстрації, переходи між функціональними екранами, коректність завантаження даних, створення та редагування проєктів, робота Kanban-дошки, переміщення карток між колонками, фільтрація завдань, керування спринтами, додавання коментарів і вкладень, відображення аналітичних показників та повідомлення про результати операцій.

Окрема увага приділяється перевірці реакції інтерфейсу на помилки сервера. Для цього відтворюються ситуації з недійсним JWT-токеном, відсутністю доступу до проєкту, некоректними параметрами, повторним ключем проєкту та спробою запуску другого активного спринту.

Серверна частина в таких випадках повинна повертати відповідні HTTP-коди: 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 409 Conflict та 429 Too Many Requests. Клієнтський застосунок опрацьовує

отримані відповіді й відображає користувачу зрозуміле повідомлення. Для тривалих операцій використовується індикатор завантаження, а для видалення даних — модальне підтвердження.

Файлові вкладення перевіряються окремо. Тестування охоплює завантаження дозволеного файла, перевищення максимального розміру, спробу додавання непідтримуваного формату, захищене отримання вкладення та його видалення. Максимальний допустимий розмір файла становить 10 МБ.

Таким чином, сформоване тестове середовище дозволяє перевірити роботу клієнтської та серверної частин, взаємодію з реальною базою даних, механізми автентифікації, розмежування доступу й обробку файлових вкладень. Поєднання ручного тестування інтерфейсу та автоматизованих інтеграційних тестів забезпечує комплексну перевірку основних функціональних можливостей TaskFlow.

5.2. Автоматизоване інтеграційне тестування основних функцій системи

Для перевірки коректності та надійності серверної частини веб-застосунку TaskFlow розроблено набір автоматизованих інтеграційних тестів. Їхнє призначення полягає у перевірці спільної роботи REST API, middleware автентифікації, контролерів, механізмів розмежування доступу, Prisma ORM та бази даних PostgreSQL.

На відміну від модульного тестування, під час якого окремі функції або класи перевіряються ізольовано, інтеграційні тести TaskFlow охоплюють повний ланцюг опрацювання HTTP-запиту. Запит надходить до Express-застосунку, проходить через middleware, передається відповідному контролеру, після чого виконується операція з реальною базою даних. Такий підхід дозволяє оцінити не лише правильність окремих операцій, а й узгодженість взаємодії між компонентами серверної частини.

5.2.1. Засоби та методика тестування

Для створення та виконання автоматизованих тестів використано такі інструменти:

• Jest — середовище запуску тестів і перевірки очікуваних результатів;

• Supertest — бібліотека для формування HTTP-запитів безпосередньо до Express-застосунку;

• ts-jest — засіб виконання тестового коду, написаного мовою TypeScript;

• Prisma Client — використовується для підготовки тестових даних і додаткової перевірки стану PostgreSQL.

Конфігурацію Express-застосунку винесено у файл app.ts, який містить підключення middleware та маршрутів, але не запускає прослуховування мережевого порту. Запуск сервера виконується окремо у файлі index.ts. Завдяки такому поділу Supertest може імпортувати об’єкт застосунку та надсилати до нього HTTP-запити без запуску окремого мережевого сервера.

Тести організовано відповідно до підходу AAA:

1. Arrange — підготовка користувачів, проєктів, завдань, спринтів та інших початкових даних;

2. Act — виконання HTTP-запиту до відповідного маршруту;

3. Assert — перевірка коду відповіді, структури отриманих даних і, за потреби, стану записів у базі даних.

Для забезпечення повторюваності тестів у блоці beforeAll створюються необхідні користувачі та проєктні сутності. Реєстрація тестових користувачів виконується через маршрут POST /api/auth/register, після чого отримані JWT-токени використовуються для доступу до захищених ресурсів.

Електронні адреси та ключі проєктів формуються з використанням часової мітки Date.now(). Це запобігає конфліктам унікальності під час повторних або паралельних запусків тестів.

Після завершення перевірок у блоці afterAll створені тестові записи видаляються. Видалення проєкту також спричиняє каскадне очищення

пов’язаних завдань, коментарів, спринтів та історії змін. Такий підхід не допускає накопичення службових даних у PostgreSQL.

5.2.2. Структура набору інтеграційних тестів

У межах тестування реалізовано сім тестових наборів, які містять загалом 45 автоматизованих перевірок. Для систематизації їхній склад і призначення наведено в таблиці 5.2.

Таблиця 5.2

Структура набору інтеграційних тестів TaskFlow

Набір тестів	**Кількість перевірок**	**Основне призначення**
auth.test.ts	8	Реєстрація, вхід, перевірка облікових даних та отримання поточного користувача
access.test.ts	10	Автентифікація, розмежування доступу між проєктами та перевірка ідентифікаторів
tasks.test.ts	6	Створення, отримання й оновлення завдань, зміна статусу та формування історії
comments.test.ts	6	Створення, отримання та видалення коментарів із перевіркою прав доступу
members.test.ts	5	Перегляд, додавання та вилучення учасників проєкту
sprints.test.ts	5	Створення спринтів і перевірка правила одного активного спринту
activity.test.ts	5	Авторизація та перевірка коректності стрічки активності проєкту
Разом	**45**	

Як видно з таблиці 5.2, автоматизовані перевірки охоплюють основні серверні модулі TaskFlow: автентифікацію, авторизацію, проєктні ресурси, завдання, спринти, коментарі, учасників та стрічку активності.

Використання функцій beforeAll і afterAll відповідає передбаченому Jest підходу до одноразової підготовки тестового середовища перед виконанням набору перевірок та його очищення після завершення тестування [35].

5.2.3. Тестування автентифікації

Набір auth.test.ts перевіряє повний цикл роботи з обліковими записами користувачів.

Фрагмент реалізації інтеграційних тестів автентифікації у файлі auth.test.ts наведено на рисунку 5.2.

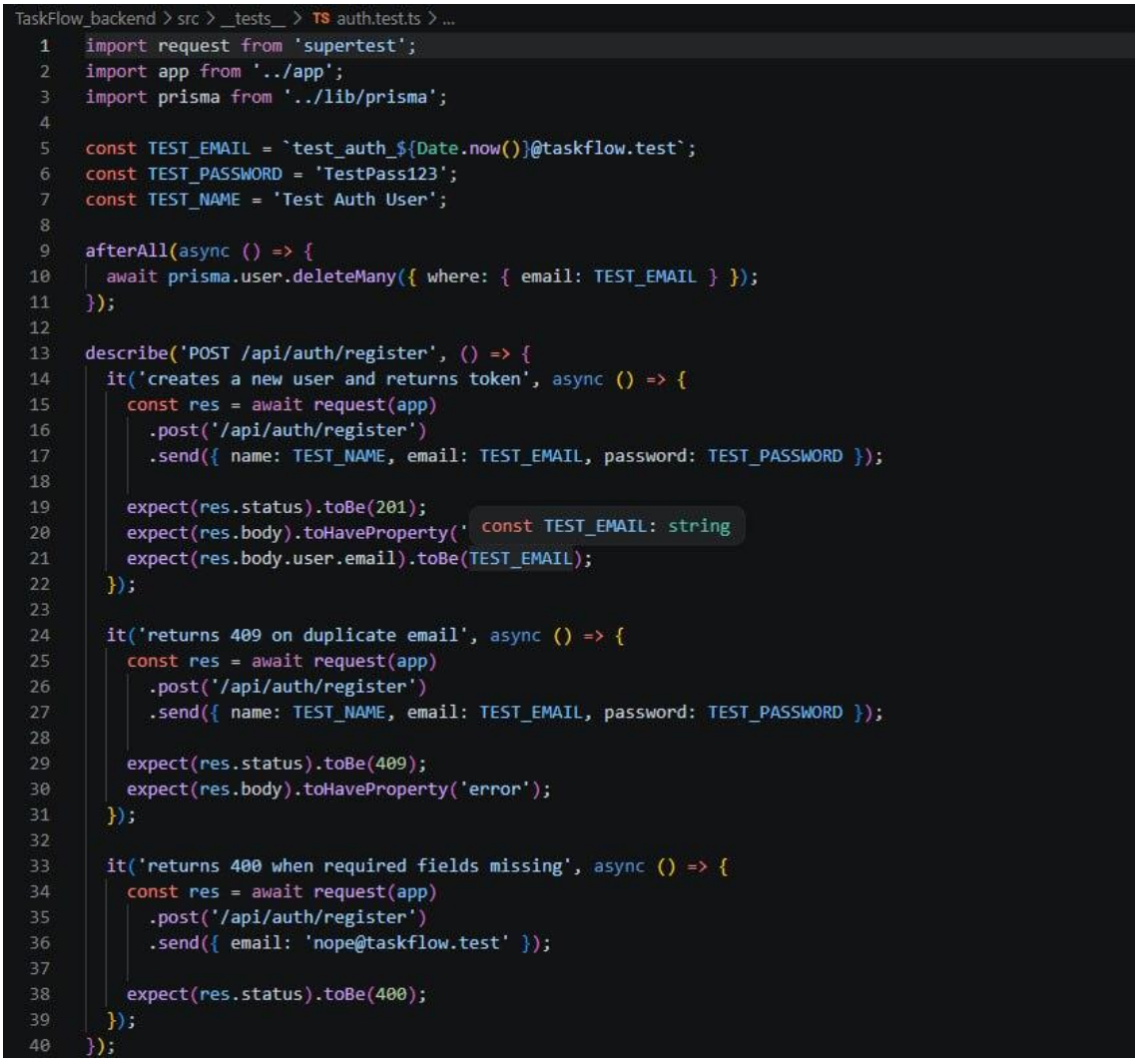


Рисунок 5.2 – Фрагмент реалізації інтеграційних тестів автентифікації у файлі auth.test.ts

За успішної реєстрації сервер повинен повернути код 201 Created, створити запис у таблиці users і сформувати JWT-токен. У разі повторного використання електронної пошти очікується код 409 Conflict, а за відсутності необхідних параметрів — 400 Bad Request.

Перевірка входу охоплює використання правильних і неправильних облікових даних. За коректної електронної пошти та пароля сервер повертає код 200 OK і JWT-токен. Якщо пароль є неправильним або користувача не існує, очікується код 401 Unauthorized.

5.2.4. Тестування роботи із завданнями

Набір tasks.test.ts перевіряє основні операції з центральною сутністю системи — завданням.

Тести охоплюють:

- створення завдання власником проєкту;
- заборону створення завдання користувачем, який не має доступу до проєкту;
- валідацію обов’язкових параметрів;
- отримання детальної інформації про завдання;
- зміну статусу;
- автоматичне формування запису в історії змін.

Після успішного створення завдання перевіряється код відповіді, структура отриманого об’єкта та наявність запису в PostgreSQL. Під час запиту детальної інформації сервер повинен повернути саме завдання та пов’язані з ним коментарі, вкладення й історію.

5.2.5. Тестування спринтів

Набір sprints.test.ts перевіряє створення й керування ітераціями, а також ключове бізнес-правило TaskFlow: у межах одного проєкту одночасно може бути активним лише один спринт.

Для перевірки цього правила виконуються такі дії:

1. створюються два заплановані спринти;
2. перший переводиться у стан ACTIVE;
3. виконується спроба активувати другий;
4. перевіряється повернення коду 409 Conflict;

5. перший спринт переводиться у стан COMPLETED;

6. повторно виконується активація другого спринту.

Фрагмент реалізації інтеграційних тестів керування спринтами у файлі sprints.test.ts наведено на рисунку 5.3.

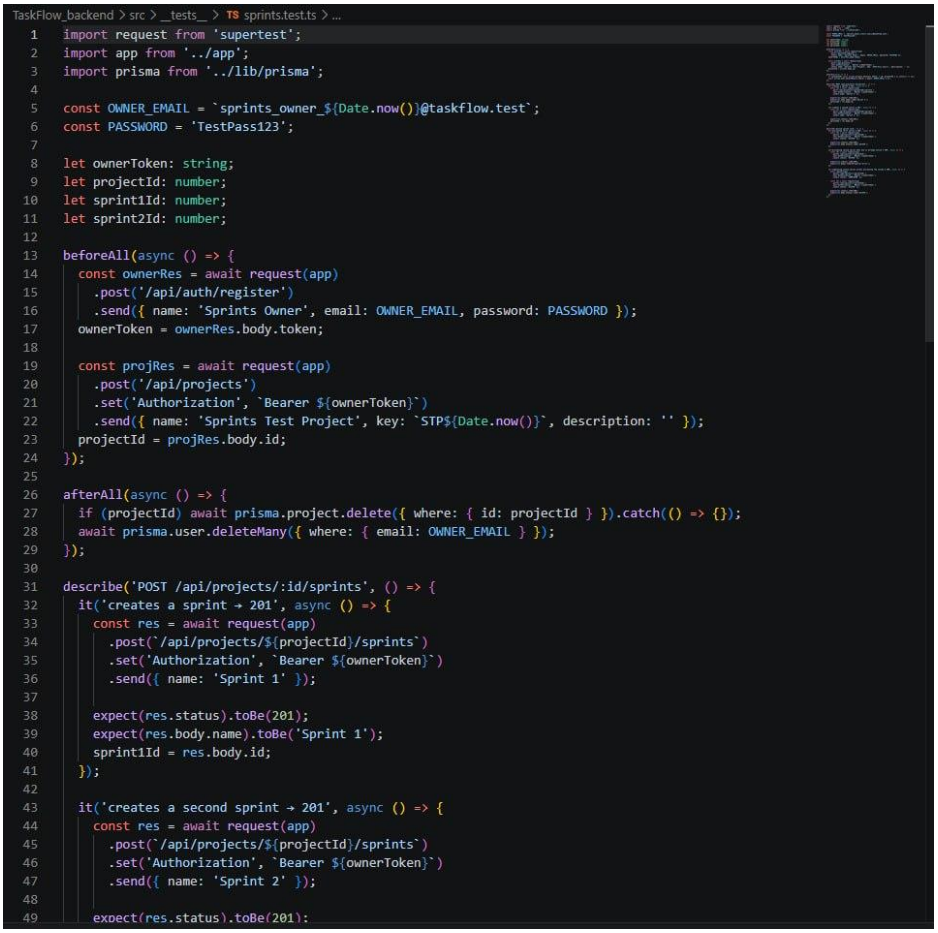


Рисунок 5.3 – Фрагмент реалізації інтеграційних тестів керування спринтами у файлі sprints.test.ts

Після завершення першого спринту другий повинен успішно перейти в активний стан. Таким чином підтверджується не лише блокування некоректної операції, а й правильність подальшої зміни стану системи.

5.2.6. Результати автоматизованого тестування

Повний набір інтеграційних тестів запускається командою:npm test

Результат запуску автоматизованих тестів серверної частини TaskFlow наведено на рисунку 5.4.

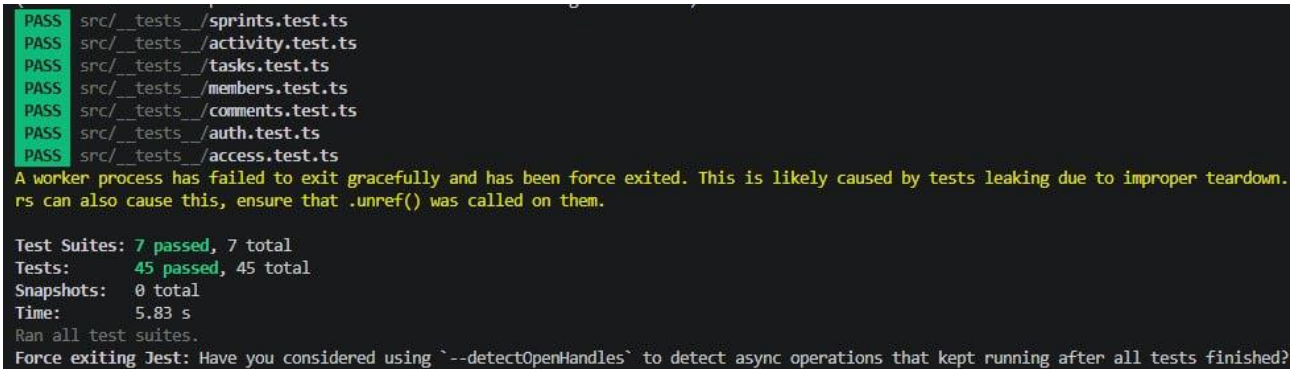


Рисунок 5.4 – Результати виконання автоматизованих тестів серверної частини TaskFlow.

Як видно з рисунка 5.4, автоматизовані тести завершилися успішно, що підтверджує коректність роботи перевірених маршрутів, механізмів автентифікації, розмежування доступу та операцій із даними.

Отримані результати підтверджують коректність роботи механізмів реєстрації та входу, перевірки JWT, розмежування доступу, керування завданнями, коментарями, учасниками та спринтами. Також підтверджено правильність ведення історії змін і формування стрічки активності.

Отже, автоматизоване інтеграційне тестування дає змогу швидко перевіряти серверну частину після внесення змін і своєчасно виявляти регресії, тоді як візуальне відображення та зручність інтерфейсу перевіряються окремо під час ручного функціонального тестування

5.3. Приклади сценаріїв використання та інтерфейс застосунку

Для перевірки роботи клієнтської частини TaskFlow проведено ручне функціональне тестування основних сценаріїв використання системи. На відміну від автоматизованого інтеграційного тестування, яке зосереджене на роботі REST API, ручна перевірка дає змогу оцінити коректність відображення даних, доступність елементів інтерфейсу, послідовність переходів між екранами та реакцію застосунку на дії користувача.

Після успішної автентифікації користувачу відкривається головне вікно TaskFlow, яке забезпечує доступ до основних функціональних модулів системи. У бічній панелі розміщено засоби переходу до головної сторінки, проєктів, Kanban-дошки, спринтів, учасників команди та аналітики. Головне вікно веб-застосунку наведено на рисунку 5.5.

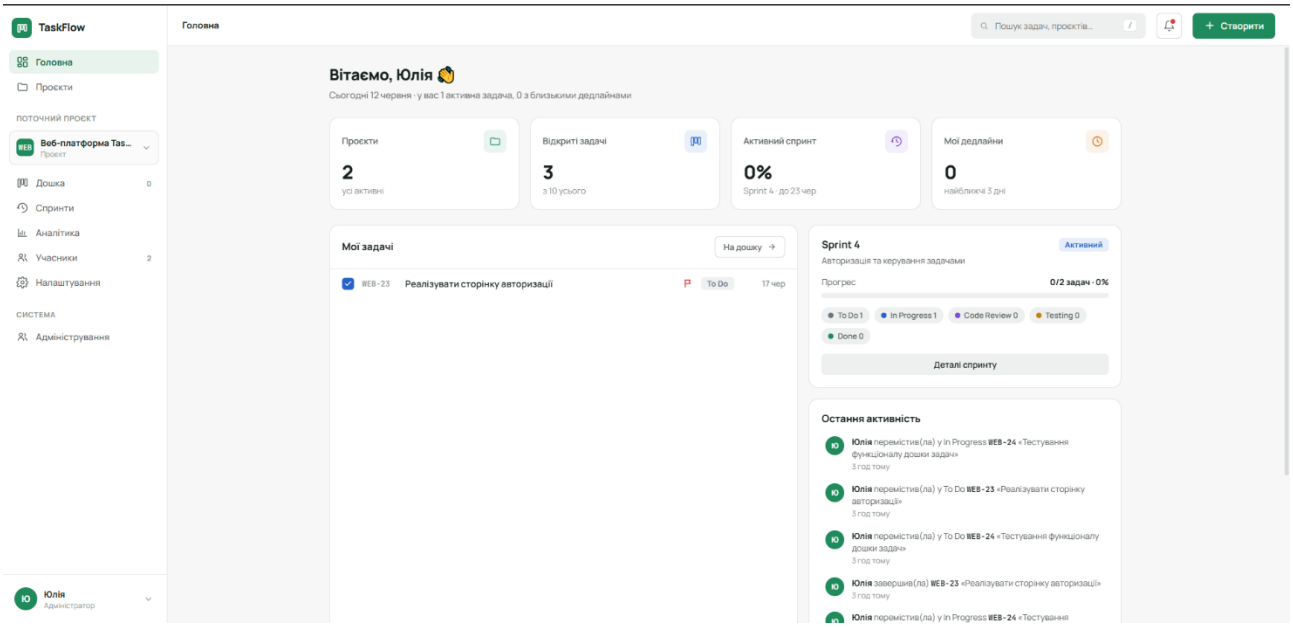


Рисунок 5.5 – Головне вікно веб-застосунку та інтерфейс керування завданнями

Як видно з рисунка 5.5, інтерфейс застосунку має єдину структуру навігації та забезпечує швидкий перехід між основними робочими екранами. Доступність окремих елементів керування залежить від системної та проєктної ролі авторизованого користувача.

Сценарій автентифікації користувача

Після відкриття веб-застосунку користувачу відображається екран автентифікації. Для входу необхідно ввести електронну пошту та пароль. Після натискання кнопки входу клієнтська частина надсилає запит до серверного маршруту автентифікації.

У разі успішної перевірки облікових даних сервер повертає JWT-токен і дані користувача. Токен зберігається в локальному сховищі браузера, після

чого відкривається основний інтерфейс TaskFlow.

Відповідний інтерфейс наведено на рисунку 5.6.

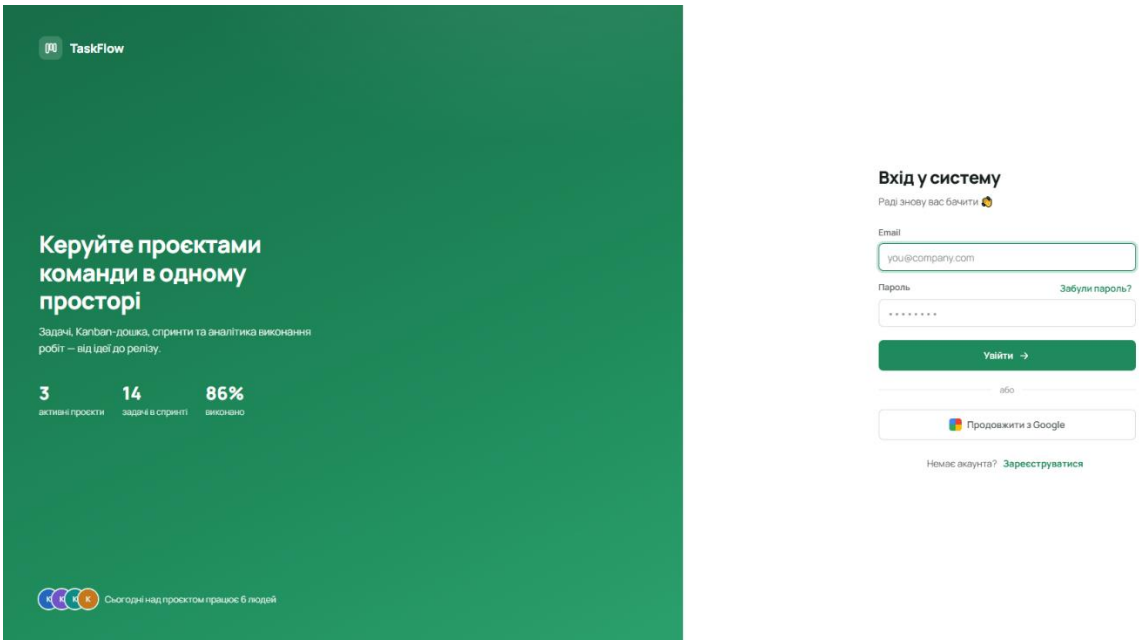


Рисунок 5.6 – Форма автентифікації користувача TaskFlow

Якщо користувач вводить неправильний пароль або неіснуючу електронну пошту, система не виконує вхід і відображає повідомлення про помилку. Таким чином перевіряється не лише успішний сценарій, а й реакція інтерфейсу на некоректні облікові дані.

Сценарій створення проєкту

Після входу користувач переходить до екрана проєктів. Для створення нового проєкту відкривається модальне вікно, у якому необхідно вказати назву, унікальний ключ і опис.

Після підтвердження клієнтська частина надсилає запит до серверного API. Сервер створює проєкт і автоматично додає його автора до складу команди з роллю власника. У разі введення ключа, який уже використовується іншим проєктом, сервер повертає повідомлення про конфлікт, а клієнтська частина відображає його користувачу.

Відповідний інтерфейс наведено на рисунку 5.7.

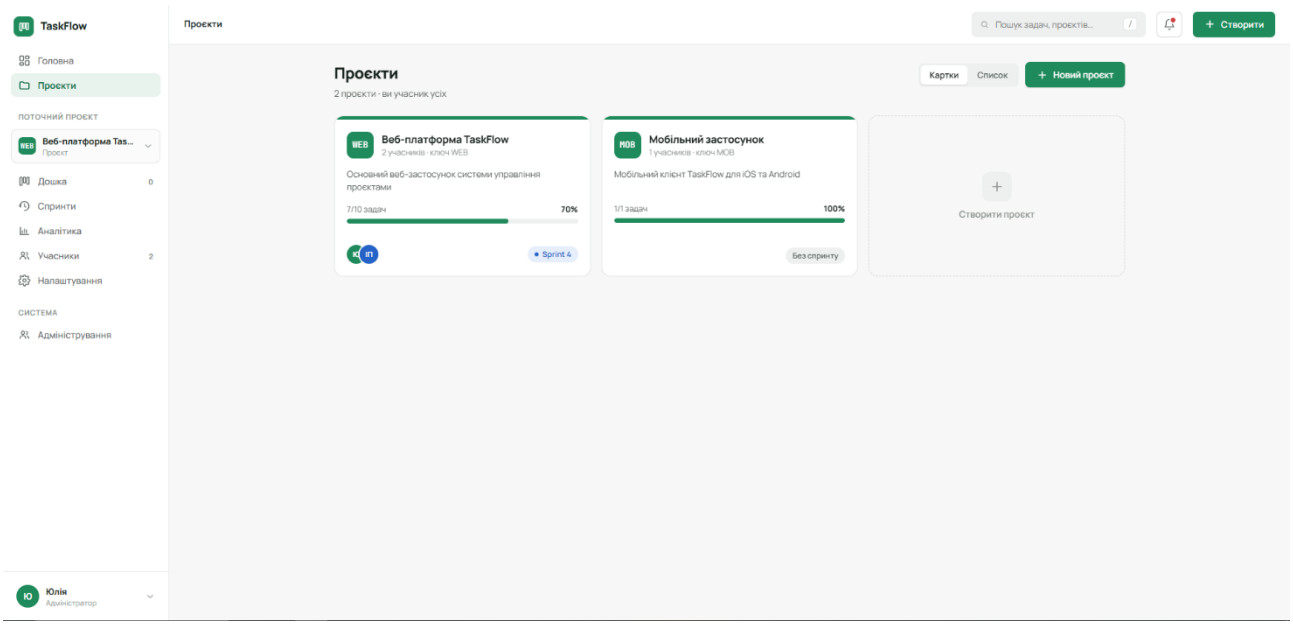


Рисунок 5.7 – Інтерфейс керування проєктами веб-застосунку TaskFlow

Сценарій формування команди

Власник або менеджер проєкту може перейти до екрана учасників і переглянути поточний склад команди. Для додавання нового учасника вибирається зареєстрований користувач і визначається його роль у межах проєкту.

Після виконання операції новий учасник відображається у списку разом зі своєю роллю. Користувачі без керівних повноважень не бачать або не можуть використовувати елементи додавання й видалення учасників.

Сценарій створення завдання

Для створення завдання користувач із відповідними повноваженнями відкриває форму й указує назву, опис, тип, пріоритет, виконавця, строк виконання, оцінку трудомісткості та спринт.

Після збереження сервер створює запис у PostgreSQL, а клієнтська частина додає завдання до відповідної колонки Kanban-дошки. Якщо спринт не визначено, завдання залишається в беклозі.

Сценарій роботи з Kanban-дошкою

Kanban-дошка є основним інтерфейсом контролю стану завдань. Вона

містить колонки «Заплановано», «У роботі», «Перевірка коду», «Тестування» та «Завершено».

Завдання відображаються у вигляді карток із назвою, типом, пріоритетом, виконавцем і строком виконання. Користувач може застосовувати фільтри за спринтом, типом, пріоритетом і виконавцем.

Відповідний інтерфейс наведено на рисунку 5.8.

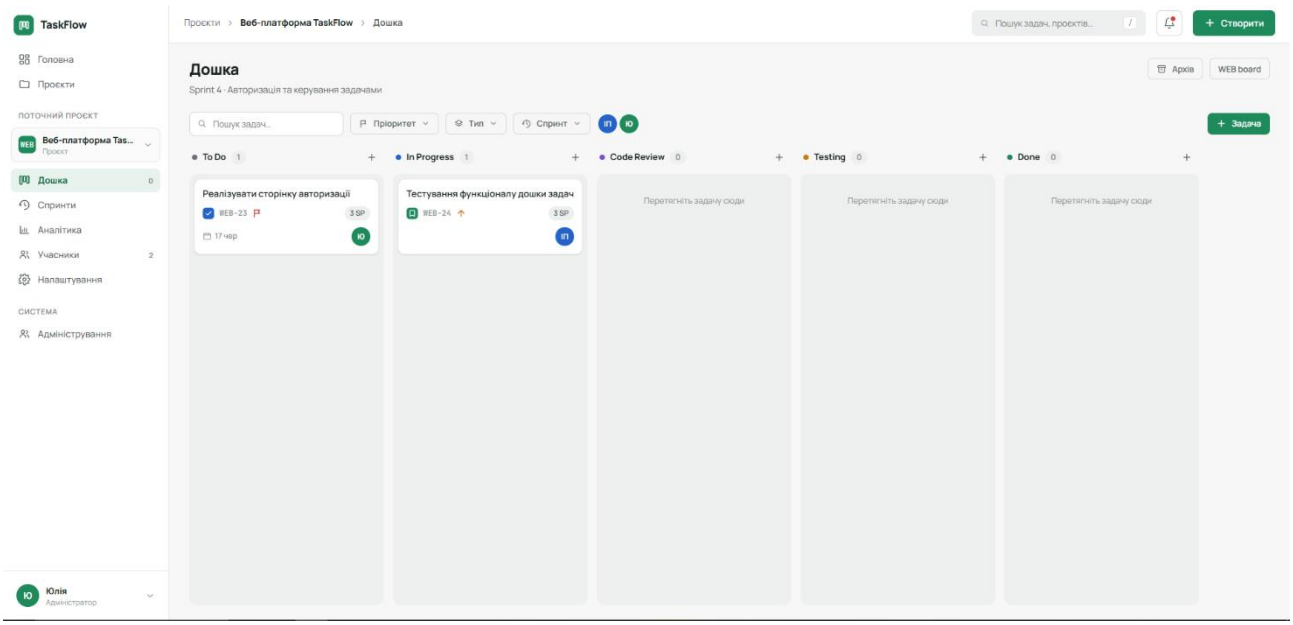


Рисунок 5.8 – Інтерфейс керування завданнями на Kanban-дошці TaskFlow

Для зміни стану роботи картка переміщується до іншої колонки методом перетягування. Після завершення операції клієнтська частина надсилає запит на зміну статусу. Сервер оновлює завдання та створює запис у журналі змін.

Сценарій перегляду деталей завдання

Після натискання на картку відкривається модальне вікно з детальною інформацією про завдання. У ньому відображаються основні поля, коментарі, вкладення та журнал змін.

У межах цього вікна користувач може додати коментар, завантажити

файл або переглянути історію зміни статусу, пріоритету, виконавця та строку виконання. Відповідний інтерфейс наведено на рисунку 5.9.

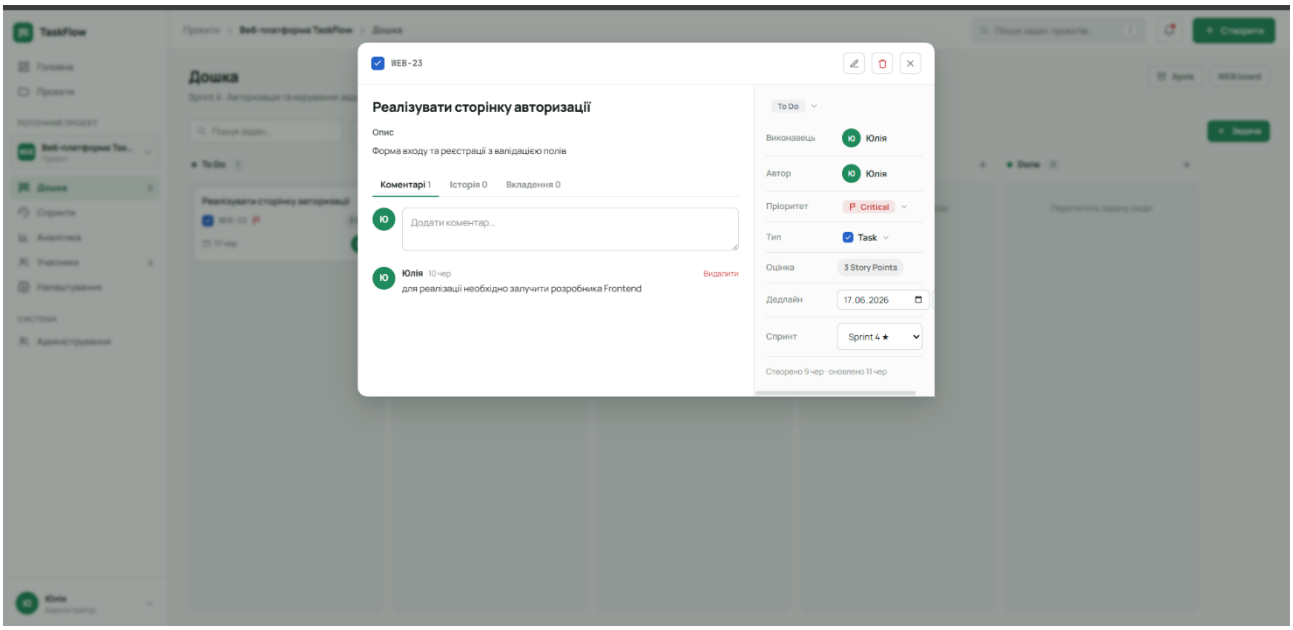


Рисунок 5.9 – Вікно перегляду детальної інформації про завдання

Сценарій планування спринту

Для організації ітераційної роботи користувач переходить до екрана спринтів. Новий спринт створюється із зазначенням назви, мети, дати початку та завершення.

Після створення спринт перебуває в запланованому стані. Користувач із відповідними правами може активувати його або завершити після закінчення роботи.

Під час ручної перевірки також відтворюється спроба запустити другий спринт за наявності активного. У такому випадку система відхиляє операцію та відображає повідомлення про те, що одночасно може бути активною лише одна ітерація.

Сценарій командної взаємодії

Користувачі TaskFlow можуть обговорювати виконання роботи шляхом додавання коментарів до конкретного завдання. Такий підхід

дозволяє зберігати уточнення, домовленості та технічні пояснення безпосередньо в контексті тієї роботи, до якої вони належать. Після введення тексту повідомлення сервер автоматично пов’язує коментар із поточним авторизованим користувачем і відповідним завданням.

Коментар відображається разом з ім’ям автора й датою створення. Автор може видалити власне повідомлення.

До завдання також можна прикріпити файл. Після завантаження він відображається у списку вкладень і стає доступним користувачам, які мають право переглядати відповідне завдання. Такий механізм дозволяє зберігати технічні матеріали, зображення, документи або інші файли, пов’язані з виконанням конкретної роботи.

Сценарій перегляду аналітики

Аналітичний модуль TaskFlow відображає узагальнену інформацію про стан проєкту. Після переходу до відповідного екрана клієнтська частина надсилає запит до серверної частини та отримує агреговані показники, сформовані на основі актуальних даних із бази даних. Це дозволяє користувачу швидко оцінити загальний перебіг виконання робіт без необхідності переглядати кожне завдання окремо.

На екрані відображаються розподіл завдань за статусами, типами та пріоритетами, кількість завдань для кожного виконавця і прогрес активного спринту.

Відповідний інтерфейс наведено на рисунку 5.10.

Якщо проєкт не містить достатньої кількості даних або активний спринт відсутній, система відображає відповідний порожній стан замість некоректних показників.

Аналіз результатів ручного тестування

Під час виконання наведених сценаріїв перевірено послідовність переходів між екранами, коректність відображення даних і реакцію інтерфейсу на успішні та помилкові операції.

Ручне тестування підтвердило, що користувачі можуть виконувати доступні їм операції відповідно до системних і проєктних ролей. Дані, створені або змінені через інтерфейс, зберігаються в PostgreSQL і залишаються доступними після повторного завантаження сторінки.

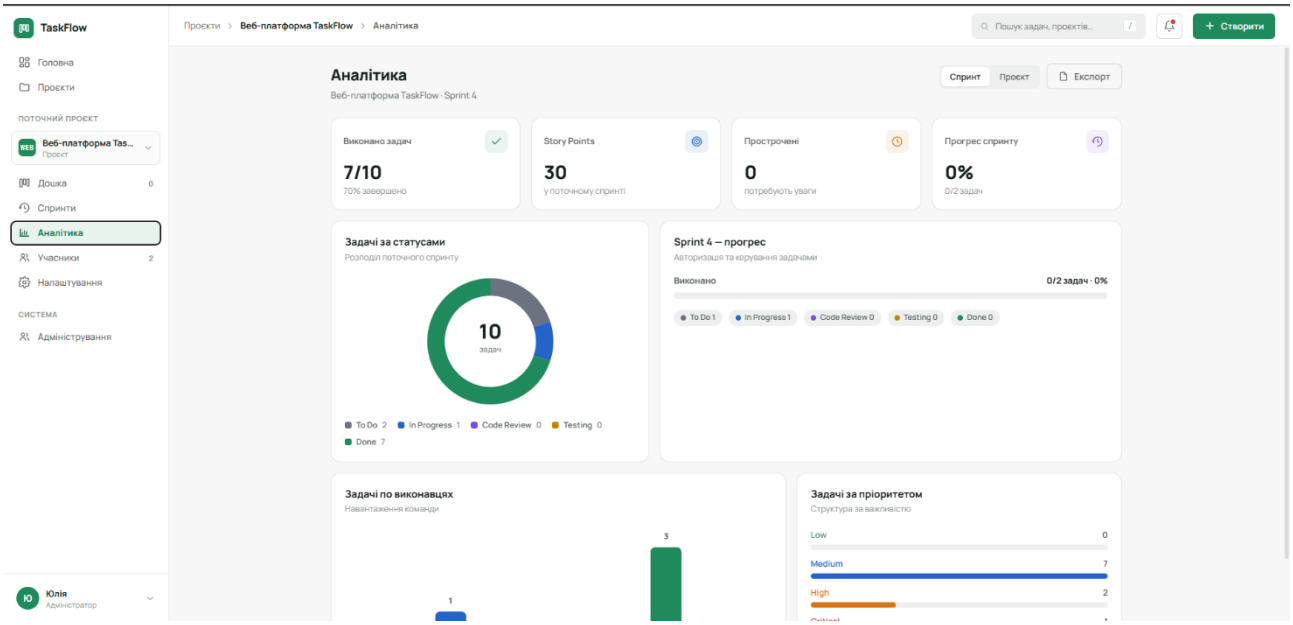


Рисунок 5.10 – Аналітичний модуль веб-застосунку TaskFlow

Таким чином, ручне функціональне тестування підтвердило працездатність основних сценаріїв використання TaskFlow і коректну взаємодію користувацького інтерфейсу із серверною частиною. Результати демонструють, що реалізовані функціональні модулі відповідають визначеним вимогам і можуть використовуватися для організації командної роботи над програмними проєктами.

5.4. Висновки по розділу

У п’ятому розділі налаштовано локальне тестове середовище та виконано автоматизоване інтеграційне й ручне функціональне тестування веб-застосунку TaskFlow

У п'ятому розділі виконано комплексну перевірку працездатності та відповідності вимогам веб-застосунку TaskFlow шляхом поєднання автоматизованого інтеграційного і ручного функціонального тестування.

Для автоматизованого тестування налаштовано середовище на основі Jest, Supertest і ts-jest. Розгортання PostgreSQL у Docker-контейнері забезпечило відтворюваність тестового оточення незалежно від операційної системи. Конфігурацію Express винесено у файл app.ts, що дозволило Supertest надсилати HTTP-запити безпосередньо до застосунку без відкриття мережевого порту.

Розроблено та виконано 45 автоматизованих інтеграційних тестів, організованих у сім наборів відповідно до підходу AAA (Arrange – Act – Assert). Охоплення охоплює такі компоненти:

8. перевірок автентифікації (auth.test.ts) — реєстрація, вхід, перевірка токена, обробка дублікатів і відсутніх полів;

10. перевірок авторизації (access.test.ts) — розмежування доступу між проєктами, перевірка системних і проєктних ролей;

6. перевірок завдань (tasks.test.ts) — створення, отримання, зміна статусу, автоматичне формування запису в TaskHistory;

6. перевірок коментарів (comments.test.ts) — створення, перегляд, видалення з перевіркою прав автора;

5. перевірок учасників (members.test.ts) — додавання, перегляд, вилучення зі складу команди;

Усі 45 тестів завершилися успішно. Підтверджено коректність механізмів автентифікації, дворівневого розмежування доступу, операцій із завданнями, коментарями, учасниками, спринтами та ведення журналу змін.

Ручне функціональне тестування у браузері Google Chrome охопило дев'ять ключових сценаріїв: автентифікацію, реєстрацію, створення та редагування проєкту, формування команди, створення та переміщення завдань на Kanban-дошці, керування спринтами, додавання коментарів і вкладень, відображення аналітичних показників. Перевірено реакцію

інтерфейсу на серверні помилки з кодами 400, 401, 403, 404, 409 і 429 — у всіх випадках застосунок відображав зрозуміле повідомлення без збоїв у роботі.

Поєднання автоматизованого і ручного тестування забезпечило перевірку як серверної логіки і механізмів безпеки, так і відповідності клієнтського інтерфейсу визначеним сценаріям використання. Жодних критичних помилок у ході тестування не виявлено, що підтверджує готовність TaskFlow до практичного розгортання.

ВИСНОВКИ

У бакалаврській роботі вирішено актуальну задачу розроблення веб-застосунку для управління проєктами, орієнтованого на невеликі команди розробників. Необхідність такого рішення обґрунтовано результатами порівняльного аналізу платформ Jira, Trello та Asana: жодна з них не забезпечує оптимального поєднання функціональності Agile-планування і простоти освоєння без надмірного адміністративного навантаження.

Сформовано функціональні й нефункціональні вимоги, побудовано UML-діаграми сценаріїв взаємодії, спроєктовано клієнт-серверну архітектуру, REST API, дворівневу модель доступу та реляційну базу даних.

Клієнтську частину реалізовано на основі React і TypeScript, серверну - з використанням Node.js та Express. Дані зберігаються в PostgreSQL, а взаємодія з базою реалізована через Prisma ORM. У системі створено модулі керування користувачами, проєктами, учасниками, завданнями та спринтами, Kanban-дошку, коментарі, вкладення, історію змін, стрічку активності й аналітичний модуль.

Елементом удосконалення TaskFlow є узгоджений механізм контролю робочого процесу, який поєднує автоматичну фіксацію змін, формування стрічки активності, рольове розмежування доступу, серверний контроль бізнес-правил і побудову аналітики даних.

Працездатність системи підтверджено автоматизованим інтеграційним і ручним функціональним тестуванням. Усі інтеграційні перевірки завершилися успішно, а критичних помилок під час виконання основних користувацьких сценаріїв не виявлено.

Отже, розроблений веб-застосунок відповідає визначеним вимогам і може використовуватися для організації роботи невеликих команд. Подальший розвиток системи може передбачати інтеграцію із системами контролю версій, сповіщення в реальному часі та розширення аналітичних можливостей.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Atlassian. Jira Features [Електронний ресурс]. URL: https://www.atlassian.com/software/jira/features (дата звернення: 03.05.2026).

2. Atlassian. Jira Workflows [Електронний ресурс]. URL: https://www.atlassian.com/software/jira/features/workflows (дата звернення: 04.05.2026).

3. Trello. How to Use Trello Boards and Cards [Електронний ресурс]. URL: https://trello.com/guide/trello-101 (дата звернення: 06.05.2026).

4. Trello. How to Use Trello Automation for Task Automation [Електронний ресурс]. URL: https://trello.com/guide/automate-anything (дата звернення: 07.05.2026).

5. Trello. Trello Power-Ups [Електронний ресурс]. URL: https://trello.com/power-ups (дата звернення: 08.05.2026).

6. Asana. Explore Project Views: List, Board, Calendar, Timeline and Gantt [Електронний ресурс]. URL: https://asana.com/features/project-management/project-views (дата звернення: 10.05.2026).

7. Asana. How to Use Task Dependencies [Електронний ресурс]. URL: https://help.asana.com/s/article/task-dependencies (дата звернення: 11.05.2026).

8. TypeScript. The TypeScript Handbook: Introduction [Електронний ресурс]. URL: https://www.typescriptlang.org/docs/handbook/intro.html (дата звернення: 14.05.2026).

9. Node.js. The Node.js Event Loop [Електронний ресурс]. URL: https://nodejs.org/en/learn/asynchronous-work/event-loop-timers-and-nexttick (дата звернення: 15.05.2026).

10. Express.js. Using Middleware [Електронний ресурс]. URL: https://expressjs.com/en/guide/using-middleware.html (дата звернення: 17.05.2026).

11. Express.js. Multer Middleware [Електронний ресурс]. URL: https://expressjs.com/en/resources/middleware/multer.html (дата звернення: 18.05.2026).

12. Prisma. Prisma Schema Overview [Електронний ресурс]. URL: https://www.prisma.io/docs/orm/prisma-schema/overview (дата звернення: 20.05.2026).

13. Prisma. Prisma Migrate Overview [Електронний ресурс]. URL: https://www.prisma.io/docs/orm/prisma-migrate (дата звернення: 21.05.2026).

14. PostgreSQL Global Development Group. PostgreSQL Documentation: Constraints [Електронний ресурс]. URL: https://www.postgresql.org/docs/current/ddl-constraints.html (дата звернення: 23.05.2026).

15. PostgreSQL Global Development Group. PostgreSQL Documentation: Glossary. ACID [Електронний ресурс]. URL: https://www.postgresql.org/docs/current/glossary.html (дата звернення: 24.05.2026).

16. React. React Documentation: Quick Start [Електронний ресурс]. URL: https://react.dev/learn (дата звернення: 26.05.2026).

17. React. Built-in React Hooks [Електронний ресурс]. URL: https://react.dev/reference/react/hooks (дата звернення: 27.05.2026).

18. Vite. Getting Started [Електронний ресурс]. URL: https://vite.dev/guide/ (дата звернення: 29.05.2026).

19. Vite. Hot Module Replacement API [Електронний ресурс]. URL: https://vite.dev/guide/api-hmr (дата звернення: 30.05.2026).

20. Axios. Interceptors [Електронний ресурс]. URL: https://axios-http.com/docs/interceptors (дата звернення: 01.06.2026).

21. Jones M., Bradley J., Sakimura N. JSON Web Token (JWT): RFC 7519 [Електронний ресурс]. RFC Editor, 2015. URL: https://www.rfc-editor.org/rfc/rfc7519.html (дата звернення: 02.06.2026).

22. Sheffer Y., Hardt D., Jones M. JSON Web Token Best Current Practices: RFC 8725 [Електронний ресурс]. RFC Editor, 2020. URL: https://www.rfc-editor.org/rfc/rfc8725.html (дата звернення: 03.06.2026).

23. Jest. Getting Started [Електронний ресурс]. URL: https://jestjs.io/docs/getting-started (дата звернення: 05.06.2026).

24. Supertest. HTTP Server Testing Library [Електронний ресурс]. URL: https://github.com/forwardemail/supertest (дата звернення: 07.06.2026).

25. Docker. What Is a Container? [Електронний ресурс]. URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/ (дата звернення: 09.06.2026).

26. Shwaber K., Sutherland J. The Scrum Guide [Електронний ресурс]. 2020. URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf (дата звернення: 05.05.2026).

27. Kanban Guides. The Kanban Guide [Електронний ресурс]. 2025. URL: https://kanbanguides.org/the-kanban-guide/ (дата звернення: 09.05.2026).

28. OWASP Foundation. OWASP API Security Top 10 – 2023 [Електронний ресурс]. URL: https://owasp.org/API-Security/editions/2023/en/0x11-t10/ (дата звернення: 19.05.2026).

29. OWASP Foundation. Password Storage Cheat Sheet [Електронний ресурс]. URL: https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html (дата звернення: 27.05.2026).

30. Fielding R., Nottingham M., Reschke J. HTTP Semantics: RFC 9110 [Електронний ресурс]. RFC Editor, 2022. URL: https://www.rfc-editor.org/info/rfc9110/ (дата звернення: 04.06.2026)

31. International Organization for Standardization. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model [Електронний ресурс]. URL: https://www.iso.org/standard/78176.html (дата звернення: 10.06.2026).

32. MDN Web Docs. Cross-Origin Resource Sharing (CORS) [Електронний ресурс]. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CORS (дата звернення: 11.06.2026).

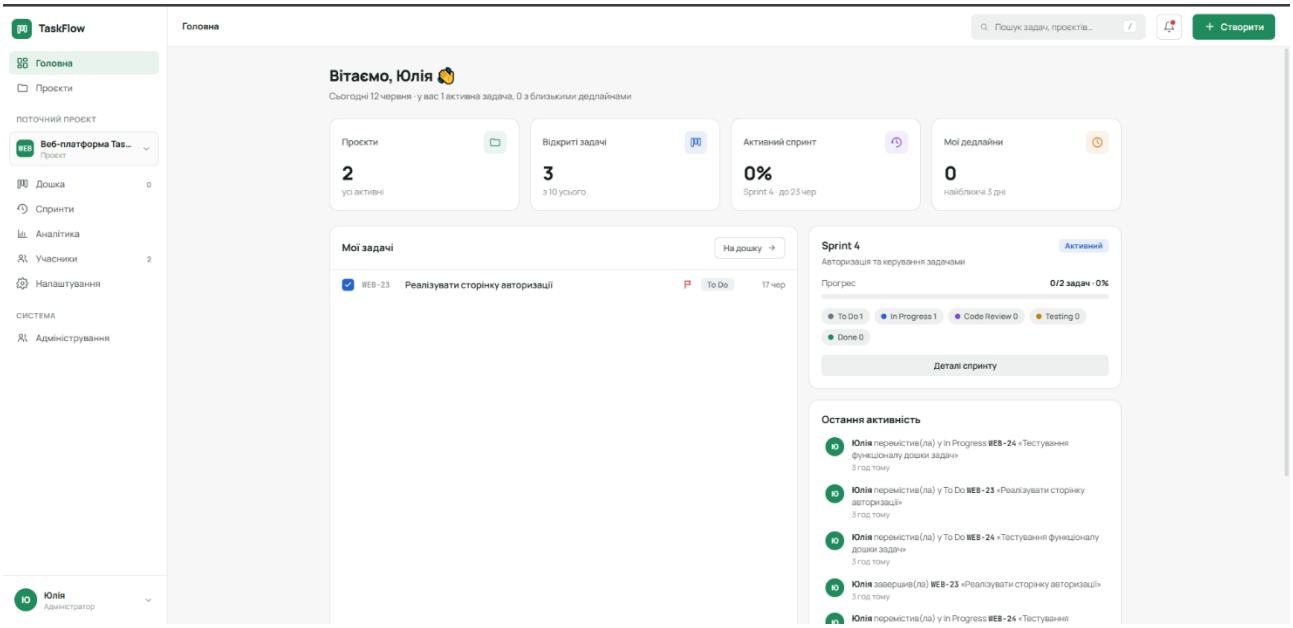
33. express-rate-limit. Overview [Електронний ресурс]. URL: https://express-rate-limit.mintlify.app/overview (дата звернення: 12.06.2026).

34. OWASP Foundation. File Upload Cheat Sheet [Електронний ресурс]. URL: https://cheatsheetseries.owasp.org/cheatsheets/File_Upload_Cheat_Sheet.html (дата звернення: 13.06.2026).

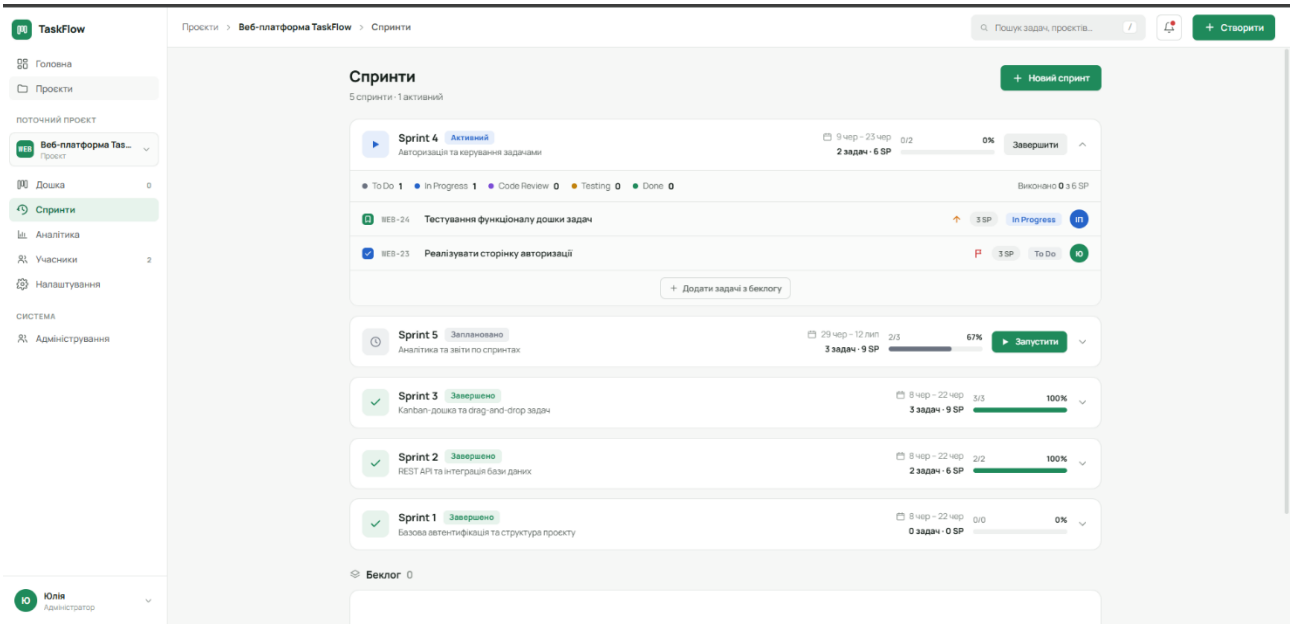
35. Jest. Setup and Teardown [Електронний ресурс]. URL: https://jestjs.io/docs/setup-teardown (дата звернення: 14.06.2026).

ДОДАТКИ

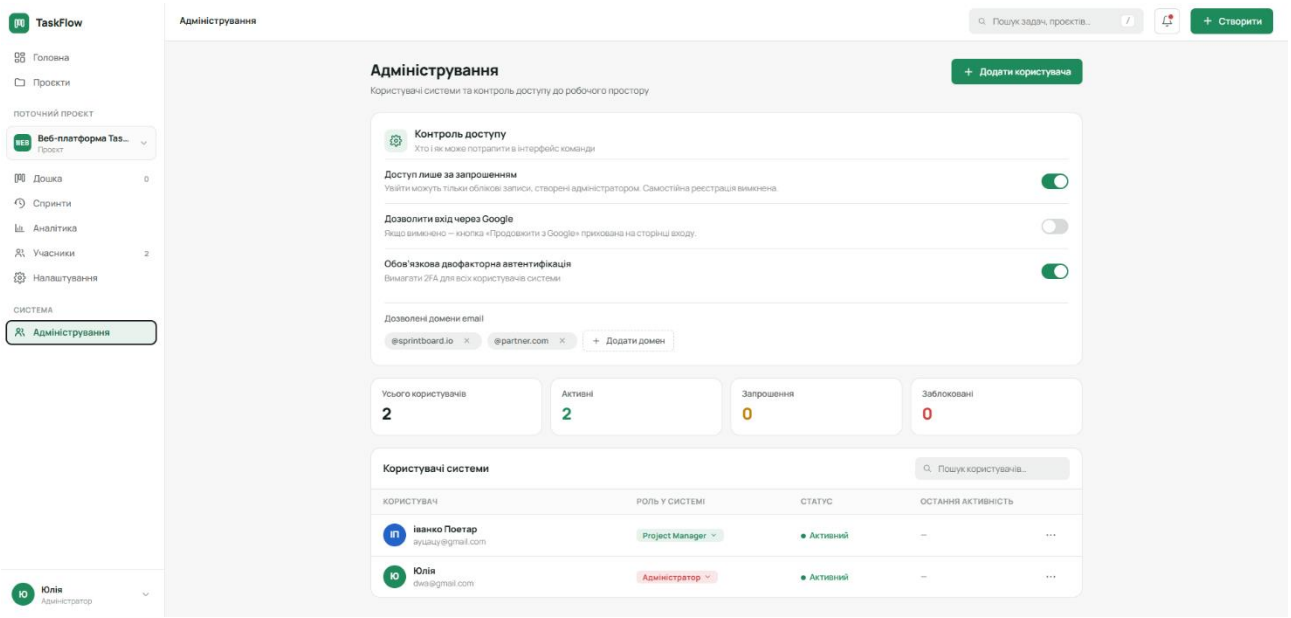
ДОДАТОК А



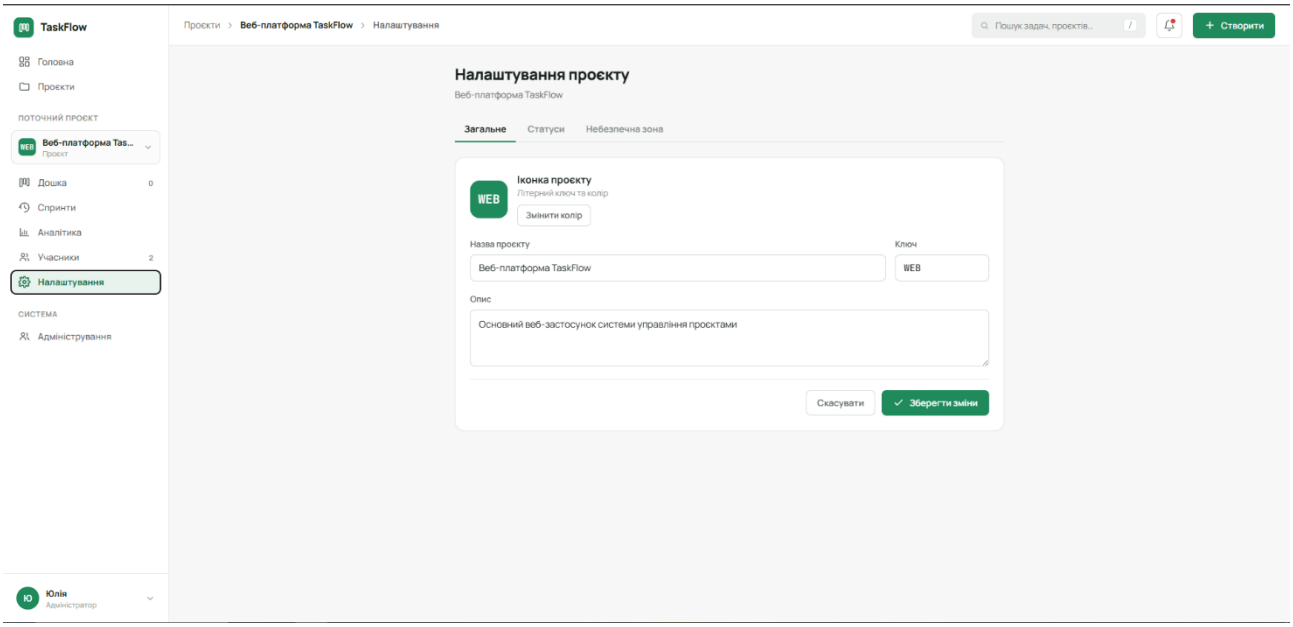
Додаток А.1 - Головна сторінка веб-застосунку



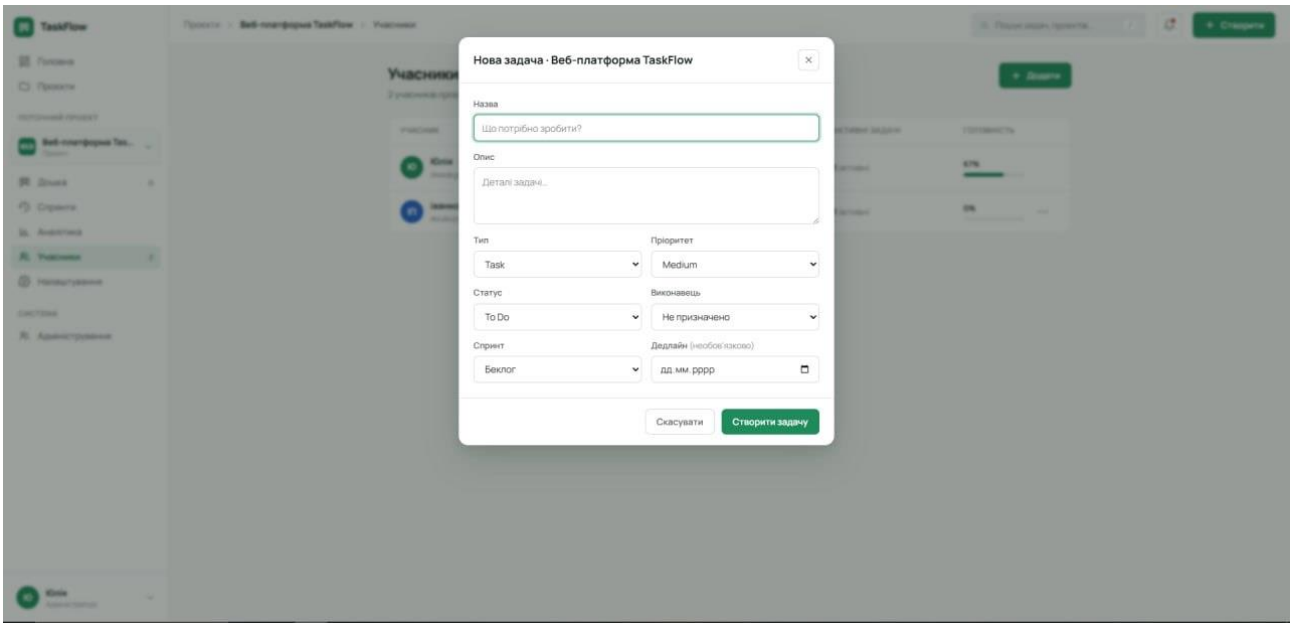
Додаток А.2 – Керування спринтами



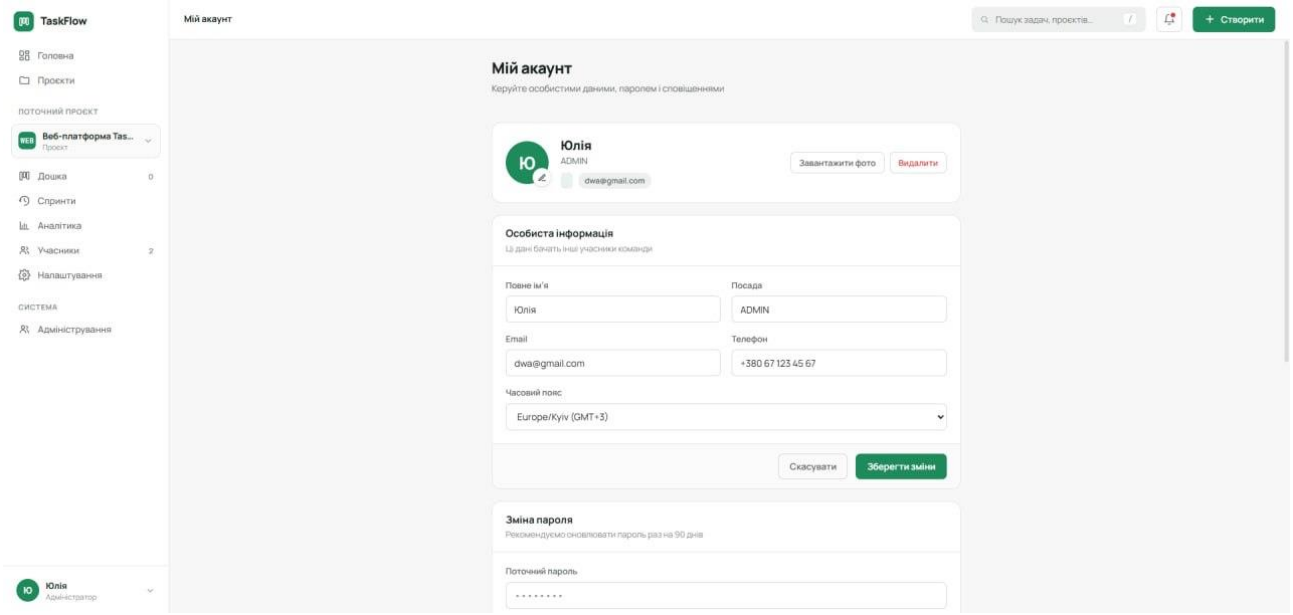
Додаток А.3 – Панель адміністратора



Додаток А.4 -Панель налаштування проєкту



Додаток А.5 – Функціонал створення завдання



Додаток А.6 – Панель налаштування акаунта

БІБЛІОГРАФІЧНА ДОВІДКА

Тема бакалаврської роботи: «Розроблення веб-застосунку для управління проєктами».

Обсяг пояснювальної записки: 84 аркуші

Дата закінчення бакалаврської роботи «11» червня 2026 р.

Підпис студента _____