МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ШМАНОВ ЯРОСЛАВ КОСТЯНТИНОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
канд. техн. наук, доцент
_____ О.В. Зелінська
«_____» _____ 2025 р.

ВЕБДОДАТОК ДЛЯ СПІЛЬНОЇ КОМУНІКАЦІЇ РОЗРОБНИКІВ ІТ-ПРОЄКТУ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Потапова Н. А., доцент кафедри
інформаційних технологій,
к. е. н., доцент

_____

Оцінка: _____/_____ / _____
(бали за шкалою ЄКТС / за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2025

АНОТАЦІЯ

Шманов Я.К. Вебдодаток для спільної комунікації розробників IT-проєкту. Спеціальність 122 «Комп’ютерні науки», освітня програма «Комп’ютерні науки». Донецький національний університет ім. Василя Стуса, Вінниця, 2025.

У кваліфікаційній (бакалаврській) роботі досліджено принципи розробки вебдодатків для управління проєктами та завданнями. Був розроблений вебдодаток, для надання користувачам інструменту для візуального управління проєктами та завданнями за Kanban-методологією, включаючи створення, редагування, відстеження статусів та переміщення завдань за допомогою drag-and-drop. Реалізовано систему автентифікації та управління користувачами.

Ключові слова: вебдодаток, управління проектами, Kanban, управління завданнями, React, Redux, Node.js, Express, PostgreSQL, REST API, веб-розробка, drag-and-drop.

50 с., 4 рис., 40 джерел.

ABSTRACT

Shmanov Y.S. Web application for joint communication of IT project developers. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia 2025.

In the qualification (bachelor’s) work, the principles of developing web applications for project and task management were researched. The main purpose of the developed application is to provide users with a tool for visual management of projects and tasks using the Kanban methodology, including creating, editing, tracking statuses, and moving tasks using drag-and-drop. An authentication and user management system was implemented.

Keywords: web application, project management, Kanban, task management, React, Redux, Node.js, Express, PostgreSQL, REST API, web development, drag-and-drop.

50 p., 4 figures, 40 sources.

ЗМІСТ

ВСТУП

В умовах стрімкого розвитку інформаційних технологій, управління проєктами та завданнями стає ключовим фактором успіху як для окремих фахівців, так і для команд будь-якого масштабу. Зростання складності завдань, необхідність координації роботи розподілених команд та прагнення до підвищення продуктивності зумовлюють високий попит на цифрові інструменти, що допомагають організовувати робочі процеси.

Водночас ринок програмного забезпечення пропонує широкий спектр таких рішень, які часто відрізняються за складністю, вартістю та можливостями кастомізації. Існує постійна потреба у розробці нових, гнучких та інтуїтивно зрозумілих вебдодатків, які б використовували переваги сучасних технологій веб-розробки, таких як Node.js для серверної частини та React для клієнтської, забезпечуючи високу продуктивність, масштабованість та зручний користувацький досвід. Розробка власного Kanban-додатку на базі актуального технологічного стеку є актуальною задачею як з практичної точки зору (створення конкурентного продукту), так і з науково-технічної (дослідження та застосування сучасних підходів до веб-розробки).

Аналіз існуючих систем управління проєктами та завданнями показує наявність багатьох потужних платформ (Trello, Jira, Asana тощо). Однак, часто користувачі стикаються з вибором між простими, але функціонально обмеженими інструментами, та складними, перевантаженими функціями корпоративними системами. Залишається ніша для вебдодатків, які пропонують збалансований набір функцій для ефективного управління за Kanban-методологією, мають сучасний, швидкий та адаптивний інтерфейс, і побудовані на технологіях, що забезпечують легкість підтримки та подальшого розвитку.

Метою роботи є проєктування та розробка повнофункціонального вебдодатку для спільної комунікації розробників IT-проєкту по управлінню завданнями, який реалізує візуальний підхід на основі Kanban-методології та надає користувачам інструменти для організації робочих процесів та командної

взаємодії, з використанням сучасного стеку веб-технологій (Node.js, Express, PostgreSQL, React, Redux).

Для досягнення поставленої мети необхідно було вирішити наступні *завдання дослідження*:

1. Провести аналіз предметної області управління проєктами за Kanban-методологією та огляд існуючих програмних рішень.
2. Обґрунтувати вибір технологій для розробки клієнтської та серверної частин додатку.
3. Спроєктувати клієнт-серверну архітектуру системи та розробити структуру бази даних для зберігання інформації про користувачів, проекти, колонки та завдання.
4. Реалізувати серверну частину (Backend) додатку з використанням Node.js, Express, Sequelize, включаючи розробку RESTful API та механізмів автентифікації/авторизації.
5. Розробити клієнтську частину (Frontend) як односторінковий додаток (SPA) з використанням React та Redux, реалізувати користувацький інтерфейс Kanban-дошки з функціоналом drag-and-drop.
6. Забезпечити взаємодію між клієнтською та серверною частинами через розроблене API.
7. Провести тестування основних функціональних можливостей розробленого вебдодатку.

Об'єктом дослідження є процес проєктування та розробки вебдодатків для управління проєктами та завданнями.

Предметом дослідження є моделі, методи та технології побудови вебсистеми для управління завданнями за Kanban-методологією з використанням стеку технологій, включаючи архітектуру додатку, структуру бази даних, реалізацію RESTful API, управління станом клієнтської частини та реалізацію інтерактивного користувацького інтерфейсу.

Структура роботи складається зі вступу, трьох основних розділів, висновків, списку використаних джерел та додатків. У першому розділі

проводиться аналіз предметної області та існуючих рішень, обґрунтовується вибір технологій, проектується архітектура системи та структура бази даних. Другий розділ присвячений обґрунтуванню вибору технологій та інструментів для розробки вебдодатку. В третьому розділі наведено результати практичної розробки серверної та клієнтської частин додатку, а також механізмів налаштування проєкту, реалізація UI компонентів, управління станом за допомогою Redux, взаємодія з API та реалізація ключового функціоналу Kanban-дошки.

Практичне значення роботи полягає в тому, що дана розробка може бути використана як основна складова системи управління для команди розробників IT-проєкту в компаніях різного розміру та форми власності.

Апробація результатів даної роботи була представлена на VI Всеукраїнській науково-практичній конференції «Прикладні інформаційні технології» (м. Вінниця, 22 травня 2025 року) в доповіді «Вебдодаток для спільної комунікації розробників в межах IT-проєкту».

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ ПРОЕКТУВАННЯ ВЕБДОДАТКУ

1.1 Теоретичні засади управління ІТ-проєктами

У галузі виробництва програмного забезпечення найбільш популярною темою, навколо якої ведеться багато суперечок є методології для управління ІТ проєктами. Всі фахівці шукають найбільш прийнятний варіант для реалізації управління командами розробників.

Поняття "проєкт" об’єднує різноманітні види діяльності, що характеризуються рядом загальних ознак, найбільш загальними з яких є наступні:

- спрямованість на досягнення конкретних цілей, визначених результатів;
- координоване виконання численних взаємозалежних дій;
- обмеженість у ресурсах та у часі з певним початком і кінцем.

Відмінність проєкту від виробничої діяльності полягає в тому, що проєкт є одноразовою, нециклічною діяльністю. Цим проєкт відрізняється від бізнес-процесів, дії яких можуть багаторазово повторюватись у вигляді екземплярів бізнес-процесів. Виробничі цикли в чистому вигляді не є проєктами. Однак останнім часом проєктний підхід все частіше застосовується і до процесів, орієнтованим на безперервне виробництво. Проєкт як система діяльності існує рівно стільки часу, скільки його потрібно для отримання кінцевого результату. Концепція проєкту, однак, не суперечить концепції ІТ-компанії і цілком сумісна з нею. Проєкт часто стає основною формою діяльності ІТ-компанії.

В основі управління ІТ-проєктами покладені принципи управління на основі жорсткої та гнучкої методології. Елементи підходу управління на основі гнучких методологій були систематизовані в Agile маніфесті. Основними серед них є:

- Люди та їх взаємодія важливіші за процеси та інструменти.
- Готовий продукт важливіший за документацію по ньому.

Принцами Agile підходу є:

1. Наш вищий пріоритет – це задоволення замовника за допомогою частих та безперервних поставок продукту, який цінний для нього.

2. Ми приймаємо зміни у вимогах до проєкту, навіть на пізніх етапах реалізації проекту.

3. Представники бізнесу і команда розробки повинні працювати разом над проєктом.

4. Успішні проєкти будуються мотивованими людьми. Робоче програмне забезпечення – головна міра прогресу проєкту.

5. Гнучкі процеси сприяють безперервному розвитку. Всі учасники проєкту повинні вміти витримувати постійний темп.

6. Архітектура, вимоги, дизайн створюються в командах, що самоорганізуються.

7. Команда постійно шукає способи стати більш ефективною, шляхом налаштування та адаптації своїх процесів.

Найбільш доцільною з позиції управління командами в ІТ-проєктах стала методологія Канбан. Канбан є стратегією оптимізації потоку поставки цінності за допомогою процесу, який використовує візуальну систему, що витягує процеси, має обмеження незавершеної роботи.

Центральне місце в визначенні Канбана займає потік – це рух цінності по всій системі розробки продукту, циклу прозорості, інспекції та адаптації, тобто час циклу зворотного зв’язку. Канбан реалізується в межах проєктів побудованих за фремворком Скрам, команди розробників є Скрам-командами.

Чотири основні метрики потоку, які слід відстежувати Скрам-командам, що використовують Канбан це:

1. Незавершена робота – кількість робочих елементів, які розпочаті, але незакінчені. Команда може використовувати метрику незавершеної роботи для забезпечення прозорості свого прогресу щодо її скорочення та покращення свого потоку виконаних вимог.

2. Час циклу – часовий інтервал між початком та завершенням роботи над робочим елементом.

3. Час життя робочого елемента – часовий інтервал між початком роботи над робочим елементом та поточним часом. Це стосується лише робочих елементів у процесі.

4. Пропускна здатність – кількість завершених робочих елементів за одиницю часу.

Ключовий постулат теорії управління рухом потоку по Канбан – Закон Літтла. Закон Літтла говорить, що для процесу з цією пропускною спроможністю, що більше елементів перебувають у роботі у кожен час (у середньому), то більше необхідно часу (у середньому) для їх завершення. Якщо час циклів дуже великий, перше, що варто зробити – це зменшити незавершені роботи. Більшість інших елементів Канбана спираються на взаємозв'язок між незавершеною роботою та часом циклу. Закон Літтла також демонструє, як теорія потоку покладається на емпіризм, використовуючи метрики потоку та дані для того, щоб досягти прозорості історичних даних про потік, а потім застосувати їх в експериментах з моніторингу та адаптації.

Скрам-команди розробників можуть оптимізувати потік за допомогою наступних чотирьох складових: візуалізація потоку, обмеження, активне управління незавершеними робочими елементами, моніторинг та адаптація.

Візуалізація за допомогою Канбан дошки – це спосіб зробити життєвий цикл робочих елементів команди розробників прозорим. Конфігурація дошки повинна підштовхувати команду до необхідних обговорень у потрібний момент та активно пропонувати можливості для покращення. Візуалізація повинна включати наступне:

• Визначено точки, в яких Скрам-команда вважає роботу розпочатою та завершеною.

• Визначення робочих елементів – індивідуальних елементів із створення цінності, тобто, які проходять через систему Скрам-команди.

• Визначення етапів життєвого циклу, такого, що робочі елементи проходять від його початку до кінця.

• Явні правила переходу робочих елементів з одного стану до іншого.

• Правила, які обмежують кількість незавершених робіт.

Незавершені роботи є робочими елементами, які Скрам-команда взяла в роботу та ще не закінчила. Скрам-команди, які використовують Канбан, повинні явно обмежувати кількість незавершених робочих елементів. Скрам-команда може явно задати це обмеження на свій розсуд, але далі повинна його дотримуватися.

Основний результат використання обмеження для незавершених робіт –це створення системи, що витягує. Витягуюча система отримала свою назву тому, що ІТ-команда розробників бере елемент в роботу (втягує) тільки тоді, коли у ІТ-команди з'являється можливість його взяти. Коли незавершена робота опускається нижче встановленого ліміту, це сигнал, що можна брати в роботу новий елемент. Слід зазначити, що така система відрізняється від штовхаючої системи, в якій робочі елементи беруться в роботу відразу після появи, незалежно від наявності можливості їх обробити. Обмеження незавершеної роботи допомагає потоку та покращує самоорганізацію, сфокусованість, відданість та взаємодію всередині команди розробників. Тому, одним із центральних питань є створення таких вебдодатків, які змогли б вдало реалізувати комунікацію між розробниками та моніторинг виконання їх завдань.

1.2 Аналіз існуючих рішень для управління завданнями та проєктами

Сьогодні вміння організовувати роботу над завданнями та проектами – це не просто корисний навик, а обов'язкова вимога. Незалежно від того, чи працюєте ви самотужки, чи в складі команди, без чіткої системи керування процесами ефективність різко падає. Проєкти ускладнюються, кількість учасників зростає, і в таких умовах на допомогу приходять цифрові інструменти. Варіантів розвитку систем управління проєктами та завданнями є безліч: від елементарних списків справ до комплексних корпоративних систем.

Найпопулярніші рішення для управління завданнями відрізняються за:

- аудиторією (індивідуальні користувачі, малі команди, великі компанії);
- підтримуваними методологіями (Kanban, Scrum, Waterfall, Agile);

- функціоналом (базовий чи розширений);

- способом розгортання (хмарні сервіси чи локальні інсталяції).

Розглянемо основні програмні інструменти управління проєктами та завданнями:

1. Trello. Один із найпростіших та найзручніших варіантів для роботи з Kanban-дошками. Основні переваги – інтуїтивний інтерфейс, візуальна наглядність та легкість у використанні. Завдання представлені у вигляді карток, які можна переміщати між колонками (етапами виконання). Ідеально підходить для невеликих проектів, але для складних завдань з аналітикою та контролем ресурсів його можливостей може бути недостатньо.

2. Jira. Потужний інструмент від Atlassian, який став стандартом для IT-команд. Підтримує Agile-методології (Scrum, Kanban), дозволяє гнучко налаштовувати процеси, інтегрується з іншими сервісами (Bitbucket, Confluence) та надає детальні звіти. Однак його часто критикують за складність у освоєнні, необхідність тонкого налаштування та високу вартість для великих організацій.

3. Asana. Універсальний інструмент, який поєднує різні формати роботи: списки, Kanban-дошки, календарі та діаграми Ганта. Підтримує командну роботу, автоматизацію рутинних завдань, управління цілями та портфелями проектів. Але для простих індивідуальних завдань він може здатися занадто навантаженим.

4. Monday.com. Позиціонується як «операційна система для роботи» (Work OS), що підкреслює її гнучкість. Має яскравий інтерфейс, безліч шаблонів для різних сфер (маркетинг, розробка, продажі), потужні інструменти автоматизації та інтеграції. Головний мінус – висока ціна, особливо для малих команд.

5. ClickUp. Амбітний проект, який намагається замінити собою кілька інструментів одночасно. Пропонує різні види візуалізації (дошки, списки, календарі, діаграми Ганта, Mind Maps), вбудовані документи, чат та інші функції. Безкоштовна версія досить функціональна, але через велику кількість можливостей інтерфейс може здаватися перевантаженим.

6. Microsoft Project & Planner. Microsoft Project – потужний інструмент для

класичного управління проектами з детальним плануванням, діаграмами Ганта та контролем ресурсів. Орієнтований на великі компанії. Microsoft Planner – спрощений варіант для команд, інтегрований з Office 365 та Teams. Нагадує Trello, але з акцентом на корпоративні потреби.

Будь-який сучасний інструмент управління завданнями має базовий набір функцій: створення завдань, призначення відповідальних, встановлення дедлайнів, коментування та прикріплення файлів. Більш просунуті системи додають:

- різні види візуалізації (дошки, календарі, діаграми Ганта);
- автоматизацію робочих процесів;
- контроль залежностей між завданнями;
- аналітику та звіти.

Вибір конкретного рішення залежить від розміру команди, типу проектів, використовуваних методологій та бюджету. Однак навіть при такому розмаїтті варіантів залишається місце для нових рішень – більш спеціалізованих, інтегрованих або побудованих на сучасних технологіях. Вивчення існуючих інструментів допомагає визначити ключові функції та оптимальні підходи для розробки власного рішення.

1.3 Підходи до розробки додатку для управління командами в межах ІТ-проєкту

Попри різноманітність інструментів для управління проектами, багато з них мають суттєві недоліки. Одні надмірно складні для малих команд і вимагають значних ресурсів, інші – занадто обмежені, не дозволяють адаптуватися під конкретні потреби або не використовують сучасні технологічні рішення. Це створює потребу в інтуїтивних, гнучких та технологічно просунутих системах, орієнтованих на спільну роботу.

Серед різноманіття методологій управління проектами, Kanban здобув значну популярність завдяки своїй візуальній простоті, гнучкості та фокусі на безперервному потоці робіт. Kanban-дошки дозволяють командам наочно

відстежувати статус завдань, виявляти вузькі місця у процесах та ефективно розподіляти навантаження. Вебдодатки, що реалізують Kanban-методологію, стали невід'ємною частиною інструментарію багатьох сучасних компаній та індивідуальних користувачів.

Функціонально вебдодаток для управління завданнями має поєднувати:

- Візуальний підхід на основі Kanban-методології.
- Сучасний стек технологій для швидкої розробки та простоти підтримки.
- Зручність у використанні для команд будь-якого розміру.

Підхід в розробці має ґрунтуватись на виконанні ключових етапів: проектування архітектури, розробка серверної частини, створення клієнтської частини, інтеграція компонентів, тестовий контроль.

1. Проектування архітектури передбачає:
- визначення структури клієнт-серверної взаємодії;
- розробку схеми бази даних (користувачі, проекти, завдання, колонки);
- вибір формату обміну даними між клієнтом і сервером (наприклад, JSON).

2. Розробка серверної частини (Backend) має реалізувати функціональні вимоги за допомогою технологій: Node.js, Express, PostgreSQL, Sequelize (ORM). Реалізований функціонал включає:
- RESTful API для взаємодії з клієнтом;
- Система аутентифікації (JWT) та авторизації користувачів;
- CRUD-операції для керування проектами, колонками, завданнями;
- Логіка додавання/видалення учасників проектів.

3. Створення клієнтської частини (Frontend) базується на використанні технологій: React, Redux (для управління станом), SPA-архітектура. – Особливостями розробки клієнтської частини є:
- Інтерактивний Kanban-інтерфейс з drag-and-drop функціоналом;
- Форми для реєстрації, входу, створення/редагування завдань;
- Візуалізація деталей завдань (опис, мітки, відповідальні);
- Синхронізація з сервером через API.

4. Інтеграція компонентів вебдодатку враховує:

- Налаштування стабільного зв’язку між Frontend і Backend;

- Перевірка коректності передачі даних (наприклад, оновлення статусу завдання після перетягування).

5. Тестування функціонування вебдодатку передбачає: перевірку роботи основних сценаріїв; виявлення та усунення помилок у логіці взаємодії.

Пріоритетом розробки має бути створення стабільного ядра додатку з основним функціоналом для управління завданнями за методологією Kanban. Очікуваний результат – робочий прототип вебдодатку, який: демонструє можливість організації проектів через Kanban-дошки; готовий до впровадження в невеликих командах; слугує основою для подальшого розширення функціоналу (додавання модулів, інтеграцій).

Такий підхід дозволить розробці стати альтернативою існуючим рішенням, поєднуючи простоту, сучасні технології та орієнтацію на потреби користувачів.

РОЗДІЛ 2
ТЕХНОЛОГІЇ ТА ІНСТРУМЕНТИ ДЛЯ ПРОЄКТУВАННЯ ВЕБДОДАТКУ

2.1. Вибір технологій та інструментів розробки

Вибір інструментів для створення програмного забезпечення – критично важливий етап, який впливає на продуктивність, масштабованість та подальшу підтримку додатку. Для веб-застосунку з управління проектами потрібні технології, що поєднують швидкість розробки, стабільність роботи та зручність для користувачів. У цьому розділі розглядаються рішення, обрані для серверної та клієнтської частин, а також їх переваги.

Розглянемо технології та інтрументи для розробки серверної частини (Backend) вебдодатку:

1. Вибір платформи Node.js обумовлений:

- Подієвою архітектурою – ефективна обробка багатьох запитів одночасно;
- Уніфікацією мови – JavaScript використовується і на сервері, і на клієнті, що спрощує розробку.

2. Вибір фреймвору Express.js обгрунтовується тим, що це:

- Мінімалістичний, але потужний інструмент для побудови RESTful API;
- Велика спільнота, безліч готових middleware для розширення функціоналу.

3. Вибір СУБД PostgreSQL обумовлений критеріями:

- Надійна об’єктно-реляційна СУБД з підтримкою складних запитів;
- Відповідність стандартам ACID (надійність транзакцій).

4. ORM – Sequelize:

- Дозволяє працювати з базою даних через JavaScript-об’єкти;
- Підтримує міграції для контролю змін у структурі БД.

5. Автентифікація на основі JWT + Passport.js:

- JSON Web Tokens (JWT) для безпечної автентифікації без зберігання сесій;

- Passport.js – гнучкий фреймворк для різних стратегій входу.

6. Побудова безпеки на використанні bcryptjs:

- Хешування паролів перед збереженням у БД.

7. Додаткові інструменти:

- CORS – для безпечної взаємодії між клієнтом і сервером;

- Express Async Handler – спрощення обробки асинхронних помилок.

Клієнтська частина (Frontend) розробляється із використанням інтурментів:

1. Бібліотека React:

- Компонентний підхід для повторного використання коду;

- Virtual DOM для швидкого оновлення інтерфейсу.

2. Управління станом Redux:

- Централізоване сховище даних;

- Передбачуваність змін стану завдяки чистій архітектурі.

- Додатки: redux-thunk (асинхронні дії), redux-form (робота з формами).

3. Маршрутизація React Router:

- Навігація без перезавантаження сторінки (SPA).

4. HTTP-клієнт Axios:

- Простий інструмент для роботи з API;

- Підтримка перехоплення запитів/відповідей.

5. UI-бібліотеки:

- react-beautiful-dnd – інтерактивне перетягування завдань;

- FontAwesome – іконки;

- React Textarea Autosize – динамічні поля введення.

Інструменти розробки:

- npm – менеджер пакунків для встановлення залежностей;

- Create React App (react-scripts) – готові налаштування Webpack/Babel.

Обраний стек (Node.js + Express + PostgreSQL для Backend, React + Redux для Frontend) є сучасним, надійним і оптимальним для створення продуктивних вебдодатків. Він дозволяє:

- Використовувати JavaScript на всіх етапах розробки;

- Забезпечити швидку взаємодію з користувачем;

- Легко масштабувати проект у майбутньому.

Ці технології повністю відповідають цілям дипломної роботи та забезпечать стабільну основу для подальшого вдосконалення додатку.

2.2. Проектування архітектури вебдодатку

Після вибору технологічного стеку наступним ключовим етапом стало проектування архітектури системи. Для вебдодатку з управління проектами було обрано класичну клієнт-серверну модель, яка передбачає чіткий поділ функціональності між двома основними компонентами:

- Клієнтська частина (Frontend) – відповідає за інтерфейс користувача та взаємодію в реальному часі.

- Серверна частина (Backend) – обробляє бізнес-логіку, зберігає дані та забезпечує безпеку.

Зв’язок між компонентами здійснюється через REST API – стандартизований інтерфейс, що використовує HTTP-запити (GET, POST, PUT, DELETE) для обміну даними у форматі JSON. Чому саме така архітектура?

1. Гнучкість розробки:

- Frontend і Backend можуть розроблятися паралельно.

- Можливість використовувати різні технології (хоча в даному випадку JavaScript уніфікований для обох частин).

2. Масштабованість:

- Серверну частину можна розгортати на потужніших машинах при зростанні навантаження.

- Клієнтський код виконується на стороні користувача, що зменшує нагрузку на сервер.

3. Безпека:

- Чутливі дані (наприклад, паролі) обробляються лише на сервері.

- Клієнт отримує лише необхідну інформацію через API.

4. Підтримка різних платформ:

- Один Backend може обслуговувати вебдодаток, мобільні застосунки або інші сервіси.

Основні компоненти вебдодатку:

1. Клієнтська частина (Frontend):

- Технології: React, Redux, Axios, React Router

- Функціонал:

- Відображення інтерфейсу (Kanban-дошки, форми, списки завдань).

- Обробка дій користувача (перетягування завдань, редагування тексту).

- Надсилання запитів до API через Axios.

- Кешування даних за допомогою Redux для швидкої роботи.

- Місце виконання: Браузер користувача (SPA – односторінковий додаток).

2. Серверна частина (Backend):

- Технології: Node.js, Express, Sequelize, JWT

- Функціонал:

- Обробка запитів від клієнта (REST API).

- Автентифікація (Passport.js + JWT) та авторизація користувачів.

- Валідація даних перед збереженням у БД.

- Взаємодія з PostgreSQL через Sequelize.

- Місце виконання: Хмарний або локальний сервер.

3. База даних (PostgreSQL) :

- Структура: Реляційна модель з таблицями:

- `Users` (користувачі)

- `Projects` (проекти)

- `Tasks` (завдання)

- `Columns` (статуси/Kanban-колонки)

- Зв’язки: ORM Sequelize забезпечує зв’язки між таблицями (наприклад, "один проект має багато завдань").

4. API (RESTful):

- Формат обміну даними: JSON

- Приклади ендпоінтів:
  - `GET /api/projects` – отримати список проектів.
  - `POST /api/tasks` – створити нове завдання.
  - `PUT /api/tasks/:id` – оновити завдання.
- Захист:
  - JWT-токен у заголовках запитів для авторизованих дій.
  - CORS – дозвіл запитів лише з довірених джерел.

Схема взаємодії компонентів:

1. Користувач відкриває додаток у браузері → Frontend (React) завантажується.
2. При авторизації Frontend надсилає запит до `/api/login` → Backend перевіряє дані і повертає JWT-токен.
3. Після авторизації:
   - Frontend робить запит до `/api/projects` для отримання даних.
   - Backend звертається до БД через Sequelize і повертає відповідь у JSON.
   - React відображає Kanban-дошку з отриманими завданнями.
4. При перетягуванні завдання між колонками:
   - Frontend надсилає `PUT /api/tasks/:id` з новим статусом.
   - Backend оновлює запис у БД і підтверджує успіх.

Обрана архітектура (клієнт-серверна з REST API) є оптимальною для нашого завдання, оскільки:

- Забезпечує швидкість (SPA + асинхронні запити).
- Дозволяє легко масштабувати серверну частину.
- Дає можливість розширювати функціонал (наприклад, додати мобільний додаток з тим самим API).
- Забезпечує безпеку за рахунок JWT та валідації даних.

Ця структура стане основою для подальшої реалізації та вдосконалення додатку.

2.3. Проектування бази даних

Для забезпечення надійного та структурованого зберігання даних у нашому вебдодатку було обрано реляційну СУБД PostgreSQL у поєднанні з ORM Sequelize. Таке рішення дозволяє:

1. Забезпечити цілісність даних через чітку схему таблиць та зв’язків.
2. Спростити роботу з БД за рахунок абстракції Sequelize.
3. Легко масштабувати структуру за допомогою міграцій.

Основні сутності та їх взаємозв’язки:

1. Користувачі (Users) відповідає за аутентифікацію та авторизацію. Поля:
   - `id` (PK) - унікальний ідентифікатор;
   - `username` - унікальне ім’я користувача;
   - `password` - захешований пароль (bcrypt);
   - `createdAt/updatedAt` - автоматичні мітки часу.
2. Проекти (Projects) - основна одиниця організації роботи. Поля:
   - `id` (PK);
   - `name` - назва проекту;
   - `background` - параметри оформлення;
   - `ownerId` (FK → Users) - власник проекту.
3. Колонки (Columns) відповідають за етапи виконання (Kanban). Поля:
   - `id` (PK);
   - `name` - назва статусу;
   - `position` - порядок відображення;
   - `projectId` (FK → Projects) - приналежність до проекту.
4. Завдання (Tasks) – основні робочі одиниці. Поля:
   - `id` (PK);
   - `title` - короткий опис;
   - `description` - деталі завдання;
   - `position` - порядок у колонці;
   - `labels` - мітки у форматі JSON;
   - `columnId` (FK → Columns) - поточний статус;

- `projectId` (FK → Projects) - для швидкого доступу.

Зв’язки багато-до-багатьох. Для реалізації складних взаємозв’язків використовуються проміжні таблиці:

1. Учасники проектів (ProjectMembers):

- `userId` (FK → Users);
- `projectId` (FK → Projects);
- Унікальний ключ: комбінація userId + projectId.

2. Виконавці завдань (TaskAssignees):

- `userId` (FK → Users);
- `taskId` (FK → Tasks);
- Унікальний ключ: комбінація userId + taskId.

Для забезпечення цілісності даних система використовує:

- Первинні ключі (PK) - гарантують унікальність записів.
- Зовнішні ключі (FK) - підтримують реляційні зв’язки.
- Каскадні операції - автоматичне оновлення/видалення залежних записів.
- Обмеження NOT NULL для обов’язкових полів.

Перевагами обраного підходу є:

1. Ефективність запитів, яка полягає в тому, що індексовані поля прискорюють пошук, а нормалізація зменшує дублювання даних.

2. Гнучкість роботи полягає в можливості складних запитів через JOIN та підтримці транзакцій для критичних операцій.

3. Безпека даних забезпечується хешуванням чутливих даних та валідацією на рівні БД.

Реалізація через Sequelize. ORM забезпечує:

- Простий опис моделей у JavaScript.
- Автоматичну генерацію SQL-запитів.
- Підтримку міграцій для контролю версій схеми.
- Зручні методи для CRUD-операцій.

Приклад моделі Task:

```
const Task = sequelize.define('Task', {
```

```
  title: { type: DataTypes.STRING, allowNull: false },
  description: { type: DataTypes.TEXT },
  position: { type: DataTypes.INTEGER },
  labels: { type: DataTypes.JSONB }
}, {
  timestamps: true
});
Task.belongsTo(Column);
Task.belongsTo(Project);
Task.belongsToMany(User, { through: 'TaskAssignees' });
```

Така структура оптимально підходить для нашого додатку, забезпечуючи баланс між гнучкістю, продуктивністю та простотою підтримки. Використання ORM дозволяє зосередитись на бізнес-логіці, не заглиблюючись у деталі SQL.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБДОДАТКУ

3.1. Програмна реалізація серверної частини вебдодатку

Налаштування середовища Node.js та Express. Серверна частина – це "мозок" вебдодатку, який відповідає за логіку роботи, зберігання даних та комунікацію з клієнтом. У цьому розділі розглядаються перші кроки створення сервера на Node.js: ініціалізація проекту, підключення бібліотек та налаштування основного файлу.

Створення проекту та встановлення бібліотек. Спочатку було ініціалізовано Node.js-проект за допомогою команди npm init, яка створила package.json – файл із налаштуваннями та списком залежностей.

Далі встановлено необхідні пакети для роботи з бібліотеками та даними.

Основні бібліотеки:

- express – фреймворк для побудови API.
- sequelize – ORM для роботи з PostgreSQL.
- pg та pg-hstore – драйвери для підключення до бази даних.
- bcryptjs – хешування паролів.
- jsonwebtoken – генерація JWT для автентифікації.
- passport та passport-jwt – управління доступом.
- cors – обробка міждоменних запитів.
- express-async-handler – спрощення обробки помилок.

Додаткові інструменти для розробки:

- sequelize-cli – генерація міграцій та моделей.

2. Налаштування сервера (index.js)

Цей файл – ядро бекенду. Ось основні етапи його налаштування:

Підключення бібліотек:

Імпортуються express, cors, passport, налаштування бази даних (sequelize) та маршрути (наприклад, для завдань, проектів, авторизації).

Ініціалізація додатку:

```
const app = express();
```

Додавання middleware:

- cors() – дозвіл запитів з інших доменів.
- express.json() – обробка JSON-даних у запитах.
- passport.initialize() – підготовка автентифікації.

Налаштування Passport. Викликається спеціальна функція для роботи з JWT:

```
require('./middleware/passport')(passport);
```

Підключення до бази даних – використовується sequelize.authenticate(). У разі успіху виводиться повідомлення, у разі помилки – її деталі.

Реєстрація маршрутів:

1. Кожен модуль (наприклад, tasks, auth) підключається через app.use() зі своїм префіксом (наприклад, /api/tasks).

2. Запуск сервера:

app.listen(5000, () => console.log('Сервер запущено на порті 5000')).

Після виконання описаних кроків було налаштовано базове серверне середовище. Проект Node.js ініціалізовано, встановлено всі необхідні залежності, а основний файл index.js сконфігуровано для запуску Express-сервера, підключення до бази даних PostgreSQL через Sequelize, використання middleware для обробки запитів та автентифікації, а також реєстрації маршрутів API. Сервер готовий до подальшої розробки бізнес-логіки та реалізації конкретних API ендпоінтів.

Після підготовки серверного середовища та підключення до PostgreSQL, необхідно визначити структуру даних у додатку та реалізувати механізми роботи з БД. У цьому проекті для цих цілей використовується Sequelize – ORM (Object-Relational Mapper), яка дозволяє працювати з реляційною базою даних через JavaScript-об'єкти. Робота з базою даних передбачає наступні етапи:

1. Визначення моделей у Sequelize. ORM Sequelize базуються на моделях – абстракціях, що відповідають таблицям у базі даних. Кожна модель описує:

- назву таблиці;
- стовпці (атрибути) та їх типи;
- обмеження (первинні ключі, унікальність, зовнішні ключі тощо);
- зв’язки з іншими моделями.

У проекті моделі зберігаються у папці models/ і відповідають структурі БД.

1. User – користувачі:

- userId (числовий ідентифікатор, автоінкремент);
- userName (унікальний логін, обов’язковий);
- password (пароль, обов’язковий).

2. Project – проекти:

- projectId (первинний ключ);
- name (назва);
- background (фон).
- Column – колонки завдань:
- columnId (первинний ключ);
- name (назва);
- position (позиція у списку);
- projectId (зовнішній ключ на проект).

3. Task – завдання:

- taskId (первинний ключ);
- taskName (назва);
- description (опис);
- position (порядок у колонці);
- markers (мітки у форматі JSON або тексту);
- columnId (зовнішній ключ на колонку);
- projectId (зовнішній ключ на проект).

Sequelize дозволяє визначати відношення між таблицями за допомогою методів hasMany, belongsTo та belongsToMany. Реалізація зв’язку «один до багатьох» будується на характеристиках:

1. Проект містить кілька колонок:

```
javascript
Project.hasMany(Column, { foreignKey: 'projectId' });
Column.belongsTo(Project, { foreignKey: 'projectId' });
```

2. Колонка містить кілька завдань:

```
javascript
Column.hasMany(Task, { foreignKey: 'columnId' });
Task.belongsTo(Column, { foreignKey: 'columnId' });
```

Реалізація зв'язку «багато до багатьох» будується на характеристиках:

1. Користувачі та проекти (через проміжну таблицю UserProjects):

```
javascript
User.belongsToMany(Project, { through: 'UserProjects', foreignKey: 'userId' });
Project.belongsToMany(User, { through: 'UserProjects', foreignKey: 'projectId' });
```

2. Користувачі та завдання (через UserTasks):

```
javascript
User.belongsToMany(Task, { through: 'UserTasks', foreignKey: 'userId' });
Task.belongsToMany(User, { through: 'UserTasks', foreignKey: 'taskId' });
```

Ці зв'язки автоматично створюють зовнішні ключі в БД та додають спеціальні методи для роботи з пов'язаними даними.

Взаємодія з БД відбувається у Express-роутах (наприклад, routes/tasks.js, routes/projects.js) через методи Sequelize.

Розглянемо приклади основних операцій:

1. Додавання запису (create) – створення нового завдання:

```
javascript
   const newTask = await Task.create({
    taskName: req.body.taskName,
    columnId: req.body.columnId,
    projectId: req.params.projectId,
    position: req.body.position
```

```
});
```

2. Читання (findAll, findOne) – отримання колонок проекту разом із завданнями:

```
javascript
   const columns = await Column.findAll({
    where: { projectId: req.params.projectId },
    include: [{ model: Task, include: [User] }], // Завантаження завдань та учасників
    order: [['position', 'ASC']]
});
```

3. Оновлення (update) – зміна назви завдання:

```
javascript
   await Task.update(
    { taskName: req.body.taskName },
    { where: { taskId: req.params.taskId } }
);
```

4. Видалення (destroy) – видалення завдання:

```
javascript
   await Task.destroy({
    where: { taskId: req.params.id }
});
```

5. Робота зі зв'язками – додавання користувача до проекту:

```
javascript
await project.addUser(userId);
```

Міграції через Sequelize CLI. Для керування змінами в структурі БД використовується sequelize-cli. Міграції – це файли, які описують зміни (додавання таблиць, зміну стовпців тощо). Після створення їх можна застосувати командою:

```
bash
npx sequelize-cli db:migrate
```

Це забезпечує узгодженість схеми БД між середовищами.

Sequelize значно спрощує роботу з PostgreSQL, дозволяючи працювати з даними через об'єкти JavaScript. Визначення моделей, CRUD-операції та

автоматичні зв’язки скорочують обсяг рутинного SQL-коду, а міграції допомагають ефективно керувати змінами в базі даних.

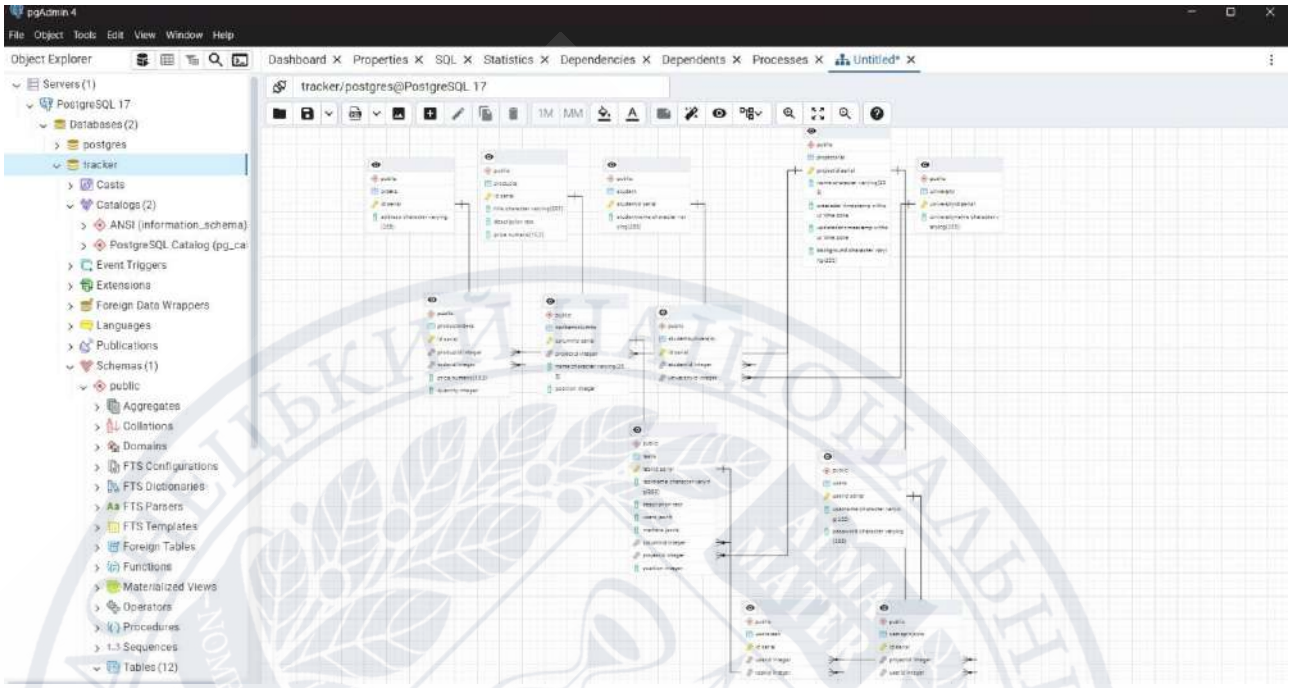

Рисунок 3.1 – Реалізація моделей даних PostgreSQL

3.2 Клієнтська частина вебдодатку та її налаштування

Фронтенд нашого вебдодатку відповідає за інтерфейс користувача, обробку взаємодій та комунікацію з бекендом через API. Основа проєкту – бібліотека React, яка дозволяє створювати динамічні та інтерактивні інтерфейси.

1. Початкова настройка проекту. Для швидкого старту ми використали Create React App - інструмент, який автоматично налаштовує:

- Базову структуру проекту.
- Локальний сервер для розробки.
- Систему автоматичного оновлення коду (Hot Module Replacement).
- Конфігурацію для збірки проекту.

Основні бібліотеки проекту:

- React та ReactDOM - ядро для роботи з компонентами.
- React Router - для клієнтської маршрутизації.
- Redux та Redux Thunk - керування станом додатку.

- Axios - робота з API.
- React Beautiful DnD - реалізація drag-and-drop функціоналу.
- Font Awesome - іконки для інтерфейсу.

2. Вхідна точка додатку. Файл `src/index.js` виконує кілька ключових функцій:

1. Підключає необхідні бібліотеки.
2. Імпортує кореневий компонент App.
3. Завантажує глобальні стилі.
4. Відображає додаток в DOM-елементі з id="root".

Для виявлення потенційних проблем використовується `<React.StrictMode>`.

3. Кореневий компонент App. Основний компонент `App.js` містить:

- Логіку перевірки автентифікації (зчитує дані з localStorage).
- Систему маршрутизації з різними макетами сторінок.
- Глобальні елементи (модальні вікна).
- Підключення до Redux store.

Маршрутизація реалізована через:

- `/` - Головна сторінка.
- `/profile` - Профіль користувача.
- `/projects` - Список проектів.
- `/projects/:projectId` - Конкретний проект.
- `/login`, `/register` - Сторінки авторизації.

4. Управління станом та маршрутизація. Додаток використовує два ключові механізми:

1. Redux - для централізованого керування станом:
   - Store доступний через `<Provider>`
   - Асинхронні дії обробляються через Redux Thunk
2. React Router - для навігації:
   - Обгорнутий в `<BrowserRouter>`
   - Підтримує історію переходів

5. Організація кодової бази. Проект має чітку структуру:

- src/
- components/ # Перевикористовувані UI-компоненти
- containers/ # Компоненти сторінок
- redux/ # Логіка Redux
- actions/
- reducers/
- store.js
- api/ # Модулі для роботи з API
- assets/ # Статичні ресурси
- layouts/ # Макети сторінок
- utils/ # Допоміжні функції
- hooks/ # Кастомні хуки

Наш фронтенд побудований на сучасному стеку технологій:

- React для інтерфейсу.
- Redux для керування станом.
- React Router для навігації.
- Axios для комунікації з API.

Такий підхід дозволяє: швидко розвивати додаток, легко підтримувати код, забезпечувати плавну роботу інтерфейсу та ефективно керувати станом програми. Структура проєкту спрощує розробку нових функцій та співпрацю між членами команди. Використання перевірених бібліотек гарантує стабільність і безпеку додатку.

Користувацький інтерфейс додатка розроблений з використанням бібліотеки React, що дозволяє створювати модульні та перевикористовувані компоненти. Кожен компонент об’єднує візуальну частину (JSX-розмітку та стилі) з логікою поведінки, забезпечуючи гнучкість і легкість підтримки коду.

1. Архітектура компонентів. Проект включає функціональні (наприклад, Home, Header, Login, Register, Navbar) та класові компоненти (наприклад,

Column, Task). Всі вони структуровані у відповідних папках для зручності навігації. Для роботи з Redux застосовується функція connect з бібліотеки react-redux, що дозволяє компонентам отримувати доступ до глобального стану.

2. Основні компоненти та їх функціонал.

1. Header.jsx (Шапка сайту). Це функціональний компонент, який адаптується до стану користувача, якщо користувач автентифікований (props.token), у шапці відображаються:

– Іконки навігації (faHome для переходу до проєктів, faUserCircle для профілю).

– Ініціал користувача (перша літера імені).

– Випадаюче меню з кнопкою виходу (закривається при кліку поза ним завдяки кастомному хуку useOutsideAlerter).

– Для неавторизованих користувачів показуються посилання на сторінки входу та реєстрації.

– Завжди відображається логотип та назва додатка – "Kanban board".

2. Login.jsx / Register.jsx (Форми авторизації) побудовані на основі redux-form для керування станом полів. Використовують кастомні поля вводу з валідацією (наприклад, обов’язкові поля перевіряються через utils/validator). Після відправки форми запускаються відповідні Redux-дії (login або register). Якщо користувач вже увійшов у систему (props.userData.token), відбувається автоматичний перехід на сторінку профілю. Додатково містяться посилання для перемикання між формами та елементи оформлення (зображення, заголовки).

3. Columns.jsx (Колонки завдань). Класовий компонент із підтримкою перетягування (react-beautiful-dnd). Має внутрішній стан для:

– Редагування назви колонки (активується кліком, зберігається при втраті фокусу або натисканні Enter).

– Додавання нових завдань (форма з TextareaAutosize, закривається при кліку поза нею).

– Відкриття контекстного меню для колонки.

– Взаємодіє з Redux для оновлення даних (зміна назви, додавання завдань, видалення колонки).

4. Tasks.jsx (Картки завдань). Класовий компонент, який підтримує перетягування. Локальний стан відповідає за: відображення іконки редагування при наведенні та швидке редагування назви (через TextareaAutosize).

Відображає:

– Кольорові маркери.

– Назву завдання.

– Іконки статусу (наприклад, faEye для підписаних користувачів).

– Ініціали учасників.

– При кліку відкриває детальну інформацію про завдання.

– Інші компоненти.

5. Home.jsx – статична сторінка з основним контентом. Якщо користувач авторизований, перенаправляє на профіль.

6. Navbar.jsx – навігаційна панель, яка змінює стилі залежно від поточного маршруту.

Стилізація реалізована через CSS Modules, що уникнуло конфліктів класів. Іконки підключені за допомогою @fortawesome/react-fontawesome. Взаємодія з користувачем забезпечується стандартними React-подіями (onClick, onChange, onKeyDown тощо), які оновлюють стан або викликають Redux-дії.

Користувацький інтерфейс побудований на компонентній архітектурі React, що дозволяє ефективно керувати станом та логікою. Використання бібліотек (react-router-dom, react-beautiful-dnd, redux-form) спрощує реалізацію ключових функцій, таких як маршрутизація, drag-and-drop та робота з формами. Ізольовані стилі та модульність компонентів забезпечують зручну підтримку та масштабування проекту.

Сучасні односторінкові додатки (SPA), такі як Kanban-дошка, потребують ефективного механізму керування станом. Він включає як дані з сервера (користувачі, проекти, завдання), так і тимчасовий стан інтерфейсу (відкриті

модальні вікна, активні форми, процес перетягування елементів). У цьому проекті для керування станом використовується Redux у поєднанні з React Redux (для інтеграції з компонентами) та Redux Thunk (для асинхронних операцій).

Redux Store – це централізоване сховище даних додатку, яке створюється за допомогою createStore. Для зручності стан розділено на логічні частини, кожна з яких керується окремим редюсером, а потім об'єднується через combineReducers.

Основні редюсери:

– authReducer – відповідає за автентифікацію (зберігає userId, userName, token).

– columnsReducer – управляє станом Kanban-дошки (колонки, завдання, порядок елементів).

– projectsReducer – обробляє список проектів та пов'язані з ними UI-стани.

– usersReducer – керує даними користувачів (списки учасників, призначення на завдання).

– formReducer – використовується для роботи з формами через redux-form.

Для обробки асинхронних дій (наприклад, запитів до API) під час створення Store застосовується middleware redux-thunk.

Редюсери – це чисті функції, які приймають поточний стан і дію (action), а потім повертають новий стан, не змінюючи оригінальний об'єкт. Ключові аспекти authReducer:

– Обробляє дію SET_AUTH_USER_DATA, яка оновлює дані авторизованого користувача.

– Зберігає токен, ім'я та ID користувача.

ColumnsReducer має складну початкову структуру (initialState), що включає:

– tasks – нормалізований об'єкт завдань (ключі – task-ID).

– columns – об'єкт колонок із масивом taskIds.

– columnOrder – порядок відображення колонок.

– taskInfo, isTaskInfo – стан детального перегляду завдання.

Обробляє дії SET_ALL_TASKS, SET_COLUMNS для оновлення даних з API.

Логіка ON_DRAG_END оновлює стан після перетягування елементів (використовується з react-beautiful-dnd).

ProjectsReducer зберігає список проектів (projects) та UI-стани (isOpenInviteList, isOpenInputEditProject).

UsersReducer відповідає за списки користувачів (users, activeUsers, participantsOnTask).

Дії (actions) – це об’єкти, які описують зміни в стані. Вони мають обов’язкове поле type та опціональне payload (дані для оновлення). Розглянемо види Action Creators. Синхронні – повертають об’єкт дії напряму. Наприклад:

– setAuthUserData – оновлює дані авторизації.

– setColumns – змінює стан колонок.

– openModal / closeModal – керує модальними вікнами.

Асинхронні (Thunks) – виконують запити до API та диспатчать інші дії після отримання відповіді. Для взаємодії з бекендом використовуються Thunk-функції, які виконують запит до API та після успішної відповіді диспатчать синхронні дії для оновлення стану.

Автентифікація (authReducer):

login, register – надсилають дані на сервер, оновлюють стан користувача.

Робота з завданнями (columnsReducer):

getColumns – завантажує колонки та завдання з API.

addNewTask, updateTaskName, removeTask – модифікують дані та синхронізують стан.

Проєкти (projectsReducer):

getAllProjects – отримує список проєктів.

createNewProject, editProject – додає або змінює проєкти.

Користувачі (usersReducer):

getAllUsers – завантажує список користувачів.

addNewParticipant, removeFromProject – керує учасниками завдань.

Робота Redux будується на односпрямованому потоці. Користувач взаємодіє з інтерфейсом (натискає кнопку, перетягує завдання). Компонент диспатчить дію (action). Якщо дія асинхронна, redux-thunk виконує запит до API, після чого диспатчить синхронну дію з результатом. Редюсер отримує дію та оновлює відповідну частину стану. Store сповіщає підписані компоненти через react-redux. Інтерфейс автоматично оновлюється з новими даними.

Використання Redux дозволяє централізовано керувати станом додатку, забезпечуючи передбачуваність і легкість розробки. Поділ на редюсери, чіткі дії (actions) та асинхронні Thunk-функції спрощують обробку даних і взаємодію з API. Завдяки React Redux стан автоматично синхронізується з інтерфейсом, що робить додаток реактивним і зручним для користувача.

3.3 Розробка API маршрутів

Після налаштування моделей даних та механізмів роботи з базою даних, наступним етапом є створення API (Application Programming Interface) – інтерфейсу, через який клієнтська частина (Frontend) зможе надсилати запити до сервера. У цьому проекті API реалізовано на Express.js з використанням REST-підходу, що забезпечує стандартизовану роботу з даними.

1. Організація маршрутів. Для зручності підтримки коду маршрути розділені за модульним принципом. Кожен тип даних має окремий файл у папці `routes/`:

- `auth.js` – автентифікація та реєстрація;
- `users.js` – управління користувачами;
- `projects.js` – операції з проектами;
- `columns.js` – робота з колонками завдань;
- `tasks.js` – управління завданнями;
- `usersprojects.js` – зв’язки користувачів і проектів;
- `userstasks.js` – призначення завдань на користувачів.

Підключення маршрутів у `index.js`:

```
javascript
const tasksRouter = require('./routes/tasks');
const projectsRouter = require('./routes/projects');
// ...інші маршрути
app.use('/api/tasks', tasksRouter);
app.use('/api/projects', projectsRouter);
// ...решта шляхів
```

Таке групування дозволяє тримати код структурованим і масштабованим.

2. Принципи RESTful API. API побудовано з урахуванням стандартних практик REST:

- Ресурси: Кожна сутність (користувач, проект, завдання) має унікальний URL.

- HTTP-методи:

  - `GET` – отримання даних;

  - `POST` – створення нового запису;

  - `PUT/PATCH` – оновлення даних;

  - `DELETE` – видалення.

- Формат даних: Обмін відбувається у JSON (використовується `express.json()`).

- Stateless: Кожен запит містить усю необхідну інформацію (автентифікація через JWT-токени у заголовках).

3. Основні ендпоїнти API.

Автентифікація та користувачі:

- `POST /api/login` – вхід в систему;

- `POST /api/users` – реєстрація нового користувача.

Проекти:

- `GET /api/projects?userId={id}` – список проектів користувача;

- `POST /api/projects` – створення проекту;

- `DELETE /api/projects/{projectId}` – видалення проекту.

Колонки:

- `GET /api/columns/{projectId}` – отримання колонок проекту;
- `PUT /api/columns/position` – зміна порядку колонок.

Завдання:

- `POST /api/tasks/{projectId}` – додавання завдання;
- `PUT /api/tasks/position/{projectId}` – оновлення позицій завдань.

Спільна робота:

- `POST /api/users/projects/active` – додати учасника до проекту;
- `DELETE /api/task/user/{taskId}/{userId}` – зняти користувача із завдання.

4. Логіка обробки запитів.

Кожен маршрут включає:

1. Валідацію даних (параметри URL, тіло запиту).
2. Запит до БД через Sequelize (пошук, створення, оновлення).
3. Відправку відповіді у форматі JSON з відповідним HTTP-статусом.

Приклад обробника для створення завдання:

```
javascript
router.post('/:projectId', async (req, res) => {
  try {
    const task = await Task.create({
      taskName: req.body.taskName,
      projectId: req.params.projectId
    });
    res.status(201).json(task);
  } catch (error) {
    res.status(500).json({ error: 'Помилка сервера' });
  }
});
```

5. Захист API.

Для обмеження доступу використовується JWT-автентифікація через Passport.js:

```
javascript
const passport = require('passport');
```

```
// Захищений маршрут
router.get(
  '/',
  passport.authenticate('jwt', { session: false }),
  (req, res) => {
    const userId = req.user.userId;
    // Логіка для автентифікованих користувачів
  }
);
```

- `passport.authenticate('jwt')` перевіряє токен із заголовка `Authorization`.
- Якщо токен валідний, у `req.user` з'являються дані користувача.
- При помилці повертається 401 Unauthorized.

Створений API забезпечує повноцінну взаємодію Frontend із сервером:

- Модульна структура маршрутів спрощує розширення функціоналу.
- REST-підхід робить API інтуїтивно зрозумілим.
- JWT-автентифікація гарантує безпеку даних.

Це дозволяє легко інтегрувати будь-який клієнтський додаток (наприклад, на React або Vue.js) із серверною частиною.

Розглянемо реалізацію процесів автентифікації та авторизації. Захист вебдодатку починається з правильної реалізації системи входу та розмежування прав доступу. У нашому проєкті для цього використано сучасні технології та бібліотеки, що гарантують надійність і безпеку:

1. Реєстрація нових користувачів. Процес створення акаунта включає кілька ключових кроків:

– Користувач вводить логін та пароль через інтерфейс.

– Сервер перевіряє унікальність імені користувача.

– Пароль обробляється за допомогою алгоритму хешування bcryptjs.

– У базу даних записується новий запис з логіном та захешованим паролем.

– Користувач отримує підтвердження успішної реєстрації.

– Паролі ніколи не зберігаються у відкритому вигляді. Натомість зберігається їх хеш - унікальний цифровий відбиток, який неможливо зворотно перетворити на оригінальний пароль.

2. Вхід у систему. Процедура автентифікації відбувається так:

– Користувач надсилає свої облікові дані.

– Система знаходить запис у базі даних за логіном.

– Порівнюється хеш введеного пароля зі збереженим хешем.

– При успішній перевірці генерується JWT-токен.

– Токен повертається клієнту для подальшої роботи.

JWT містить зашифровану інформацію про користувача та має обмежений термін дії. Це дозволяє ідентифікувати користувача без необхідності постійно вводити пароль.

3. Налаштування перевірки токенів. Для роботи з JWT використовується бібліотека Passport.js зі спеціальною стратегією:

– Токен витягується з заголовка Authorization.

– Перевіряється його підпис за допомогою секретного ключа.

– Якщо токен валідний, система отримує дані користувача.

– Ці дані додаються до об’єкту запиту для подальшого використання.

У разі будь-яких невідповідностей (неправильний підпис, прострочений термін дії тощо) доступ до захищених ресурсів блокується.

4. Захист API-ендпоінтів. Для обмеження доступу до чутливих даних використовується спеціальний middleware:

```
javascript
router.get('/secure-data',
  passport.authenticate('jwt', { session: false }),
  (req, res) => {
    // Тут обробляється запит тільки для авторизованих користувачів
    const userData = req.user; // Дані користувача з токена
    // ... логіка обробки ...
  }
```

```
);
```

Цей підхід гарантує, що:

- До захищених даних мають доступ лише авторизовані користувачі.
- Кожен запит супроводжується перевіркою прав доступу.
- Не потрібно зберігати стан сесії на сервері.

Реалізована система безпеки поєднує:

- Надійне зберігання паролів за допомогою хешування.
- Сучасний механізм автентифікації через JWT.
- Гнучкий контроль доступу до API.
- Просту інтеграцію з існуючою кодобазою.

Такий підхід забезпечує захист даних користувачів і ресурсів системи, дотримуючись сучасних стандартів веб-безпеки.

Ефективна комунікація між клієнтською частиною (Frontend) та серверною частиною (Backend) є необхідною умовою функціонування сучасного вебдодатку. Frontend повинен отримувати дані для відображення (списки проектів, завдань, інформацію про користувачів) та надсилати дані на сервер для збереження змін (створення нових завдань, оновлення статусів, редагування проектів). У даному проекті для цієї взаємодії використовується бібліотека Axios, популярний HTTP-клієнт на основі Promise. Розглянемо основні елементи взаємодії:

1. Конфігурація Axios та базовий URL. Для спрощення та централізації API запитів, у проекті створено спеціальний модуль (src/api/api.js), де налаштовується екземпляр Axios.

```
// Приклад конфігурації з api.js
import axios from "axios";
import {baseUrl} from '../common/config/config'; // Імпорт базового URL
const instance = axios.create({
  baseURL: baseUrl, // Встановлення базового URL для всіх запитів
});
```

Створення окремого екземпляру instance з параметром baseURL дозволяє уникнути повторення повної адреси сервера у кожному запиті, роблячи код чистішим та легшим для конфігурації у різних середовищах (розробка, продакшен).

2. Автоматичне додавання токена автентифікації (Interceptor). Оскільки більшість запитів до API вимагають автентифікації, реалізовано механізм автоматичного додавання JWT-токена до заголовків запитів за допомогою перехоплювача запитів (Request Interceptor) Axios.

```
// Приклад Interceptor з api.js
instance.interceptors.request.use(
config => {
 // Отримання токена з localStorage (збереженого після логіну)
 const token = JSON.parse(localStorage.getItem('user'))?.token
|| '';
 if (token) {
 // Додавання заголовка Authorization
  config.headers.Authorization = `Bearer ${token}`;
  } else {
   // Видалення заголовка, якщо токена немає
 delete config.headers.Authorization;
 }
 return config;
 },
 error => Promise.reject(error)
);
```

Цей перехоплювач спрацьовує перед кожним запитом, зробленим через instance. Він перевіряє наявність токена у localStorage, і якщо токен знайдено, додає заголовок Authorization зі значенням Bearer <token>. Це звільняє розробника від необхідності вручну додавати токен до кожного захищеного запиту.

3. Структурування API запитів. Логіка виконання конкретних API запитів інкапсульована у функції, згруповані за ресурсами в об'єкти (authAPI, usersApi,

projectsApi, columnsApi, tasksAPI). Кожна функція відповідає за певний ендпоінт і використовує відповідний метод Axios (instance.get, instance.post, instance.put, instance.delete) для надсилання запиту:

```
// Приклади функцій з api.js
export const projectsApi = {
  getAllProjects(userId) {
    return instance.get(`projects/`, { params: { userId } });
  },
  createNewProject(projectName, userId, background) {
    return instance.post(`projects/`, { name: projectName, userId: userId, background });
  },
  // ... інші методи ...
};

export const tasksAPI = {
  addNewTask(taskName, columnId, projectId, position) {
    return instance.post(`tasks/${projectId}`, { taskName, columnId, position });
  },
  updateTaskName(taskname, projectid, taskid) {
    return instance.put(`/tasks/${projectid}/${taskid}`, { taskname });
  },
  // ... інші методи ...
};
```

Такий підхід робить код, що викликає API, більш читабельним та абстрагує його від деталей реалізації HTTP-запитів.

4. Використання у Redux Thunks. Основне використання цих API-функцій відбувається всередині асинхронних action creators (thunks) у файлах редюсерів. Thunk викликає потрібну функцію з модуля api.js, очікує на відповідь (проміс, що повертається Axios) і потім диспатчить інші синхронні actions для оновлення стану в Redux store.

```
// Приклад використання в thunk
export const getAllProjects = (userId) => async (dispatch) =>
{
  try {
    let response = await projectsApi.getAllProjects(userId);
    if (response.statusText === 'OK') {
      dispatch(setProjects(response.data));
    } // ... обробка помилок ...
  } catch (err) { /* ... */ }
};

export const addNewTask = (taskName, columnId, projectId,
position) => async (dispatch) => {
  try {
    let response = await tasksAPI.addNewTask(taskName,
columnId, projectId, position);
    // Часто після успішної зміни даних - повторний запит для
оновлення стану
    dispatch(getColumns(projectId));
  } catch (e) { /* ... */ }
};
```

5. Обробка помилок. Запити до API можуть завершуватися невдало (помилки мережі, помилки сервера 4xx/5xx). Код у thunks використовує блоки try...catch для перехоплення цих помилок. У поточній реалізації помилки переважно виводяться в консоль (console.log(err)), але в більш розвиненому додатку тут могла б бути логіка для відображення повідомлень про помилки користувачеві через оновлення стану Redux.

Взаємодія Frontend-додатку з Backend API реалізована за допомогою бібліотеки Axios. Створення конфігурованого екземпляру Axios з baseURL та автоматичним додаванням JWT-токена через interceptor спрощує та убезпечує процес комунікації. Інкапсуляція логіки API-запитів у спеціалізованому модулі (api.js) та його використання в Redux thunks забезпечує чіткий поділ

відповідальності та робить процес отримання даних та відправки змін на сервер структурованим та підтримуваним.

3.4. Реалізація функціоналу управління проєктами та завданнями

Основне призначення розробленого вебдодатку – надання користувачам можливості ефективно управляти своїми проектами та завданнями в рамках цих проектів, використовуючи візуальний Kanban-інтерфейс. Цей функціонал реалізовано через взаємодію UI компонентів, системи управління станом Redux та Backend API. Далі описано ключові аспекти реалізації цього функціоналу.

1. Управління проектами:

- Перегляд списку проектів. Компонент ProjectsContainer відповідає за відображення списку проектів. При його завантаженні диспатчиться thunk getAllProjects, який викликає projectsApi.getAllProjects для отримання списку проектів поточного користувача з сервера. Отримані дані зберігаються у projectsReducer і передаються компоненту для рендерингу.
- Створення проекту. Користувач ініціює створення через відповідний UI елемент (наприклад, кнопку). Це призводить до диспатчу thunk createNewProject, який приймає назву проекту та інші параметри. Thunk викликає projectsApi.createNewProject, а після успішного створення на сервері, диспатчить getAllProjects для оновлення списку проектів у стані Redux та UI.
- Редагування/Видалення проекту. Аналогічно до створення, взаємодія користувача з UI (наприклад, кнопка редагування або видалення) запускає відповідні thunks (editProject або removeProject). Ці thunks викликають методи projectsApi (editProject, removeProject), а потім оновлюють стан, зазвичай перезавантажуючи список проектів (getAllProjects). При видаленні проекту також може викликатися getAllUsers для оновлення списку активних користувачів.
- Управління учасниками проекту. Додавання або видалення користувачів з проекту реалізується через thunks addToProject та removeFromProject, які викликають відповідні методи usersApi і потім оновлюють списки користувачів

через getAllUsers та getParticipantsOnTask.

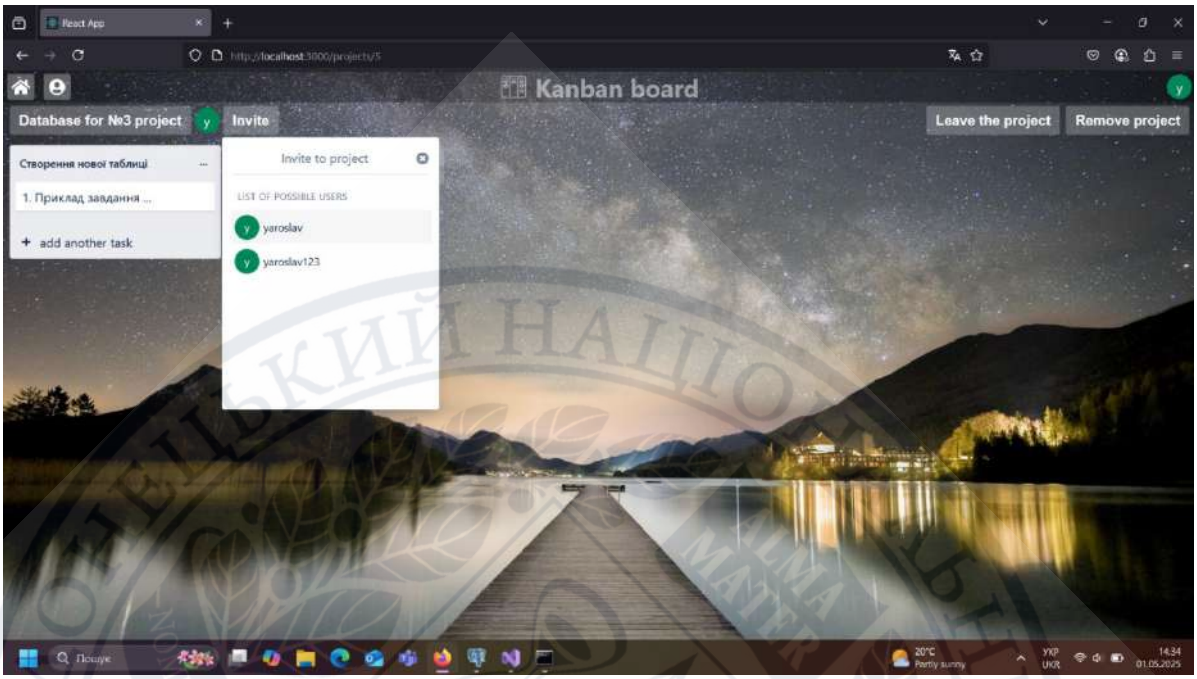

Рисунок 3.2 – Вигляд сторінки управління проєктами

2. Управління колонками (Статусами):

- Відображення колонок. При відкритті конкретного проекту (компонент Project), диспатчиться thunk getColumns. Він отримує з сервера дані про колонки та завдання цього проекту через columnsApi.getColumns. Дані обробляються та зберігаються у columnsReducer у нормалізованому вигляді (columns, tasks, columnOrder). Компонент Project використовує ці дані для рендерингу компонентів Columns у порядку, визначеному columnOrder.
- Створення колонки. Дія користувача (клік на кнопку "Додати колонку") призводить до диспатчу thunk createNewColumn, який викликає columnsApi.createNewColumn і потім оновлює стан дошки через getColumns.
- Редагування/Видалення колонки. Компонент Columns містить логіку для редагування назви (зміна локального стану isInput, виклик thunk onUpdateColumn при збереженні) та видалення (через меню, виклик thunk onRemoveColumn). Ці thunks взаємодіють з columnsApi та оновлюють стан через getColumns.
- Зміна порядку колонок. Реалізується за допомогою react-beautiful-dnd.

Коли користувач перетягує колонку, спрацьовує onDragEnd, диспатчиться дія onDragEnd з type: ‘column’. Редюсер columnsReducer синхронно оновлює масив columnOrder у стані Redux. Після цього (або паралельно) диспатчиться thunk onUpdateColumnsPosition, який надсилає новий порядок колонок на сервер через columnsApi.updateColumnsPosition і може викликати getColumns для фінальної синхронізації стану.

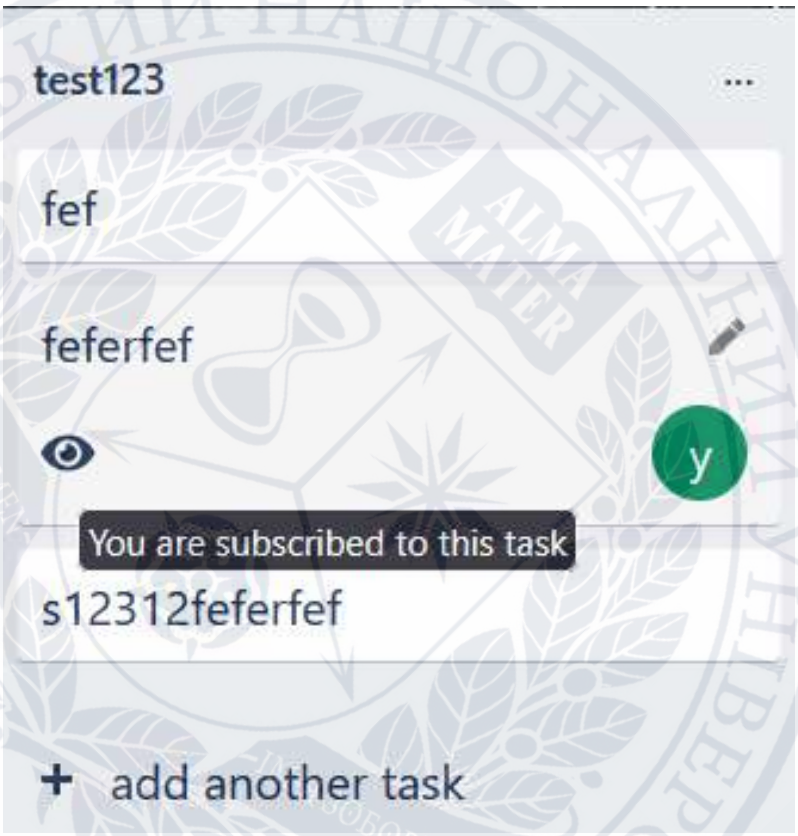

Рисунок 3.3 – Вигляд вікна редагування колонок

3. Управління завданнями:

- Відображення завдань. Компонент Columns ітерує по масиву taskIds (з columnsReducer state) та рендерить компоненти Tasks для кожного завдання, передаючи йому дані завдання з об’єкту tasks (також з columnsReducer state).
- Створення завдання. Користувач взаємодіє з формою додавання завдання у компоненті Columns. При збереженні диспатчиться thunk addNewTask, який викликає tasksAPI.addNewTask і потім оновлює всю дошку через getColumns.
- Редагування завдання (Назва, Опис, Маркери). Реалізовано через швидке

редагування назви у компоненті Tasks (викликає thunk updateTaskName) та через компонент TaskInfo для інших полів (викликає thunks updateDescription, addNewMarker). Thunks взаємодіють з tasksAPI та оновлюють стан через getColumns.

- Призначення/Зняття учасників. Через UI диспатчаться thunks addNewParticipant або removeParticipant. Вони викликають tasksAPI і потім оновлюють списки учасників та стан дошки через getColumns, getParticipantOnTask, getParticipantsOnTask.

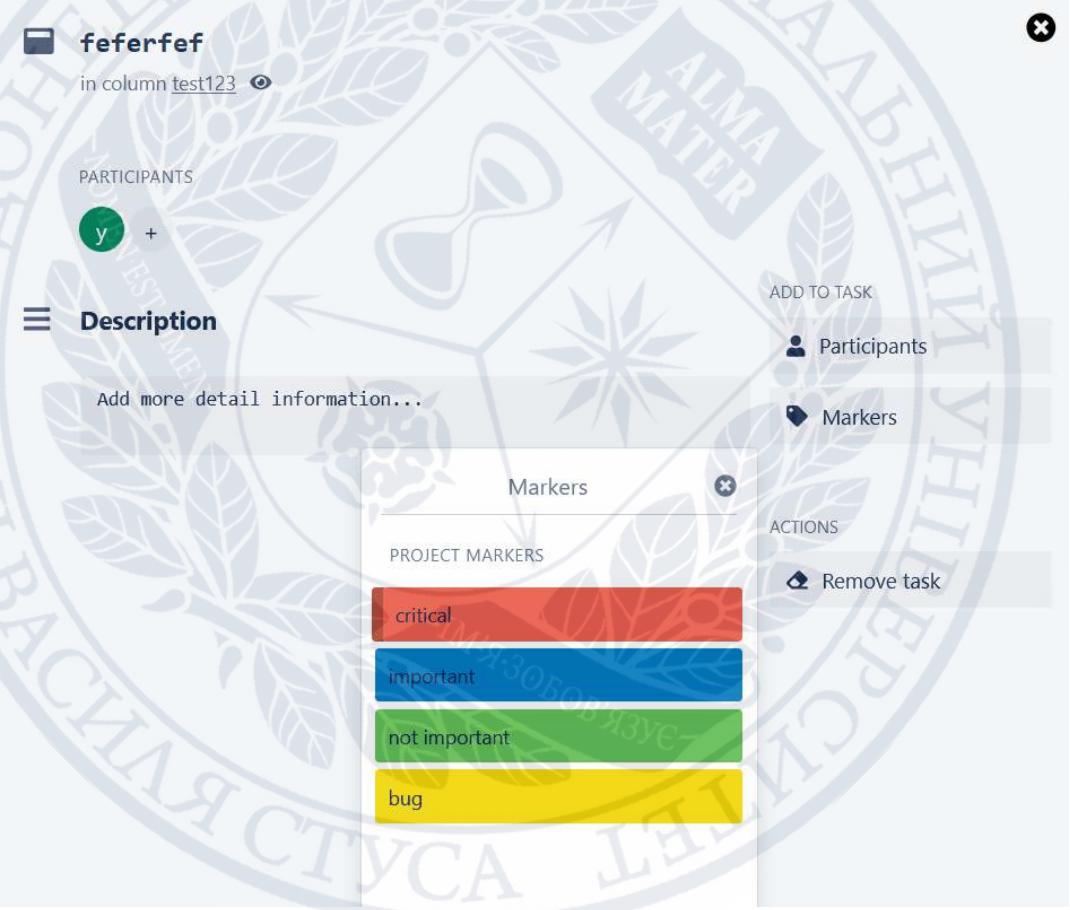

Рисунок 3.4 – Вигляд вікна редагування завдання

- Переміщення/Зміна порядку завдань. Реалізується через react-beautiful-dnd. Перетягування картки завдання (Tasks) ініціює onDragEnd, диспатчиться дія onDragEnd з type: ‘task’. Редюсер columnsReducer синхронно оновлює масиви taskIds у відповідних колонках стану Redux. Потім диспатчиться thunk

updateTasksPosAndColumnId, який надсилає на сервер інформацію про нові позиції та/або колонки завдань через tasksAPI.updateTasksPosAndColumnId. Для фінальної синхронізації викликається getColumns.

- Видалення завдання. Дія користувача призводить до диспатчу thunk removeTask, який викликає tasksAPI.removeTask і оновлює стан через getColumns.

Функціонал управління проєктами та завданнями реалізовано через тісну інтеграцію React компонентів, Redux та Backend API. Компоненти відповідають за відображення даних та ініціацію дій користувача. Redux управляє станом додатку, обробляє дії (включаючи складну логіку drag-and-drop) та оркеструє асинхронні операції за допомогою thunks. Thunks інкапсулюють логіку взаємодії з API, викликаючи відповідні функції з модуля api.js для надсилання запитів на сервер та оновлення стану додатку після отримання відповіді. Такий підхід забезпечує реалізацію всіх основних CRUD-операцій та функціоналу Kanban-дошки.

ВИСНОВКИ

У рамках даної роботи було зпроєктовано та розроблено повнофункціональний вебдодаток для спільної комунікації розробників IT-проєкту по управлінню завданнями, який реалізує візуальний підхід на основі Kanban-методології та надає користувачам інструменти для організації робочих процесів та командної взаємодії, з використанням сучасного стеку веб-технологій. Основним результатом став готовий прототип системи, який дозволяє якісно організовувати робочий процес за допомогою інтерактивної дошки з завданнями. За результатами проведеного дослідження можна зробити наступні висновки:

1. В результаті аналізу предметної області управління IT-проєктами було систематизовано основні методологічні підходи до їх управління. Виявлено, що такий підхід як Скрам в управлінні ко мандами розробників показує найкращі результати з позиції моніторингу та адаптації груп розробників до поставлених завдань.

2. Під час розробки було досліджено ринок рішень IT-продуктів, які дозволяють реалізувати системи по управлінню IT-проєктами. Ними є такі програмні продукти як: Trello, Jira Asana Monday.com, ClickUp, Microsoft Project, Microsoft Planner. Усі вони мають базовий набір функцій, як-от: створення завдань, призначення відповідальних, встановлення дедлайнів, коментування та прикріплення файлів.

3. Обґрунтовано вибір інструментів та технологій для реалізації вебдодатку, а також сформовано вимоги до функціоналу та обрано стек технологій. Для серверної частини додатку: Node.js + Express, PostgreSQL (ORM Sequelize). Для клієнтської частини: React + Redux, Axios для HTTP-запитів. Додаткові інструменти: JWT-аутентифікація, drag-and-drop (react-beautiful-dnd).

4. Була розроблена архітектура клієнт-серверної взаємодії на основі REST API та спроектована структура бази даних.

5. Особливостями програмної реалізації серверної частини стали

налаштування та використані моделі даних. Налаштовано сервер на Node.js з використанням Express. Реалізовано моделі даних для користувачів, проектів, колонок і завдань (Sequelize + PostgreSQL). Розроблено API для CRUD-операцій та бізнес-логіки (наприклад, призначення завдань учасникам). Забезпечено безпеку: хешування паролів (bcryptjs), автентифікація через JWT та Passport.js.

6. Розробка клієнтської частини (Frontend) реалізувала в собі ряд вимог. Створено SPA на React з модульною структурою компонентів. Для керування станом використано Redux (з Redux Thunk для асинхронних запитів). Drag-and-drop функціонал забезпечено бібліотекою react-beautiful-dnd. Взаємодія з бекендом забезпечена Axios з автоматичним додаванням JWT-токена. Основні реалізовані вимоги це:

– Система авторизації (реєстрація/вхід).

– Інтерфейс проекту з Kanban-дошкою (колонки, завдання, перетягування).

– Детальний перегляд завдань та управління учасниками.

7. Основний функціонал, який забезпечує розроблений додаток є: реєстрація та вхід у систему; створення проектів із Kanban-дошками; додавання колонок (статусів) та завдань; візуальне керування завданнями через drag-and-drop; призначення учасників та контроль прогресу. Виявлені перспективи розвитку дослідження показують, що проєкт може бути допрацьований за такими напрямками як: додавання коментарів, звітів, сповіщень; інтеграція зі зовнішніми сервісами (наприклад, Google Calendar); оптимізація продуктивності та розгортання на продакшен-сервері.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

1. Trello. Офіційний сайт. URL: https://trello.com/
2. Jira Software. Офіційний сайт компанії Atlassian. URL: https://www.atlassian.com/software/jira
3. Asana. Офіційний сайт. URL: https://asana.com/
4. Monday.com. Офіційний сайт. URL: https://monday.com/
5. ClickUp. Офіційний сайт. URL: https://clickup.com/
6. Node.js Documentation. URL: https://nodejs.org/en/docs/
7. Express.js Documentation. URL: https://expressjs.com/
8. PostgreSQL Official Documentation. URL: https://www.postgresql.org/docs/
9. Sequelize ORM Documentation.URL: https://sequelize.org/
10. Passport.js Documentation. URL: http://www.passportjs.org/docs/
11. React Documentation. URL: https://react.dev/
12. Redux Documentation. URL: https://redux.js.org/
13. Axios Documentation. URL: https://axios-http.com/
14. React Beautiful DnD Documentation. URL: https://github.com/atlassian/react-beautiful-dnd
15. bcryptjs Documentation. URL: https://github.com/dcodeIO/bcrypt.js
16. JWT (JSON Web Token) Introduction. URL: https://jwt.io/introduction
17. Мартін Р. Чистий код. Створення, аналіз та рефакторинг. Фабула, 2019. 416 с.
18. Круг С. Не змушуйте мене думати. Вебюзабіліті: практичний підхід. ArtHuss, 2024. 198 с.
19. Сучасний підручник з JavaScript. URL: https://uk.javascript.info/
20. Яблонські Д. Закони UX дизайну. Психологія в дизайні продуктів. O'Reilly, 2022. 160 с.
21. Ґотельф Д., Сейден Д. Lean UX: Створення класних продуктів з Agile-командами. ArtHuss, 2024. 206 с.
22. Фрімен Е., Робсон Е. та ін. Head First. Патерни проєктування. Фабула, 2020. 672 с.

23. EPAM Campus. (2023). Scrum vs Agile vs Kanban: що обрати? URL: https://campus.epam.ua/ua/blog/577

24. Hahn E. Express in Action: Writing, building, and testing Node.js applications. Manning Publications, 2016. 256 p.

25. Brown E. Web Development with Node and Express: Leveraging the JavaScript Stack 2nd Edition. O'Reilly Media, 2019. 343 p.

26. Mardan, A. Pro Express.js: Master Express.js: The Node.js Framework For Your Web Development. Apress, 2014. 372 p.

27. Casciaro M., Mammino L. Node.js Design Patterns. Packt Publishing, 2020. 712 p.

28. Lim G. Beginning Node.js, Express & MongoDB Development. Greg Lim, 2019. 154 p.

29. Garreau M. Redux in Action. Manning Publications, 2018. 312 p.

30. Mukhiya S. K., Wei T., Lee, J. Redux Quick Start Guide: A Beginner's Guide to Managing App State with Redux. Packt Publishing, 2019. 204 p.

31. Anderson D. J. Kanban: Successful Evolutionary Change for Your Technology Business. Blue Hole Press, 2010. 278 p.

32. Brechner E. Agile Project Management with Kanban (Developer Best Practices). Microsoft Press, 2015. 160 p.

33. Benson J., DeMaria Barry, T. Personal Kanban: Mapping Work | Navigating Life. Modus Cooperandi Press, 2011. 216 p.

34. Hammarberg M., Sunden J. Kanban in Action. Manning Publications, 2014. 360 p.

35. Anderson D. J., Carmichael A. Essential Kanban Condensed. Lean Kanban University Press, 2016. 102 p.

36. Osmani A. Learning JavaScript Design Patterns. O'Reilly Media, 2012. 251 p.

37. Sequelize Docs. PostgreSQL. URL: https://sequelize.org/docs/v7/databases/postgres/

38. MDN Web Docs. JavaScript. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript

39. ECMA International. ECMAScript Language Specification. URL: https://262.ecma-international.org/

ДОДАТКИ

ДОДАТОК А

Основна частина програмного коду

Backend:

Index.js:

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const passport = require("passport");
const keys = require("./config/keys");
const app = express();
const cors = require("cors");
const pool = require("./db");

const tasks = require('./routes/tasks');
const columns = require('./routes/columns');
const projects = require('./routes/projects');
const auth = require('./routes/auth');
const users = require('./routes/users');
const usersprojects = require('./routes/usersprojects');
const userstasks = require('./routes/userstasks');

const sequelize = require('./config/database')

//Test DB

sequelize.authenticate().then(() => {
    console.log("Database connected...")
}).catch((err) => {
    console.log("Error:" + err);
})

app.use(cors());
// app.use(auth);
app.use(express.json());
app.use(tasks);
app.use(columns);
app.use(projects);
app.use(auth);
app.use(users);
app.use(usersprojects);
app.use(userstasks);


app.use(passport.initialize());
require('./middleware/passport')(passport);

app.listen(5000, () => {
```

```
    console.log("Server has started on 5000 port");
     });
```

Db.js:

```
const Pool = require("pg").Pool;

const pool = new Pool({
    user: "postgres",
    password: "4110",
    host: "localhost",
    port: 5432,
    database: "tracker"
})

module.exports = pool;
```

passport.js:

```
const JwtStrategy = require('passport-jwt').Strategy;
const ExtractJwt = require('passport-jwt').ExtractJwt;
const keys = require('../config/keys');
const pool = require("../db");

const options = {
    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
    secretOrKey: keys.jwt
}

const Users = require('../models/Users');

module.exports = passport => {
    passport.use(
        new JwtStrategy(options, async (payload, done) => {
            try {

                // const user = await pool.query("SELECT * FROM
registration WHERE userId=$1", [payload.userId]);
                const user = await
Users.findUserByUserId(payload.userId);
                if(user.length !== 0) {
                    done(null, user)
                }
                else {
                    done(null, false);
                }
            }
            catch(err) {
```

```
                console.log(err);
            }

        })
    )
}
```

Frontend:

Index.js:

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want to start measuring performance in your app, pass a
function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more:
https://bit.ly/CRA-vitals
reportWebVitals();
```

App.js:

```
import './App.css';
import React from 'react';
import {BrowserRouter, Switch} from "react-router-dom";
import {connect, Provider} from "react-redux";
import store from "./redux/store";
import {compose} from "redux";
import Profile from "./components/Profile";
import Projects from "./components/ProjectsContainer";
import Project from "./components/Project";
import Register from "./components/Register/Register";
import Login from "./components/Login/Login";
import LoginLayoutRoute from "./layouts/loginLayout";
import DashboardLayoutRoute from "./layouts/DashboardLayout";
import Home from "./components/Home/Home";
import {setAuthUserData} from "./redux/reducers/authReducer";
```

```
import ModalContainer from "./components/Modal/ModalContainer";
import TaskInfo from "./components/Tasks/TaskInfo/TaskInfo";
import HomeLayoutRoute from "./layouts/HomeLayout";


class App extends React.Component {

    componentDidMount() {
        if (JSON.parse(localStorage.getItem('user'))) {
            let user = JSON.parse(localStorage.getItem('user'));
            if (user.timestamp > Date.now() - 3600000) {
                this.props.setAuthUserData(user.userId,
user.userName, user.token)
            } else {
                window.localStorage.removeItem('user');
                this.props.setAuthUserData(null, null, null)
            }
        }
    }

    render() {
        return <div className="App">
            <div className='app-wrapper-content'>
                <ModalContainer/>
                <Switch>
                    <HomeLayoutRoute exact path='/'
component={Home}/>
                    <DashboardLayoutRoute path='/profile'
component={Profile}/>
                    <DashboardLayoutRoute
path='/projects/:projectId/:taskId' component={TaskInfo}/>
                    <DashboardLayoutRoute
path='/projects/:projectId' component={Project}/>
                    <DashboardLayoutRoute path='/projects'
component={Projects}/>
                    <LoginLayoutRoute path='/register'
component={Register}/>
                    <LoginLayoutRoute path='/login'
component={Login}/>
                </Switch>
            </div>
        </div>
    }

}

const mapStateToProps = (state) => ({});

let AppContainer = compose(
    connect(mapStateToProps, {setAuthUserData}))
(App);
```

```
const mainApp = () => {
    console.log()
    return (
        <BrowserRouter>
            <Provider store={store}>
                <AppContainer/>
            </Provider>
        </BrowserRouter>
    );
}

export default mainApp;
```

store.js:

```
import {applyMiddleware, combineReducers, createStore} from
"redux";
import thunkMiddleware from 'redux-thunk';
import authReducer from "./reducers/authReducer";
import { reducer as formReducer } from 'redux-form';
import projectsReducer from "./reducers/projectsReducer";
import columnsReducer from "./reducers/columnsReducer";
import usersReducer from "./reducers/usersReducer";


let reducers = combineReducers({
    auth: authReducer,
    projectsPage: projectsReducer,
    columnsPage: columnsReducer,
    usersPage: usersReducer,
    form: formReducer,
});

let store = createStore(reducers,
applyMiddleware(thunkMiddleware));

export default store;
```

api.js:

```
import axios from "axios";
import {baseUrl} from '../common/config/config';

const instance = axios.create({
    baseURL: baseUrl,
})

// const headers = {"Content-Type": "multipart/form-data"}
```

```
instance.interceptors.request.use(
    config => {
        const token = JSON.parse(localStorage.getItem('user')) ?
JSON.parse(localStorage.getItem('user')).token : '';

        if (token) {
            config.headers.Authorization = token;
        } else {
            delete instance.defaults.headers.common.Authorization;
        }
        return config;
    },

    error => Promise.reject(error)
);

export const usersApi = {
    getAllUsers() {
        return instance.get(`users/`);
    },
    addToProject(userid, projectid) {
        console.log("userid, projectid", userid, projectid)
        return instance.post(`users/projects/active`, {
            userid, projectid
        });
    },
    removeFromProject(userid, projectid) {
        return instance.delete(`users/projects/active`, {
            params: {
                userid, projectid
            }
        });
    },
    getActiveUsers(projectId) {
        return instance.get(`users/active`, {
            params: {
                projectId
            }
        });
    }
}

export const projectsApi = {
    createNewProject(projectName, userId, background) {
        return instance.post(`projects/`, {
            name: projectName,
            userId: userId,
            background
        })
    },
    editProject(id, name, userId) {
```

```
        return instance.put(`projects/${id}`, {
            name, userId
        })
    },
    removeProject(projectId, userId) {
        return instance.delete(`projects/${projectId}`)
    },
    getAllProjects(userId) {
        return instance.get(`projects/`, {
            params: {
                userId
            }
        });
    }
};

export const columnsApi = {
    removeColumn(id) {
        return instance.delete(`columns/${id}`)
    },

    updateColumn(id, name) {
        return instance.put(`columns/${id}`, {
            name
        })
    },

    createNewColumn(name, projectListId, position) {
        return instance.post(`columns/${projectListId}`, {
            name, position
        })
    },

    getColumns(projectId) {
        return instance.get(`columns/${projectId}`);
    },

    updateColumnsPosition(newColumns) {
        console.log("firstId, lastId, firstPosition,
lastPosition", newColumns)
        return instance.put('columns/position', {
            newColumns
        })
    }
};

export const tasksAPI = {
    updateTaskName(taskname, projectid, taskid) {
        return instance.put(`/tasks/${projectid}/${taskid}`, {
            taskname
        })
    },
```

```
    getParticipantOnTask(projectid, taskid) {
        return instance.get(`/task/user/${projectid}/${taskid}`)
    },

    updateTasksPosAndColumnId(tasksArr, projectId) {
        return instance.put(`/tasks/position/${projectId}`, {
            tasksArr
        })
    },
    updateTaskDescription(description, projectId, taskId) {
        return instance.put(`/tasks/${projectId}/${taskId}`, {
            description
        })
    },

    getTasksUsers(projectId, userId) {
      return instance.get(`tasks/${projectId}/${userId}`)
    },

    getAllTasks(projectId) {
        return instance.get(`tasks/` + projectId);
    },

    addNewTask(taskName, columnId, projectId, position) {
        return instance.post(`tasks/${projectId}`, {
            taskName, columnId, position
        });
    },

    addNewParticipant(taskId, userId) {
        console.log("userId, taskId", userId, taskId);
        return instance.post(`/task/user`, {
            userId, taskId
        })
    },

    removeParticipant(taskId, userId) {
        console.log("userId, projectId, taskId", userId, taskId);
        return instance.delete(`/task/user/${taskId}/${userId}`)
    },

    addNewMarker(markers, projectId, taskId) {
        return instance.put(`/tasks/${projectId}/${taskId}`, {
            markers
        })
    },

    removeTask(id, projectId) {
        return instance.delete(`/tasks/${projectId}/${id}`)
    }
}
```

```
export const authAPI = {
    login(password, userName) {
        return axios.post(`${baseUrl}login`, {
            password,
            userName
        })
    },

    register(password, userName) {
        return axios.post(`${baseUrl}users`, {
            password,
            userName
        })
    },
};
```

Приклад компоненту Home.jsx:

```
import React from 'react';
import classes from './Home.module.css';
import hero from '../../assets/image/hero.png'
import {Redirect} from "react-router-dom";
import {connect} from "react-redux";

const Home = (props) => {
    // if()
    if (props.userData.token) {
        return <Redirect to={"/profile"}/>
    }
    return <div className="Home">
        <div className={classes.container}>
            <div className={classes.itemLeft}>
                <p className={classes.title}><span
className={classes.firstTextItem}>Kanban</span> board helps teams
get
                    the job done.</p>
                <p className={classes.text}>Collaborate, manage
projects and increase your productivity. Wherever you
                    work - in a large building or in a home office
- with Kanban board your team will succeed.</p>
            </div>
            <div className={classes.itemRight}>
                <div className={classes.wrapImg}>
                    <img className={classes.img} src={hero}
alt=""/>
                </div>
            </div>
        </div>
    </div>
}
```

```
const mapStateToProps = (state) => ({
    userData: state.auth
})

export default connect(mapStateToProps, {})(Home);
```

ДОДАТОК Б

Приклади інтерфейсу користувача

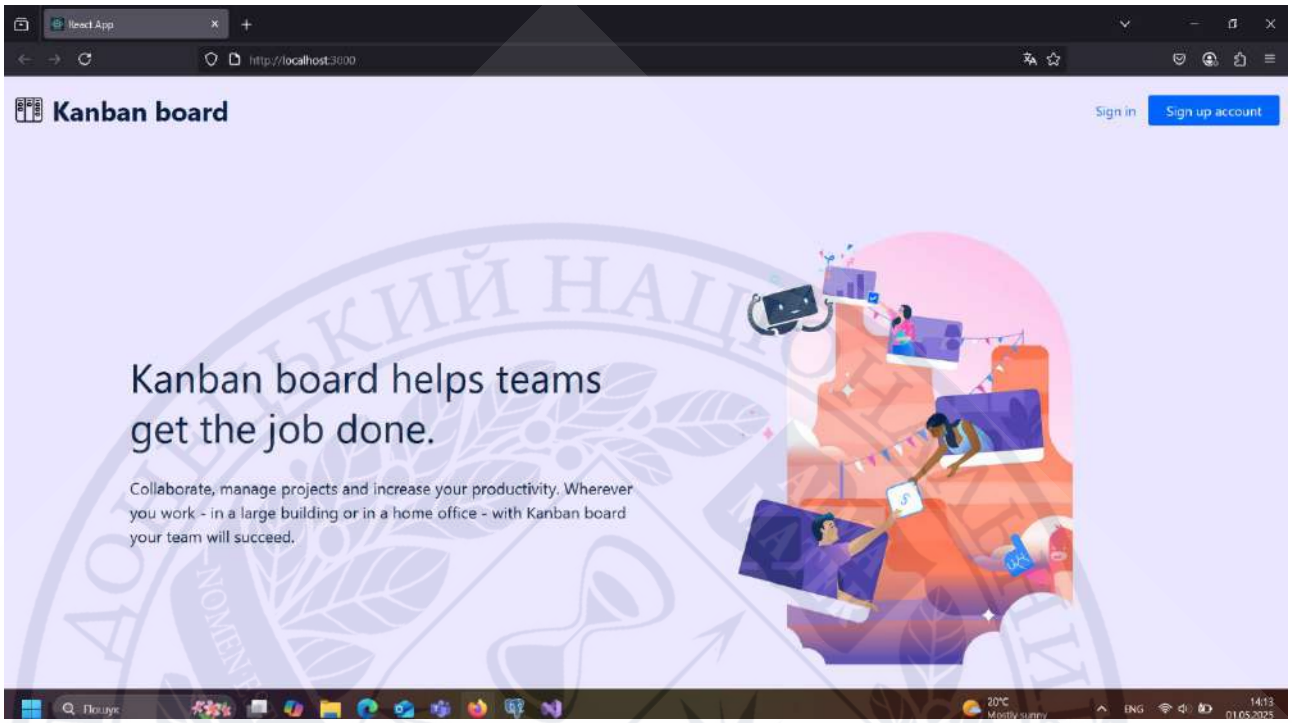

Рисунок Б. 1 – Головна сторінка сайту

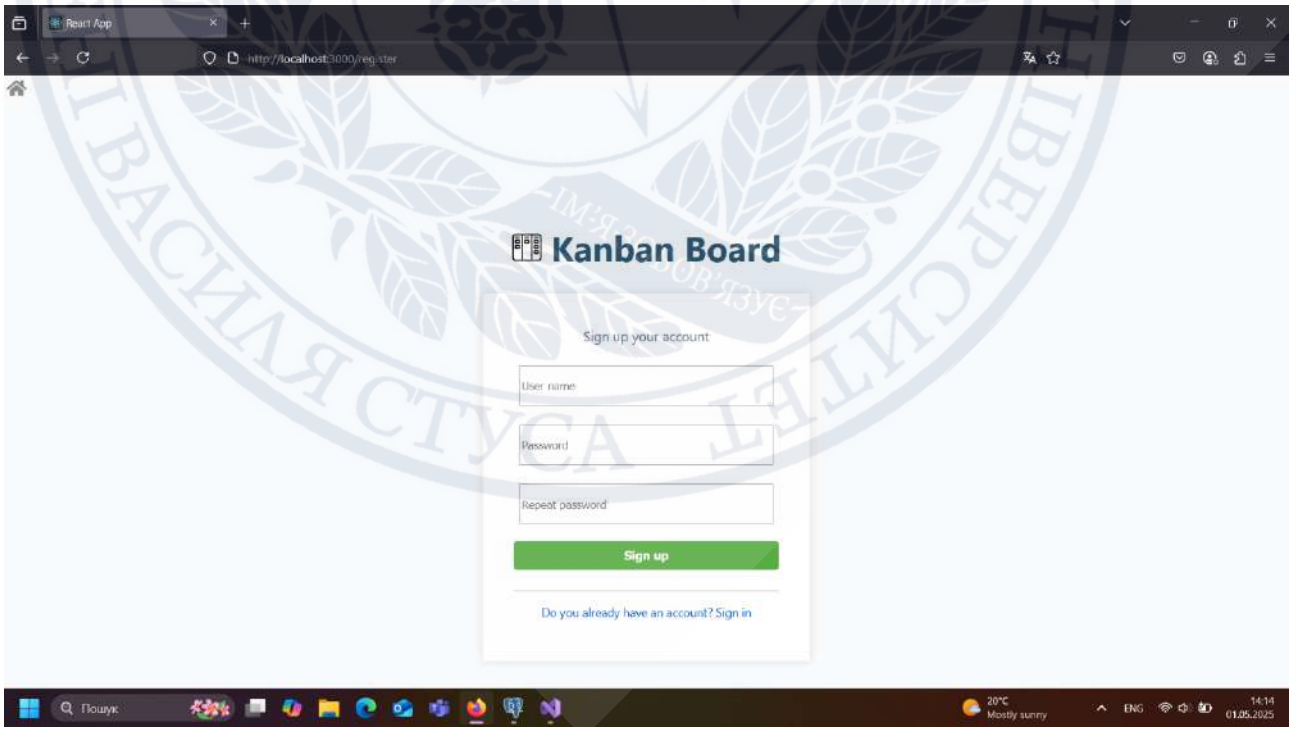

Рисунок Б.2 – Форма реєстрації користувача

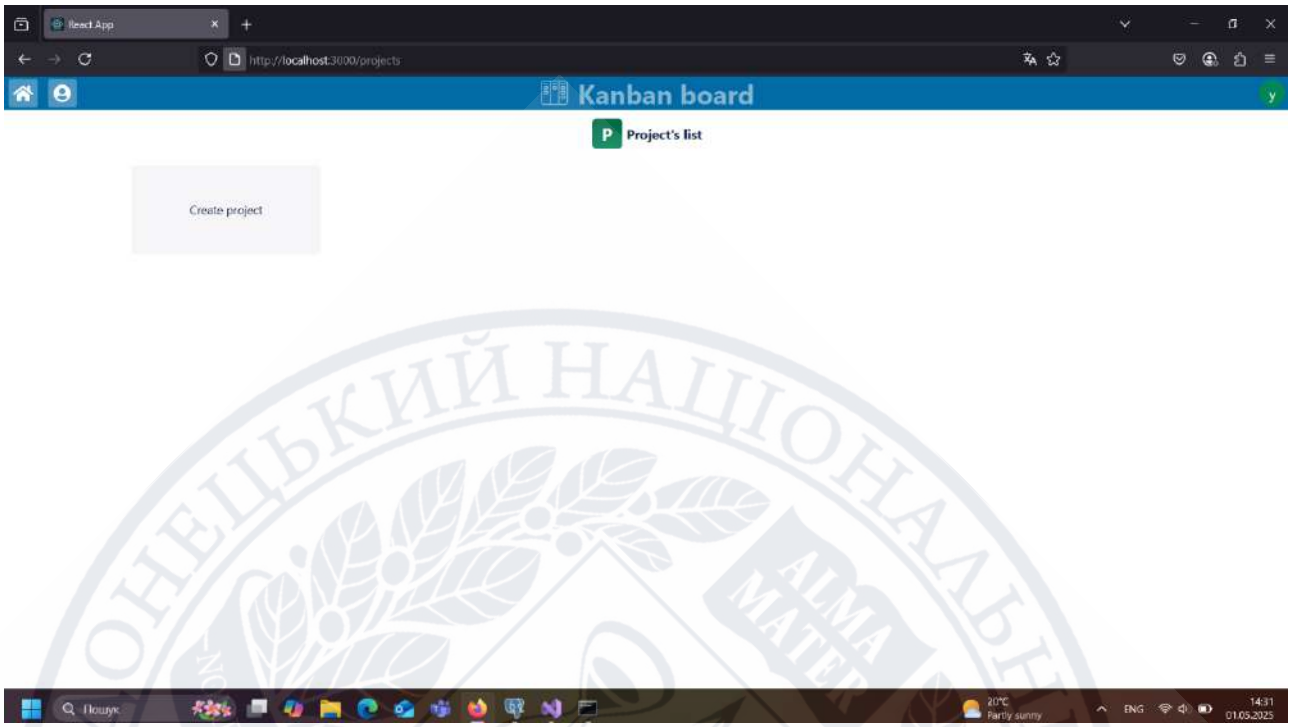

Рисунок Б.3 – Сторінка зі списком проектів користувача

Рисунок Б.4 – Інтерфейс створення нового проекту

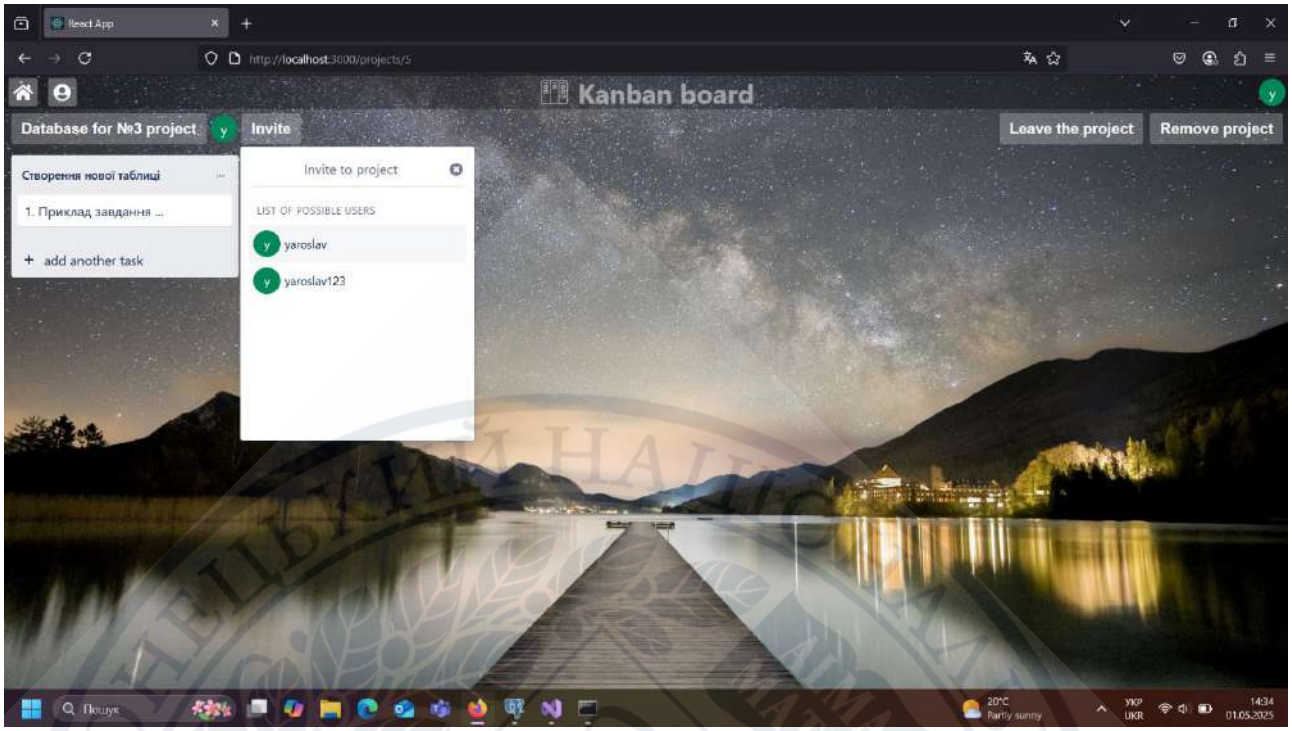

Рисунок Б.5 – Вигляд сторінки зі створеним проектом

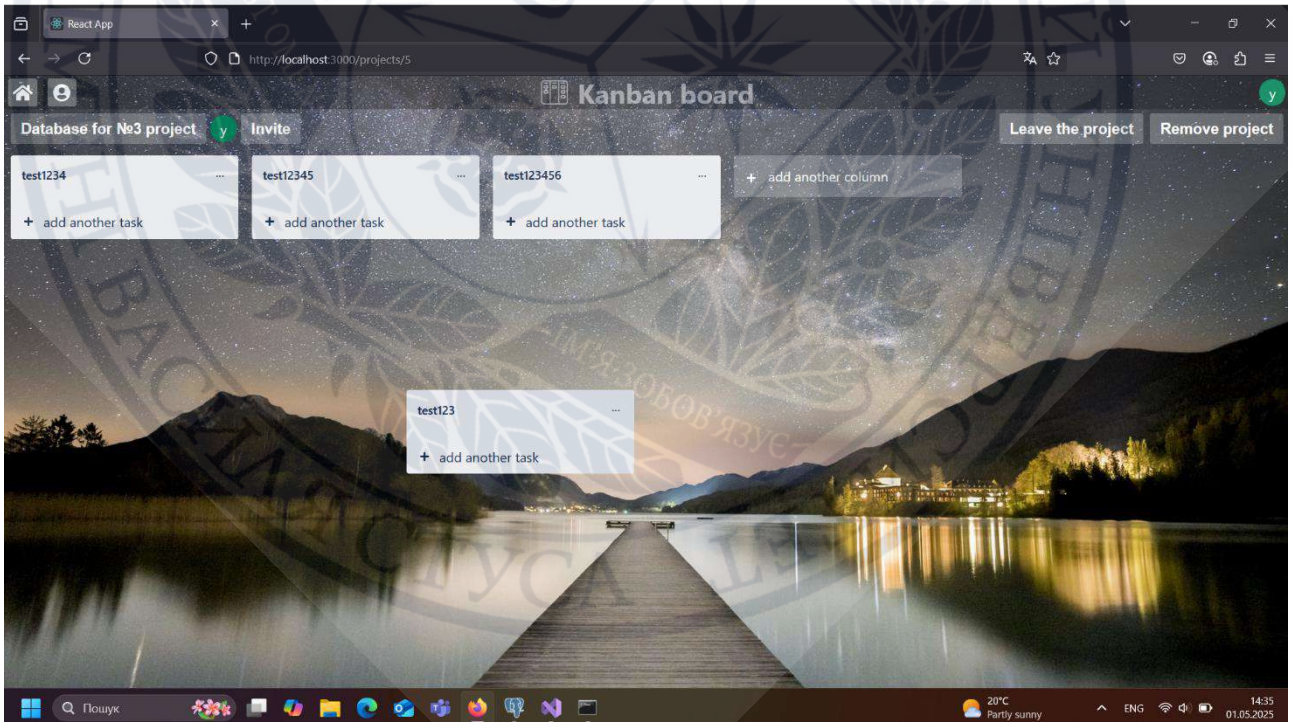

Рисунок Б.6 – Приклад роботи функціоналу drag-and-drop

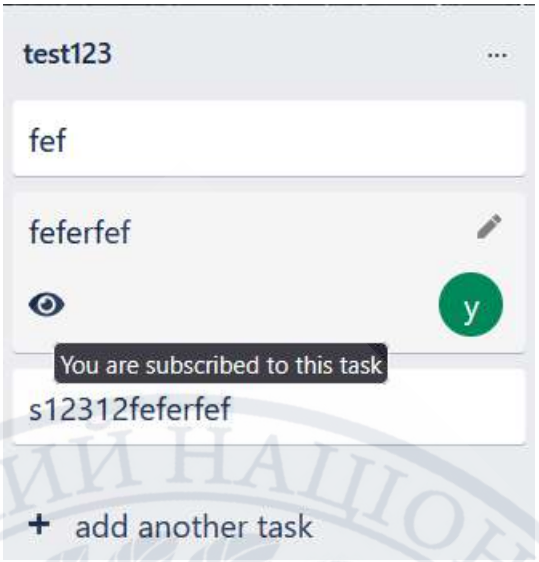

Рисунок Б.7 – Призначений користувач до завдання

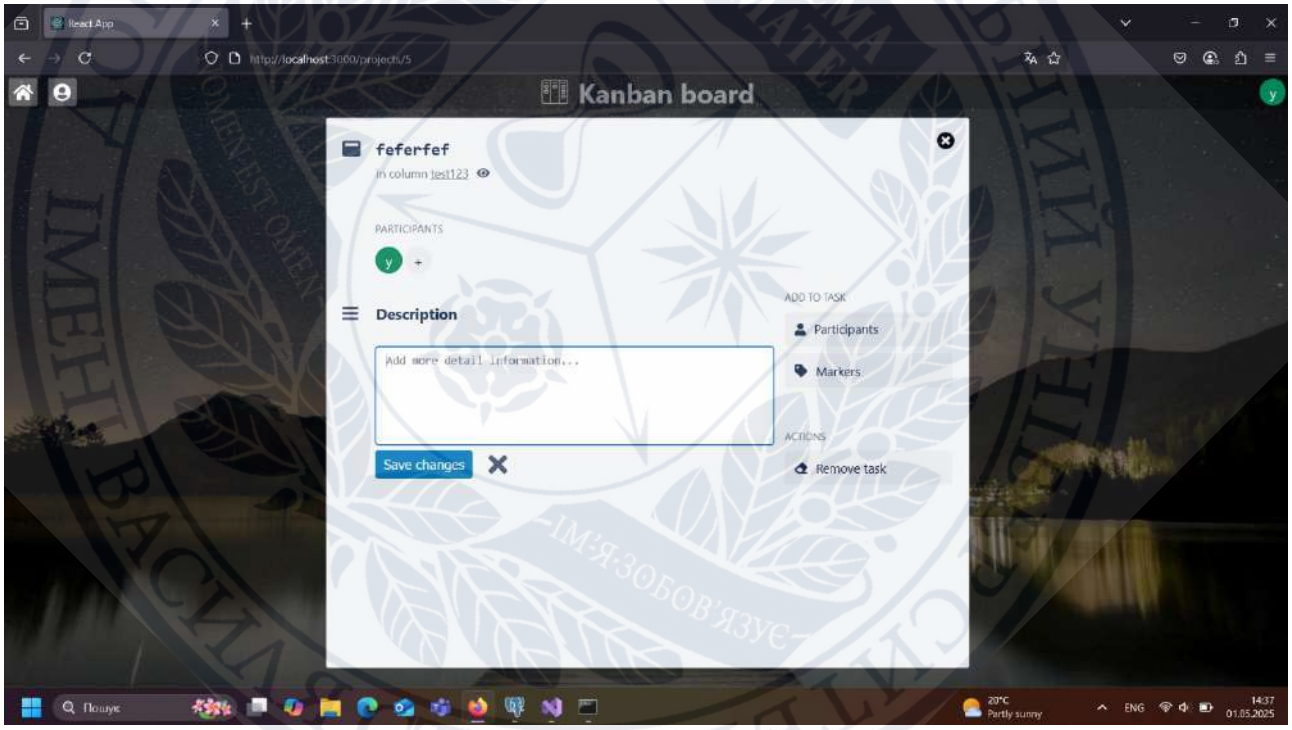

Рисунок Б.8 – Деталі завдання

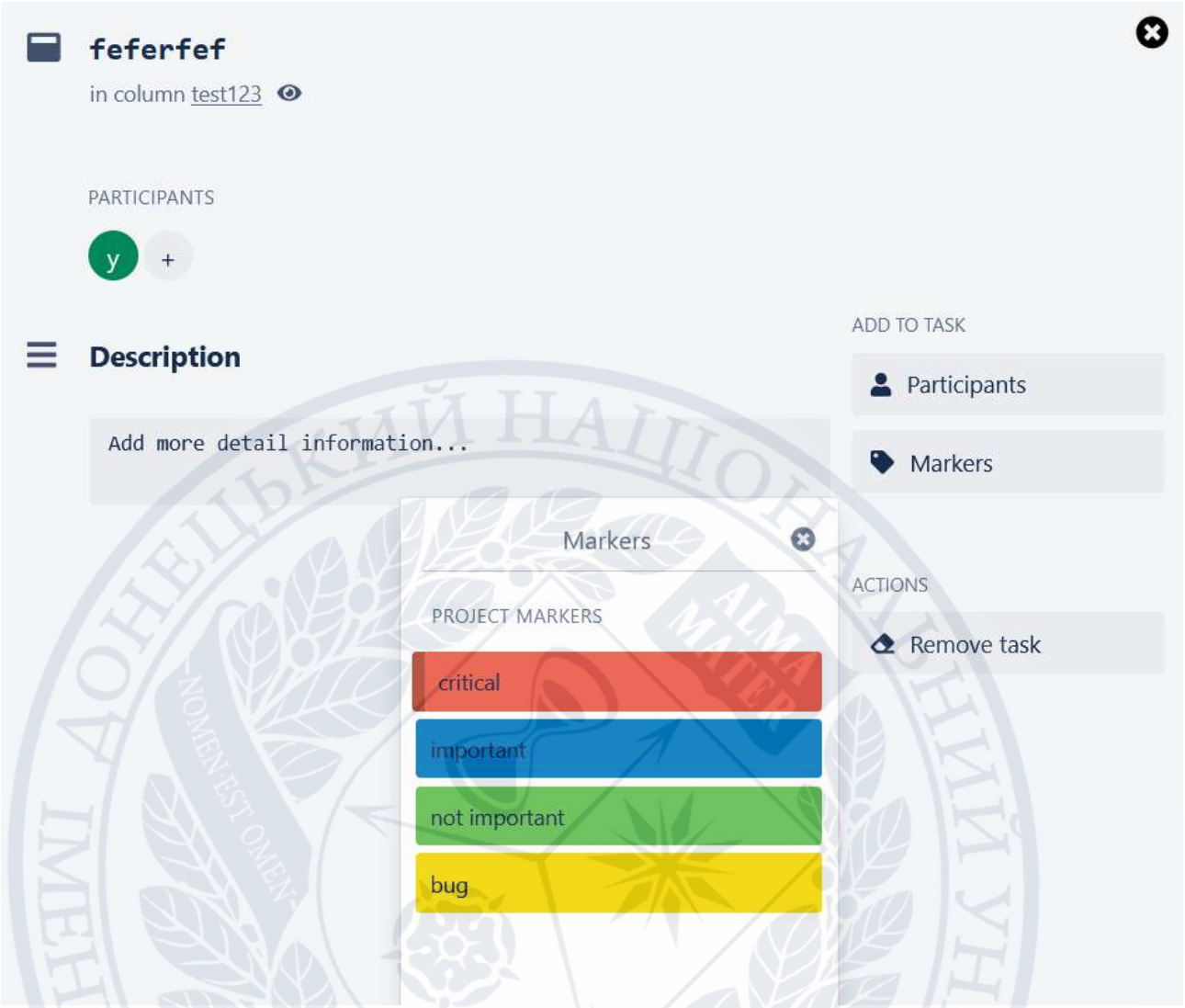

Рисунок Б.9 – Маркери позначення важливості завдання

ДОДАТОК В

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»; що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації; що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів; що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)