МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ З ВИКОРИСТАННЯМ ANGULAR ТА NODE.JS»**

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

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

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Лахуібі Мураду
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи управління завданнями з використанням Angular та Node.js

керівник роботи Лісняк Андрій Олександрович, к.ф.-м.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

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

2. Строк подання студентом роботи 06.02.2026 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі, аналіз предметної області.
2. Проєктування.
3. Реалізація та тестування.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 14.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	28.03.2025	
2.	Збір вихідних даних.	18.04.2025	
3.	Обробка методичних та теоретичних	16.06.2025	
	джерел.		
4.	Розробка першого та другого розділу.	15.09.2025	
5.	Розробка третього розділу.	29.12.2025	
6.	Оформлення та нормоконтроль	02.02.2026	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	20.02.2026	

Студент _____ М. Лахуібі
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.О. Лісняк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи управління завданнями з використанням Angular та Node.js»: 74 с., 33 рис., 1 табл., 20 джерел, 5 додатків.

ANGULAR, ANGULAR MATERIAL, EXPRESS, JWT, KANBAN, NODE.JS, MYSQL, REST API, RXJS, UML.

Об’єкт дослідження – вебсистеми управління завданнями.

Мета роботи – розробити систему управління завданнями.

Методи дослідження – аналітичний, моделювання, проєктування, програмний.

У роботі розглянуто проєктування та реалізацію клієнт-серверної системи управління завданнями з урахуванням базових сценаріїв роботи команди: реєстрація та авторизація користувачів, створення та редагування проєктів, створення, оновлення, видалення завдань, призначення виконавців, зміна статусу завдань через форму та drag-and-drop на канбан-дошці. Клієнтська частина виконана засобами Angular та Angular Material. Серверна частина реалізована на Node.js з використанням Express.js і взаємодіє з реляційною базою даних MySQL. Взаємодія між клієнтом і сервером реалізована через REST API з авторизацією через JWT.

Побудовано UML-діаграми варіантів використання, діяльності, послідовності та розгортання, які формалізують основні сценарії та архітектурні рішення системи. Інтерфейс має наступну структуру: сторінки входу та реєстрації, список проєктів і канбан-дошку завдань з кольоровим кодуванням пріоритетів та виділенням зон перетягування. Забезпечено валідацію форм, обробку типових помилок та базове unit-тестування ключових компонентів.

Таким чином, розроблена система забезпечує зручний інтерфейс для керування завданнями, підтримує спільну роботу над проєктами, легко розширюється завдяки модульній архітектурі та може бути використана як основа для подальшого розвитку функціоналу управління проєктами.

SUMMARY

Master’s qualifying paper “Development of a Task Management System Using Angular and Node.js”: 74 pages, 33 figures, 1 table, 20 references, 5 supplements.

ANGULAR, ANGULAR MATERIAL, EXPRESS, JWT, KANBAN, NODE.JS, MYSQL, REST API, RXJS, UML.

The object of the study is task management web systems.

The aim of the study is to develop a task management system.

The methods of research are analytical, modeling, design, programming.

The paper considers the design and implementation of a client-server task management system, taking into account the basic scenarios of the team’s work: user registration and authorization, project creation and editing, task creation, updating, deletion, assignment of executors, changing the status of tasks via a form and drag-and-drop on a kanban board. The client part is implemented using Angular and Angular Material. The server part is implemented on Node.js using Express.js and interacts with a relational MySQL database. Interaction between the client and server is implemented via REST API with authorization via JWT.

UML diagrams of use cases, activities, sequences, and deployment have been built to formalize the main scenarios and architectural solutions of the system. The interface has the following structure: login and registration pages, a list of projects, and a Kanban task board with color-coded priorities and highlighted drag-and-drop areas. Form validation, typical error handling, and basic unit testing of key components are provided.

Thus, the developed system provides a convenient interface for task management, supports collaborative work on projects, is easily expandable thanks to its modular architecture, and can be used as a basis for further development of project management functionality.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Технічне завдання 10
1.1 Терміни та визначення 10
1.1.1 Загальні терміни 10
1.1.2 Технічні терміни 10
1.2 Функціональні вимоги 11
1.2.1 Призначення і цілі створення системи 11
1.2.2 Загальні функціональні можливості системи 11
1.3 Нефункціональні вимоги 12
1.4 Опис предметної області 13
1.5 Опис системи 13
1.6 Огляд та порівняння аналогів 14
2 Проєктування 16
2.1 Використання UML під час розробки системи 16
2.2 Діаграма варіантів використання 16
2.2.1 Опис варіантів використання 18
2.3 Діаграма діяльності 26
2.4 Діаграма послідовності 29
2.5 Діаграма розгортання 30
3 Програмна реалізація та тестування 33
3.1 Опис інструментів розробки 33
3.2 Основні компоненти та класи системи 34
3.3 Встановлення та налаштування оточення 36
3.4 Опис API 39

3.5 Тестування проєкту 41
3.6 Керівництво користувача 43
3.6.1 Вхід та реєстрація 43
3.6.2 Керування проєктами 47
3.6.3 Керування завданнями 48
Висновки 52
Перелік посилань 53
Додаток А Angular TaskDialogComponent 55
Додаток Б Angular TaskService 59
Додаток В Express Task Controller 62
Додаток Г Express Task Model 67
Додаток Д Unit-test Register Component 71

ВСТУП

З поширенням дистанційної роботи, гнучких форматів зайнятості та командної розробки програмного забезпечення зростає потреба в інструментах, що дозволяють централізовано планувати, відслідковувати та координувати виконання завдань. Вебсистеми управління завданнями відіграють важливу роль в організації робочих процесів, адже забезпечують доступ до актуального стану проєктів з будь-якого пристрою та у будь-який час.

Ринок пропонує багато рішень для управління задачами, однак частина з них має надмірний функціонал і високий поріг входження, інші – орієнтовані на комерційне використання з обмеженим безкоштовним доступом, або ж не дозволяють легко адаптувати систему під специфічні освітні чи невеликі командні сценарії. В умовах навчальних та невеликих проєктних команд постає задача створити просту, зрозумілу та відкриту систему управління завданнями з мінімальним, але достатнім набором функцій.

Актуальність роботи полягає у розробці вебсистеми управління завданнями, яка поєднує мінімалістичний інтерфейс із сучасними інженерними практиками. Система має забезпечувати базові сценарії роботи з проєктами й завданнями, включно з можливістю призначення виконавців і зміною статусів задач у форматі канбан-дошки з підтримкою drag-and-drop.

З огляду на це, сформульовано цілі та задачі дослідження.

Мета: розробити розробити систему управління завданнями.

Завдання:

- сформулювати функціональні та нефункціональні вимоги до системи управління завданнями;
- проаналізувати, описати та порівняти готові рішення;
- спроєктувати архітектуру системи;
- реалізувати систему управління завданнями;
- протестувати систему управління завданнями, а також надати повне керівництво користувача.

Об’єкт дослідження: вебсистеми управління завданнями.

Предмет дослідження: Angular, Angular Material, Node.js, Express.js, MySQL, JWT, REST API.

Методи дослідження: аналітичний, моделювання, проєктування, програмний.

У першому розділі описано технічне завдання, функціональні та нефункціональні вимоги до системи, надано опис предметної області, а також огляд і порівняння готових рішень.

У другому розділі обґрунтовано використання UML для розробки системи управління завданнями, спроєктовано архітектуру системи та наведено основні UML-діаграми для ключових прецедентів.

У третьому розділі описано інструменти розробки, основні класи та компоненти системи, встановлення та налаштування середовища, надано опис REST API, приклади unit-тестування клієнтських компонентів, а також наведено повне та покрокове керівництво користувача з основних сценаріїв роботи системи.

1 ТЕХНІЧНЕ ЗАВДАННЯ

1.1 Терміни та визначення

1.1.1 Загальні терміни

Система – вебзастосунок для управління завданнями.

Angular – платформа та фреймворк для розробки SPA-застосунків [1].

Angular Material – бібліотека UI-компонентів для Angular, що забезпечує сучасний, адаптивний інтерфейс у стилі Material Design [2].

Node.js – середовище виконання JavaScript на стороні сервера [11].

Express.js – мінімалістичний серверний фреймворк для Node.js [6].

MySQL – реляційна система керування базами даних, що використовується для зберігання інформації про користувачів, проєкти та завдання [10].

Користувач – суб’єкт, що взаємодіє з системою.

Проєкт – логічна сутність, що об’єднує набір взаємопов’язаних завдань.

Завдання – одиниця роботи в межах проєкту, що має назву, опис, виконавця, статус, пріоритет та дедлайн.

1.1.2 Технічні терміни

UI – інтерфейс користувача.

UX – досвід взаємодії користувача з системою.

SPA-застосунок – вебзастосунок, який завантажується на одній сторінці та динамічно оновлює контент без повного перезавантаження.

REST API – стиль побудови вебінтерфейсів, у якому клієнт і сервер взаємодіють через стандартизовані HTTP-запити [15].

JSON – текстовий формат обміну даними між клієнтом і сервером.

JWT – токен у форматі JSON Web Token, що використовується для авторизації та передачі інформації про користувача [14].

Drag-and-drop – механізм перетягування елементів інтерфейсу мишкою або дотиком.

Канбан-дошка – візуальне представлення завдань у вигляді колонок за статусами.

Компонент – структурна одиниця Angular, що поєднує шаблон, стилі та логіку.

1.2 Функціональні вимоги

1.2.1 Призначення і цілі створення системи

Функціональне призначення системи – надати користувачам можливість створювати проєкти та завдання, призначати відповідальних, змінювати статуси, відслідковувати прогрес виконання та організовувати спільну роботу над задачами.

Експлуатаційне призначення – система експлуатується користувачами для планування, організації та контролю виконання завдань у межах проєктів.

Мета створення системи – розробити мінімалістичну, зручну та масштабовану систему управління завданнями з сучасним інтерфейсом та надійним REST API.

1.2.2 Загальні функціональні можливості системи

Система має надавати користувачам такі можливості:

– вхід/вихід/реєстрація у системі;
– перегляд/створення/редагування/видалення проєктів;

– перегляд/створення/редагування/видалення завдань;
– змінити статус/пріоритет завдання;
– призначити/змінити виконавця завдання;
– фільтрувати завдання за статусом/пріоритетом/виконавцем.

1.3 Нефункціональні вимоги

Інтерфейс користувача. Система повинна коректно відображатися на всіх типах сучасних пристріїв. Система повинна мати наступні елементи: верхня панель навігації, що відображається тільки на захищених сторінках та містить назву системи, меню навігації та логотип; головний вміст з адаптивною сіткою; діалогові вікна для створення/редагування проєктів та завдань, реалізовані засобами Angular Material; канбан-дошка з зонами для перетягування завдань між статусами, візуальним виділенням пріоритетів, виконавця та дедлайну.

Підтримка браузерів. Система повинна коректно працювати в актуальних десктопних та мобільних версіях основних браузерів: Firefox, Chrome, Edge.

Вимоги до продуктивності. Початкове завантаження застосунку має займати не більше 1-2 секунд на сучасних пристроях за умови стабільного Інтернет-з’єднання. Операції створення, редагування та зміни статусу завдань повинні виконуватися з мінімальною затримкою. Запити до бази даних мають бути оптимізовані для типових сценаріїв.

Вимоги до безпеки та приватності. Доступ до закритих API-ендпоінтів повинен здійснюватися лише після успішної автентифікації користувача. Для авторизації використовуються токени, що передаються з клієнта на сервер у заголовках запитів. Паролі користувачів мають зберігатися у базі даних у хешованому вигляді. Дані про завдання та проєкти повинні бути доступні лише авторизованим користувачам із відповідними правами.

1.4 Опис предметної області

Предметною областю є планування, організація та супровід виконання завдань у рамках одного або декількох проєктів.

Розглянемо типові сценарії використання.

Створення проєкту – користувач відкриває сторінку «Merge», завантажує 2 або більше PDF-файлів, змінює порядок перетягуванням, запускає обробку, після чого отримує один злитий PDF.

Створення завдання – користувач відкриває проєкт, натискає «Додати завдання», вводить назву, опис, вибирає статус, пріоритет, дедлайн та за потреби виконавця з пошуком за email. Після збереження завдання з’являється у відповідній колонці канбан-дошки.

Зміна статусу завдання – користувач відкриває канбан-дошку проєкту, перетягує завдання до відповідної колонки. Статус зберігається на сервері, а інтерфейс оновлюється.

Призначення виконавця – користувач відкриває картку завдання, вводить у полі пошуку email, вибирає користувача зі списку, зберігає зміни. Проєкт з’являється у списку доступних.

Доступ до основної функціональності можливий лише після реєстрації та входу в систему.

1.5 Опис системи

Система побудована за клієнт-серверною архітектурою з використанням SPA на Angular для клієнтської частини та REST API на Node.js/Express для серверної частини. Базу даних реалізовано на MySQL.

Розглянемо основні компоненти системи.

Клієнтську частину створено на Angular з використанням Angular Material для реалізації сучасного, адаптивного інтерфейсу.

Серверна частина створена як API-сервер на Node.js з використанням Express, що реалізує REST API для взаємодії з клієнтом.

Реляційна база даних реалізована на MySQL. Містить таблиці користувачів, проєктів, завдань та пов’язані.

Наданий поділ дозволяє незалежно розвивати клієнтську і серверну частини, розширювати модулі, не порушуючи основної логіки, а також масштабувати серверний компонент у разі зростання кількості користувачів.

1.6 Огляд та порівняння аналогів

На ринку існує велика кількість систем управління завданнями та проєктами. Розглянемо деякі поширені готові рішення.

Trello є вебсервісом для візуального управління задачами на основі канбан-дошок. Дозволяє створювати дошки, списки та картки, додавати виконавців, дедлайни, чеклісти. Перевагою є простота та інтуїтивність, однак при збільшенні кількості проєктів і карток структура може ускладнюватися, а розширений функціонал доступний переважно в платних тарифах [17].

Jira представляє собою систему для управління задачами та інцидентами, орієнтована на розробку програмного забезпечення. Підтримує складні робочі процеси, ролі, інтеграції з іншими системами. Перевагою є гнучкість та широкі можливості налаштування, але інтерфейс та модель можуть бути надмірно складними для невеликих команд або простих навчальних проєктів [8].

Asana є системою управління завданнями та проєктами із фокусом на командну взаємодію. Дозволяє створювати списки, дошки, таймлайни, призначати завдання, налаштовувати сповіщення. Підходить для різних доменів, але містить велику кількість функцій, що може збільшувати когнітивне навантаження та вимоги до навчання користувачів [3].

ClickUp є платформою, що поєднує управління задачами, документами, чатами та іншими аспектами командної роботи. Дає широкий набір інструментів,

проте інтерфейс і налаштування можуть бути перевантаженими для користувачів, які потребують лише базової канбан-дошки [5].

У таблиці 1.1 наведено порівняння аналогів з розроблюваною системою за базовими критеріями.

Таблиця 1.1 – Порівняння аналогів

Функція / Платформа	**Наша система**	**Trello**	**Jira**	**Asana**	**ClickUp**
Канбан-дошка	Так	Так	Так	Так	Так
Призначення виконавців	Так	Так	Так	Так	Так
Статуси та пріоритети завдань	Так	Частково	Так	Так	Так
Фокус на мінімалістичному UI	Так	Частково	Ні	Частково	Ні
Інтеграція з іншими сервісами	Мінімальна	Широка	Широка	Широка	Широка
Орієнтація на навчальні/маленькі команди	Так	Так	Ні	Так	Частково

Аналіз систем показує, що незважаючи на схожість базових можливостей, більшість розглянутих платформ орієнтовані або на широкий спектр сценаріїв, або на інтеграцію з великою кількістю зовнішніх сервісів та використання в комерційних середовищах. Це призводить до збільшення складності інтерфейсу, кількості налаштувань та кроків для досягнення результату.

Розроблювана система фокусується на цільовій функціональності для невеликих команд і навчальних проєктів. Такий підхід дозволяє зменшити когнітивне навантаження на користувача та скоротити час до отримання результату.

2 ПРОЄКТУВАННЯ

2.1 Використання UML під час розробки системи

Unified Modeling Language є стандартною нотацією для специфікації, візуалізації та документування моделей програмних систем. Організація OMG визначає UML як універсальну мову моделювання, яка допомагає описувати структуру й поведінку системи на різних етапах життєвого циклу від аналізу вимог до проєктування та документування реалізації [18].

У контексті розробки системи управління завданнями UML використовується як інструмент зв’язку між вимогами, архітектурними рішеннями та конкретною реалізацією на Angular і Node.js.

У системі використовуються наступні діаграми:

– варіантів використання щоб формалізувати очікувану поведінку системи з точки зору користувача;
– діаграма діяльності щоб описати логіку виконання ключових сценаріїв;
– діаграма послідовності щоб показати взаємодію клієнта, сервера та бази даних у часі;
– діаграма розгортання щоб зафіксувати фізичну архітектуру системи.

Надані діаграми є ключовим, що напряму підтримує розуміння та полегшує розробку системи.

2.2 Діаграма варіантів використання

Діаграма варіантів використання належить до поведінкових діаграм UML і відображає акторів та сценарії взаємодії з системою. Використовується для фіксації функціональних вимог і показує, які саме дії може виконувати актор у межах системи. Варіанти використання зображуються у вигляді еліпсів, а актори у вигляді фігур людини, між ними проводяться асоціації [19].

У проєкті системи управління завданнями діаграма варіантів використання [7, 19]:

– виділяє основного актора «Користувач», який взаємодіє із системою через вебінтерфейс;
– фіксує базові прецеденти: «Реєстрація у системі», «Вхід до системи», «Вихід з системи», «Створити проєкт», «Редагувати проєкт», «Видалити проєкт», «Переглянути проєкт», «Створити завдання», «Редагувати завдання», «Видалити завдання», «Змінити статус завдання», «Призначити виконавця завдання», «Фільтрувати завдання»;
– демонструє відношення include/extend між загальними та деталізованими сценаріями.

Такий опис дозволяє на рівні вимог чітко окреслити межі системи, очікувану поведінку та підтримувані сценарії роботи користувача [7, 19].

На рисунку 2.1 зображена діаграма варіантів використання для системи управління завданнями. Представлено актора «Користувач», який взаємодіє з системою на рівні UI.

Виділимо один із основних прецедентів системи – «Створити завдання». Цей варіант використання є ключовим і доступний усім авторизованим користувачам, що мають доступ до проєкту.

Після натискання на кнопку «Додати завдання», відкривається модальне вікно з формою створення.

Користувач заповнює назву завдання, додає опис, обирає статус, пріоритет, дедлайн та опціонально виконавця, після чого натискає на кнопку «Зберегти».

Система виконує валідацію введених даних, надсилає запит до REST API, створює запис завдання в базі даних та оновлює інтерфейс, додаючи нову картку до відповідної колонки дошки.

У разі помилок користувач отримує інформативне повідомлення з рекомендаціями щодо виправлення помилок або повторення спроби.

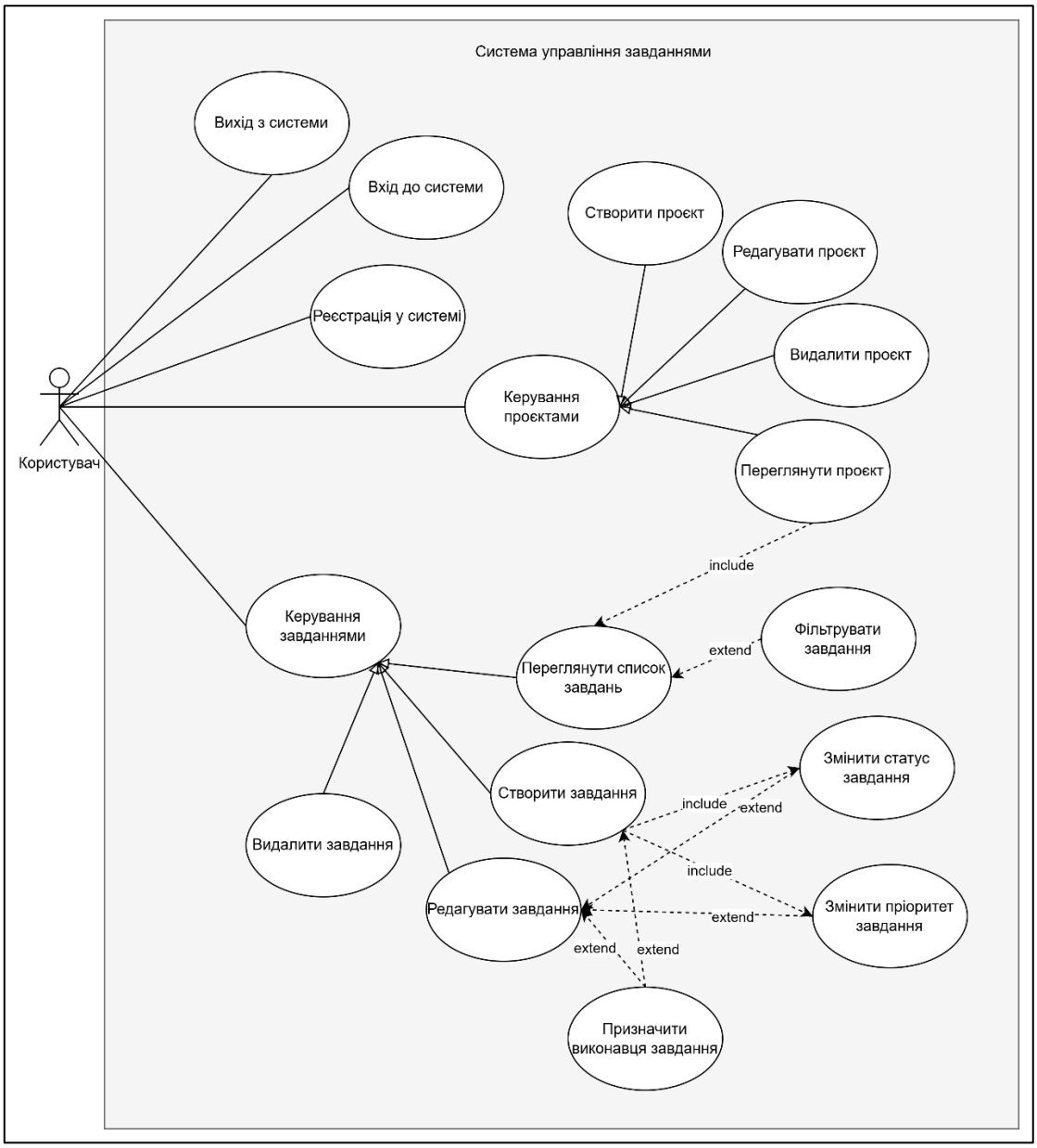


Рисунок 2.1 – Діаграма варіантів використання

2.2.1 Опис варіантів використання

Прецедент «Вхід до системи».

Призначення: даний варіант використання надає можливість користувачу автентифікуватися для доступу до списку проєктів та задач.

Передумови: користувач зареєстрований у системі.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму входу, вводить email і пароль та натискає

кнопку «Вхід». Система перевіряє облікові дані, створює токен авторизації та надає доступ до розділів «Проєкти» та інших функцій відповідно до ролі.

Виняткова ситуація 1: неправильні облікові дані – система виводить відповідне повідомлення про помилку авторизації та пропонує перевірити email/пароль.

Виняткова ситуація 2: сервер недоступний – система виводить відповідне повідомлення про технічну помилку та радить спробувати знов пізніше.

Прецедент «Вихід з системи».

Призначення: даний варіант використання надає можливість користувачу вийти з системи.

Передумови: користувач увійшов у систему.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає на кнопку «Вихід» у лівому нижньому куті бокової панелі. Система видаляє токен авторизації та перенаправляє користувача на сторінку авторизації.

Виняткова ситуація 1: токен невалідний або протермінований – система очищує локальні дані та перенаправляє користувача на сторінку входу.

Виняткова ситуація 2: сервер недоступний – система виконує локальний вихід, виводить повідомлення про можливі проблеми на сервері та пропонує повторно увійти пізніше.

Прецедент «Реєстрація у системі».

Призначення: даний варіант використання надає можливість користувачу створити новий обліковий запис.

Передумови: користувач не зареєстрований у системі з вказаним email.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму реєстрації, вводить ім’я, email, пароль, підтвердження пароля та натискає кнопку «Зареєструватися». Система виконує валідацію даних, створює обліковий запис із ролью за замовчуванням та автоматично виконує вхід.

Виняткова ситуація 1: вказаний email уже використовується – система виводить попередження про існуючий обліковий запис з вказаним email та

пропонує відновити пароль або використати іншу адресу.

Виняткова ситуація 2: невідповідність вимогам до пароля/валідації – система підсвічує відповідні поля, виводить повідомлення з інструкціями та блокує відправку форми до виправлення помилок.

Прецедент «Керування проєктами».

Призначення: даний варіант використання надає можливість користувачу виконувати базові операції над проєктами: створення, перегляд, редагування, видалення та архівація.

Передумови: користувач увійшов у систему.

Основний потік подій: даний варіант використання включає послідовне або вибіркове виконання прецедентів «Створити проєкт», «Переглянути проєкт», «Редагувати проєкт», «Видалити проєкт». Користувач переходить до розділу «Проєкти», переглядає список, за потреби створює нові проєкти, відкриває існуючі для перегляду або редагування, а непотрібні видаляє чи архівує.

Виняткова ситуація 1: відсутні проєкти – система відображає порожній стан із повідомленням «Проєкти відсутні».

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення про технічну помилку при завантаженні/зміні списку проєктів та пропонує повторити запит.

Прецедент «Створити проєкт».

Призначення: даний варіант використання надає можливість користувачу створити новий проєкт.

Передумови: користувач увійшов у систему та знаходиться на сторінці «Проєкти».

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Створити проєкт». Система відкриває модальну форму, у якій користувач заповнює назву проєкту та опис. Після натискання «Зберегти» система виконує валідацію даних, створює проєкт у базі даних та додає його до списку проєктів користувача.

Виняткова ситуація 1: невалідні дані – система підсвічує відповідні поля та виводить повідомлення про помилки валідації, блокуючи збереження до їх виправлення.

Виняткова ситуація 2: помилка створення на сервері – система виводить відповідне повідомлення про технічну помилку та пропонує повторити спробу.

Прецедент «Редагувати проєкт».

Призначення: даний варіант використання надає можливість користувачу оновлювати основну інформацію про існуючий проєкт.

Передумови: користувач увійшов у систему, має доступ до проєкту та знаходиться на сторінці списку проєктів або детальному перегляді проєкту.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає дію «Редагувати» для конкретного проєкту. Система відкриває форму з поточними даними. Користувач вносить зміни та натискає «Зберегти». Система виконує валідацію даних, оновлює запис проєкту в базі даних та оновлює інформацію в інтерфейсі.

Виняткова ситуація 1: відсутні права на редагування – система не виконує змін та виводить повідомлення про відсутність прав доступу.

Виняткова ситуація 2: помилка збереження на сервері – система показує повідомлення про технічну помилку, залишає форму відкритою та пропонує повторити спробу пізніше.

Прецедент «Видалити проєкт».

Призначення: даний варіант використання надає можливість користувачу видалити проєкт.

Передумови: користувач увійшов у систему та має права на видалення відповідного проєкту.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач у списку проєктів або в детальному перегляді обирає дію «Видалити». Система відображає діалог підтвердження видалення. Після підтвердження система видаляє проєкт та пов’язані з ним завдання із бази даних і оновлює список проєктів.

Виняткова ситуація 1: користувач скасовує дію – система закриває діалог підтвердження, жодних змін до даних не вноситься.

Виняткова ситуація 2: помилка видалення на сервері – система виводить повідомлення про технічну помилку та повідомляє, що проєкт не був видалений, пропонуючи повторити спробу пізніше.

Прецедент «Переглянути проєкт».

Призначення: даний варіант використання надає користувачу можливість переглянути детальну інформацію про конкретний проєкт та пов’язані із ним завдання.

Передумови: проєкт існує, користувач має права доступу до нього.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає проєкт зі списку «Проєкти». Система завантажує детальну інформацію про проєкт та відображає канбан-дошку завдань, пов’язаних із проєктом.

Виняткова ситуація 1: проєкт не знайдено або видалено – система показує повідомлення «Проєкт не знайдено» та повертає користувача до списку проєктів.

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення про технічну помилку та елементи повторного завантаження.

Прецедент «Керування завданнями».

Призначення: даний варіант використання надає можливість користувачу виконувати базові операції над завданнями в межах обраного проєкту.

Передумови: користувач увійшов у систему, обрав конкретний проєкт для перегляду.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку проєкту. Система завантажує перелік завдань проєкту, дозволяє створювати нові завдання, відкривати їх для редагування, змінювати статуси та видаляти зайві. Для взаємодії використовується канбан-дошка.

Виняткова ситуація 1: відсутні завдання – система відображає порожній стан із підказкою «Створіть перше завдання» та кнопкою для додавання.

Виняткова ситуація 2: сервер недоступний – система показує відповідне повідомлення про технічну помилку при завантаженні або зміні завдань.

Прецедент «Переглянути список завдань».

Призначення: даний варіант використання надає можливість користувачу переглянути список завдань, прив’язаних до обраного проєкту, у вигляді списку або канбан-дошки.

Передумови: користувач увійшов у систему, обрав проєкт для перегляду.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває сторінку проєкту. Система надсилає запит до серверу, отримує перелік завдань, згрупованих за статусами, та відображає їх у вигляді канбан-дошки або таблиці, забезпечуючи базову навігацію та можливість відкриття деталей.

Виняткова ситуація 1: сервер недоступний – система показує відповідне повідомлення про технічну помилку та елементи повторного завантаження.

Виняткова ситуація 2: відсутні завдання – система показує порожній стан із підказкою щодо створення першого завдання.

Прецедент «Створити завдання».

Призначення: даний варіант використання надає можливість користувачу створити нове завдання в межах обраного проєкту.

Передумови: користувач увійшов у систему, знаходиться на сторінці конкретного проєкту та має права на створення завдань.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Додати завдання» на канбан-дошці або у списку завдань. Система відкриває форму, в якій користувач задає назву завдання, опис, статус, пріоритет, дедлайн та за потреби вибирає виконавця через пошук за email. Після натискання кнопки «Зберегти» система виконує валідацію та створює нове завдання, додаючи його до відповідної колонки або списку.

Виняткова ситуація 1: невалідні дані – система підсвічує помилкові поля та блокує збереження до виправлення.

Виняткова ситуація 2: помилка збереження на сервері – система показує відповідне повідомлення про технічну помилку та пропонує повторити дію.

Прецедент «Редагувати завдання».

Призначення: даний варіант використання надає можливість користувачу змінювати властивості існуючого завдання.

Передумови: користувач увійшов у систему, має доступ до проєкту та завдання, знаходиться на сторінці проєкту.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає конкретне завдання та відкриває форму редагування. Система підтягує поточні дані завдання. Користувач вносить зміни та натискає кнопку «Зберегти». Система перевіряє дані, оновлює запис завдання в базі та оновлює відображення на дошці.

Виняткова ситуація 1: відсутні права на редагування – система блокує дію та показує відповідне повідомлення.

Виняткова ситуація 2: збій під час оновлення на сервері – система показує повідомлення про технічну помилку, зберігає введені у формі дані та пропонує повторити спробу.

Прецедент «Видалити завдання».

Призначення: даний варіант використання надає можливість користувачу видалити завдання з проєкту.

Передумови: користувач увійшов у систему, має права на редагування/видалення завдань у даному проєкті.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач натискає кнопку «Видалити» у формі редагування. Система відображає діалог підтвердження. Після підтвердження система видаляє завдання з бази даних і видаляє картку з канбан-дошки.

Виняткова ситуація 1: користувач скасовує дію – система закриває діалог підтвердження без внесення змін до даних.

Виняткова ситуація 2: помилка видалення на сервері – система виводить повідомлення про технічну помилку і пояснює, що завдання не було видалено.

Прецедент «Змінити статус завдання».

Призначення: даний варіант використання надає можливість користувачу змінювати статус завдання.

Передумови: користувач увійшов у систему, має доступ до відповідного завдання та знаходиться на сторінці проєкту.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму редагування завдання або перетягує картку завдання на іншу колонку канбан-дошки. Система фіксує новий статус, надсилає запит на сервер, зберігає зміну в базі даних і оновлює позицію завдання на дошці.

Виняткова ситуація 1: невалідний статус – система не застосовує зміну та відображає повідомлення про причину відмови.

Виняткова ситуація 2: помилка оновлення на сервері – система повертає завдання до попереднього статусу та показує повідомлення про технічну помилку.

Прецедент «Змінити пріоритет завдання».

Призначення: даний варіант використання надає можливість користувачу змінити пріоритет завдання.

Передумови: користувач увійшов у систему, має доступ до завдання та права на його редагування.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму редагування завдання та обирає інший пріоритет зі списку. Система зберігає новий пріоритет у базі даних і візуально оновлює картку.

Виняткова ситуація 1: невалідне значення – система не застосовує зміни, показує повідомлення про некоректне значення та повертає попередній стан.

Виняткова ситуація 2: помилка збереження на сервері – система показує відповідне повідомлення та повертає попереднє значення пріоритету для завдання.

Прецедент «Призначити виконавця завдання».

Призначення: даний варіант використання надає можливість користувачу призначити або змінити виконавця завдання.

Передумови: користувач увійшов у систему, має права на редагування завдання, у системі є інші користувачі, яких можна призначити виконавцями.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач відкриває форму редагування завдання. У полі виконавця користувач вводить частину email виконавця, система виконує пошук користувачів і відображає список збігів. Користувач обирає потрібного виконавця та натискає «Зберегти». Система записує ідентифікатор виконавця в завдання, зберігає зміни в базі даних та оновлює відображення картки. У призначеного виконавця проєкт з’являється у списку доступних.

Виняткова ситуація 1: користувача з введеним email не знайдено – система показує відповідне повідомлення та пропонує змінити критерії пошуку.

Виняткова ситуація 2: помилка збереження на сервері – система не змінює виконавця і показує повідомлення про технічну помилку.

Прецедент «Фільтрувати завдання».

Призначення: даний варіант використання надає користувачу можливість обмежити список завдань за заданими критеріями.

Передумови: користувач увійшов у систему, знаходиться на сторінці канбан-дошки завдань конкретного проєкту.

Основний потік подій: даний варіант використання починає виконуватися, коли користувач обирає критерії фільтрації у панелі фільтрів. Система застосовує обрані критерії до списку завдань і відображає відфільтрований результат.

Виняткова ситуація 1: недійсні параметри фільтра – система ігнорує некоректні параметри та повертає базовий список.

Виняткова ситуація 2: відсутні завдання за вказаними фільтрами – система відображає порожній стан із рекомендаціями змінити критерії фільтрації.

2.3 Діаграма діяльності

Діаграма діяльності є важливою поведінковою діаграмою UML, яка моделює потік робіт, процесів або алгоритмів. Її часто розглядають як розширену

блок-схему, де показано послідовність дій, гілкування, паралельне виконання і завершення процесу [7, 9].

Основними елементами діаграми діяльності є [7, 9]:

– початковий вузол – точка старту сценарію;
– дії – прямокутники із заокругленими кутами, які відображають кроки процесу;
– вузли керування потоком – ромби та вузли розгалуження/злиття, які задають умовні переходи та паралельність;
– кінцеві вузли – завершення окремої гілки або всієї діяльності.

У системі управління завданнями діаграма діяльності використовується для детального опису основних сценаріїв, зокрема прецеденту «Створити завдання» (див. рис. 2.2).

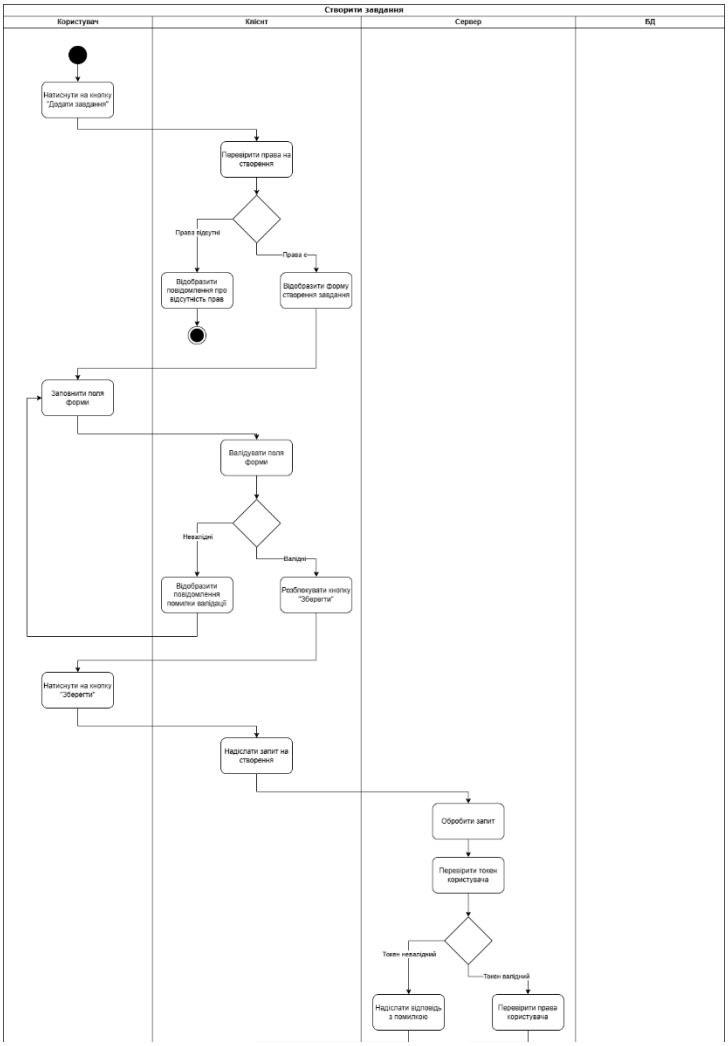


Рисунок 2.2 – Діаграма діяльності основного прецеденту (аркуш 1)

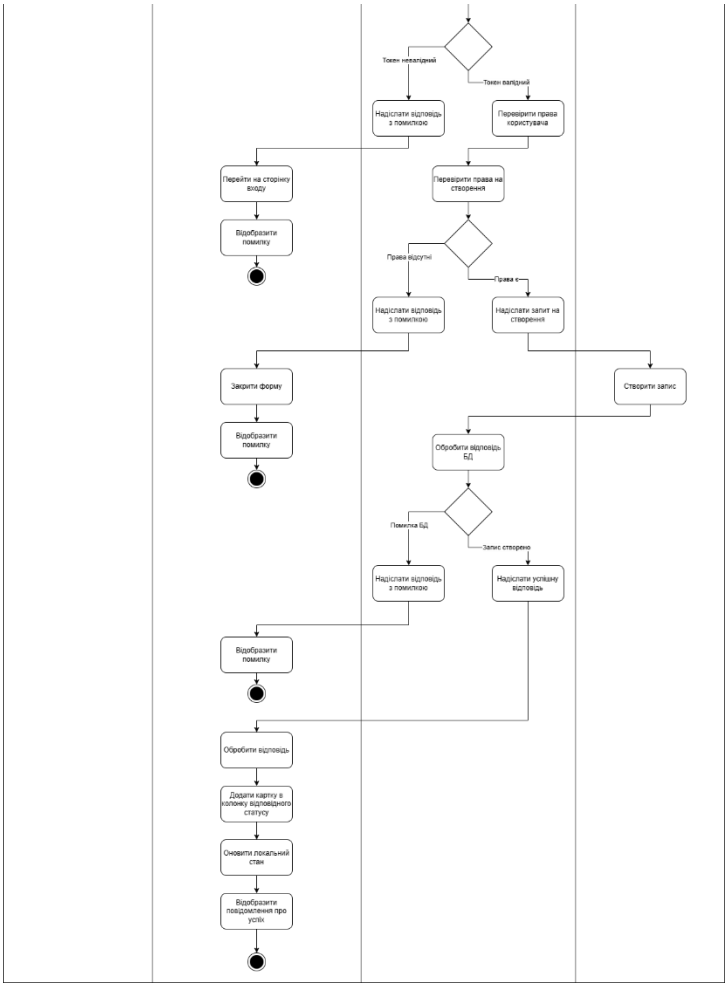


Рисунок 2.2 – Діаграма діяльності основного прецеденту (аркуш 2)

Діаграма відображає:

– дії користувача: відкриття сторінки проєкту, натискання кнопки «Додати завдання», заповнення форми, підтвердження створення;
– реакцію клієнтської частини: валідація форми, формування запиту до REST API;
– логіку на сервері: перевірка прав доступу, валідація бізнес-правил, збереження в базі даних;
– обробку виняткових ситуацій: помилки валідації, відсутність прав, збої сервера.

Такий опис дозволяє формально зафіксувати алгоритм роботи для ключових прецедентів та слугує основою для реалізації та тестування [7, 9].

2.4 Діаграма послідовності

Діаграма послідовності належить до діаграм взаємодії та описує динамічну поведінку системи, зосереджуючись на обміні повідомленнями між об’єктами або компонентами у часі. Вертикальна вісь діаграми відображає плин часу, а горизонтально розташовуються «життєві лінії» учасників взаємодії [7, 9].

Діаграми послідовності дозволяють [7, 9]:

– показати точний порядок викликів між клієнтом, сервером і базою даних;
– конкретизувати сценарій, описаний у варіанті використання;
– виявити відсутні або надлишкові операції в API та логіці системи.

У межах даної дипломної роботи діаграма послідовності побудована, зокрема, для прецеденту «Створити завдання» (див. рис. 2.3). На ній моделюється взаємодія між:

– актором «Користувач»;
– клієнтською частиною на Angular;
– серверним REST API на Node.js/Express;
– базою даних MySQL.

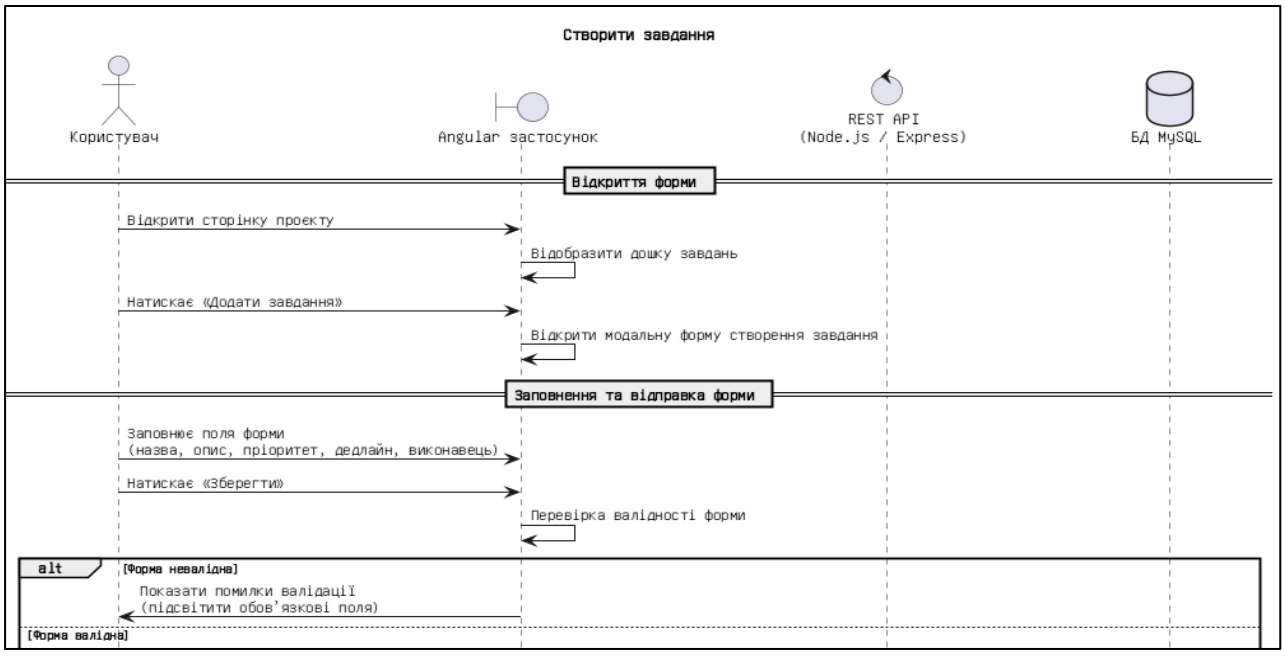


Рисунок 2.3 – Діаграма послідовності основного прецеденту (аркуш 1)

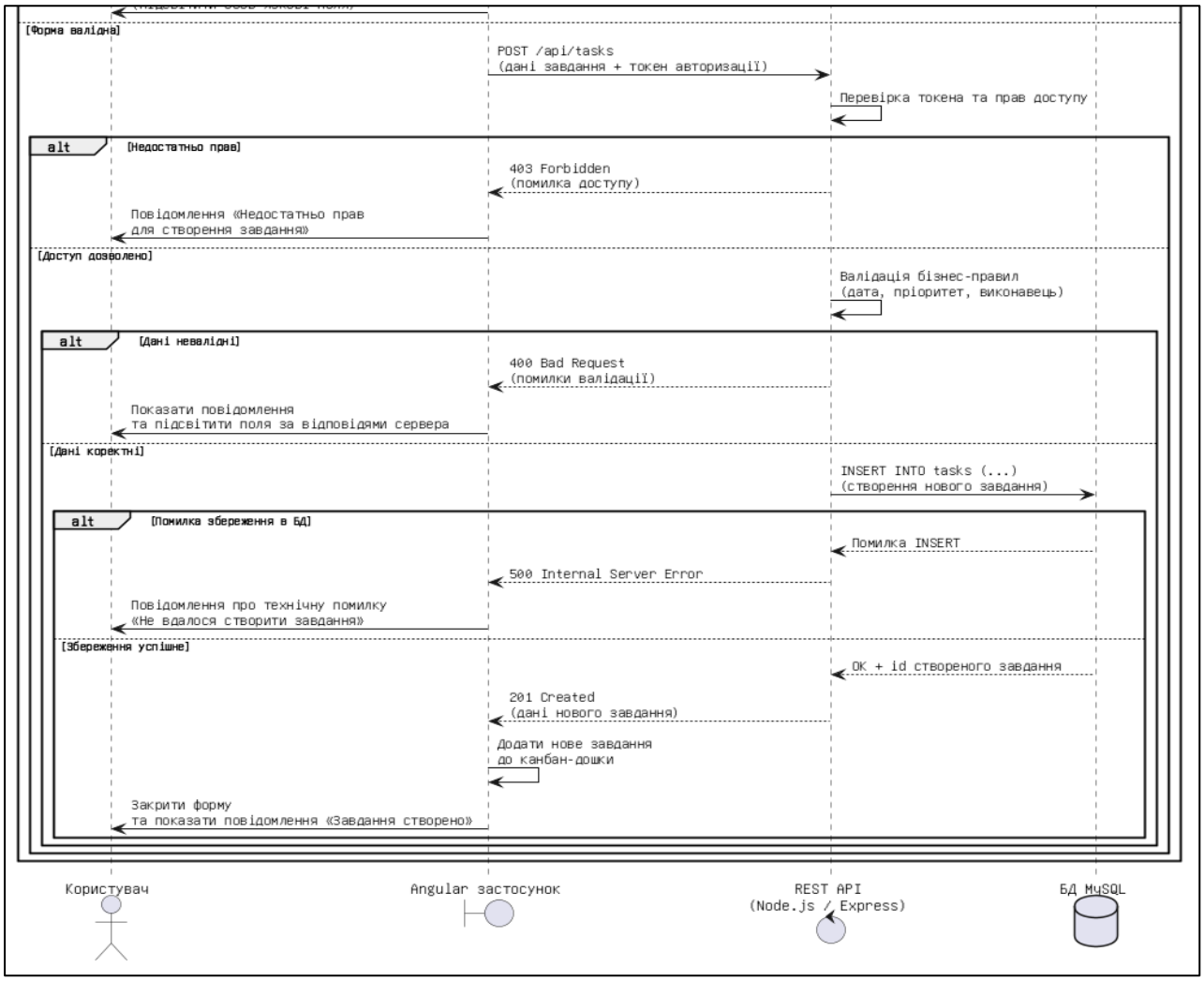


Рисунок 2.3 – Діаграма послідовності основного прецеденту (аркуш 2)

Діаграма фіксує такі кроки, як відкриття сторінки проєкту, виклик діалогу створення, валідація та відправка форми, виклик ендпоінта POST /api/tasks, перевірка прав, вставка запису в таблицю tasks та повернення відповіді з новим завданням, а також обробку можливих помилкових відповідей. Це забезпечує прозорий зв’язок між текстовим описом прецеденту, діаграмою та фактичною реалізацією на рівні коду.

2.5 Діаграма розгортання

Діаграма розгортання належить до структурних діаграм UML і моделює фізичну архітектуру системи: вузли, на яких розгортаються програмні

артефакти, та зв’язки між ними. Вона показує, як компоненти програмного забезпечення розміщуються на апаратних ресурсах – серверах, клієнтських пристроях, мережевій інфраструктурі [7, 9].

У типових вебсистемах діаграма розгортання відображає [7, 9]:

– клієнтські пристрої користувачів з веббраузером;
– сервер фронтенду;
– сервер застосунку;
– сервер бази даних;
– мережеві з’єднання між вузлами.

У проєкті системи управління завданнями діаграма розгортання (див. рис. 2.4) формалізує обрану клієнт-серверну архітектуру, а саме: Angular-застосунок завантажується у браузері клієнта, звертається до серверного REST API на Node.js/Express, а той, у свою чергу, працює з реляційною базою даних MySQL.

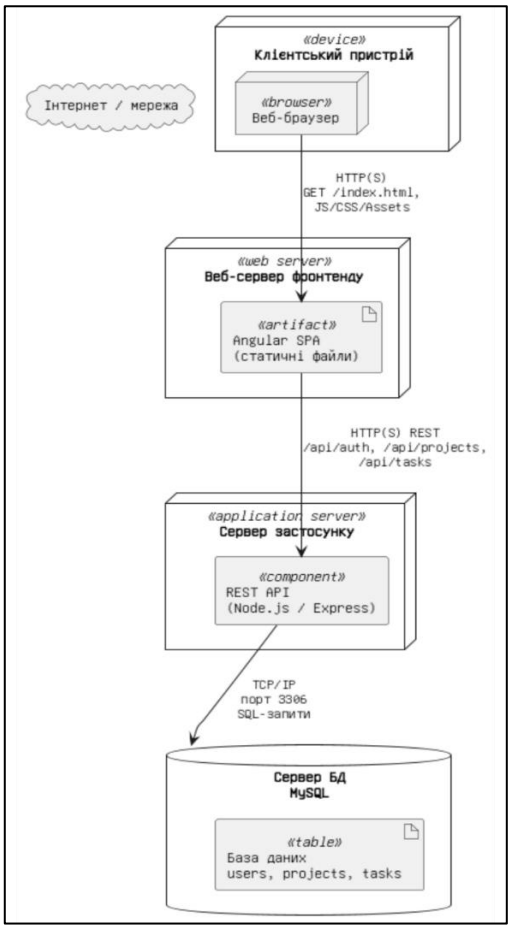


Рисунок 2.4 – Діаграма розгортання

Така візуалізація дозволяє чітко показати, на яких фізичних вузлах розташовані компоненти системи, які протоколи використовуються для взаємодії та як відбувається розподіл відповідальності між рівнями.

Отже, використання діаграм UML при розробці системи управління завданнями дозволило формально описати як функціональні можливості, так і архітектурні рішення проєкту. Діаграма варіантів використання окреслила основні сценарії взаємодії користувача з системою, діаграма діяльності деталізувала послідовність дій у ключових процесах, діаграма послідовності показала обмін повідомленнями між клієнтом, сервером і базою даних у часі, а діаграма розгортання відобразила обрану архітектуру клієнт-сервер на основі Angular, Node.js/Express і MySQL. Сукупно це забезпечило прозорість структури системи, узгодженість вимог із реалізацією та створило зручну основу для подальшого супроводу і розширення.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ

3.1 Опис інструментів розробки

Система реалізована з використанням сучасних підходів та інструментів, що надає високий рівень гнучкості, простоту використання, підтримки та розширення.

Angular – фреймворк для розробки SPA з компонентним підходом, декларативними шаблонами та реактивною моделлю роботи з даними [1].

Angular Material – бібліотека UI-компонентів для швидкого створення адаптивного інтерфейсу згідно з рекомендаціями Material Design, використовується для форм авторизації, діалогових вікон та відображення списків проєктів і завдань [2].

RxJS – бібліотека реактивних потоків, підписок та операторів, що використовується на клієнтській стороні для роботи з асинхронними HTTP-запитами, станами завантаження та оновленням списків завдань [16].

Node.js – серверна платформа з неблокуючою моделлю вводу/виводу, на якій реалізовано логіку авторизації, управління проєктами та завданнями [11].

Express.js – мінімалістичний фреймворк для Node.js, що відповідає за маршрутизацію HTTP-запитів, підключення middleware для автентифікації, обробку помилок та взаємодію з базою даних [6].

MySQL – реляційна система керування базами даних, у якій зберігаються користувачі, проєкти та пов’язані з ними завдання [10].

JWT – механізм токен-базованої авторизації, який використовується для захисту API та перевірки прав доступу користувача до операцій над проєктами та завданнями [14].

Jasmine/Karma – інструменти unit-тестування клієнтської частини, що використовуються для перевірки роботи компонентів та взаємодії з сервісами [1, 4].

Bcrypt – бібліотека для хешування паролів на сервері перед зберіганням у базі даних [12].

3.2 Основні компоненти та класи системи

У системі є декілька процесів, які можна назвати основними бізнес-процесами: робота з проєктами та завданнями. Відповідно, основними компонентами/класами є ті, що забезпечують цей процес.

Зі сторони **клієнта** можна виділити наступні компоненти й сервіси.

MainLayoutComponent – Angular-компонент, що відповідає за загальне компонування захищеної частини застосунку: верхню панель навігації, бокове меню (за потреби) та область контенту, де відображаються список проєктів чи дошка завдань.

LoginComponent – Angular-компонент, що реалізує форму входу в систему. Містить поля email/пароль, обробляє валідацію та ініціює запит до AuthService.

RegisterComponent – Angular-компонент із формою реєстрації нового користувача. Виконує перевірку полів та відправляє дані до AuthService.

ProjectListComponent – Angular-компонент, що відображає список проєктів користувача, надає можливість створити новий проєкт, редагувати або видалити існуючі, а також відкрити вибраний проєкт для перегляду завдань.

ProjectDialogComponent – діалоговий компонент на Angular Material для створення та редагування проєкту.

TaskBoardComponent – Angular-компонент, що відображає канбан-дошку завдань обраного проєкту. Використовує можливості drag-and-drop для зміни статусів завдань, а також відображає кольорове виділення пріоритетів і виконавців.

TaskDialogComponent – діалоговий компонент для створення/редагування завдання.

ProjectService – Angular-сервіс для взаємодії з REST API проєктів.

AuthService – Angular-сервіс, що відповідає за реєстрацію, вхід, вихід із системи та збереження/зчитування токена авторизації.

TaskService – Angular-сервіс для взаємодії з REST API завдань.

UserService – Angular-сервіс, що відповідає за пошук користувачів за email для форми створення/редагування завдання.

Розглянемо класи та модулі зі сторони **серверу**.

App.ts / server.ts – точка входу серверного застосунку, підключення middleware та маршрутів.

AuthController – модуль, що відповідає за реєстрацію та авторизацію користувачів.

ProjectController – контролер для CRUD-операцій над проєктами.

TaskController – контролер для CRUD-операцій над заданнями.

UserController – контролер для пошуку при призначенні виконавця завдання.

Моделі User, Project, Task описують структуру сутностей користувача, проєкту та завдання й містять функції для взаємодії з базою даних MySQL.

На рисунку 3.1 наведено функцію створення завдання зі сторону серверу.

```
exports.createTask = async (req, res, next) :Promise<...> => {
    try {
        const userId :BufferSource = req.user.id;
        const {
            projectId,
            title,
            description,
            status,
            priority,
            dueDate,
            assigneeId
        } = req.body;

        if (!projectId || !title) {
            return res
                .status(400)
                .json({ message: 'projectId і title є обов'язковими' });
        }

        const project = await Project.findByIdAndOwner(projectId, userId);
        if (!project) {
            return res
                .status(404)
                .json({ message: 'Проєкт не знайдено або немає доступу' });
        }

        const task = await Task.createTask( {projectId, creatorId, assigneeId, title, description, status, priority, dueDate}: {
            projectId,
            creatorId: userId,
            assigneeId,
            title,
            description,
            status,
            priority,
            dueDate
        });
```

Рисунок 3.1 – Функція створення завдання зі сторону серверу

З повними реалізаціями TaskDialogComponent / TaskService можна ознайомитися у додатках А, Б, з відповідними серверними обробниками – у додатках В, Г.

3.3 Встановлення та налаштування оточення

Для початку розробки системи необхідно налаштувати відповідне оточення. Оскільки клієнтська і серверна частини реалізовані на JavaScript/TypeScript, потрібно встановити актуальні версії Node.js та npm.

У якості операційної системи обрано Linux Ubuntu, тому у терміналі необхідно виконати команди sudo apt install nodejs та sudo apt install npm для встановлення Node.js та npm [11, 13]. Після встановлення, перевіряємо версії командою node -v та npm -v [11, 13].

Також необхідно встановити MySQL-сервер за допомогою команди sudo apt install mysql-server. Після встановлення створюємо базу даних task_manager та користувача з відповідними правами доступу [10].

Структура проєкту організована у вигляді батьківської теки task-manager з двома основними дочірніми теками: client та server (рис. 3.2).

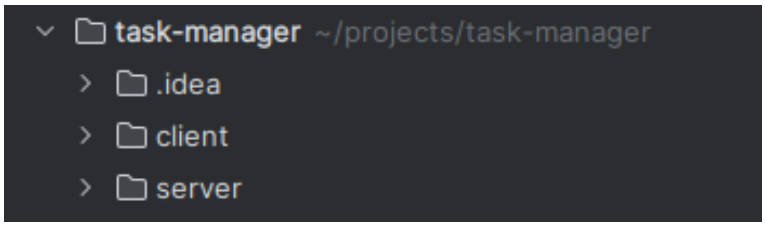


Рисунок 3.2 – Структура проєкту

У теці client розташована клієнтська частина на Angular, у теці server – серверна частина на Node.js/Express.

Для створення клієнтської частини необхідно для початку встановити Angular CLI глобально за допомогою команди npm install -g @angular/cli. Далі з кореня проєкту викликається команда ng new client --routing --no-standalone --

style=scss (рис. 3.3). Ця команда створює базовий каркас SPA-застосунку з підтримкою маршрутизації та SCSS-стилізації [1, 4].

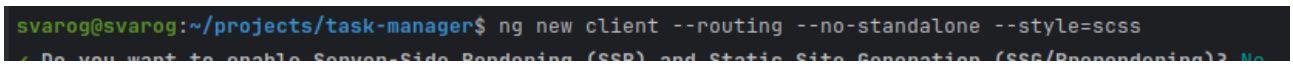


Рисунок 3.3 – Створення клієнтської частини

У теці server ініціалізується Node-проєкт за допомгою команди npm init -y та встановлюються необхідні залежності через npm install express cors mysql2 jsonwebtoken bcrypt body-parser (рис. 3.4) [6, 12 – 14].

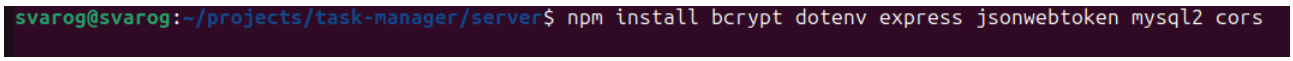


Рисунок 3.4 – Встановлення залежностей серверної чатсини

Як результат, створено проєкт з чіткою та зрозумілою структурою серверної й клієнтської частин (див. рис. 3.5 – 3.6) [1, 20].

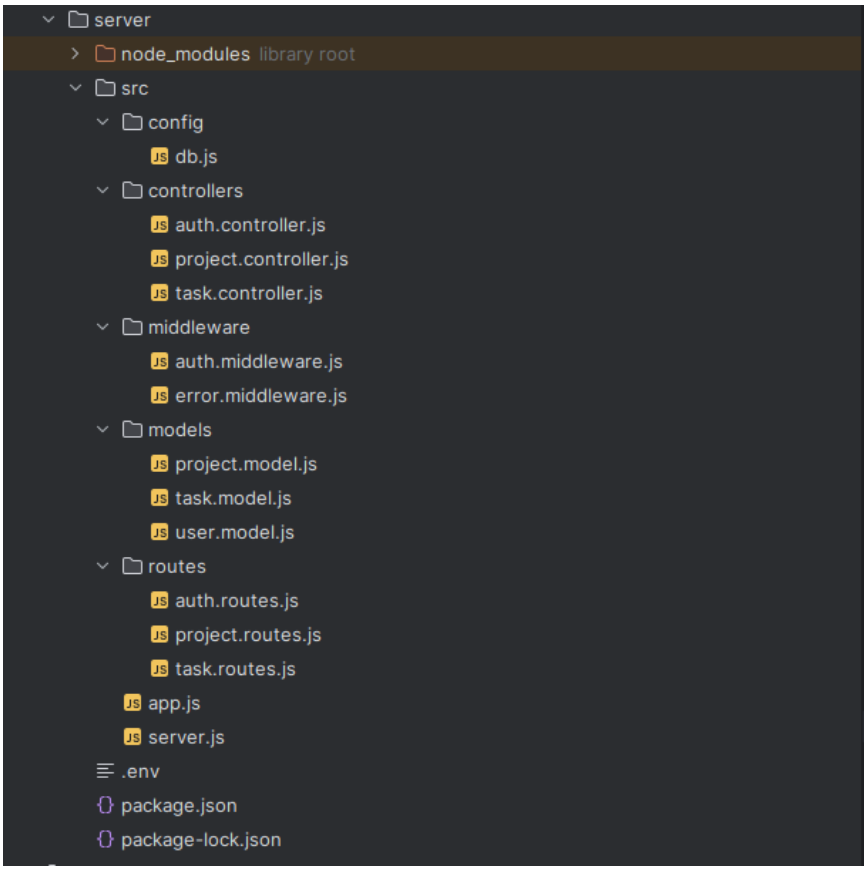


Рисунок 3.5 – Структура серверної частини

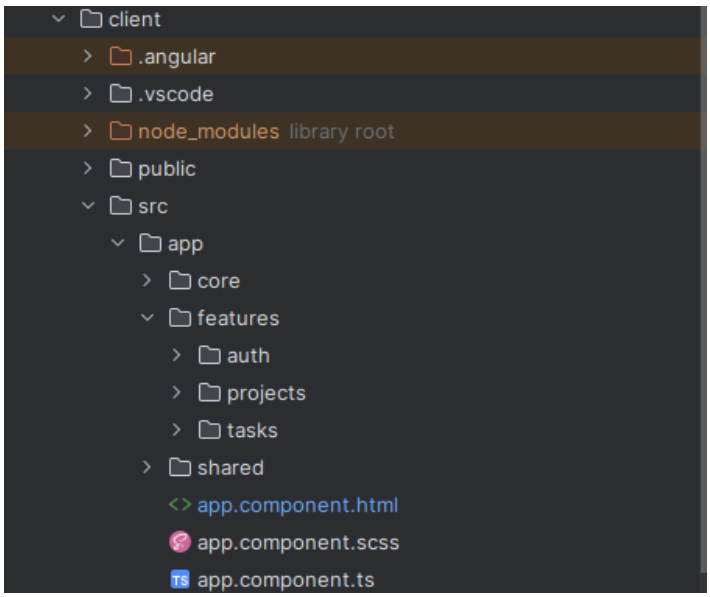


Рисунок 3.6 – Структура клієнтської частини

Після створення структури налаштовуємо конфігураційні файли для підключення до бази даних (рис. 3.7) та змінні оточення (рис. 3.8).

```
const mysql :{…} = require('mysql2/promise');
require('dotenv').config();

const pool :Pool = mysql.createPool( config: {
    host: process.env.DB_HOST,
    port: process.env.DB_PORT,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0
});

module.exports = pool;
```

Рисунок 3.7 – Файл db.js

```
PORT=4000

DB_HOST=localhost
DB_PORT=3306
DB_USER=root
DB_PASSWORD=root
DB_NAME=task_manager
```

Рисунок 3.8 – Файл .env

Для запуску серверної частини використовуємо команду npm run start (рис. 3.9) [13].

```
svarog@svarog:~/projects/task-manager/server$ npm run start

> task-manager-server@1.0.0 start
> node src/server.js

[dotenv@17.2.3] injecting env (8) from .env -- tip: 🔑 add access controls to secrets: htt
[dotenv@17.2.3] injecting env (0) from .env -- tip: 👥 sync secrets across teammates & mac
MySQL підключено успішно
Server is running on http://localhost:4000
```

Рисунок 3.9 – Запуск серверної частини

Для запуску клієнтської частини запускаємо команду ng serve (рис. 3.10) [1].

```
svarog@svarog:~/projects/task-manager/client$ ng serve
Component HMR has been enabled, see https://angular.dev/hmr for more info.
Initial chunk files | Names          |  Raw size
polyfills.js        | polyfills      |  89.77 kB |
main.js             | main           |  33.78 kB |
styles.css          | styles         |   8.01 kB |
chunk-7YWHJTTQ.js   | -              |   2.30 kB |
chunk-PZ5AY32C.js   | -              | 234 bytes |

                    | Initial total  | 134.09 kB

Lazy chunk files    | Names          |  Raw size
chunk-GDFPI6BX.js   | project-module | 143.34 kB |
chunk-PM23SEON.js   | auth-module    |  42.77 kB |

Application bundle generation complete. [2.068 seconds]

Watch mode enabled. Watching for file changes...
NOTE: Raw file sizes do not reflect development server per-request transformations.
  ➜  Local:   http://localhost:4200/
```

Рисунок 3.10 – Запуск клієнтської частини

3.4 Опис API

У системі реалізовано REST API, яке забезпечує взаємодію між клієнтською та серверною частинами через HTTP-запити. Основні операції

виконуються методами HTTP GET, POST, PUT, PATCH та DELETE з використанням формату JSON [15].

Усі запити обробляються через фреймворк Express.js. Для закритих маршрутів використовується middleware авторизації, яке перевіряє JWT-токен у заголовку «Authorization: Bearer ...». Сервіс працює у stateless-режимі [6, 14, 15].

Розглянемо основні API-endpoints.

POST /api/auth/register відповідає за реєстрацію нового користувача. Приймає JSON з полями email, password, name. Виконує валідацію, хешує пароль та створює запис у таблиці users. У відповідь повертає повідомлення про успішну реєстрацію.

POST /api/auth/login відповідає за вхід у систему. Приймає JSON з полями email та password. У разі успіху повертає JWT-токен та базову інформацію про користувача.

GET /api/projects відповідає за отримання списку проєктів для поточного користувача. Повертає масив проєктів, де користувач є автором або має завдання як виконавець.

POST /api/projects відповідає за створення нового проєкту. Приймає JSON з полями name, description. Повертає створений проєкт.

PUT /api/projects/:id відповідає за оновлення існуючого проєкту. Дозволяє змінити назву та опис проєкту за умови наявності прав.

DELETE /api/projects/:id відповідає за видалення проєкту. Видаляє проєкт і пов’язані з ним завдання за умови, що користувач має відповідні права.

GET /api/projects/:id/tasks відповідає за отримання списку завдань для конкретного проєкту. Повертає масив завдань із полями title, description, status, priority, dueDate, assignee.

POST /api/projects/:id/tasks відповідає за створення завдання у проєкті. Приймає JSON із зазначенням назви, опису, статусу, пріоритету, дедлайну та, опціонально, ідентифікатора виконавця. Повертає створене завдання.

PUT /api/tasks/:id відповідає за редагування існуючого завдання. Дозволяє змінювати основні поля завдання.

DELETE /api/tasks/:id відповідає за видалення завдання.

PATCH /api/tasks/:id/status відповідає за зміну статусу завдання. Приймає новий статус та оновлює його в базі даних.

GET /api/users відповідає за пошук користувачів за email для призначення виконавця. Приймає рядок запиту «?email=...» та повертає список користувачів, email яких відповідає шаблону.

Для помилок повертаються наступні коди та повідомлення [15]:

– 400 Bad Request – відсутні або некоректні поля, помилки валідації;
– 401 Unauthorized – відсутній або невалідний JWT-токен;
– 403 Forbidden – користувач не має прав на виконання операції;
– 404 Not Found – запитаний проєкт або завдання не знайдено;
– 500 Internal Server Error – внутрішня помилка сервера.

Опис API-структури наведено у Swagger UI (рис. 3.11).

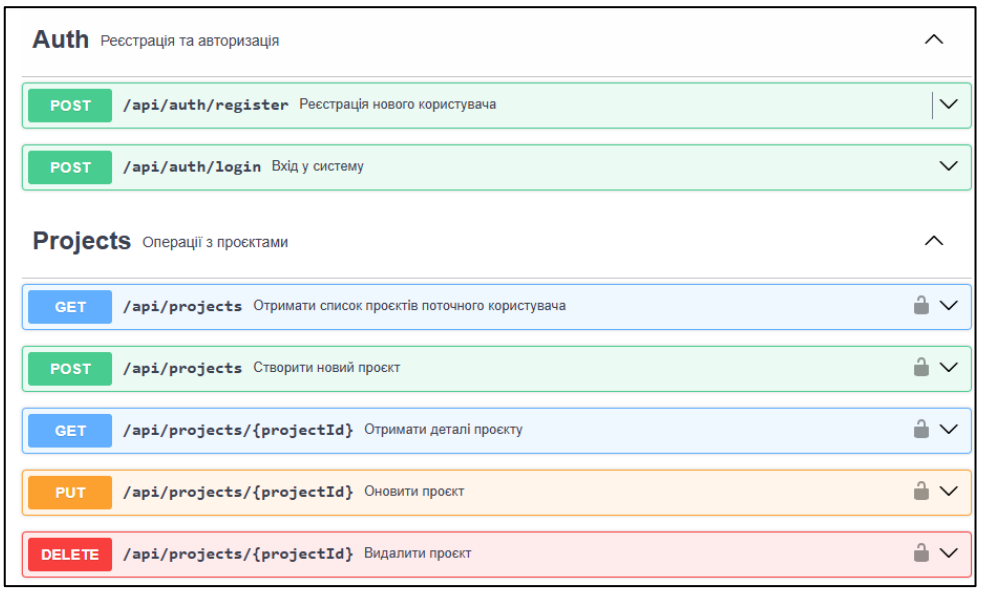


Рисунок 3.11 – Опис API у Swagger UI

3.5 Тестування проєкту

У процесі розробки системи було проведено unit-тестування для забезпечення цілісності основної бізнес-логіки, надійності взаємодії з API та коректної роботи користувацького інтерфейсу.

Для тестування клієнтської частини використовувалися вбудовані в Angular CLI фреймворки Jasmine та Karma [1, 4]. Основний акцент зроблено на перевірці роботи форм авторизації та реєстрації, а також взаємодії з сервісами, що звертаються до REST API.

Зокрема, для компонента RegisterComponent було реалізовано декілька unit-тестів, що перевіряють:

– коректне створення форми та початкові значення полів;
– поведінку валідації;
– стан кнопки «Зареєструватися»;
– виклик методу AuthService при успішному заповненні форми;
– відображення повідомлень про помилки у разі невдалої відповіді сервера.

На рисунку 3.12 наведено приклад unit-тесту для RegisterComponent.

```
it('should validate required fields', () => {
  component.form.setValue({
    fullName: '',
    email: '',
    password: '',
    confirmPassword: ''
  });

  expect(component.form.valid).toBeFalse();
  expect(component.fullName?.hasError('required')).toBeTrue();
  expect(component.email?.hasError('required')).toBeTrue();
  expect(component.password?.hasError('required')).toBeTrue();
  expect(component.confirmPassword?.hasError('required')).toBeTrue();
});
```

Рисунок 3.12 – Приклад unit-тесту

Тести запускаються командою ng test, що відкриває браузер з інтерфейсом Karma та паралельно виводить результати у консоль (див. рис. 3.13 – 3.14) [1, 4].

Проведене тестування дозволяє виявляти помилки в логіці форм та у взаємодії з сервером на ранніх етапах, забезпечуючи стабільну роботу інтерфейсу для кінцевого користувача.

Детальні приклади тестів для RegisterComponent наведено у додатку Д.

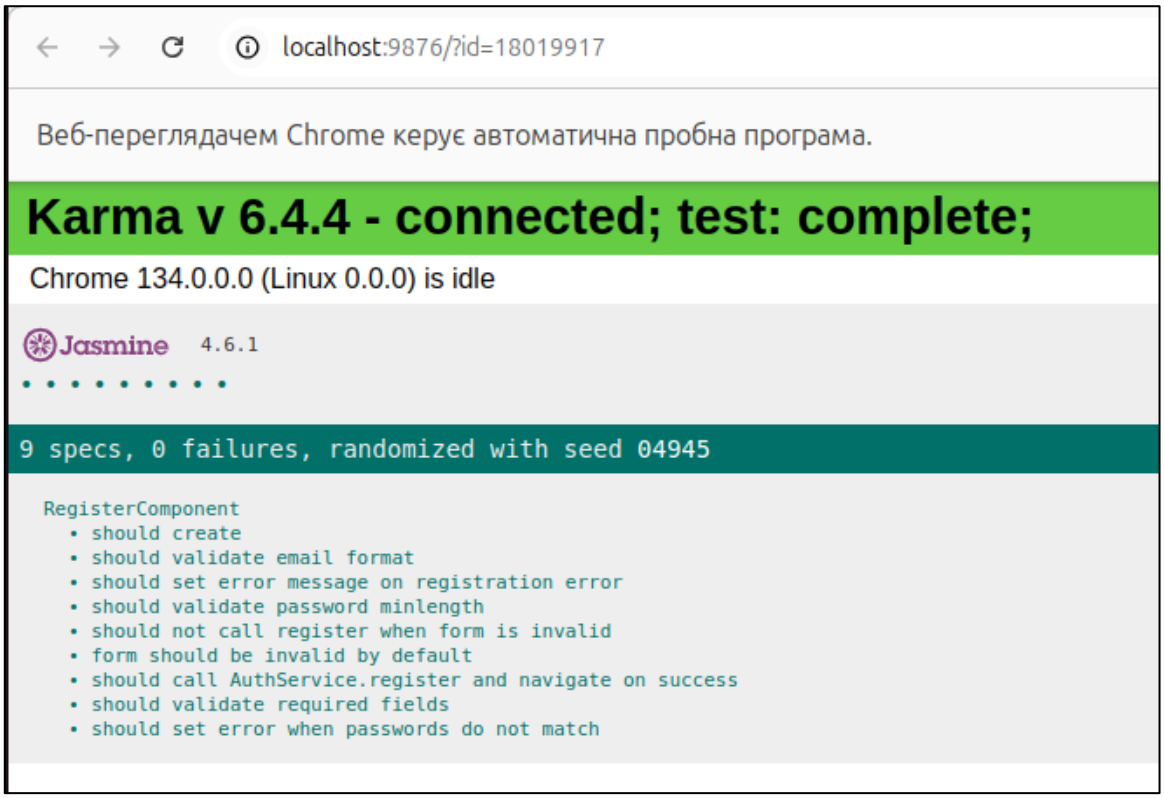


Рисунок 3.13 – Запуск тестів у браузері

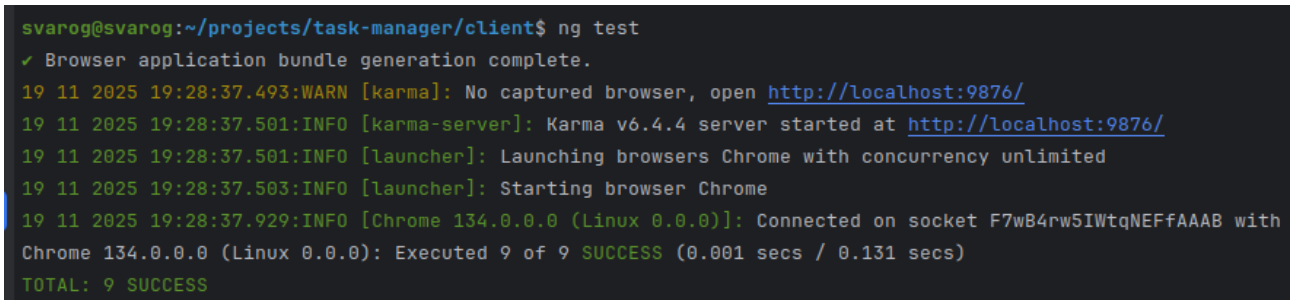


Рисунок 3.14 – Запуск тестів у консолі

3.6 Керівництво користувача

3.6.1 Вхід та реєстрація

Після відкриття вебзастосунку користувач потрапляє на сторінку авторизації. На ній відображається форма входу з полями email та пароль, а також підсказку щодо дій, якщо користувач ще не має облікового запису (рис. 3.15).

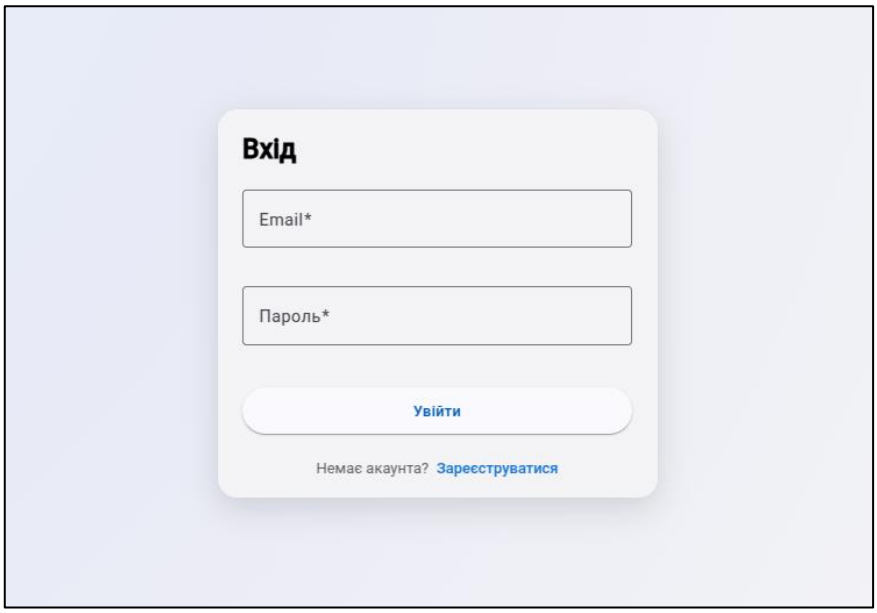


Рисунок 3.15 – Форма входу

Для входу користувачу необхідно ввести свій email та пароль, після чого натиснути на кнопку «Увійти». У разі успіху система перенаправляє його до списку проєктів (рис. 3.16).

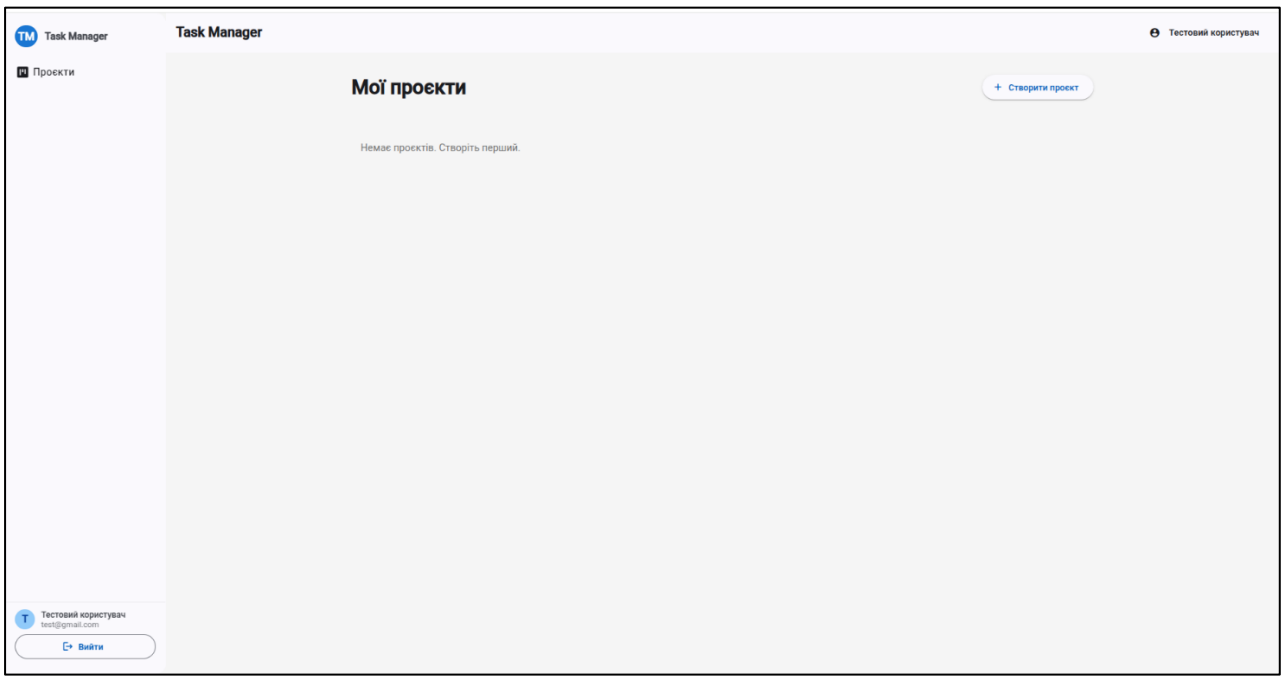


Рисунок 3.16 – Список проєктів

У випадку помилки входу або валідації під формою/полем відображається відповідне повідомлення (рис. 3.17 – 3.18).

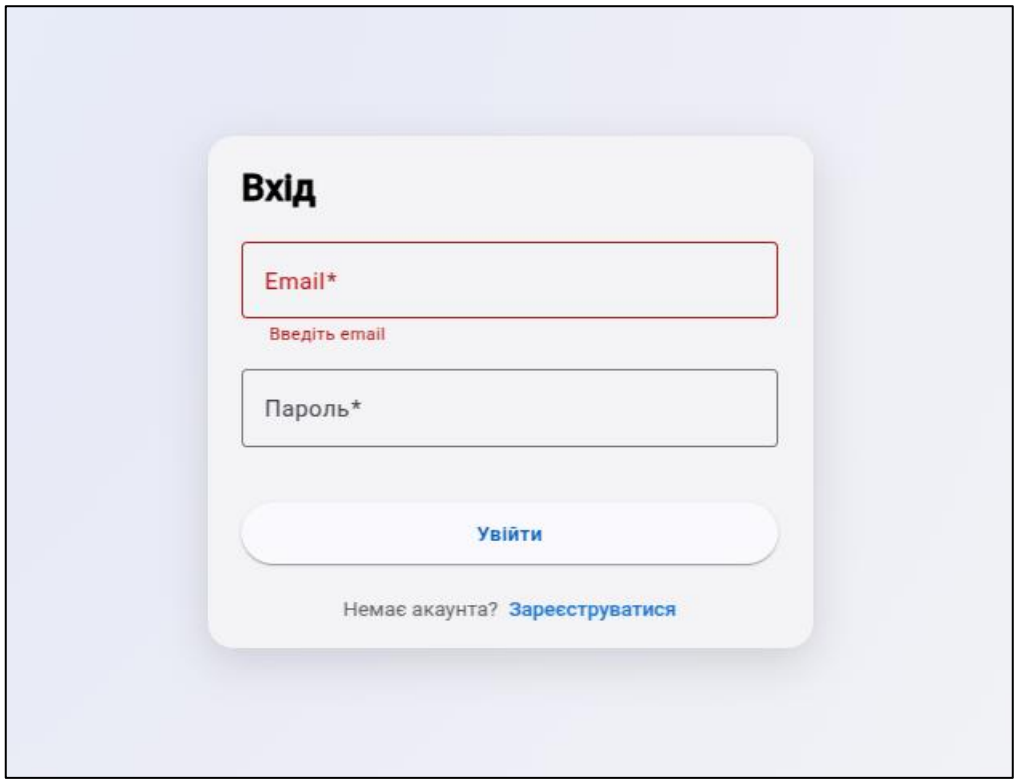


Рисунок 3.17 – Невалідне поле

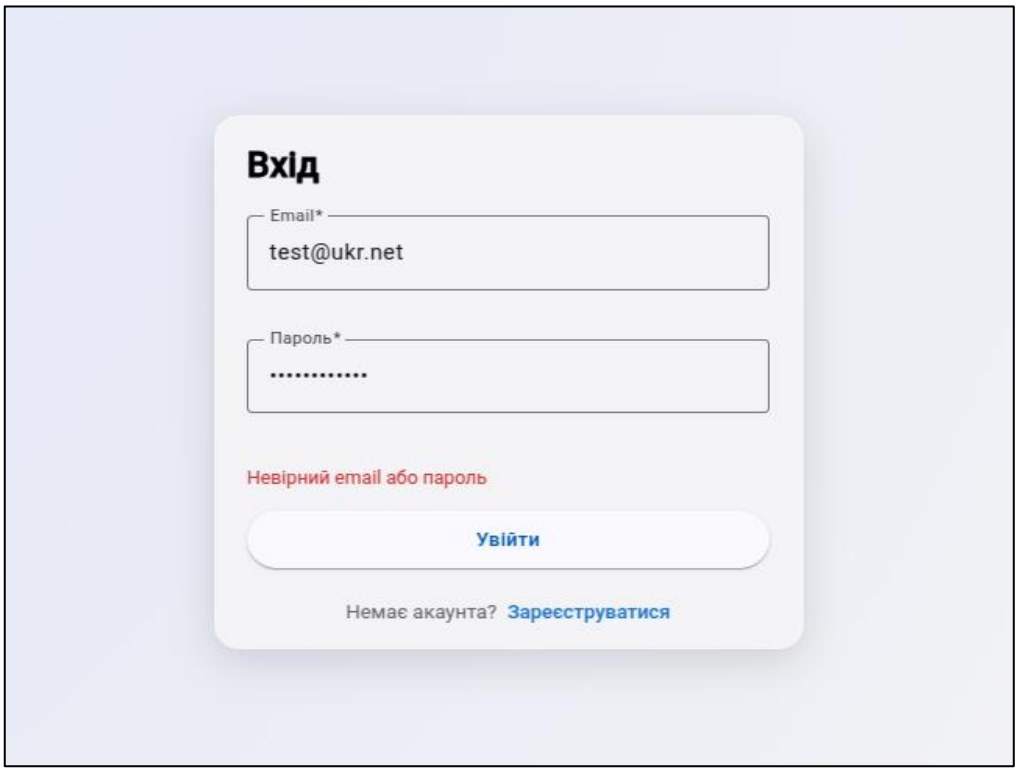


Рисунок 3.18 – Неправильні облікові дані

Для створення нового облікового запису користувач переходить до форми реєстрації за допомогою посилання «Зареєструватися» (рис. 3.19).

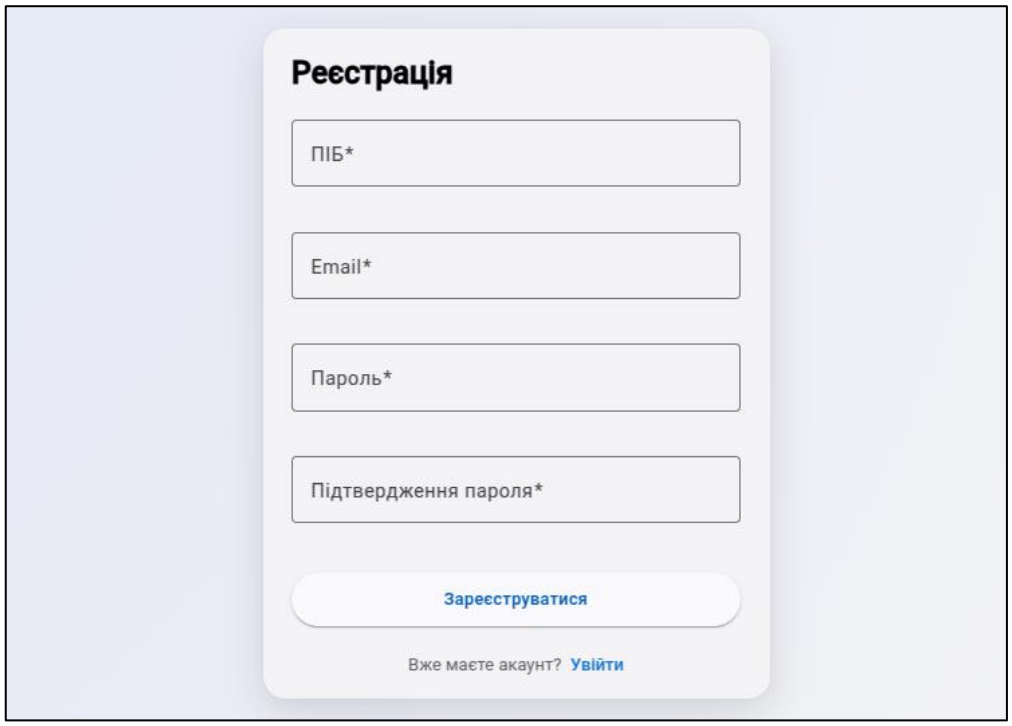


Рисунок 3.19 – Форма реєстрації

Форма містить поля для введення ПІБ, email, пароля та підтвердження пароля. Після коректного заповнення та натискання на кнопку «Зареєструватися» система створює новий обліковий запис і автоматично виконує вхід. Аналогічно до форми входу виконується валідація полів та обробка помилок сервера (рис. 3.20 – 3.21).

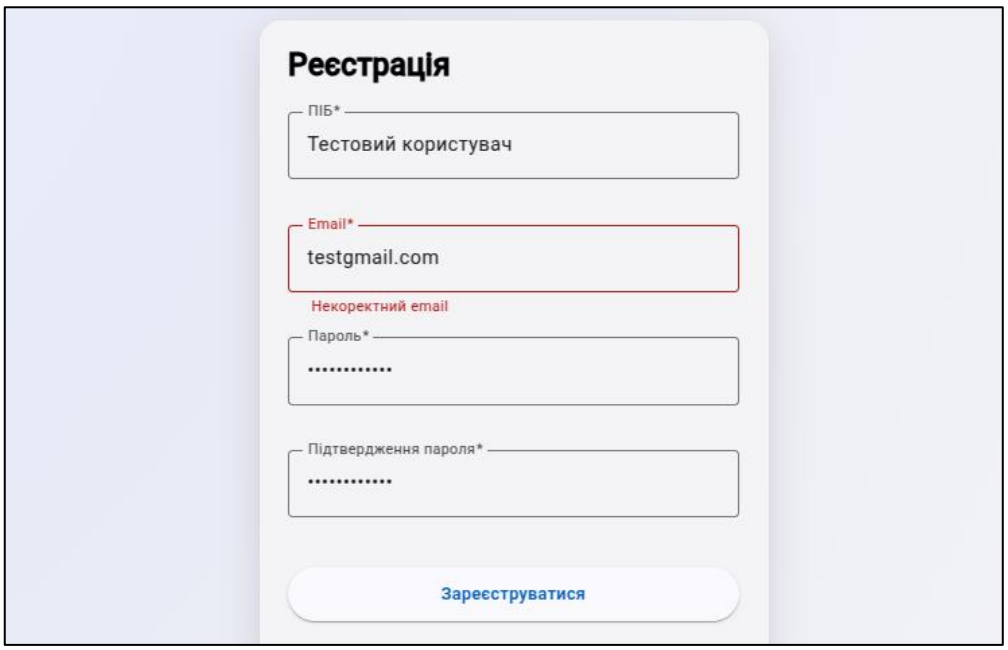


Рисунок 3.20 – Невалідний email

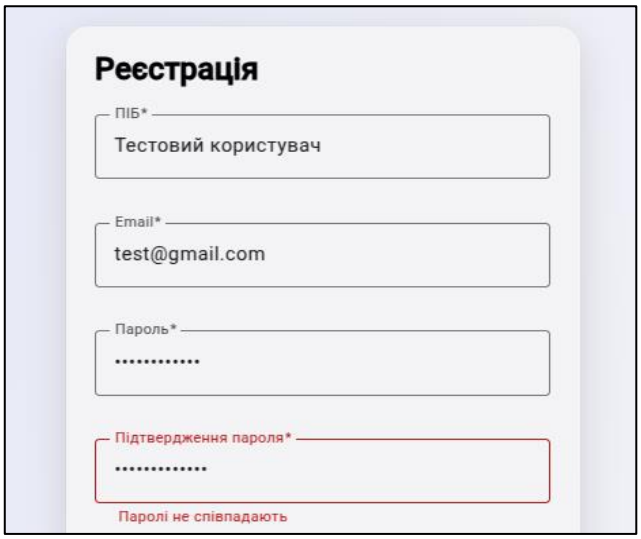


Рисунок 3.21 – Не співпадають паролі

3.6.2 Керування проєктами

Після успішного входу користувач потрапляє на сторінку «Проєкти», яка є головною робочою сторінкою системи (див. рис. 3.16).

У верхній частині сторінки розташована кнопка «Створити проєкт», яка відкриває діалогове вікно для створення нового проєкту (рис. 3.22).

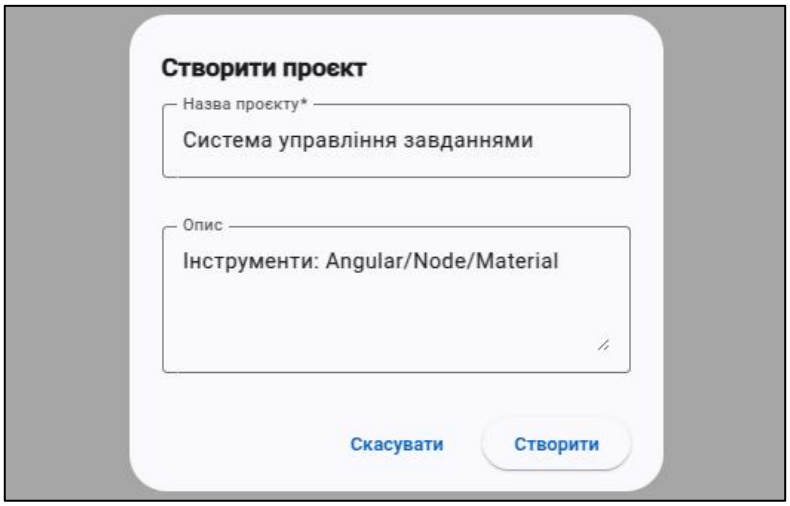


Рисунок 3.22 – Форма створення нового проєкту

Після створення, проєкт відображається на сторінці у адаптивному списку карток проєктів. Кожна картка містить назву проєкту, короткий опис, додаткову

інформацію та функціональні кнопки: «Відкрити», «Редагувати», «Видалити» (рис. 3.23).

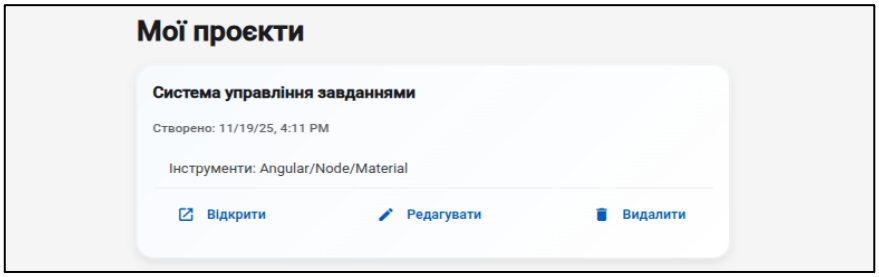


Рисунок 3.23 – Картка проєкту

Клік по назві або по кнопці «Переглянути» відкриває детальний перегляд обраного проєкту з дошкою завдань.

3.6.3 Керування завданнями

Після вибору проєкту користувач переходить на сторінку з канбан-дошкою (рис. 3.24). Дошка розділена на колонки за статусами. У кожній колонці відображаються картки завдань, що належать до відповідного статусу.

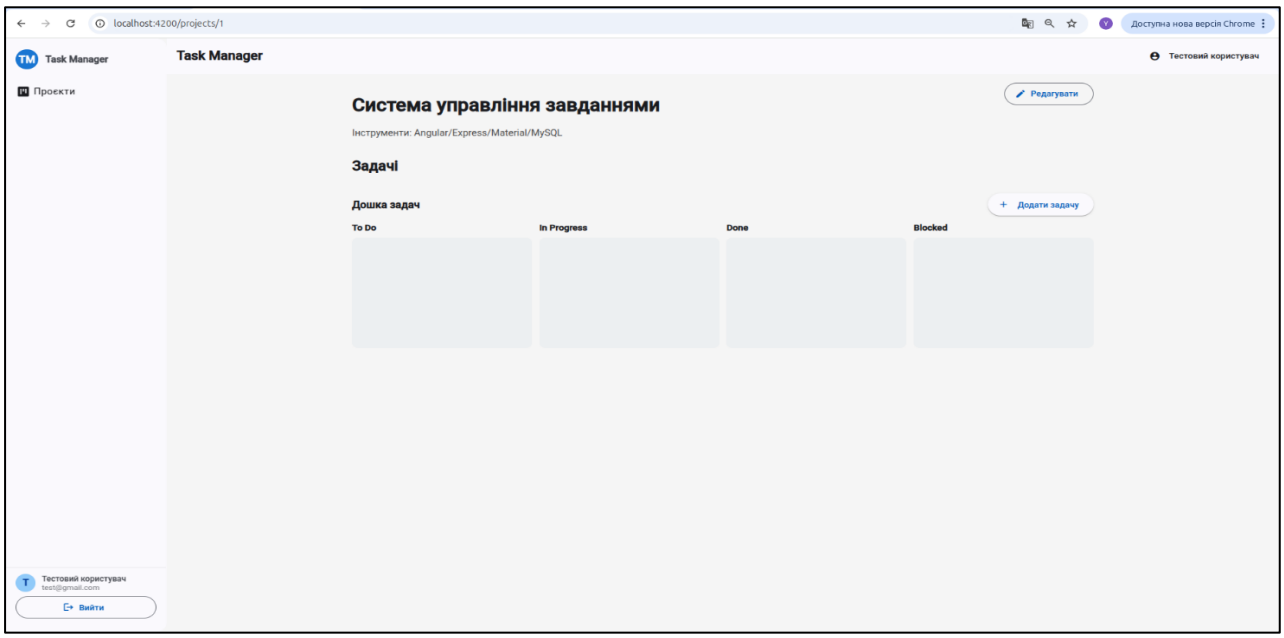


Рисунок 3.24 – Сторінка проєкту

Для додавання нового завдання у проєкт користувач натискає кнопку «Додати задачу» у верхній правій частині дошки. Система відображає модальне вікно з формою створення завдання (див. рис. 3.25).

Форма містить наступні поля:

– назва задачі;
– опис;
– статус;
– пріоритет;
– дедлайн;
– виконавець.

Для вибору виконавця користувач починає вводити email у відповідному полі. Система надсилає запит до серверу та відображає список користувачів, чиї email-адреси відповідають введеному фрагменту. Користувач обирає потрібну адресу зі списку, після чого дані виконавця прив’язуються до завдання.

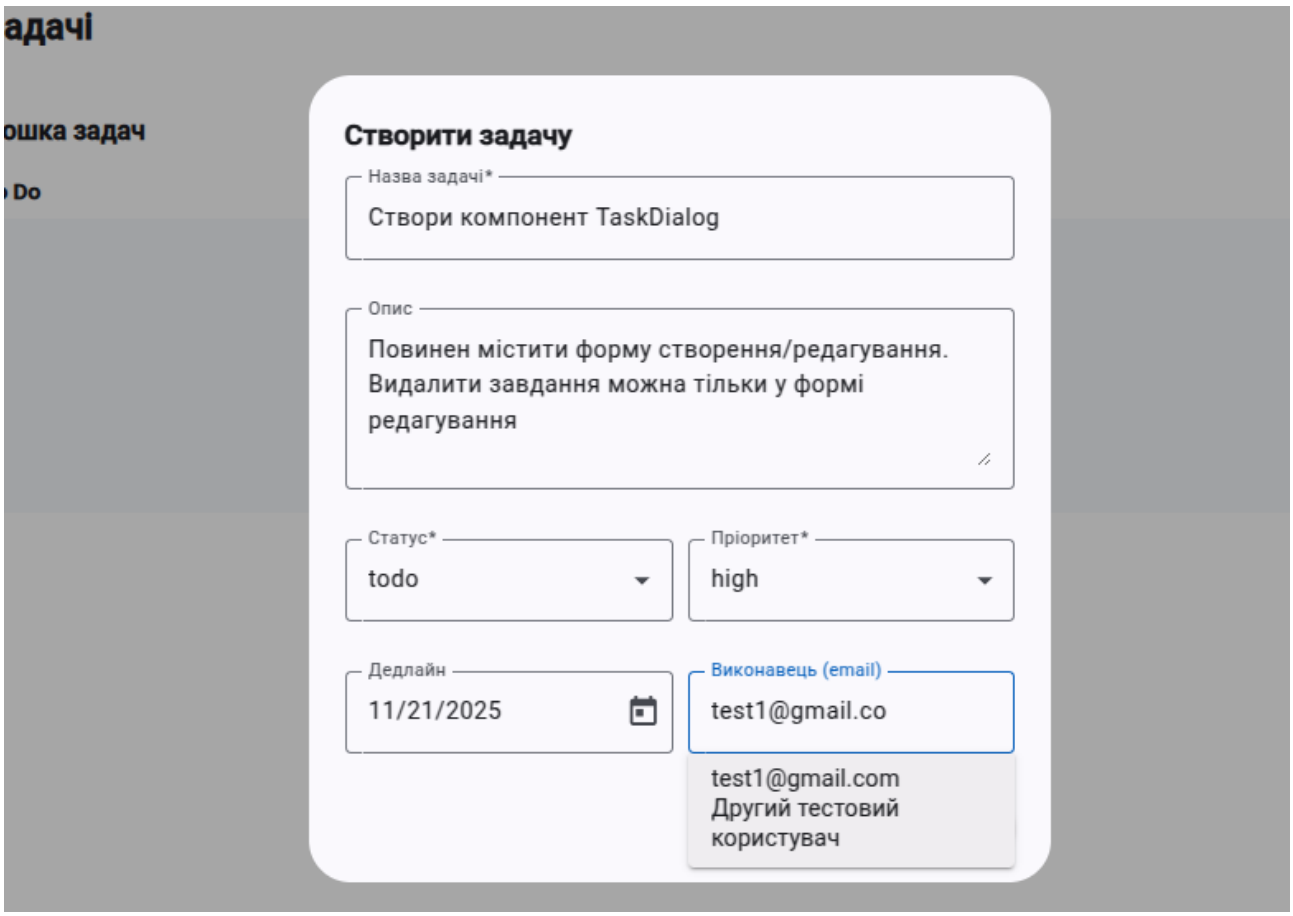


Рисунок 3.25 – Форма створення завдання

Після заповнення форми та натискання «Зберегти» система створює нове завдання і відображає його у відповідній колонці канбан-дошки (див. рис. 3.26). При помилках валідації поля підсвічуються, а під формою відображається відповідне повідомлення. При серверній помилці користувач бачить інформативне повідомлення та може повторити спробу.

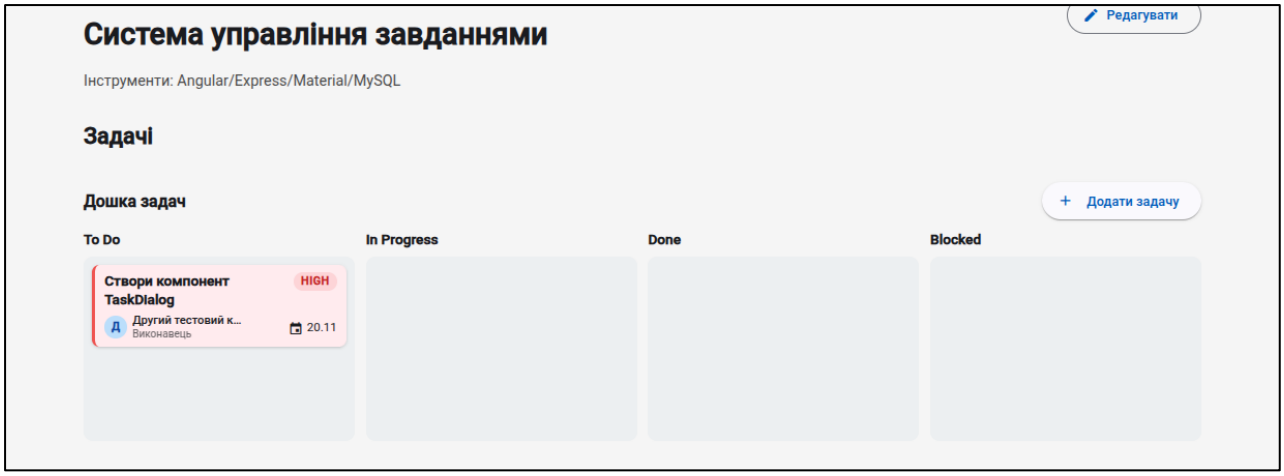


Рисунок 3.26 – Створене завдання

Картка завдання містить скорочену інформацію: назву, пріоритет, виконавця та дедлайн. За кліком на картці відкривається діалог з детальною інформацією та можливістю редагування (рис. 3.27).

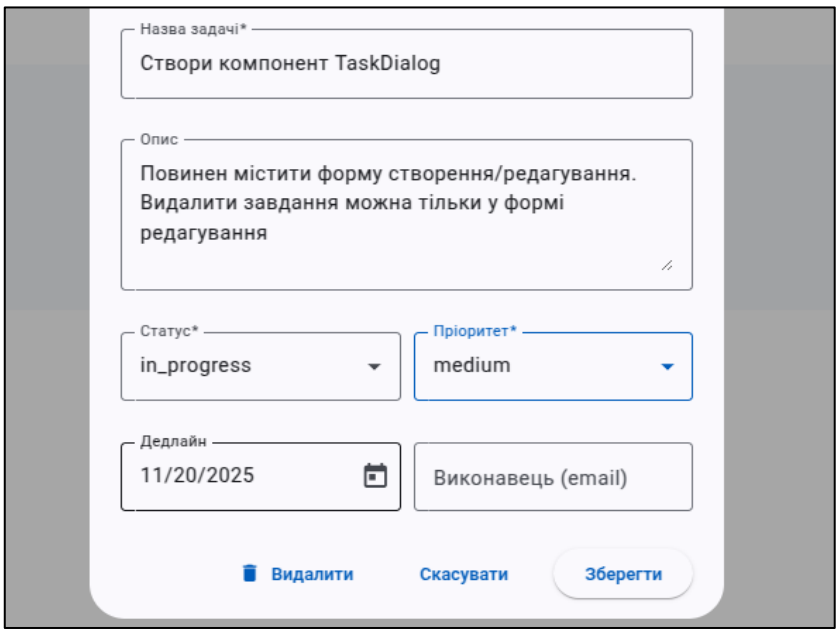


Рисунок 3.27 – Форма редагування

Зміна статусу завдання здійснюється або за допомогою форми, або за допомогою drag-and-drop: користувач перетягує картку із однієї колонки в іншу. Після відпускання картки система автоматично оновлює статус завдання на сервері та візуально фіксує його у новій колонці (рис. 3.28).

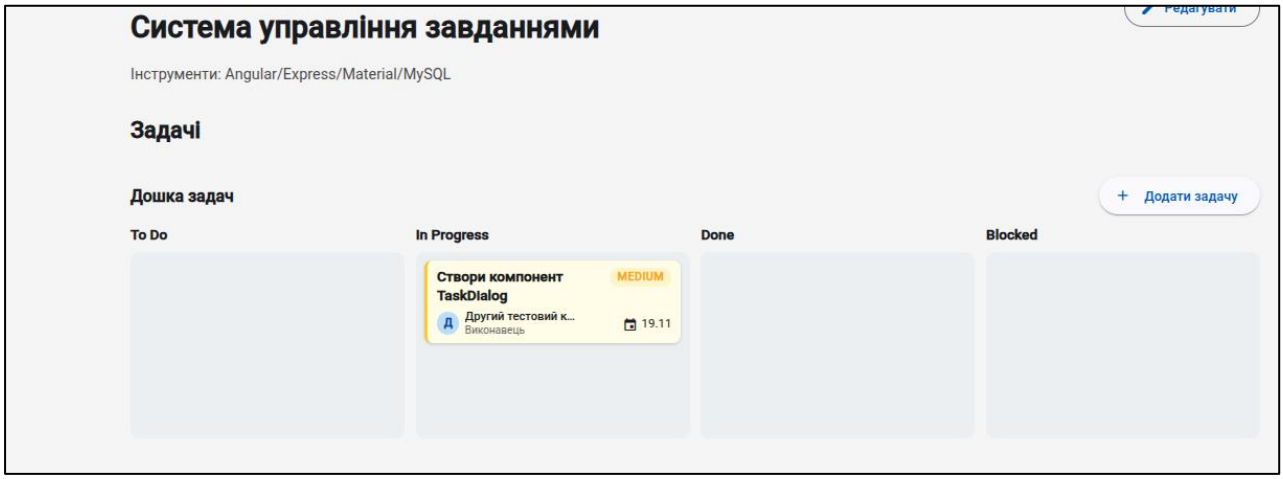


Рисунок 3.28 – Зміна статусу

Зовнішній вигляд картки залежить від того, чи було назначено для завадання виконавця, чи протерміновані дедлайни або який статус має (рис. 3.29).

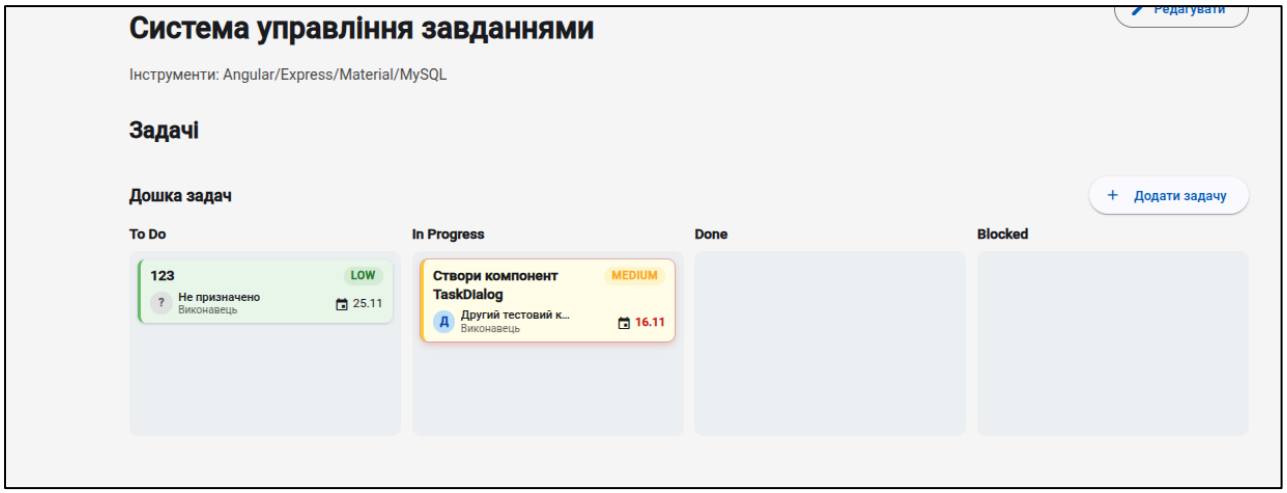


Рисунок 3.29 – Відображення карток

ВИСНОВКИ

У ході виконання роботи розроблено клієнт-серверну систему управління завданнями, що підтримує базові сценарії командної роботи з проєктами: реєстрацію та авторизацію користувачів, створення та редагування проєктів, створення та зміну параметрів завдань, призначення виконавців, зміну статусу задач і роботу з канбан-дошкою. Система надає можливість організувати робочий процес у вигляді візуальної дошки, де завдання групуються за статусами та можуть перетягуватися між колонками, що підвищує наочність контролю виконання.

Відповідно до мети кваліфікаційної роботи реалізовано систему із застосуванням наступних технологій:

– Angular та Angular Material для клієнтської частини, побудови SPA та адаптивного інтерфейсу;
– Node.js та Express.js для серверної частини та реалізації REST API;
– MySQL як реляційної системи керування базами даних для зберігання користувачів, проєктів і завдань.

За поставленими задачами виконано такі етапи:

– сформулювані функціональні та нефункціональні вимоги до системи управління завданнями, а також надано опис предметної області та системи;
– проаналізовані, описані та порівняні готові рішення з нашою системою;
– спроєктовано архітектуру системи;
– реалізовано систему управління завданнями;
– протестовано систему управління завданнями, а також надано повне та покрокове керівництво користувача.

ПЕРЕЛІК ПОСИЛАНЬ

1. Angular Developer Documentation. URL: https://angular.io/docs/ (дата звернення: 17.08.2025).

2. Angular Material Guide. URL: https://material.angular.dev/guides (дата звернення: 26.08.2025).

3. Asana. URL: https://asana.com/ (дата звернення: 02.04.2025).

4. Bampakos A., Deeleman P. Learning Angular: A no-nonsense guide to building web applications with Angular 15. Birmingham : Packt Publishing, 2023. 446 p.

5. ClickUp. URL: https://clickup.com/ (дата звернення: 12.04.2025).

6. Express.js Guide. URL: https://expressjs.com/ (дата звернення: 06.08.2025).

7. Fowler M. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2003. 208 p.

8. Jira. URL: https://www.atlassian.com/software/jira (дата звернення: 02.04.2025).

9. Mrzyglocki F. UML Summarized: Key Concepts and Diagrams for Software Engineers, Architects, and Designers. Phillip Mrzyglocki, 2023. 59 p.

10. MySQL Documentation. URL: https://dev.mysql.com/doc/ (дата звернення: 06.08.2025).

11. Node.js Documentation. URL: https://nodejs.org/docs/latest/api/ (дата звернення: 06.08.2025).

12. Npm bcrypt. URL: https://www.npmjs.com/package/bcrypt (дата звернення: 11.08.2025).

13. Npm Docs. URL: https://docs.npmjs.com/ (дата звернення: 06.08.2025).

14. Npm jsonwebtoken. URL: https://www.npmjs.com/package/jsonwebtoken (дата звернення: 11.08.2025).

15. Postman. API documentation. URL: https://www.postman.com/api-platform/api-documentation/ (дата звернення: 07.08.2025).

16. RxJS Documentation. URL: https://rxjs.dev/api (дата звернення: 19.08.2025).

17. Trello. URL: https://trello.com/ (дата звернення: 14.04.2025).

18. Visual Paradigm. What is Unified Modeling Language (UML)? URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-uml/ (дата звернення: 15.06.2025).

19. Visual Paradigm. Use Case Diagram Guide. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 16.06.2025).

20. Understanding the MVC Pattern with Node.js. URL: https://dev.to/abhishekjaiswal_4896/understanding-the-mvc-pattern-with-nodejs-1fpg (дата звернення: 07.08.2025).

ДОДАТОК А

Angular TaskDialogComponent

```
import { Component, Inject } from '@angular/core';
import {FormBuilder, Validators, FormControl, FormGroup} from '@angular/forms';
import { MAT_DIALOG_DATA, MatDialogRef } from '@angular/material/dialog';
import { Task, TaskPriority, TaskStatus } from '../../../shared/models/task.model';
import { TaskService } from '../task.service';
import { UserService } from '../../../core/services/user.service';
import { User } from '../../../shared/models/user.model';

import { Observable, of } from 'rxjs';
import { debounceTime, distinctUntilChanged, switchMap } from 'rxjs/operators';
import { MatAutocompleteSelectedEvent } from '@angular/material/autocomplete';

export interface TaskDialogData {
  projectId: number;
  task?: Task;
}

@Component({
  selector: 'tm-task-dialog',
  templateUrl: './task-dialog.component.html',
  styleUrls: ['./task-dialog.component.scss'],
  standalone: false
})
export class TaskDialogComponent {
  isEdit!: boolean;
  title = this.isEdit ? 'Редагувати задачу' : 'Створити задачу';

  statuses: TaskStatus[] = ['todo', 'in_progress', 'done', 'blocked'];
  priorities: TaskPriority[] = ['low', 'medium', 'high'];

  loading = false;
  error: string | null = null;

  form: FormGroup;

  assigneeSearchControl = new FormControl<string>('', { nonNullable: true });
  filteredUsers$: Observable<User[]> = of([]);

  constructor(
    private fb: FormBuilder,
    private taskService: TaskService,
    private userService: UserService,
    private dialogRef: MatDialogRef<TaskDialogComponent>,
```

```
    @Inject(MAT_DIALOG_DATA) public data: TaskDialogData
  ) {

    this.form = this.fb.group({
      title: ['', [Validators.required, Validators.maxLength(255)]],
      description: [''],
      status: ['todo' as TaskStatus, Validators.required],
      priority: ['medium' as TaskPriority, Validators.required],
      dueDate: [null as Date | null],
      assigneeId: [null as number | null]
    });
    this.isEdit = !!this.data.task;
    if (this.isEdit && data.task) {
      this.form.patchValue({
        title: data.task.title,
        description: data.task.description || '',
        status: data.task.status,
        priority: data.task.priority,
        dueDate: data.task.dueDate ? new Date(data.task.dueDate) : null,
        assigneeId: data.task.assigneeId ?? null
      });

    }

    this.filteredUsers$ = this.assigneeSearchControl.valueChanges.pipe(
      debounceTime(300),
      distinctUntilChanged(),
      switchMap(value =>
        value && value.length >= 2
          ? this.userService.searchByEmail(value)
          : of([])
      )
    );
  }

  displayUser(user: User | string | null): string {
    if (!user || typeof user === 'string') {
      return user ?? '';
    }
    return `${user.email}`;
  }
q1
  onAssigneeSelected(event: MatAutocompleteSelectedEvent): void {
    const user = event.option.value as User;
    this.form.patchValue({ assigneeId: user.id });
  }

  submit(): void {
    if (this.form.invalid) {
      this.form.markAllAsTouched();
```

```
    return;
  }

  this.loading = true;
  this.error = null;

  const value = this.form.value;
  const dueDateStr = value.dueDate
    ? value.dueDate.toISOString().substring(0, 10)
    : null;

  if (!this.isEdit) {
    this.taskService.createTask({
      projectId: this.data.projectId,
      title: value.title!,
      description: value.description || null,
      status: value.status!,
      priority: value.priority!,
      dueDate: dueDateStr,
      assigneeId: value.assigneeId ?? null
    }).subscribe({
      next: (task) => {
        this.loading = false;
        this.dialogRef.close(task);
      },
      error: (err) => {
        this.loading = false;
        this.error = err.error?.message || 'Помилка створення задачі';
      }
    });
  } else if (this.data.task) {
    this.taskService.updateTask(this.data.task.id, {
      title: value.title!,
      description: value.description || null,
      status: value.status!,
      priority: value.priority!,
      dueDate: dueDateStr,
      assigneeId: value.assigneeId ?? null
    }).subscribe({
      next: (task) => {
        this.loading = false;
        this.dialogRef.close(task);
      },
      error: (err) => {
        this.loading = false;
        this.error = err.error?.message || 'Помилка оновлення задачі';
      }
    });
  }
}
```

```
  deleteTask(): void {
    if (!this.isEdit || !this.data.task) {
      return;
    }

    const confirmed = window.confirm('Видалити це завдання? Дію не можна буде скасувати.');
    if (!confirmed) {
      return;
    }

    this.loading = true;
    this.error = null;

    this.taskService.deleteTask(this.data.task.id).subscribe({
      next: () => {
        this.loading = false;
        this.dialogRef.close({ action: 'deleted', id: this.data.task?.id });
      },
      error: (err) => {
        this.loading = false;
        this.error = err.error?.message || 'Помилка видалення задачі';
      }
    });
  }

  cancel(): void {
    this.dialogRef.close();
  }
}
```

ДОДАТОК Б

Angular TaskService

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Task, TaskPriority, TaskStatus } from '../../shared/models/task.model';
import { Observable, map } from 'rxjs';

interface TaskDto {
  id: number;
  project_id: number;
  creator_id: number;
  assignee_id: number | null;
  title: string;
  description: string | null;
  status: TaskStatus;
  priority: TaskPriority;
  due_date: string | null;
  order_index: number;
  creator_name?: string | null;
  assignee_name?: string | null;
  created_at: string;
  updated_at: string | null;
}

@Injectable({
  providedIn: 'root'
})
export class TaskService {
  private readonly apiUrl = 'http://localhost:4000/api/tasks';

  constructor(private http: HttpClient) {}

  private mapTask(dto: TaskDto): Task {
    return {
      id: dto.id,
      projectId: dto.project_id,
      creatorId: dto.creator_id,
      assigneeId: dto.assignee_id,

      title: dto.title,
      description: dto.description,

      status: dto.status,
      priority: dto.priority,
      dueDate: dto.due_date,
```

```
    orderIndex: dto.order_index,

    creatorName: dto.creator_name ?? undefined,
    assigneeName: dto.assignee_name ?? undefined,

    createdAt: dto.created_at,
    updatedAt: dto.updated_at
  };
}

getTasksByProject(projectId: number): Observable<Task[]> {
  return this.http
    .get<TaskDto[]>(`${this.apiUrl}/project/${projectId}`)
    .pipe(map(dtos => dtos.map(dto => this.mapTask(dto))));
}

createTask(payload: {
  projectId: number;
  title: string;
  description?: string | null;
  status?: TaskStatus;
  priority?: TaskPriority;
  dueDate?: string | null;
  assigneeId?: number | null;
}): Observable<Task> {
  return this.http
    .post<TaskDto>(this.apiUrl, payload)
    .pipe(map(dto => this.mapTask(dto)));
}

updateTask(id: number, payload: Partial<Task>): Observable<Task> {
  const body: any = {
    title: payload.title,
    description: payload.description,
    status: payload.status,
    priority: payload.priority,
    dueDate: payload.dueDate,
    assigneeId: payload.assigneeId
  };

  return this.http
    .put<TaskDto>(`${this.apiUrl}/${id}`, body)
    .pipe(map(dto => this.mapTask(dto)));
}

deleteTask(id: number): Observable<void> {
  return this.http.delete<void>(`${this.apiUrl}/${id}`);
}

changeStatus(id: number, status: TaskStatus, orderIndex?: number): Observable<Task> {
```

```
    return this.http
      .patch<TaskDto>(`${this.apiUrl}/${id}/status`, { status, orderIndex })
      .pipe(map(dto => this.mapTask(dto)));
  }

  changeAssignee(id: number, assigneeId: number | null): Observable<Task> {
    return this.http
      .patch<TaskDto>(`${this.apiUrl}/${id}/assignee`, { assigneeId })
      .pipe(map(dto => this.mapTask(dto)));
  }
}
```

ДОДАТОК В

Express Task Controller

```
const Task = require('../models/task.model');
const Project = require('../models/project.model');
const User = require('../models/user.model');

exports.getTasksByProject = async (req, res, next) => {
  try {
    const userId = req.user.id;
    const projectId = req.params.projectId;

    const project = await Project.findByIdAndOwner(projectId, userId);
    if (!project) {
      return res
        .status(404)
        .json({ message: 'Проєкт не знайдено або немає доступу' });
    }

    const tasks = await Task.getTasksByProject(projectId);
    res.json(tasks);
  } catch (err) {
    next(err);
  }
};

exports.createTask = async (req, res, next) => {
  try {
    const userId = req.user.id;
    const {
      projectId,
      title,
      description,
      status,
      priority,
      dueDate,
      assigneeId
    } = req.body;

    if (!projectId || !title) {
      return res
        .status(400)
        .json({ message: 'projectId і title є обов’язковими' });
    }

    const project = await Project.findByIdAndOwner(projectId, userId);
```

```
        if (!project) {
            return res
                .status(404)
                .json({ message: 'Проєкт не знайдено або немає доступу' });
        }

        const task = await Task.createTask({
            projectId,
            creatorId: userId,
            assigneeId,
            title,
            description,
            status,
            priority,
            dueDate
        });

        res.status(201).json(task);
    } catch (err) {
        next(err);
    }
};

exports.updateTask = async (req, res, next) => {
    try {
        const userId = req.user.id;
        const taskId = req.params.id;
        const {
            title,
            description,
            status,
            priority,
            dueDate,
            assigneeId
        } = req.body;

        const existing = await Task.findWithProjectOwner(taskId);
        if (!existing) {
            return res.status(404).json({ message: 'Завдання не знайдено' });
        }

        if (existing.owner_id !== userId) {
            return res
                .status(403)
                .json({ message: 'Немає прав для редагування цього завдання' });
        }

        const updated = await Task.updateTask(taskId, {
            title: title || existing.title,
            description: description ?? existing.description,
```

```
      status: status || existing.status,
      priority: priority || existing.priority,
      dueDate:
        typeof dueDate !== 'undefined' ? dueDate : existing.due_date,
      assigneeId:
        typeof assigneeId !== 'undefined'
          ? assigneeId
          : existing.assignee_id
    });

    res.json(updated);
  } catch (err) {
    next(err);
  }
};

exports.deleteTask = async (req, res, next) => {
  try {
    const userId = req.user.id;
    const taskId = req.params.id;

    const existing = await Task.findWithProjectOwner(taskId);
    if (!existing) {
      return res.status(404).json({ message: 'Завдання не знайдено' });
    }

    if (existing.owner_id !== userId) {
      return res
        .status(403)
        .json({ message: 'Немає прав для видалення цього завдання' });
    }

    await Task.deleteTask(taskId);
    res.status(204).send();
  } catch (err) {
    next(err);
  }
};

exports.changeStatus = async (req, res, next) => {
  try {
    const userId = req.user.id;
    const taskId = req.params.id;
    const { status, orderIndex } = req.body;

    if (!status) {
      return res.status(400).json({ message: 'Поле status є обов’язковим' });
    }

    const existing = await Task.findWithProjectOwner(taskId);
```

```
    if (!existing) {
      return res.status(404).json({ message: 'Завдання не знайдено' });
    }

    if (existing.owner_id !== userId) {
      return res
        .status(403)
        .json({ message: 'Немає прав для зміни статусу' });
    }

    const updated = await Task.updateStatus(taskId, {
      status,
      orderIndex:
        typeof orderIndex !== 'undefined'
          ? orderIndex
          : existing.order_index
    });

    res.json(updated);
  } catch (err) {
    next(err);
  }
};

exports.changeAssignee = async (req, res, next) => {
  try {
    const userId = req.user.id;
    const taskId = req.params.id;
    const { assigneeId } = req.body;

    const existing = await Task.findWithProjectOwner(taskId);
    if (!existing) {
      return res.status(404).json({ message: 'Завдання не знайдено' });
    }

    if (existing.owner_id !== userId) {
      return res
        .status(403)
        .json({ message: 'Немає прав для передачі цього завдання' });
    }

    if (assigneeId) {
      const user = await User.findById(assigneeId);
      if (!user) {
        return res
          .status(400)
          .json({ message: 'Вказаний користувач не існує' });
      }
    }
```

```
        const updated = await Task.updateAssignee(taskId, assigneeId || null);
        res.json(updated);
    } catch (err) {
        next(err);
    }
};
```

ДОДАТОК Г

Express Task Model

```
const pool = require('../config/db');

async function getTasksByProject(projectId) {
   const [rows] = await pool.query(
      `SELECT t.*,
         c.full_name AS creator_name,
         a.full_name AS assignee_name
    FROM tasks t
    LEFT JOIN users c ON t.creator_id = c.id
    LEFT JOIN users a ON t.assignee_id = a.id
    WHERE t.project_id = ?
    ORDER BY t.status, t.order_index`,
      [projectId]
   );
   return rows;
}

async function createTask({
                        projectId,
                        creatorId,
                        assigneeId,
                        title,
                        description,
                        status,
                        priority,
                        dueDate
                     }) {
   const [result] = await pool.query(
      `INSERT INTO tasks
    (project_id, creator_id, assignee_id, title, description, status, priority, due_date, order_index)
    VALUES (?, ?, ?, ?, ?, ?, ?, ?, 0)`,
      [
         projectId,
         creatorId,
         assigneeId || null,
         title,
         description || null,
         status || 'todo',
         priority || 'medium',
         dueDate || null
      ]
   );
```

```
  const [rows] = await pool.query(
    'SELECT * FROM tasks WHERE id = ?',
    [result.insertId]
  );

  return rows[0] || null;
}

async function findById(taskId) {
  const [rows] = await pool.query(
    'SELECT * FROM tasks WHERE id = ?',
    [taskId]
  );
  return rows[0] || null;
}

async function findWithProjectOwner(taskId) {
  const [rows] = await pool.query(
    `SELECT t.*, p.owner_id
   FROM tasks t
   JOIN projects p ON t.project_id = p.id
   WHERE t.id = ?`,
    [taskId]
  );
  return rows[0] || null;
}

async function updateTask(taskId, {
  title,
  description,
  status,
  priority,
  dueDate,
  assigneeId
}) {
  await pool.query(
    `UPDATE tasks
   SET title = ?, description = ?, status = ?, priority = ?, due_date = ?, assignee_id = ?, updated_at = NOW()
   WHERE id = ?`,
    [
      title,
      description,
      status,
      priority,
      dueDate,
      assigneeId || null,
      taskId
    ]
  );
```

```
  const [rows] = await pool.query(
    'SELECT * FROM tasks WHERE id = ?',
    [taskId]
  );
  return rows[0] || null;
}

async function updateStatus(taskId, { status, orderIndex }) {
  await pool.query(
    `UPDATE tasks
   SET status = ?, order_index = ?, updated_at = NOW()
   WHERE id = ?`,
    [status, orderIndex, taskId]
  );

  const [rows] = await pool.query(
    'SELECT * FROM tasks WHERE id = ?',
    [taskId]
  );
  return rows[0] || null;
}

async function updateAssignee(taskId, assigneeId) {
  await pool.query(
    `UPDATE tasks
   SET assignee_id = ?, updated_at = NOW()
   WHERE id = ?`,
    [assigneeId || null, taskId]
  );

  const [rows] = await pool.query(
    'SELECT * FROM tasks WHERE id = ?',
    [taskId]
  );
  return rows[0] || null;
}

async function deleteTask(taskId) {
  await pool.query(
    'DELETE FROM tasks WHERE id = ?',
    [taskId]
  );
}

module.exports = {
  getTasksByProject,
  createTask,
  findById,
  findWithProjectOwner,
```

```
    updateTask,
    updateStatus,
    updateAssignee,
    deleteTask
};
```

ДОДАТОК Д

Unit-test Register Component

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { ReactiveFormsModule } from '@angular/forms';
import { NO_ERRORS_SCHEMA } from '@angular/core';
import { of, throwError } from 'rxjs';

import { RegisterComponent } from './register.component';
import { AuthService } from '../../../core/services/auth.service';
import { Router } from '@angular/router';

describe('RegisterComponent', () => {
  let component: RegisterComponent;
  let fixture: ComponentFixture<RegisterComponent>;
  let authServiceSpy: jasmine.SpyObj<AuthService>;
  let routerSpy: jasmine.SpyObj<Router>;

  beforeEach(async () => {
    authServiceSpy = jasmine.createSpyObj<AuthService>('AuthService', ['register']);
    routerSpy = jasmine.createSpyObj<Router>('Router', ['navigateByUrl']);

    await TestBed.configureTestingModule({
      declarations: [RegisterComponent],
      imports: [ReactiveFormsModule],
      providers: [
        { provide: AuthService, useValue: authServiceSpy },
        { provide: Router, useValue: routerSpy }
      ],
      schemas: [NO_ERRORS_SCHEMA]
    }).compileComponents();
  });

  beforeEach(() => {
    fixture = TestBed.createComponent(RegisterComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('form should be invalid by default', () => {
    expect(component.form.valid).toBeFalse();
    expect(component.fullName?.valid).toBeFalse();
```

```
  expect(component.email?.valid).toBeFalse();
  expect(component.password?.valid).toBeFalse();
  expect(component.confirmPassword?.valid).toBeFalse();
});

it('should validate required fields', () => {
  component.form.setValue({
    fullName: '',
    email: '',
    password: '',
    confirmPassword: ''
  });

  expect(component.form.valid).toBeFalse();
  expect(component.fullName?.hasError('required')).toBeTrue();
  expect(component.email?.hasError('required')).toBeTrue();
  expect(component.password?.hasError('required')).toBeTrue();
  expect(component.confirmPassword?.hasError('required')).toBeTrue();
});

it('should validate email format', () => {
  component.form.patchValue({
    fullName: 'Test User',
    email: 'not-an-email',
    password: '123456',
    confirmPassword: '123456'
  });

  expect(component.email?.hasError('email')).toBeTrue();
  expect(component.form.valid).toBeFalse();
});

it('should validate password minlength', () => {
  component.form.patchValue({
    fullName: 'Test User',
    email: 'test@example.com',
    password: '123',
    confirmPassword: '123'
  });

  expect(component.password?.hasError('minlength')).toBeTrue();
  expect(component.form.valid).toBeFalse();
});

it('should set error when passwords do not match', () => {
  component.form.patchValue({
    fullName: 'Test User',
    email: 'test@example.com',
    password: '123456',
    confirmPassword: '654321'
```

```
  });

  component.form.updateValueAndValidity();

  expect(component.confirmPassword?.hasError('passwordMismatch')).toBeTrue();
  expect(component.form.valid).toBeFalse();
});

it('should call AuthService.register and navigate on success', () => {
  const fullName = 'Test User';
  const email = 'test@example.com';
  const password = '123456';

  component.form.setValue({
    fullName,
    email,
    password,
    confirmPassword: password
  });

  authServiceSpy.register.and.returnValue(of({} as any));

  component.submit();

  expect(component.loading).toBeFalse();
  expect(component.error).toBeNull();

  expect(authServiceSpy.register).toHaveBeenCalledTimes(1);
  expect(authServiceSpy.register).toHaveBeenCalledWith(email, password, fullName);

  expect(routerSpy.navigateByUrl).toHaveBeenCalledWith('/projects');
});

it('should not call register when form is invalid', () => {
  component.form.setValue({
    fullName: '',
    email: 'bad-email',
    password: '123',
    confirmPassword: '321'
  });

  component.submit();

  expect(authServiceSpy.register).not.toHaveBeenCalled();
  expect(component.loading).toBeFalse();
});

it('should set error message on registration error', () => {
  const fullName = 'Test User';
  const email = 'test@example.com';
```

```
    const password = '123456';

    component.form.setValue({
      fullName,
      email,
      password,
      confirmPassword: password
    });

    const backendError = { error: { message: 'Email already taken' } };
    authServiceSpy.register.and.returnValue(throwError(() => backendError));

    component.submit();

    expect(component.loading).toBeFalse();
    expect(component.error).toBe('Email already taken');
    expect(routerSpy.navigateByUrl).not.toHaveBeenCalled();
  });
});
```