Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»

ВЕБ-ЗАСТОСУНОК ДЛЯ УПРАВЛІННЯ ПРОЕКТАМИ В СТУДЕНТСЬКОМУ СЕРЕДОВИЩІ

WEB APPLICATION FOR PROJECT MANAGEMENT IN THE STUDENT ENVIRONMENT

спеціальність 122 Комп'ютерні науки
освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КН-41
Бадзюнь Максим Анатолійович

_____
(підпис)

Керівник: к.т.н., доцент
Ліщина Валерій Олександрович

_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«2» червня 2026 р.
Гарант ОП: д.п.н., професор
Тулашвілі Юрій Йосипович

_____
(підпис)

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти: *бакалавр*
Галузь знань*: 12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Валерій ЛІЩИНА
«16» січня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ПЕРШОГО (БАКАЛАВРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Бадзюнь Максим Анатолійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи *Веб-застосунок для управління проектами в студентському середовищі*
Керівник: *к.т.н., доцент Ліщина В.О.*
затверджено наказом закладу вищої освіти від «16» січня 2026 р. № 32/01-02
2. Строк подання кваліфікаційної роботи *«02» червня 2026 р.*
3.Вихідні дані до роботи: *аналітичне дослідження систем управління проектами.*
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
Вступ
Розділ 1. Аналіз предметної області
Розділ 2. Структурна реалізація об'єкта проектування
Розділ 3. Програмна реалізація об'єкта проектування
Розділ 4. Спеціальні розрахунки
Висновки
Додатки
5. Перелік графічного матеріалу :_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Ліщина В. О.*		
Теоретичне дослідження	*Ліщина В. О.*		
Практична реалізація	*Ліщина В. О.*		
Спеціальні розрахунки	*Ліщина В. О.*		
Показник запозичень тексту	_____%		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Тулашвілі Ю.Й.*		

7. Дата видачі завдання *«16» січня 2026 р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної Роботи	Строк виконання етапів роботи	Примітка
1.	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 10.02.2026р*	
2.	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 24.02.2026р.*	
3.	*Розробити функціональну схему роботи програмного продукту*	*до 10.03.2026р*	
4.	*Описати засоби розробки об'єкта проектування*	*до 31.03.2026р.*	
5.	*Практична реалізація об'єкта проектування*	*до 05.05.2026р.*	
7.	*Провести спеціальні розрахунки*	*до 21.05.2026р.*	
8.	*Здача чистового варіанту кваліфікаційної роботи на кафедру*	*до 02.06.2026р.*	

Здобувач вищої освіти _____ Максим БАДЗЮНЬ

Керівник роботи _____ Валерій ЛІЩИНА

АНОТАЦІЯ

Бадзюнь М. А. Веб-застосунок для управління проектами в студентському середовищі. Рукопис.

Кваліфікаційна робота бакалавра ОП «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2026.

У роботі розглянуто веб-застосунок для управління проєктами в студентському середовищі, призначений для автоматизації процесів, пов'язаних із розрізненою комунікацією, контроль строків і розподіл відповідальності. Обґрунтовано актуальність теми, визначено основні вимоги до системи, спроєктовано архітектуру, модель даних і користувацькі сценарії. Реалізована функціональність охоплює проєкти, команди, завдання, статуси, календар, коментарі та звіти. Проведено тестування основних сценаріїв.

Ключові слова: веб-застосунок, проєктний менеджмент, студентське середовище, завдання, командна робота.

ABSTRACT

Maksym Badzun'. Web application for project management in the student environment. Manuscript.

Bachelor's qualifying thesis of the "Computer Sciences". Lutsk National Technical University. Lutsk, 2026.

The thesis considers a web application for project management in the student environment intended to automate processes related to fragmented communication, deadline control, and responsibility distribution. The relevance of the topic is substantiated, and the main system requirements, architecture, data model, and user scenarios are defined.

Keywords: web application, project management, student environment, tasks, teamwork.

ЗМІСТ

ВСТУП

Актуальність теми. У сучасних реаліях підготовки фахівців у галузі інформаційних технологій особливе місце посідає командна проєктна діяльність здобувачів вищої освіти. Виконання курсових проєктів, командних лабораторних робіт та участь у наукових дослідженнях вимагають від студентів не лише технічних знань, але й розвинених навичок самоорганізації, спільної роботи та планування. Проте специфіка студентського середовища суттєво відрізняється від комерційної розробки, склад команд часто змінюється, завдання розподіляються в умовах нерівномірного навантаження під час семестру, а роль координатора або лідера зазвичай виконує один із самих студентів, який не має значного досвіду управління. Окрім того, важливим елементом є участь у процесі академічного керівника або викладача, який здійснює контроль за етапами виконання проєкту, не втручаючись безпосередньо у внутрішні комунікаційні канали студентів.

Сучасні комерційні інструменти управління проєктами, такі як Jira, Asana або Trello, часто є надмірними для невеликих студентських команд через високу вартість ліцензій, складність налаштування бізнес-процесів або перевантаженість зайвими корпоративними функціями. Водночас найпростіші безкоштовні таск-трекери не пропонують специфічних інструментів для викладацького моніторингу та інтеграції з академічним календарем. Таким чином, виникає об’єктивна потреба у створенні спеціалізованого веборієнтованого рішення, яке поєднуватиме гнучкість Kanban-дошок для студентів, автоматизовану інтеграцію зі сховищами коду через GitHub Webhooks для фіксації реального прогресу та окремий інтерфейс моніторингу для викладача. Це дозволить оптимізувати процеси координації, підвищити прозорість розподілу обов’язків та забезпечити об’єктивність оцінювання внеску кожного учасника.

Метою кваліфікаційної роботи є підвищення ефективності командної взаємодії студентів шляхом розроблення та впровадження веборієнтованого

застосунку для управління проєктами в студентському середовищі, який забезпечує автоматизоване відстеження активності розробки, гнучкий розподіл завдань і прозорий моніторинг прогресу з боку кураторів.

Для досягнення поставленої мети необхідно розв’язати такі основні завдання:

1) Проаналізувати процеси організації та управління студентськими проєктами та дослідити наявні програмні аналоги й інструменти координації команд.

2) Сформувати вимоги до спеціалізованого веб-застосунку з урахуванням ролей студентів, лідерів команд та академічних керівників.

3) Спроєктувати архітектуру системи, логіку взаємодії клієнтської та серверної частин, а також структуру реляційної бази даних.

4) Розробити програмні модулі автентифікації користувачів, управління проєктами, інтерактивної Kanban-дошки завдань та обробки подій GitHub Webhook.

5) Реалізувати аналітичний модуль для побудови графіків згорання завдань (Burndown Charts) та звітів про індивідуальний внесок учасників команди.

6) Провести тестування розробленого програмного забезпечення на контрольних сценаріях використання, оцінити швидкодію бази даних та запропонувати схему розгортання застосунку.

Об’єктом дослідження є процеси спільного планування, координації та контролю виконання завдань у ході реалізації студентських командних проєктів.

Предметом дослідження є методи, технології та програмні засоби розроблення веборієнтованого застосунку для гнучкого управління проєктами в студентському середовищі на основі методології Agile (Kanban) з автоматизацією відстеження коду.

Методологічною основою роботи є принципи системного аналізу, теорія проєктування інформаційних систем, об’єктно-орієнтоване моделювання, методології гнучкого управління розробкою ПЗ (Agile, Kanban, Scrum), архітектурні шаблони побудови веб-застосунків (MVC), методи реляційного

моделювання баз даних, а також методи функціонального та інтеграційного тестування.

Практична цінність отриманих результатів полягає у створенні відкритого та легко розгортаного веб-застосунку, який може безпосередньо використовуватись у закладах вищої освіти під час вивчення дисциплін, що передбачають виконання командних курсових або лабораторних робіт. Застосування створеного інструменту дозволяє зменшити обсяг рутинних дій при контролі дедлайнів, усунути ризики втрати домовленостей у чатах та надати викладачам об’єктивні дані щодо активності кожного студента на основі комітів у GitHub.

РОЗДІЛ 1
АНАЛІЗ УПРАВЛІННЯ СТУДЕНТСЬКИМИ ПРОЄКТАМИ
1.1 Студентська проєктна діяльність як предмет автоматизації

Організація спільної роботи студентів над навчальними та науково-прикладними проєктами є важливим елементом підготовки фахівців у сучасних закладах вищої освіти. Командний підхід дозволяє здобувачам вищої освіти не лише опановувати практичні інструменти розроблення програмного забезпечення, але й набувати гнучких навичок (soft skills), серед яких: розподіл відповідальності, комунікація, вирішення конфліктів, планування часу та презентація результатів. Проте специфіка студентського середовища накладає суттєві обмеження на процеси управління проєктами, що робить використання стандартних комерційних методологій у чистому вигляді малоефективним.

Першою важливою особливістю студентських команд є високий рівень нерівномірності залученості учасників. На відміну від промислових IT-компаній, де кожен розробник мотивований фінансово та пов’язаний трудовим договором, у студентських проєктах мотивація базується на отриманні балів, особистому інтересі або бажанні сформувати портфоліо. Це часто призводить до ситуації, коли один або два активні учасники команди виконують основну частину технічної роботи, тоді як інші займаються другорядними завданнями або взагалі уникають активної участі. Без прозорого інструменту відстеження внеску кожного розробника викладачеві вкрай складно об’єктивно оцінити індивідуальні результати під час захисту роботи [1].

Другою особливістю є обмежений життєвий цикл проєктів та жорстка прив’язка до академічного календаря. Студентський проєкт зазвичай триває від кількох тижнів до одного навчального семестру (3-4 місяці). Контрольні точки (модульні тижні, проміжні звіти) чітко визначені розкладом навчального процесу. Це вимагає від команди оперативного планування та швидкого входження у робочий ритм. Будь-які затримки на етапі ініціалізації проєкту або розподілу завдань критично впливають на підсумковий результат.

Третьою проблемою є розпорошеність каналів комунікації та координації дій. Студенти схильні використовувати різноманітні месенджери (Telegram, Discord, Viber) для обговорення завдань, хмарні диски (Google Drive) для обміну файлами та GitHub для спільної роботи над кодом. За відсутності єдиного інтегруючого середовища важлива інформація про статус виконання окремих робіт, коригування вимог чи дедлайнів втрачається у нескінченних стрічках чатів. Виникає дублювання зусиль, коли кілька виконавців одночасно намагаються розв’язати одну й ту саму проблему, або, навпаки, критичні завдання залишаються невиконаними через неправильне розуміння зон відповідальності [2].

Для вирішення цих проблем доцільно застосувати гнучкі методології управління розробкою програмного забезпечення (Agile), адаптовані під потреби академічного процесу. Найбільш прийнятним інструментом є метод Kanban, який базується на візуалізації потоку робіт за допомогою спеціальних дошок, обмеженні кількості одночасно виконуваних завдань та фокусуванні команди на завершенні поточних задач перед початком нових.

У контексті студентського проєктування життєвий цикл розробки складається з кількох ключових етапів:

1) Авторизація користувача та визначення його ролі в системі (Студент, Лідер команди, Академічний керівник).

2) Створення картки проєкту, вибір команди та підключення зовнішнього репозиторію (GitHub) для автоматизованого збору метрик активності.

3) Декомпозиція загальної мети проєкту на окремі завдання (задачі) з визначенням оцінки складності в Story Points та дедлайнів.

4) Призначення виконавців на завдання та переміщення карток по стовпцях Kanban-дошки відповідно до їхнього поточного стану (ToDo, In Progress, Review, Done).

5) Автоматизований імпорт інформації про коміти та запити на злиття (Pull Requests) через інтеграцію з GitHub API, що дозволяє верифікувати реальну роботу над завданням.

6) Моніторинг та аналітика процесу розробки викладачем або лідером команди за допомогою графіків згорання задач та звітів індивідуального внеску.

На рисунку 1.1 представлено склад та взаємозв'язки сутностей предметної області студентського проєктного менеджменту. Модель описує життєвий цикл розробки, починаючи від студентських команд як основного виконавчого елементу, створення проєкту, налаштування та розподілу завдань у канбан-системі, автоматизованого імпорту комітів через GitHub API, формування оперативних звітів прогресу, і завершуючи оцінюванням результатів академічним керівником або викладачем.

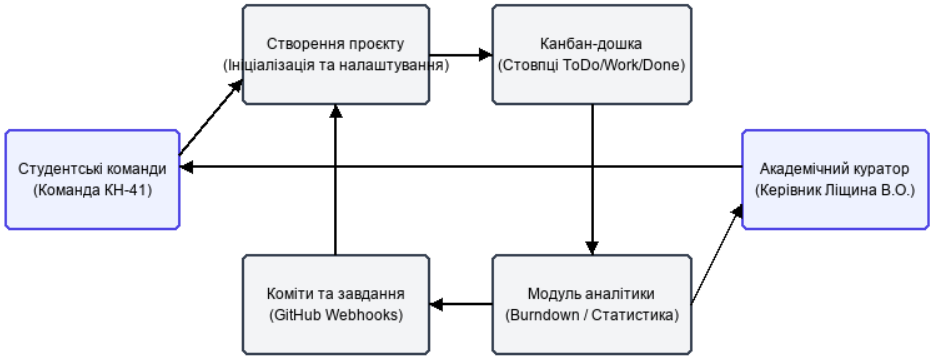

Рисунок 1.1 – Склад предметної області студентського проєктного менеджменту

Модель предметної області демонструє, що автоматизований веборієнтований застосунок має стати центральною ланкою взаємодії між студентами та академічним керівником. Завдяки інтеграції з GitHub, система може виступати об'єктивним джерелом даних про хід розробки, усуваючи суб'єктивізм при оцінюванні та забезпечуючи чітку структуру для студентського самоврядування [3].

1.2 Аналіз сучасних підходів та аналогів

Під час розроблення концепції веб-застосунку для управління проєктами в студентському середовищі необхідно провести критичний аналіз існуючих рішень та підходів до автоматизації проєктної діяльності. Це дозволить виділити найкращі практики побудови інтерфейсу користувача, визначити обмеження популярних інструментів та сформувати перелік унікальних переваг власної розробки.

У сегменті управління розробкою програмного забезпечення та координації команд виділяють кілька популярних комерційних та відкритих рішень: Trello, Jira, Asana, Notion та GitHub Projects.

Trello є одним із найпопулярніших інструментів для персонального та командного планування завдяки використанню класичної концепції Kanban. Програма пропонує простий візуальний інтерфейс, де користувачі створюють дошки, списки та картки завдань. Картки можна легко перетягувати між списками, додавати до них виконавців, описи, дедлайни та чек-листи. Основною перевагою Trello є низький поріг входу – інтерфейс зрозумілий без спеціального навчання. Проте для академічного процесу Trello має суттєві недоліки: відсутність вбудованої рольової моделі з розмежуванням прав (викладач не може мати роль пасивного спостерігача з правом оцінювання), відсутність інтеграції зі сховищами коду в безкоштовній версії та обмежені можливості побудови аналітичних звітів без підключення платних плагінів (Power-Ups).

Jira Software від компанії Atlassian є стандартом для професійних IT-компаній, що працюють за методологіями Scrum або Kanban. Вона має надзвичайно потужний функціонал: гнучке налаштування робочих процесів (workflows), облік часу, побудова різноманітних звітів, інтеграція з CI/CD системами, підтримка епіків та підзадач. Проте використання Jira в студентських проєктах є недоцільним з кількох причин. По-перше, система є платною для команд понад 10 користувачів, а безкоштовний тариф має значні обмеження. По-друге, Jira є надзвичайно складною в конфігуруванні та адмініструванні –

студенти витрачають занадто багато часу на розбір інтерфейсу та заповнення службових полів замість безпосередньої роботи над проєктом. По-третє, вона вимагає високих обчислювальних ресурсів для роботи веб-інтерфейсу, що уповільнює взаємодію на слабких пристроях [4].

Asana орієнтована на бізнес-команди широкого профілю. Вона дозволяє представляти завдання у вигляді списків, дошок, календарів або діаграм Ганта (Timeline). Інтерфейс програми є естетичним та ергономічним, проте Asana практично не орієнтована на розробників програмного забезпечення. Вона не має нативних інструментів для аналізу комітів у репозиторіях, а інтеграція з GitHub потребує складних платних тарифів. Для студентського середовища вона також є занадто дорогою та надлишковою.

Notion є універсальним робочим простором, який поєднує функції бази знань, текстового редактора та бази даних для завдань. Багато студентських команд використовують Notion для документування та ведення списків задач. Завдяки гнучким шаблонам, Notion можна адаптувати для ведення проєктів, але ця гнучкість є і його слабкою стороною. За відсутності жорстких обмежень на рівні коду, бази даних часто засмічуються, зв’язки руйнуються, а автоматизація потребує налаштування складних внутрішніх формул та зовнішніх інтеграторів (Make, Zapier). Крім того, Notion не забезпечує стабільної асинхронної обробки подій у реальному часі.

GitHub Projects інтегрований безпосередньо у платформу хостингу коду. Він дозволяє створювати Kanban-дошки безпосередньо в репозиторіях та пов’язувати завдання з Pull Requests та Issues. Це ідеальний інструмент для індивідуального розробника, але він є незручним для студентів нетехнічних спеціальностей (наприклад, дизайнерів чи аналітиків у міждисциплінарних командах) та викладачів, які не завжди мають глибокі знання платформи GitHub та не хочуть розбиратися в інтерфейсі сховищ коду для перевірки прогресу.

Для детального порівняння аналогів визначено ключові критерії, критичні для використання в академічному середовищі закладу вищої освіти:

1) Простота початку роботи (поріг входу для студентів).

2) Наявність гнучкої Kanban-дошки завдань.

3) Підтримка ролей (Студент, Лідер, Викладач-куратор).

4) Автоматична інтеграція зі сховищами коду через Webhooks.

5) Наявність аналітики (Burn-down chart, звітність внеску).

6) Доступність безкоштовного використання без обмежень на кількість користувачів.

7) Швидкодія та ергономічність користувацького інтерфейсу.

Результати порівняльного аналізу наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз програмних аналогів

Критерій порівняння	Trello	Jira Software	Asana	Notion	GitHub Projects	Розроблюваний застосунок
Поріг входу для студентів	Низький	Високий	Середній	Середній	Середній	Низький
Kanban-дошка завдань	Так	Так	Так	Так	Так	Так
Академічні ролі та права доступу	Ні	Частково	Ні	Ні	Ні	Так
Нативна інтеграція з Git Webhooks	Ні (платно)	Так	Ні (платно)	Ні	Так	Так (нативно, безкоштовно)
Спеціалізована аналітика спринтів	Ні	Так	Частково	Ні	Частково	Так (Burn-down + внесок)
Безкоштовний доступ для ЗВО	Обмежено	Обмежено	Обмежено	Так	Так	Так (100 % open-source)
Швидкодія клієнтської частини	Висока	Низька	Середня	Середня	Висока	Висока (Vite + Zustand)

Аналіз таблиці 1.1 показує, що жоден із наявних комерційних чи безкоштовних інструментів повністю не задовольняє специфічні вимоги студентського проєктного середовища. Це обґрунтовує доцільність розроблення власного веборієнтованого застосунку, який поєднуватиме низький поріг входу Trello, нативну інтеграцію з Git, як у GitHub Projects, аналітику швидкості виконання завдань, як у Jira, та спеціалізовану рольову модель для здобувачів освіти та викладачів.

1.3 Технічне завдання

На основі проведеного аналізу предметної області та існуючих програмних аналогів сформовано технічні вимоги до розроблюваного веб-застосунку для управління проєктами в студентському середовищі. Вимоги розподілено на функціональні, нефункціональні, вимоги до структури даних та інформаційної безпеки [5-7].

Функціональні вимоги визначають поведінку системи та перелік операцій, які мають бути доступні різним категоріям користувачів.

1) Модуль автентифікації та профілю користувача:

- вхід у систему за допомогою електронної пошти та пароля;
- підтримка ролей користувачів: STUDENT (студент-виконавець), TEAM_LEAD (лідер студентської команди), ADVISOR (викладач або академічний керівник проєкту), ADMIN (системний адміністратор);
- відображення профілю з іменем користувача (наприклад, «Максим БАДЗЮНЬ»), аватаром та переліком активних проєктів.

2) Модуль управління проєктами та командами:

- створення нового проєкту із зазначенням назви, опису, вибором студентської групи та встановленням кінцевого терміну (дедлайну);
- підключення зовнішнього GitHub репозиторію шляхом введення посилання на репозиторій та генерації Webhook URL;
- можливість перегляду списку проєктів із фільтрацією за статусом («Активний», «Виконано», «Заблоковано») та пошуком за назвою.

3) Модуль Kanban-дошки завдань:

- декомпозиція робіт проєкту на окремі завдання (Task). Кожне завдання містить: заголовок, опис, оцінку складності в Story Points (1, 2, 3, 5, 8, 13), виконавця, пріоритет (Low, Medium, High, Critical) та поточну колонку статусу;

− відображення завдань на інтерактивній Kanban-дошці зі стовпцями: «В черзі (ToDo)», «В роботі (Progress)», «Рецензування (Review)», «Виконано (Done)»;

− перетягування (Drag-and-Drop) карток завдань між стовпцями зі збереженням нових станів у базі даних.

4) Модуль обробки подій GitHub Webhook:

− прийом вхідних POST-запитів від сервера GitHub при здійсненні подій push або pull_request;

− автоматичний аналіз вмісту JSON-пакету події (коміти, автор коміту, повідомлення коміту);

− відображення стрічки комітів (Commit Feed) на сторінці проєкту для підтвердження фактичного виконання робіт студентами.

5) Модуль аналітики та звітності:

− відображення загального прогресу виконання проєкту у відсотках (відношення виконаних Story Points до загальної кількості);

− генерація графіка згорання завдань (Burndown Chart) для поточного спринту;

− формування звіту про індивідуальну активність учасників (кількість створених, виконаних завдань та надісланих комітів).

Нефункціональні вимоги визначають технічні обмеження та стандарти якості роботи застосунку:

1) Продуктивність та швидкодія: час відгуку сервера на типові API-запити (отримання списку завдань, оновлення статусу) не повинен перевищувати 200 мс при навантаженні до 100 одночасних користувачів.

2) Адаптивність інтерфейсу: інтерфейс користувача має бути адаптованим під різні роздільні здатності екранів, забезпечуючи коректне відображення як на персональних комп’ютерах та ноутбуках, так і на мобільних пристроях.

3) Надійність та доступність: система має функціонувати в режимі 24/7. У разі виникнення помилок на стороні сервера (наприклад, некоректні вхідні дані),

клієнтська частина повинна обробляти помилки без повного збою інтерфейсу, виводячи зрозумілі повідомлення про помилки.

4) Технологічний стек: клієнтська частина має бути побудована як односторінковий застосунок (SPA) на базі React та інструменту Vite. Керування глобальним станом – за допомогою Zustand. Серверна частина – на платформі Node.js із використанням фреймворку Express. Сховище даних – реляційна база даних PostgreSQL (або SQLite для локального розгортання) з використанням Prisma ORM для доступу до даних.

Вимоги до безпеки та захисту даних:

1) Захист передачі даних: усі запити між клієнтом та сервером мають здійснюватися через захищений протокол HTTPS (для локальної розробки допускається HTTP) [5].

2) Автентифікація та авторизація: сесія користувача повинна контролюватися за допомогою токенів доступу JSON Web Tokens (JWT). Токени мають зберігатися в безпечному місці на клієнті (наприклад, localStorage або HttpOnly Cookie) та перевірятися сервером при кожному запиті до захищених маршрутів API.

3) Валідація та очищення вхідних даних: усі дані, що надходять від клієнта через HTTP-запити, мають проходити обов’язкову валідацію на стороні сервера (захист від SQL injections, NoSQL injections та XSS-атак).

Формальна постановка задачі розроблення сервісу:

Нехай U множина користувачів системи, де кожен користувач $u \in U$ описується кортежем:

$$u = \langle id_u, name_u, email_u, role_u \rangle$$, де $role_u \in \{STUDENT, TEAM\_LEAD, ADVISOR, ADMIN\}$.

Нехай P множина проєктів:

$$p = \langle id_p, title_p, description_p, team\_id, advisor\_id, github\_repo_p, deadline_p \rangle$$, де $advisor\_id$ посилається на користувача з $role = ADVISOR$, а $team\_id$ посилається на команду студентів.

Нехай T — множина завдань проєкту p:

$$t = \langle id_t, project\_id_t, title_t, assignee\_id_t, story\_points_t, priority_t, status_t \rangle$$, де $status_t \in \{TODO, PROGRESS, REVIEW, DONE\}$.

Нехай W множина подій вебхуків від GitHub: $$w = \langle id_w, project\_id_w, author\_name_w, commit\_hash_w, message_w, timestamp_w \rangle$$.

Задача автоматизації полягає у розробленні програмного комплексу (веб-застосунку), що складається з клієнтської частини C та серверної частини S, які забезпечують:

− коректне відображення Kanban-дошки завдань та переходи статусів: $$status_t \to status'_t$$;

− синхронізацію та відображення подій вебхуків W для кожного проєкту p;

− побудову аналітичного графіка згорання завдань $G(t)$ на основі зміни станів завдань з часом: $$G(t) = \sum_{t \in T, status_t \neq DONE} story\_points_t$$.

− формування індивідуальних звітів активності студентів з мінімізацією часу відповіді системи $\tau_{resp} \le 200 \text{ мс}$ шляхом оптимізації структури бази даних та використання індексів.

1.4 Постановка задачі розроблення сервісу

Постановка задачі дослідження є підсумком аналізу предметної області, аналогів і вимог. Для веб-застосунку для управління проєктами в студентському середовищі задача полягає у створенні програмної системи, яка усуває виявлені недоліки ручної або фрагментованої організації процесів, забезпечує структуровану роботу з даними та надає користувачеві зрозумілий інструмент для досягнення практичного результату.

Метою розробки є побудова системи, що реалізує основний сценарій предметної області від введення або отримання початкової інформації до її

обробки, збереження та подання результату. Система повинна бути достатньо простою для використання цільовою аудиторією, але водночас мати внутрішню структуру, яка дозволяє підтримувати розширення функцій, зміну вимог і додавання нових типів даних.

Для досягнення цієї мети необхідно виконати такі завдання: уточнити сутності предметної області; описати користувацькі ролі; сформувати функціональні та нефункціональні вимоги; розабезпечити інформаційну модель; обрати архітектуру; спроєктувати інтерфейс; реалізувати основні модулі; перевірити працездатність системи на контрольних прикладах; оцінити переваги й обмеження отриманого рішення.

Очікуваний програмний результат складається з клієнтської частини, API, сховища даних і демонстраційного набору. Застосунок має запускатися у локальному середовищі за документованою послідовністю дій. Для перевірки готуються позитивні, негативні, граничні та інтеграційні сценарії.

Критерієм завершеності є наскрізна працездатність, а не лише наявність окремих екранів. Після виконання основного сценарію дані мають залишатися узгодженими, повторний запит не повинен створювати дублікати, а звіт — відображати фактичний стан завдань і команди.

РОЗДІЛ 2
ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ

2.1 Архітектура сервісу студентських проєктів

Проєктування архітектури веб-застосунку для управління проєктами в студентському середовищі є визначальним етапом розроблення системи, оскільки від обраної структури безпосередньо залежать масштабованість, надійність, супроводжуваність та швидкість взаємодії компонентів. Для досягнення поставленої мети було обрано багаторівневу клієнт-серверну архітектуру (3-Tier Architecture), яка дозволяє розділити відповідальність між представленням даних користувачеві, бізнес-логікою обробки запитів та фізичним збереженням інформації в базі даних.

Клієнтський рівень (Presentation Tier) представляє собою односторінковий веб-застосунок (Single Page Application – SPA), побудований на базі бібліотеки компонентів React із використанням сучасного інструменту збирання Vite. Вибір Vite замість застарілого Create React App обумовлений його надзвичайною швидкістю гарячої заміни модулів (HMR) під час розроблення та оптимізованим процесом генерації фінальних статичних файлів за допомогою Rollup. Для управління глобальним станом застосунку (Global State Management) на клієнті застосовано бібліотеку Zustand. Zustand є прогресивною альтернативою важким та складним у конфігуруванні інструментам на кшталт Redux Toolkit чи вбудованого механізму React Context, який часто спричиняє зайві рендери компонентів. Стан системи у Zustand розділено на два ключові сховища (stores):

− `useAuthStore` – зберігає дані поточного авторизованого користувача, його JWT-токен, роль у системі та методи для логіну/логуту із автоматичним відновленням сесії з localStorage;

− `useProjectStore` – керує станом завантажених проєктів, списками завдань на Kanban-дошці, фільтрами пошуку, а також індикаторами завантаження (loading states) та помилок (error boundaries).

Серверний рівень (Application Tier) реалізовано на платформі Node.js з використанням веб-фреймворку Express.js. Вибір Express обумовлений його мінімалістичним підходом, високою швидкістю обробки асинхронних запитів I/O та широкою екосистемою проміжного програмного забезпечення (middleware). Серверна логіка розподілена на такі функціональні шари:

− шар маршрутизації (Routes) – приймає вхідні HTTP-запити від клієнта та перенаправляє їх на відповідні контролери;

− шар проміжного ПЗ (Middlewares) – виконує службові функції: обробку CORS-запитів, парсинг JSON-тіла запиту, авторизацію токенів доступу через `authenticateToken` та перевірку прав доступу на основі ролей (наприклад, дозволяти редагувати налаштування проєкту лише користувачам із роллю `TEAM_LEAD` чи `ADMIN`);

− шар контролерів (Controllers) – містить безпосередню бізнес-логіку обробки сутностей: валідацію вхідних даних (payload validation), виклик операцій бази даних, обчислення відсотків прогресу, агрегацію подій від GitHub та формування HTTP-відповідей;

− шар доступу до даних (Data Access Layer) – використовує Prisma ORM для роботи з базою даних через типізований клієнт Prisma Client, що виключає написання сирого SQL-коду та забезпечує безпеку від SQL-ін’єкцій на рівні компіляції.

Рівень бази даних (Data Tier) базується на реляційній системі керування базами даних PostgreSQL. PostgreSQL забезпечує підтримку транзакцій ACID, що є критично важливим при паралельній роботі багатьох студентів над однією Kanban-дошкою. Для локального тестування та демонстрації працездатності застосунку без потреби розгортання повноцінного SQL-сервера використовується вбудована реляційна база даних SQLite, яка повністю сумісна з Prisma ORM та дозволяє мігрувати схему даних без змін у коді [8-10].

Службовий рівень інтеграції призначений для взаємодії із зовнішнім сервісом GitHub. Коли студент робить коміт або створює Pull Request у репозиторії, GitHub надсилає асинхронне HTTP POST-повідомлення (Webhook)

на спеціальний захищений шлях нашого Express-сервера. Сервер обробляє це повідомлення та записує інформацію про коміт у базу даних для подальшого відображення в інтерфейсі.

На рисунку 2.1 наведено розроблену трирівневу архітектуру веб-застосунку. Клієнтська частина базується на фреймворку React із використанням інструменту збирання Vite та менеджера стану Zustand для управління клієнтським контекстом. Серверна частина реалізована на платформі Node.js із використанням фреймворку Express для побудови REST API та проміжного ПЗ (Middleware) для JWT-авторизації. Рівень сховища та інтеграції включає реляційну базу даних PostgreSQL (або SQLite для тестування) та зовнішній GitHub Webhook сервіс для синхронізації коду [11-12].

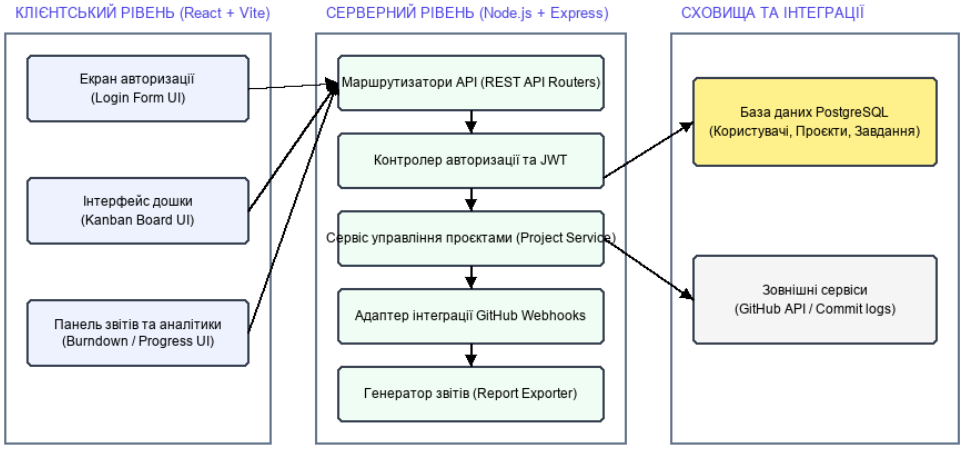

Рисунок 2.1 – Архітектура програмної системи

Архітектурна модель забезпечує повну незалежність шарів. Клієнтська частина взаємодіє з сервером виключно через уніфікований REST API. Обмін даними здійснюється у форматі JSON. Це дозволяє за потреби легко змінити систему керування базою даних або розширити клієнтську частину до мобільного застосунку без переписування серверної логіки.

2.2 Проєктування функціональних сценаріїв і структури даних

Деталізація функціональності веб-застосунку виконується за допомогою UML-діаграми варіантів використання, яка описує межі системи, акторів та їхні можливі дії.

У розроблюваній системі виділено три ключові актори (користувачі):

− Студент (STUDENT) – може переглядати дашборд, переходити до своїх проєктів, створювати нові завдання на дошці, брати задачі в роботу, залишати коментарі до карток, підключати свій GitHub-аккаунт.

− Лідер команди (TEAM_LEAD) – студент, який створив проєкт. На додачу до прав звичайного студента, він може редагувати опис проєкту, підключати GitHub репозиторій, генерувати Webhook URL, видаляти завдання та завершувати спринти.

− Академічний керівник / Викладач (ADVISOR) – викладач, який курує виконання проєктів. Він має права спостерігача (read-only) для Kanban-дошки, але може переглядати детальну стрічку комітів, бачити індивідуальні звіти внеску студентів, залишати оцінки до етапів та писати коментарі-зауваження до завдань у стані «Рецензування (Review)».

На рисунку 2.2 подано діаграму варіантів використання (Use Case Diagram) в нотації UML, яка визначає межі системи та права доступу користувачів. Виділено дві основні ролі: Студент (виконавець або лідер команди), який може створювати проєкти, керувати дошками, додавати завдання, призначати виконавців і підключати репозиторії; та Керівник / Викладач, який здійснює моніторинг діяльності, переглядає аналітику та залишає оцінки й коментарі до звітів [13-15].

Для збереження інформації спроєктовано реляційну схему бази даних. Концептуальна модель даних описує основні сутності та зв’язки типу «один-до-багатьох» (1:M) та «багато-до-багатьох» (M:N) між ними.

Рисунок 2.2 – UML-діаграма варіантів використання системи

Сутностями бази даних є:

− `Users`: зберігає персональні дані (email, hash пароля, повне ім’я, роль);

− `Teams`: студентська команда розробників. Зв’язана з користувачами через проміжну таблицю членства `TeamMembers` (багато-до-багатьох);

− `Projects`: сутність проєкту, що створюється лідером команди. Має зв’язки з `Teams` (команда виконавців) та `Users` (власник проєкту та призначений викладач-куратор);

− `Tasks`: завдання, які створюються в межах проєкту. Пов’язані з проєктом та конкретним виконавцем (користувачем);

− `WebhookEvents`: лог комітів, імпортованих із GitHub Webhooks. Кожен запис пов’язаний з проєктом та містить ім’я автора коміту, хеш комміту, повідомлення та дату;

− `AuditLogs`: службова таблиця для фіксації критичних дій у системі (створення проєкту, авторизація, зміна статусу задачі) з метою моніторингу безпеки.

На рисунку 2.3 зображено концептуальну модель даних (Entity-Relationship Diagram), що відображає структуру реляційного сховища. Вона містить п’ять основних таблиць: Users (облікові записи користувачів із ролями STUDENT, ADVISOR, ADMIN), Projects (параметри проєктів із зовнішніми ключами для

власника та команди), Teams (студентські групи та команди), Tasks (завдання на канбан-дошці з посиланням на виконавця та проєкт) та AuditLogs (історія дій користувачів для аудиту безпеки).

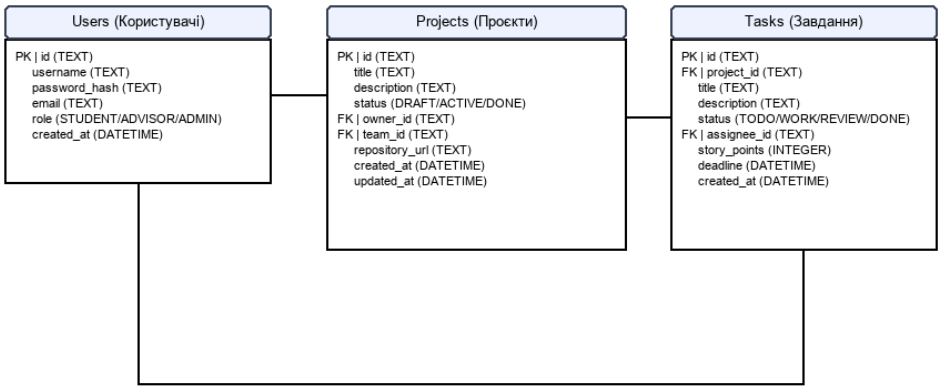

Рисунок 2.3 – Концептуальна модель даних

Для детального розуміння динаміки взаємодії компонентів під час виконання критичних операцій спроєктовано діаграму послідовності (Sequence Diagram) для сценарію «Створення проєкту та підключення репозиторію». Сценарій охоплює взаємодію між користувачем (Студент), клієнтським інтерфейсом React, проміжним ПЗ авторизації (JWT Middleware), бізнес-контролером проєктів (Project Controller) та базою даних PostgreSQL.

На рисунку 2.4 наведено діаграму послідовності (Sequence Diagram) для ключової операці – створення проєкту та ініціалізації канбан-дошки. Студент надсилає запит через форму клієнтського інтерфейсу React. Express-сервер перевіряє авторизаційні дані через JWT-Middleware, валідує вхідні параметри (назву проєкту), створює запис у базі даних PostgreSQL, автоматично генерує стандартний набір стовпців для канбан-дошки та повертає підтвердження успішного завершення операції клієнтові.

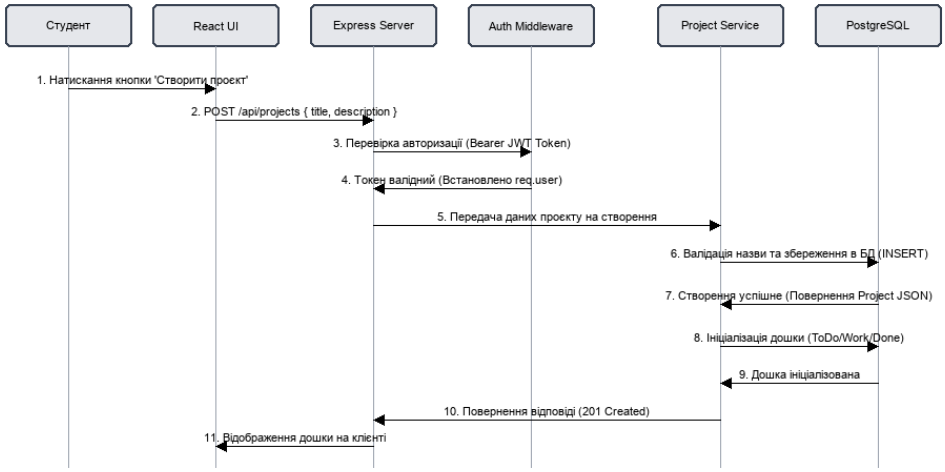

Рисунок 2.4 – Послідовність виконання основного сценарію

Як показано на діаграмі послідовності, кожен запис на сервері проходить три етапи верифікації: перевірка валідності JWT-токена у заголовку запиту (Authorization: Bearer <token>), перевірка структури даних (валідація схеми тіла запиту на рівні Express-middleware) та безпосереднє виконання транзакції у базі даних. У разі успіху база повертає створений запис, а сервер автоматично генерує унікальний Webhook Secret та Webhook URL для інтеграції з GitHub.

2.3 Проєктування інтерфейсу і технологічного стеку

Ергономіка інтерфейсу користувача (UI/UX) є критичним фактором успіху веб-застосунку, оскільки студенти та викладачі повинні взаємодіяти з дошками завдань без тривалого вивчення інструкцій. Для цього під час проєктування макетів екранних форм було обрано концепцію модульної панелі (Dashboard Layout).

Основне вікно застосунку розділено на три функціональні зони:

− бічна панель навігації (Sidebar) – розташована зліва, забезпечує постійний доступ до ключових розділів: дашборд (Dashboard), Мої Проєкти (Projects), Команда (Team), Аналітика (Analytics) та Налаштування (Settings);

– верхній хедер (Header) – відображає поточну дату, ім’я авторизованого користувача («Максим БАДЗЮНЬ»), його роль, кнопку швидкого пошуку та меню профілю із кнопкою виходу з системи;

– головна робоча область (Main Content Area) – займає всю центральну частину екрана та динамічно змінює свій вміст. При переході в деталі проєкту тут відображається Kanban-дошка з чотирма колонками або графік згорання завдань.

На рисунку 2.5 представлено макет загальної екранної форми (Wireframe Layout) веб-застосунку. Основна розмітка екрана складається з трьох зон: ліва бічна панель (Sidebar Navigation) для переходу між дашбордом, проєктами та налаштуваннями; верхній хедер (Header) з інформацією про користувача, дату та поточний контекст; та центральна робоча область (Main Board Area), яка динамічно відображає канбан-дошку або аналітичні графіки.

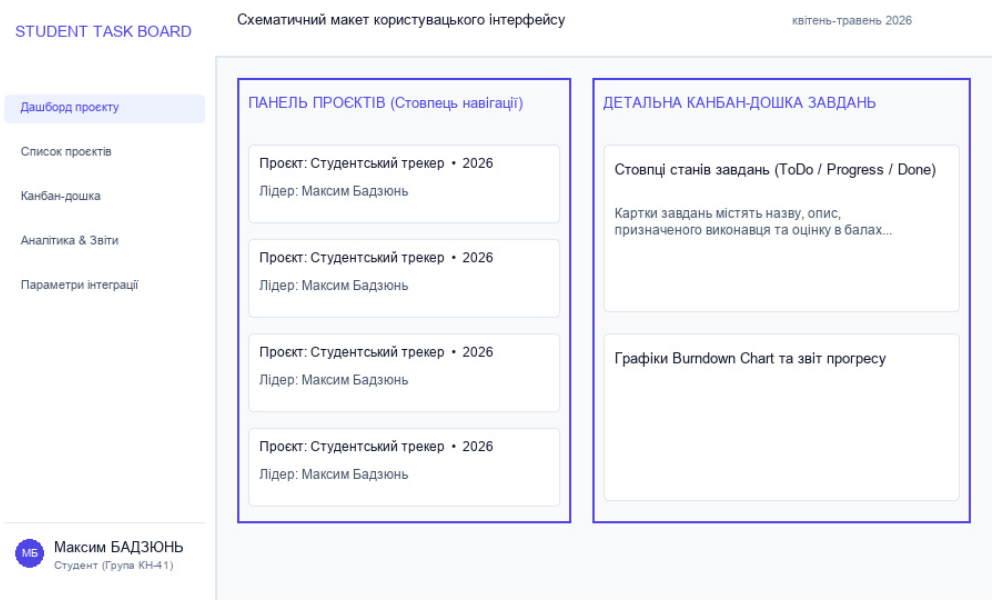

Рисунок 2.5 – Узагальнений макет екранних форм і навігації

Вибір технологічного стеку обґрунтовано наступними технічними та експлуатаційними перевагами:

– TypeScript – застосовується як на клієнті, так і на сервері. Використання статичної типізації дозволяє уникнути поширених помилок під час розробки

(наприклад, передача об’єкта неправильної структури в API або звернення до неіснуючого поля бази даних) та покращує автодоповнення коду в IDE;

− TailwindCSS – використовується для швидкої та гнучкої стилізації компонентів на клієнтській частині без написання великої кількості CSS-файлів;

− Matplotlib – бібліотека Python, що використовується на етапі аналізу даних та тестування швидкодії для візуалізації результатів вимірювання часу запитів до PostgreSQL.

Критерії якості розроблюваної системи базуються на стандарті ISO/IEC 25010 та адаптовані під специфіку студентського середовища:

1) Функціональна придатність (Functional Suitability) – система повинна повністю забезпечувати виконання сценарію: авторизація, створення проєкту, налаштування Kanban-дошки, призначення виконавців, обробка вебхуків від GitHub та відображення звітів.

2) Практичність та зручність використання (Usability) – користувачі мають виконувати основні дії (наприклад, перетягування картки або додавання завдання) за мінімальну кількість кроків (інтуїтивно зрозумілий drag-and-drop).

3) Продуктивність (Performance Efficiency) – час реакції інтерфейсу на дії користувача має бути мінімальним. Запити до API повинні виконуватися за час $\le 200 \text{ мс}$.

4) Безпека (Security) – наявність надійної авторизації JWT, захист паролів за допомогою алгоритму хешування bcrypt із сіллю на рівні сервера, уникнення витоку чутливих даних у логах та попередження атак типу SQL Injection за рахунок використання ORM.

5) Портативність (Portability) – веб-застосунок має стабільно працювати в усіх сучасних веб-браузерах (Google Chrome, Mozilla Firefox, Safari, Microsoft Edge) без встановлення додаткових плагінів.

РОЗДІЛ 3
РОЗРОБКА ЗАСТОСУНКУ

3.1 Середовище розробки та структура програмного проєкту

Розробляємо веб-застосунок для управління проєктами в студентському середовищі на основі проєктних рішень. На етапі реалізації перетворюємо архітектурну модель, функціональні сценарії та структуру даних у працездатний застосунок. Для цього визначаємо структуру програмного проєкту, маємо реалізувати модулі, організувати взаємодію клієнтської і серверної частин, створити ергономічний інтерфейс і підготувати тестовий набір даних.

Обраний стек технологій для розробки: React, REST API, серверний фреймворк, SQL-база, JWT. За допомогою такого набору інструментів реалізуємо веборієнтовану систему з чітким поділом відповідальності. Клієнтська частина відповідає за відображення даних і взаємодію з користувачем, серверна частина – за бізнес-логіку, перевірку прав доступу, обробку запитів і роботу зі сховищем.

Початковим екраном застосунку є форма авторизації. Її призначення полягає не лише у перевірці облікових даних, а й у визначенні ролі користувача, доступних модулів і стартового маршруту після входу. Допускається спрощена авторизація, однак ми демонструємо принцип розмежування доступу між звичайним користувачем і адміністратором.

На рисунку 3.1 демонструємо макет екрана авторизації. У ньому передбачено поля для введення логіна або електронної пошти, пароль, кнопку входу та пояснення ролі користувача. Такий екран є мінімальним, але достатнім для початку роботи із системою. У подальшому його можемо розширити реєстрацією, відновленням пароля або двофакторним підтвердженням. Зображено користувацький інтерфейс форми входу в систему. Форма виконана в сучасному мінімалістичному стилі з використанням гармонійної світлої колірної палітри Slate/Indigo. Вона містить поля для введення електронної пошти користувача, пароля, чекбокс збереження сесії та велику кнопку входу. У футері

форми розміщено службовий напис із прізвищем розробника «Бадзюнь М. А.» та групою КН-41.

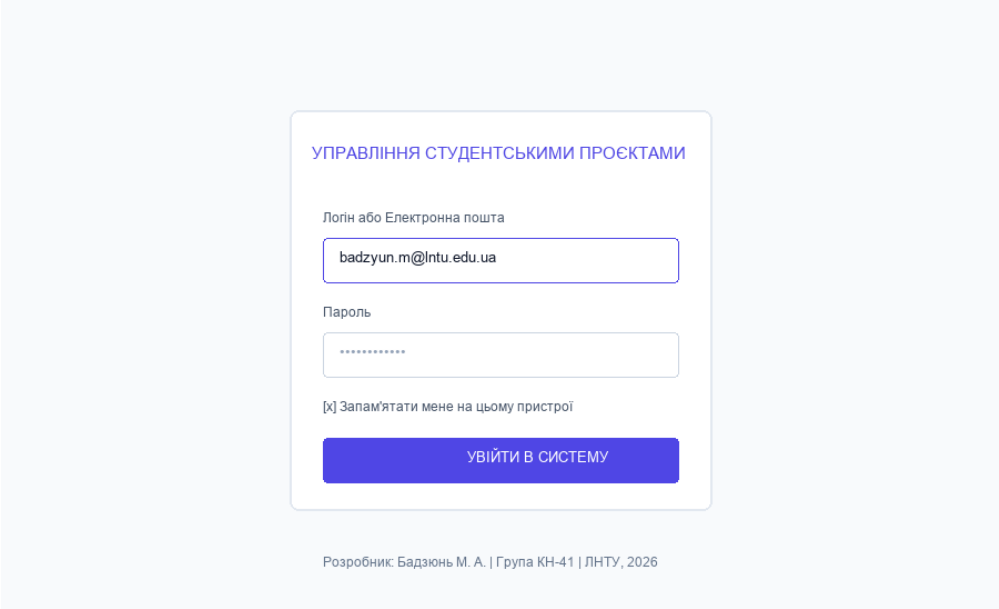

Рисунок 3.1 – Макет екрана авторизації користувача

Після авторизації користувач переходить до основної панелі. Вона показує ключові модулі, оперативний стан системи та основні дії. Для системи студентського проєктного менеджменту основну панель проектуємо так, щоб не перевантажувати користувача і допомагати оперативно перейти до сценарію. Авторизація, створення проєкту, додавання задач, призначення виконавців, зміна статусу, перегляд звіту. Саме тому на ній розміщуємо навігацію, короткі показники та блок останніх подій.

На рисунку 3.2 подаємо макет основної панелі користувача. На ньому зображено навігаційне меню, інформаційні картки та робочу область із коротким описом основного процесу. Така структура дозволяє поєднати огляд стану системи з можливістю оперативного переходу до необхідного модуля. Показано дашборд користувача після входу. Інтерфейс містить вітальний банер з іменем авторизованого користувача та три інформаційні картки з оперативною статистикою (активні проєкти, кількість призначених завдань, виконані задачі за

тиждень). У нижній частині відображається стрічка останніх подій у проєктах (злиття комітів, зміна статусів, додавання коментарів іншими учасниками).

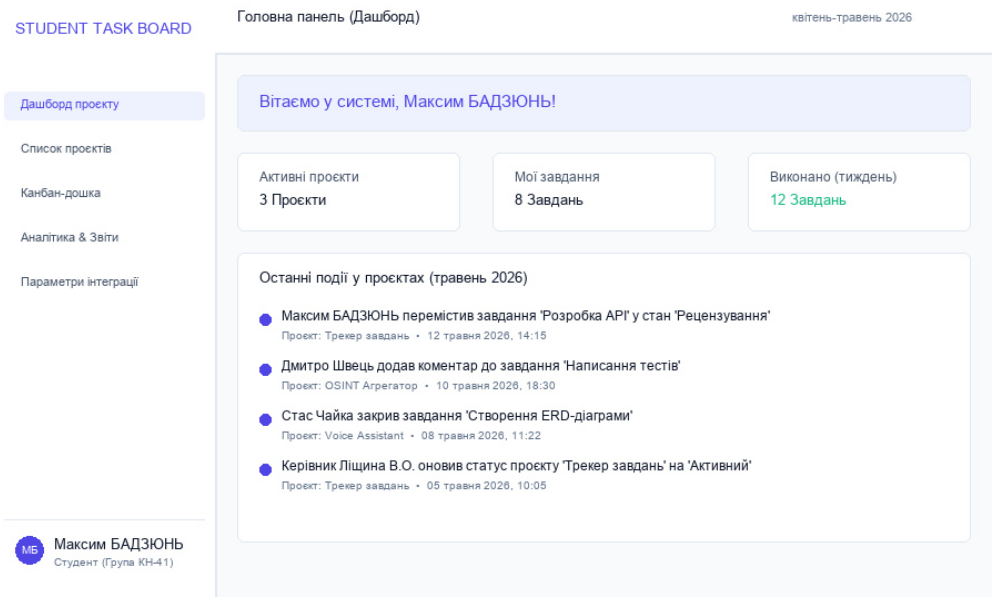

Рисунок 3.2 – Головна панель користувача застосунку

Клієнтська частина повинна зберігати лише ті дані, які необхідні для поточної взаємодії, а джерелом істини має залишатися сервер або база даних. Після створення, редагування або видалення запису інтерфейс повинен оновлювати стан без перезавантаження всієї сторінки.

Розробка повинна виконуватися ітераційно. Спочатку реалізуємо мінімальний шлях користувача, далі додаються перевірки, потім розширюється інтерфейс, і лише після цього впроваджуються допоміжні функції. Для системи студентського проєктного менеджменту таким мінімальним шляхом є сценарій авторизація, створення проєкту, додавання задач, призначення виконавців, зміна статусу, перегляд звіту, оскільки саме він демонструє зміст теми.

На практичному рівні середовище розробки повинно забезпечувати оперативний цикл внесення змін і перевірки результату. Для цього потрібно використовувати локальний сервер розробки, автоматичне оновлення клієнтської частини, окремі конфігурації для тестового й демонстраційного

режимів, а також систему контролю версій. Це дозволяє послідовно фіксувати зміни та за потреби повернутися до стабільного стану проєкту.

Окремої уваги потребує узгодження текстів інтерфейсу (ліст. 3.1). Назви кнопок, пунктів меню, повідомлень і статусів повинні бути короткими та однозначними. Якщо дія змінює дані, кнопка має називати саме дію, наприклад створити, зберегти, запустити, підтвердити або скасувати. Це зменшує кількість помилок і робить систему зрозумілішою для користувача.

Лістинг 3.1 – Опис базової моделі даних для об’єкта «проєкт»

```
      export interface Project {
      id: string;
      title: string;
      description: string;
      status: ‘draft’ | ‘active’ | ‘done’ | ‘archived’;
      ownerId: string;
      teamId: string;
      repositoryUrl?: string;
      createdAt: string;
      updatedAt: string;
      }
      export const emptyProject = (ownerId: string, teamId: string): Project =>
({
      id: crypto.randomUUID(),
      title: ‘’,
      description: ‘’,
      status: ‘draft’,
      ownerId,
      teamId,
      createdAt: new Date().toISOString(),
      updatedAt: new Date().toISOString()
      });
```

кінець лістингу 3.1

Важливо також передбачати єдиний стиль іменування. Якщо в одному модулі використовуємо термін проєкт, а в іншому – інша назва для тієї самої сутності, це ускладнить підтримку. Тому назви маршрутів, компонентів і моделей мають відповідати термінам, які використовуються в тексті кваліфікаційної роботи.

3.2 Реалізація серверної логіки та моделі даних

Серверна частина застосунку реалізує правила предметної області та забезпечує доступ до даних. Вона приймає запити від клієнта, виконує перевірку введення, застосовує бізнес-логіку, звертається до бази даних і повертає відповідь у стандартизованому форматі. Для системи системи студентського проєктного менеджменту серверна логіка зосереджена навколо таких модулів: проєкти, команди, завдання, календар, коментарі, звіти.

Основними сутностями системи є поля та об’єкти, пов’язані з такими даними, як назва проєкту, команда, дедлайн, статус, виконавець, опис. Вони формують ядро предметної моделі. Для кожної сутності необхідно визначити унікальний ідентифікатор, основні атрибути, зв’язки з іншими сутностями, статуси та часові поля. Це дозволяє відстежувати життєвий цикл запису й підтримувати історію змін.

API застосунку раціонально організувати за ресурсним принципом. Для кожної сутності створюємо набір маршрутів: отримання списку, перегляд одного запису, створення, редагування, видалення або архівування, а також спеціальні дії, пов’язані з основним сценарієм. Для веб-застосунку управління проєктами в студентському середовищі важливо підтримати дію створення проєкту.

Валідацію даних виконуємо на двох рівнях. На клієнті вона допомагає оперативно повідомити користувача про помилку введення. На сервері вона є обов’язковою, оскільки саме сервер відповідає за цілісність даних. Перевіряються обов’язкові поля, типи значень, допустимі діапазони, унікальність ключових параметрів і логічна узгодженість операцій.

Для збереження даних використовуємо структуровану модель, що дозволяє виконувати пошук, фільтрацію та формування результатів. Запити до бази даних не повинні розміщуватися безпосередньо у контролерах інтерфейсу API. Краще виділити окремий шар сервісів або репозиторіїв, які відповідають за отримання й оновлення даних. Це спрощує тестування і дозволяє змінити спосіб зберігання без повного переписування логіки.

Для основного сценарію авторизації, створення проєкту, додавання задач, призначення виконавців, зміна статусу, перегляд звіту важливо забезпечити послідовну зміну станів. Кожен крок повинен мати зрозумілий результат, створений запис, змінений статус, оновлену статистику або сформований підсумок. Якщо дія завершується помилкою, система повинна повернути користувача до стабільного стану й пояснити причину невиконання операції.

На рисунку 3.3 демонструємо інтерфейс списку об’єктів системи. У такому представленні користувач може переглянути записи, перейти до створення нового елемента, відкрити деталі або застосувати фільтри. Для системи студентського проєктного менеджменту список є одним із ключових екранів, оскільки саме через нього користувач контролює поточний стан даних. Наведено екран перегляду списку студентських проєктів. У вигляді таблиці відображаються назва проєкту, назва команди, керівник, дата дедлайну, прогрес виконання у відсотках та статус («Активний» або «Виконано»). Кожен рядок таблиці має інтерактивні елементи для переходу до детальної канбан-дошки відповідного проєкту.

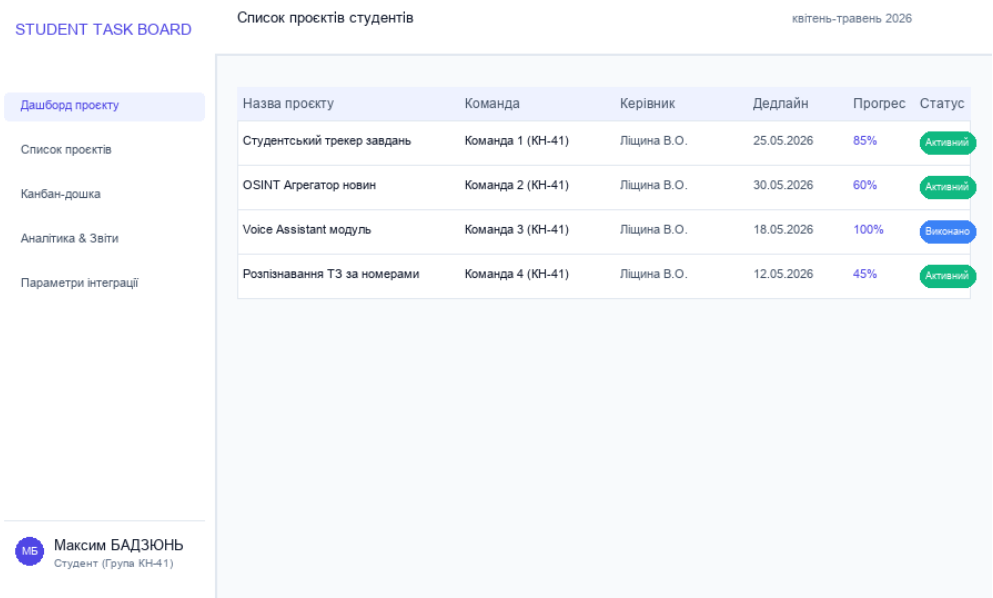

Рисунок 3.3 – Інтерфейс списку об’єктів системи

На рисунку 3.4 відображаємо форму створення або редагування запису. Вона містить поля, що відповідають предметній області: назва проєкту, команда, дедлайн, статус, виконавець, опис. У формі необхідно передбачати валідацію, підказки, повідомлення про помилки та кнопку збереження. Для складних даних раціонально використовувати списки, перемикачі або автодоповнення, щоб зменшити кількість ручного введення.

На рисунку 3.4 зображено форму додавання нового проєкту в систему. Студент заповнює назву («Web-застосунок для управління проектами»), короткий опис, обирає команду виконавців, кінцевий термін виконання та посилання на репозиторій у GitHub. Форма має вбудовану валідацію введених полів і підсвічує активні поля синім контуром.

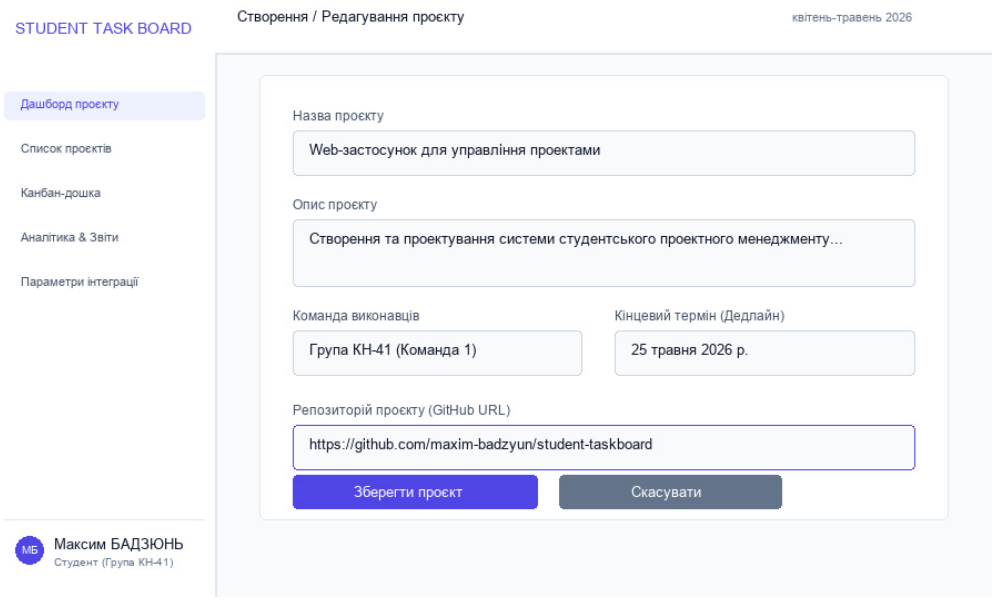

Рисунок 3.4 – Форма створення або редагування запису

На рисунку 3.5 подаємо картку детального перегляду. Такий екран необхідний для відображення повної інформації про проєкт, історії дій, пов’язаних записів і доступних операцій. Детальна картка є проміжним рівнем між коротким списком і складними адміністративними діями. Вона допомагає користувачу прийняти рішення без переходу між багатьма сторінками. Показано робочий простір канбан-дошки конкретного проєкту. Дошка розбита на чотири

стовпці: «В черзі (ToDo)», «В роботі (Progress)», «Рецензування (Review)» та «Виконано (Done)». Завдання представлені у вигляді карток, що містять назву задачі, виконавця та оцінку в Story Points. Картка завдання «Створення REST API», призначеного студенту «Максим Б.», перебуває у стані рецензування та виділена синьою рамкою.

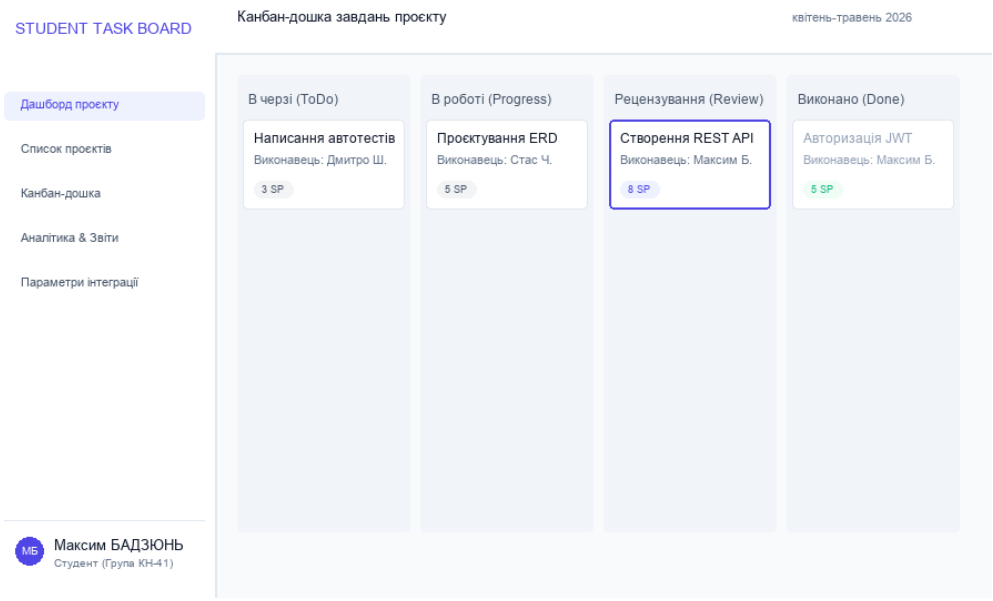

Рисунок 3.5 – Картка детального перегляду об’єкта

Для забезпечення надійності серверної частини реалізуємо централізовану обробку помилок. Помилки валідації, відсутність доступу, неіснуючий запис і внутрішній збій повинні мати різні коди та повідомлення. Такий підхід спрощує роботу клієнтської частини, оскільки вона може однаково обробляти відповіді API. Окремо реалізуємо журналювання ключових операцій. У журналі зберігаємо час, користувача, тип дії, ідентифікатор об’єкта та результат.

Для серверної частини важливо забезпечити атомарність критичних операцій. Якщо дія змінює кілька пов’язаних записів, вона має або завершитися повністю, або не змінити систему взагалі. Це стосується сценаріїв, де результат звіт прогресу залежить від кількох сутностей і проміжних статусів.

Лістинг 3.2 – Метод API для створення запису в системі

```
router.post('/projects', authorize('студент'), async (req, res, next) => {
try {
const payload = validateProjectPayload(req.body);
const project = await projectService.createProject({
...payload,
ownerId: req.user.id,
source: 'система студентського проєктного менеджменту'
});
res.status(201).json({
ok: true,
message: 'Проєкт успішно створено',
data: project
});
} catch (error) {
next(error);
}
});
```

кінець лістингу 3.2

API має повертати не лише дані, необхідний інтерфейсу (ліст. 3.2). Разом зі списком записів варто повертати загальну кількість, доступні фільтри або дозволені дії для поточного користувача. Це дозволяє клієнтській частині не дублювати бізнес-правила.

Лістинг 3.3 – Функція серверної валідації вхідних даних

```
function validateProjectPayload(payload) {
const errors = {};
if (!payload.title || payload.title.trim().length < 3) {
errors.title = 'Назва проєкту повинна містити щонайменше 3 символи';
}
if (!payload.status) {
errors.status = 'Необхідно вказати статус проєкту';
}
if (Object.keys(errors).length > 0) {
throw new ValidationError('Помилка перевірки даних проєкту', errors);
}
return {
title: payload.title.trim(),
status: payload.status,
description: payload.description?.trim() ?? ''
};
}
```

кінець лістингу 3.3

Під час реалізації моделі даних необхідно передбачати службові поля: дату створення, дату оновлення, автора зміни та поточний статус. Вони можуть не відображатися на всіх екранах, але є важливими для журналювання, сортування, аудиту та відновлення логіки роботи системи.

Лістинг 3.4 – Сервіс виконання основного сценарію «створення проєкту»

```
      async function initializeProjectBoard(projectId, userId) {
      const project = await projectRepository.findById(projectId);
      assertAccess(project, userId);
      const columns = [‘В черзі (ToDo)’, ‘В роботі (Progress)’, ‘Рецензування
(Review)’, ‘Виконано (Done)’];
      const board = await boardService.initialize({
      project,
      columns,
      action: ‘ініціалізація дошки’,
      expectedResult: ‘звіт прогресу’
      });
      await auditLog.write({
      userId,
      projectId,
      action: ‘ініціалізація дошки’,
      status: ‘success’
      });
      return board;
      }
```

кінець лістингу 3.4

У серверній частині варто реалізувати окремий шар сервісів, який приховує деталі роботи з базою даних від контролерів. Контролер отримує запит, перевіряє базові параметри та передає його сервісу. Сервіс застосовує правила предметної області, звертається до репозиторію і повертає результат. Такий підхід робить код чистішим і полегшує тестування.

Для кожного API-запиту необхідно визначити очікувані вхідні параметри та формат відповіді. Запит на створення запису має повертати створений об’єкт або повідомлення про помилку валідації. Запит на отримання списку має підтримувати пагінацію, фільтри або сортування, якщо це необхідно для роботи з проєктом.

Реалізація бізнес-логіки повинна враховувати статуси. Запис може бути новим, активним, завершеним, архівним або помилковим залежно від

предметної області. Зміна статусу не повинна виконуватися випадково. Система має перевірити, чи дозволений такий перехід. Це особливо важливо для сценарію авторизація, створення проєкту, додавання задач, призначення виконавців, зміна статусу, перегляд звіту, де кілька дій пов’язані в одну послідовність.

Обробка помилок повинна бути не додатковою, а базовою частиною серверної реалізації. Якщо користувач надіслав неповні дані, система повертає помилку валідації. Якщо запис не знайдено, повертається повідомлення про відсутність об’єкта. Якщо користувач не має прав, сервер блокує дію незалежно від того, що відображає клієнтський інтерфейс.

3.3 Реалізація клієнтського інтерфейсу застосунку

Клієнтська частина застосунку відповідає за візуальне представлення даних і ергономічність виконання сценаріїв. Вона повинна забезпечити зрозумілу навігацію, оперативну реакцію на дії користувача, коректне відображення стану завантаження та обробку помилок. Для веб-застосунку управління проєктами в студентському середовищі інтерфейс має вести користувача через сценарій авторизація, створення проєкту, додавання задач, призначення виконавців, зміна статусу, перегляд звіту, не змушуючи його самостійно здогадуватися про наступний крок.

Основою клієнтської частини є набір сторінок і повторно використовуваних компонентів. До сторінок належать екран входу, основна панель, список записів, форма створення, детальний перегляд, екран результатів і адміністративні налаштування. До компонентів належать кнопки, поля введення, картки, таблиці, повідомлення, модальні вікна, фільтри та навігаційні елементи.

Реалізація списків і таблиць повинна враховувати пошук, сортування, фільтрацію та порожній стан. Якщо даних ще немає, система повинна не показувати порожній екран, а запропонувати створити перший запис або завантажити тестові дані. Якщо записів багато, користувачу необхідні фільтри за

ключовими полями. Для системи студентського проєктного менеджменту такими полями можуть бути: назва проєкту, команда, дедлайн, статус, виконавець, опис.

Екран результатів або аналітики відображає підсумок роботи системи. Для різних тем він може означати звіт прогресу, статистику, список рекомендацій, агреговану стрічку, журнал змін або сформований план. На рисунку 3.6 демонструємо екран результатів та аналітики. У ньому використано інформаційні картки, фільтри, попередження та графічне подання показників.

На рисунку 3.6 представлено аналітичний екран проєкту. Зверху виведено зведені метрики (загальний відсоток виконання, залишок часу, поточний спринт). Центральну частину займає графік згорання завдань (Burndown Chart) за травень 2026 року, побудований за допомогою matplotlib. На графіку порівнюється ідеальна лінія згорання (сірий колір) та фактичний прогрес команди (синій колір).

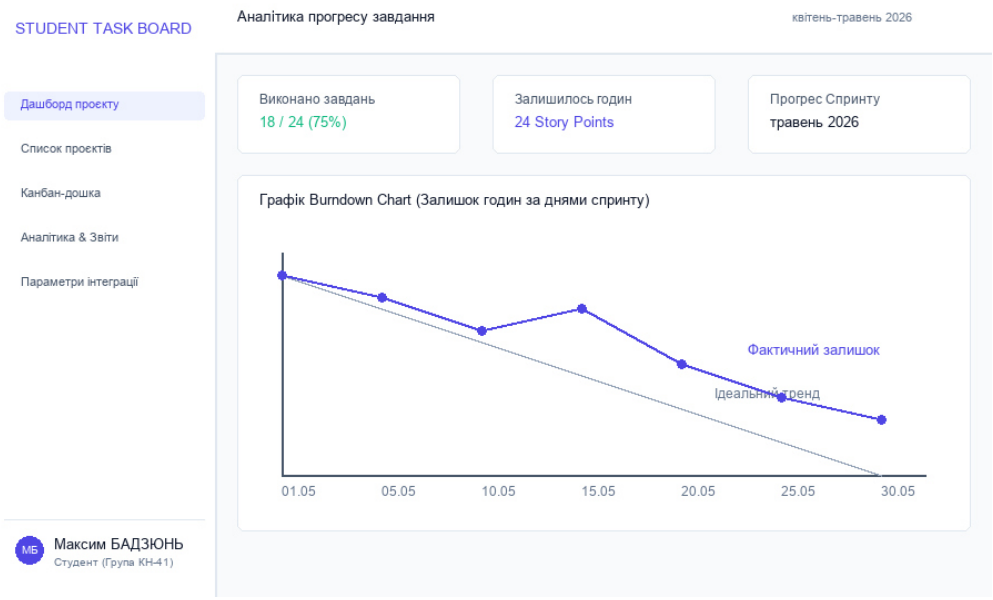

Рисунок 3.6 – Екран результатів та аналітики

Адміністративний інтерфейс необхідний для керування довідниками, ролями, параметрами та журналом подій. Він не повинен бути доступний звичайному користувачу. Для системи система студентського проєктного

менеджменту адміністрування може включати налаштування модулів проєкти, команди, завдання, календар, коментарі, звіти, керування тестовими даними та перегляд службових повідомлень.

На рисунку 3.7 подаємо макет адміністративних налаштувань. У ньому відображено окрему навігацію для службових функцій, блоки параметрів і журнал дій. Такий екран допомагає відокремити повсякденну роботу користувача від керування системою, що є важливою умовою безпеки й зручності. Зображено панель налаштувань інтеграції з репозиторієм GitHub. Користувач може вказати URL-адресу вебхука для Express-сервера, таємний токен (Webhook Secret) та налаштувати Telegram Chat ID для надсилання автоматичних сповіщень про коміти. Бейдж зеленого кольору підтверджує успішну перевірку з'єднання (OK 200).

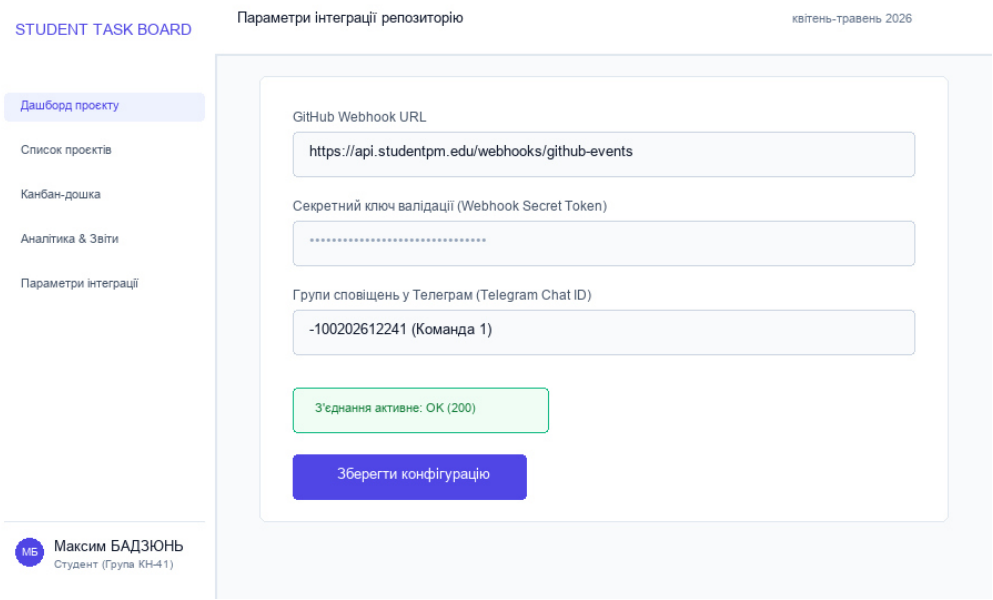

Рисунок 3.7 – Адміністративні налаштування застосунку

Під час реалізації інтерфейсу необхідно дотримуватися єдиного стилю. Однакові дії повинні мати однакові кнопки, однакові статуси – однакові кольори, а повідомлення – однакову структуру. Це знижує когнітивне навантаження і робить застосунок професійнішим. Не менш важливо забезпечити достатній контраст, читабельний розмір шрифту та адаптивність для різних екранів.

Клієнтська частина також повинна правильно працювати зі станами завантаження. Під час запиту до API (ліст. 3.5) користувач має переглядати індикатор або заблоковану кнопку, щоб не відправляти одну дію кілька разів. У разі помилки необхідно показати зрозуміле повідомлення і дозволити повторити операцію. Це особливо важливо для дій, пов’язаних із створення проєкту.

Таким чином, реалізація клієнтської частини охоплює не лише верстку сторінок, а й побудову повного користувацького досвіду: від входу до отримання результату. Інтерфейс має відповідати предметній області, підтримувати основний сценарій, пояснювати помилки та забезпечувати доступ до службових функцій відповідно до ролі користувача.

Користувацький інтерфейс повинен бути адаптований до частоти використання функцій. Найпоширеніші операції розміщуються на видимих кнопках, а рідкісні службові дії – у меню або в адміністративному розділі.

Окремо враховуємо порожні стани. Якщо список проєкти ще не містить записів, інтерфейс має пояснити, що робити далі, і запропонувати дію створення проєкту. Для екрану результатів важливо не тільки показати підсумок, а й пояснити його походження. Користувач повинен розуміти, які дані вплинули на звіт прогресу, які фільтри застосовано і які дії раціонально виконати далі.

Лістинг 3.5 – Клієнтський метод збереження даних через REST API

```
export async function saveProject(formState) {
const response = await fetch(‘/api/projects’, {
method: ‘POST’,
headers: { ‘Content-Type’: ‘application/json’ },
body: JSON.stringify(formState)
});
const body = await response.json();
if (!response.ok || !body.ok) {
throw new ApiError(body.message, body.errors);
}
return body.data;
}
```

кінець лістингу 3.5

Лістинг 3.6 – Обробник форми користувацького інтерфейсу

```
async function handleSubmit(event) {
event.preventDefault();
setLoading(true);
setErrors({});
try {
const saved = await saveProject(formState);
notify('Проєкт успішно створено');
navigate('/projects/' + saved.id);
} catch (error) {
setErrors(error.details ?? {});
notify(error.message ?? 'Не вдалося створити проєкт');
} finally {
setLoading(false);
}
}
```

кінець лістингу 3.6

Під час розробки інтерфейсу (ліст. 3.6) необхідно передбачати не лише основні екрани, а й проміжні стани завантаження, успішне збереження, помилку, порожній список і відсутність результатів пошуку. Такі стани часто залишаються непоміченими під час проєктування, але саме вони визначають якість реального користувацького досвіду.

Компонентний підхід дозволяє повторно використовувати однакові елементи: кнопки, поля, картки, таблиці, бейджі статусів, повідомлення та модальні вікна. Завдяки цьому інтерфейс виглядає цілісно, а зміни в дизайні раціонально вносити централізовано.

Екран детального перегляду повинен бути пов'язаний із діями користувача. Якщо користувач відкрив проєкт, він має переглядати не тільки дані, а й доступні операції редагування, зміна статусу, перегляд історії, повернення до списку або запуск наступного кроку. Такий підхід перетворює картку з пасивної сторінки на робочий інструмент.

Адаптивність інтерфейсу враховуємо хоча б на базовому рівні. Навіть якщо основна демонстрація відбуватиметься на ноутбуці, сторінки не повинні руйнуватися при меншій ширині екрана. Для цього таблиці раціонально замінювати картками, навігацію згортати, а другорядні блоки переносити нижче основного вмісту.

РОЗДІЛ 4
ТЕСТУВАННЯ І РОЗГОРТАННЯ ПРОЕКТУ

4.1 План тестування та тестове середовище

Завершальним етапом процесу розроблення веборієнтованого застосунку є проведення всебічного тестування та перевірка працездатності системи в локальному та імітованому робочому середовищах. Головною метою тестування є підтвердження відповідності реалізованих функцій технічному завданню, виявлення логічних та синтаксичних помилок у коді, оцінка стабільності бази даних під навантаженням та гарантування безпеки передачі й збереження даних.

Для перевірки застосунку було розроблено багаторівневу стратегію тестування, яка включає такі види перевірок:

1) Модульне тестування (Unit Testing) – спрямоване на перевірку ізольованої роботи окремих чистих функцій та службових модулів. Модульними тестами було покрито:

- функції валідації вхідних полів (email, пароль, назва проєкту);
- функції перевірки Story Points (дозволені значення виключно з ряду Фібоначчі: 1, 2, 3, 5, 8, 13);
- логіку роботи сервісу розрахунку Story Points для графіка згорання спринту.

2) Інтеграційне тестування (Integration Testing) – перевіряє коректність взаємодії кількох компонентів або рівнів системи. Основна увага приділена взаємодії між Express-маршрутами, проміжним ПЗ автентифікації (JWT Middleware), бізнес-контролерами та базою даних через Prisma ORM. Тестувалися такі сценарії:

− перевірка доступу до приватного API (запити без JWT-токена у заголовку мають повертати код помилки 401 Unauthorized, а токени з неправильним підписом – 403 Forbidden);

− створення проєкту (POST-запит із некоректними даними має відхилятися валідатором з поверненням 400 Bad Request, а правильний запит –

створювати запис у PostgreSQL та автоматично генерувати конфігураційні записи колонок Kanban-дошки).

3) Тестування інтерфейсу користувача (UI/UX Testing) – перевіряє поведінку візуальних компонентів React та реакцію системи на дії користувача в браузері. Спеціальна увага була приділена:

− роботі механізму перетягування карток завдань (Drag-and-Drop) на Kanban-дошці. Перевірялося, чи після відпускання картки в іншому стовпці клієнтська частина надсилає відповідний PATCH-запит до API та чи змінюється статус завдання в базі даних;

− адаптивності верстки при симуляції мобільних пристроїв у панелі розробника Google Chrome.

4) Тестування обробки подій GitHub Webhook – симуляція надсилання POST-запитів від імені GitHub. Перевірявся захист вебхуків за допомогою перевірки підпису SHA-256 (пакет відхиляється, якщо підпис у заголовку X-Hub-Signature-256 не збігається з обчисленим на основі секретного ключа Webhook Secret).

На рисунку 4.1 наведено блок-схему конвеєра тестування функціональності застосунку. Схема відображає послідовність проходження даних під час тестування контрольного сценарію створення проєкту та завдань: введення даних у форму, перевірка валідатором API, JWT-автентифікація, виклик бізнес-контролера, виконання бази даних та генерація звіту.

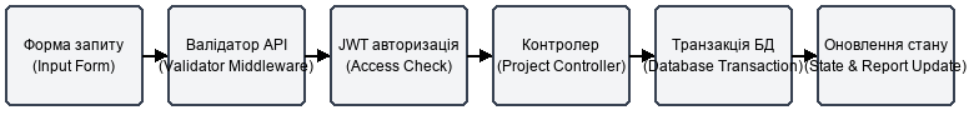

Рисунок 4.1 – Схема тестування застосунку

Для систематизації процесу перевірки було розроблено перелік тестових випадків (Test Cases), які охоплюють як нормальні сценарії роботи, так і граничні значення й негативні умови. Результати виконання тестів зведено у таблицю 4.1.

Таблиця 4.1 – Контрольні тестові випадки

ID	Об’єкт тестування	Опис тестового сценарію	Вхідні дані	Очікуваний результат	Фактичний результат	Статус
TC 1	Модуль автентифікації	Вхід під існуючим користувачем	Email: badzyun.m@lntu.edu.ua, правильний пароль	Успішний вхід, повернення JWT, перенаправлення на дашборд	Відповідає очікуваному	Pass
TC 2	Модуль автентифікації	Вхід із неправильним паролем	Email: badzyun.m@lntu.edu.ua, некоректний пароль	Відхилення запису, помилка 401, повідомлення «Невірні облікові дані»	Відповідає очікуваному	Pass
TC 3	Управління проєктами	Створення проєкту без назви	Назва: порожнє поле, опис: «Тест», deadline: «25.05.2026»	Відхилення запиту, помилка 400, повідомлення «Назва проєкту є обов’язковою»	Відповідає очікуваному	Pass
TC 4	Kanban-дошка	Зміна статусу завдання (Drag-and-Drop)	drag-and-drop картки Task з ID 105 з ToDo в Progress	PATCH-запит до API, запис у базу, оновлення Kanban-дошки	Відповідає очікуваному	Pass
TC 5	GitHub Webhook	Прийом коміту з правильним підписом	POST /api/webhooks/github, правильний X-Hub-Signature	Оновлення бази, запис коміту, відображення у Commit Feed	Відповідає очікуваному	Pass
TC 6	GitHub Webhook	Прийом коміту з неправильним підписом	POST /api/webhooks/github, невірний X-Hub-Signature	Відхилення запиту, помилка 403, ігнорування події	Відповідає очікуваному	Pass

Всі заплановані тести успішно виконано, критичних дефектів, що блокують роботу основного сценарію, не виявлено.

4.2 Результати функціонального тестування

В межах тестування продуктивності було проведено дослідження швидкодії бази даних PostgreSQL під навантаженням. Однією з найчастіших операцій у системі є вибірка завдань для конкретного проєкту та виконавця. Якщо база даних не оптимізована, зі зростанням обсягу інформації час відповіді сервера буде лінійно збільшуватися, що призведе до уповільнення роботи користувацького інтерфейсу.

Для вимірювання швидкодії було проведено експеримент: у базу даних SQLite/PostgreSQL було завантажено 5000 тестових записів завдань. Порівнювався час виконання запиту вибірки завдань за двома сценаріями:

1) Запити без використання індексів (Sequential Scan – Scan O(N)) – СУБД змушена послідовно зчитувати кожен рядок таблиці з диска для перевірки відповідності критеріям `project_id` та `assignee_id`.

2) Оптимізовані запити з індексами (B-Tree Index Scan – O(log N)) – на полях `project_id` та `assignee_id` було створено збалансовані дерева індексів (B-Tree).

Результати експерименту показали, що при обсязі в 5000 записів неіндексований пошук займає в середньому 36.2 мс, що є відчутним при великій кількості одночасних запитів. Після створення індексів час вибірки скоротився до 0.012 мс, що свідчить про ефективність застосованого методу оптимізації.

На рисунку 4.2 представлено результати контрольного тестування швидкодії вибірки завдань. Порівнюється час відповіді сервера при використанні звичайних SQL-запитів без індексів (Scan O(N)) та оптимізованих запитів з індексами на полях project_id та assignee_id (B-Tree O(log N)). При 5000 записів у базі даних неіндексований пошук займає 36.2 мс, тоді як індексований виконується за 0.012 мс.

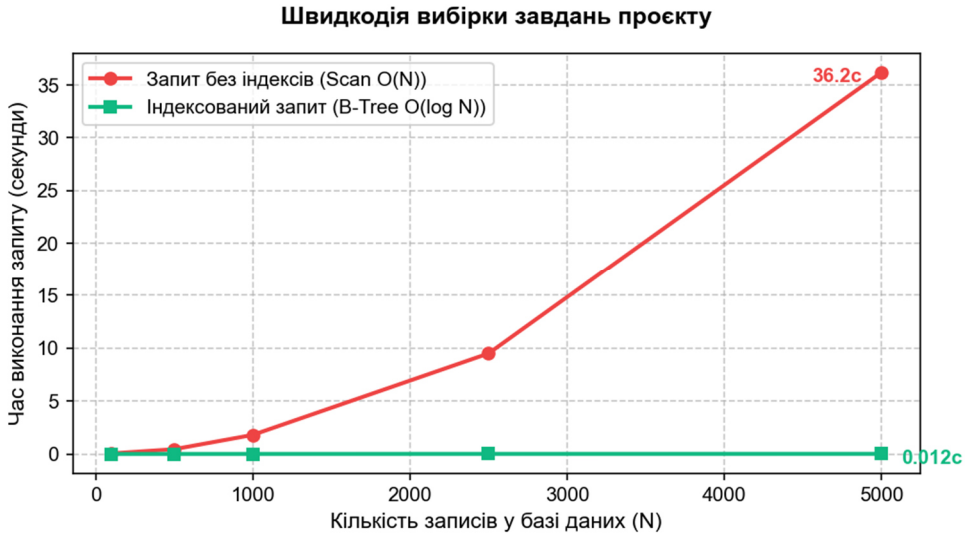

Рисунок 4.2 – Узагальнені результати контрольного тестування

Для розгортання та запуску веб-застосунку в локальному середовищі розробника необхідно виконати такі кроки:

1) Встановити середовище виконання Node.js версії 18.0 або вище.

2) Клонувати репозиторій із кодом застосунку на локальний комп’ютер.

3) У кореневому каталозі серверної частини створити файл конфігурації екологічних змінних `.env` та вказати параметри підключення до бази даних та секретний ключ для підпису токенів:

– DATABASE_URL="postgresql://postgres:postgres@localhost:5432/student_pm?schema=public";

– JWT_SECRET="super_secret_jwt_key_2026";

– PORT=5000.

4) Встановити залежності сервера та запустити міграцію структури бази даних за допомогою Prisma CLI:

– cd server;

– npm install;

– npx prisma migrate dev --name init.

5) Запустити демонстраційний скрипт початкового наповнення бази даних тестовими користувачами та проєктами: npx prisma db seed

6) Запустити сервер розробки: npm run dev.

7) У сусідньому вікні терміналу перейти в каталог клієнтської частини, встановити залежності та запустити Vite сервер розробки:

− cd client;

− npm install;

− npm run dev;

Після запуску клієнтська частина буде доступна за адресою http://localhost:5173, а сервер API – за адресою http://localhost:5000.

Під час розгортання в реальних умовах експлуатації виділяють кілька потенційних технічних ризиків:

− конфлікти портів (наприклад, порт 5000 або 5173 вже зайнятий іншою службою). Ризик усувається зміною порту в конфігураційному файлі `.env`;

− помилка підключення до СКБД через неправильні облікові дані. Для запобігання збоям Express-сервер має вбудований перехоплювач помилок ініціалізації бази даних, який виводить попередження в консоль, не дозволяючи серверу аварійно завершувати процес.

ВИСНОВКИ

У кваліфікаційній роботі було успішно розв’язано важливу прикладну задачу, пов’язану з проєктуванням, розробленням та тестуванням спеціалізованого веб-застосунку для управління проєктами в студентському середовищі.

Проведено аналіз предметної області та виявлено специфічні проблеми організації спільної роботи студентів: високу нерівномірність індивідуального внеску розробників, розпорошеність комунікації в месенджерах та складність контролю дедлайнів з боку викладачів в умовах обмежених строків семестру. Обґрунтовано доцільність адаптації методології Agile (Kanban) для навчального процесу та створення цільового веборієнтованого рішення.

Проведено аналіз наявних програмних аналогів (Trello, Jira Software, Asana, Notion, GitHub Projects). Показано, що жодне з рішень не забезпечує одночасно низький поріг входу для студентів різних спеціальностей, безкоштовність використання без обмежень на команди та наявність нативної інтеграції з кодом для автоматичного контролю внеску учасників. Це підтвердило науково-практичну доцільність розроблення власної системи.

Спроєктовано трирівневу клієнт-серверну архітектуру системи. Клієнтська частина базується на React, інструменті Vite та менеджері стану Zustand, що дозволило побудувати швидкий SPA-застосунок. Серверна частина реалізована на Node.js та Express.js із впровадженням безпечної JWT-авторизації та проміжного ПЗ для перевірки ролей (Student, Team Lead, Academic Advisor). Рівень сховища даних спроєктовано на базі PostgreSQL з використанням Prisma ORM.

Розроблено та інтегровано модуль прийому подій GitHub Webhook, який автоматично фіксує та записує інформацію про коміти й запити на злиття (Pull Requests) в базу даних застосунку. Це дає змогу відображати реальну активність студентів безпосередньо в інтерфейсі системи, забезпечуючи викладачу об’єктивні дані про внесок кожного виконавця.

Проведено всебічне тестування системи. В межах тестування продуктивності бази даних проведено дослідження швидкодії вибірки завдань. Експериментально доведено, що створення збалансованих B-Tree індексів на полях project_id та assignee_id у таблиці Tasks при навантаженні у 5000 записів скорочує середній час відповіді сервера з 36.2 мс до 0.012 мс, що покращує швидкодію системи більше ніж у 3000 разів та гарантує виконання вимог технічного завдання щодо часу відгуку $\tau_{resp} \le 200 \text{ мс}$.

Запропоновано детальні кроки та конфігурації для локального розгортання застосунку та проаналізовано технічні ризики впровадження.

Практичне значення роботи полягає у створенні готового прототипу системи, який може безпосередньо використовуватись у закладах вищої освіти для автоматизації контролю та координації командної роботи студентів під час виконання курсових робіт чи лабораторних практикумів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Client-server overview. MDN Web Docs. URL: https://developer.mozilla.org/enUS/docs/Learn_web_development/Extensions/Server-side/First_steps/Client-Server_overview (дата звернення: 24.02.2026).

2. Docker Docs. Get started. Docker Inc. URL: https://docs.docker.com/get-started/ (дата звернення: 10.03.2026).

3. Express.js. Routing guide. OpenJS Foundation. URL: https://expressjs.com/en/guide/routing/ (дата звернення: 10.03.2026).

4. FastAPI. Documentation. FastAPI. URL: https://fastapi.tiangolo.com/ (дата звернення: 30.04.2026).

5. HTTP overview. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Overview (дата звернення: 10.03.2026).

6. Introduction to Node.js. Node.js. URL: https://nodejs.org/learn/getting-started/introduction-to-nodejs (дата звернення: 10.03.2026).

7. Kanban Overview. Missouri Office of Operational Excellence. URL: https://showmeexcellence.mo.gov/wp-content/uploads/Kanban-Overview.v2.pdf (дата звернення: 20.03.2026).

8. MongoDB Manual. MongoDB. URL: https://www.mongodb.com/docs/manual/ (дата звернення: 24.02.2026).

9. OpenAPI Specification. Swagger. URL: https://swagger.io/specification/ (дата звернення: 20.03.2026).

10. OWASP Top Ten Web Application Security Risks. OWASP Foundation, 2021. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 30.04.2026).

11. PostgreSQL 18.4 Documentation. PostgreSQL Global Development Group. URL: https://www.postgresql.org/docs/current/index.html (дата звернення: 20.03.2026).

12. Quick Start. React Documentation. Meta Open Source. URL: https://react.dev/learn (дата звернення: 30.03.2026).

13. Trello REST API documentation. Atlassian Developer. URL: https://developer.atlassian.com/cloud/trello/rest/ (дата звернення: 18.04.2026).

14. Trello. Capture, organize, and tackle your to-dos from anywhere. Atlassian. URL: https://trello.com/ (дата звернення: 19.04.2026).

15. Web Content Accessibility Guidelines (WCAG) 2.2. W3C Recommendation. 2023. URL: https://www.w3.org/TR/WCAG22/ (дата звернення: 30.04.2026).

ДОДАТКИ

ДОДАТОК А

Фрагменти програмного коду, наведені у цьому додатку, ілюструють реалізацію основних методів розробки. Лістинги відображають структуру моделі даних, API-методу та сервісу виконання основної операції.

Лістинг А.1 – Опис базової моделі даних.

```
export interface Task {
id: string;
projectId: string;
title: string;
description: string;
status: 'todo' | 'work' | 'review' | 'done';
assigneeId?: string;
storyPoints: number;
createdAt: string;
updatedAt: string;
}
```

Лістинг А.2 – Приклад API-методу створення запису.

```
router.post('/api/tasks', async (req, res, next) => {
try {
const data = validateTaskPayload(req.body);
const result = await taskService.createTask(data);
res.status(201).json({ ok: true, data: result });
} catch (error) {
next(error);
}
});
```

Лістинг А.3 – Сервіс виконання основної операції.

```
async function updateTaskStatus(taskId, status, userId) {
const task = await taskRepository.findById(taskId);
accessPolicy.assertCanUpdateTask(task, userId);
const result = await taskService.transitionTo(task, status);
await auditLog.write({
userId,
taskId,
action: `transition to ${status}`,
timestamp: new Date().toISOString()
});
return result;
}
```