

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ДЕРЖАВНИЙ
УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА» ФАКУЛЬТЕТ
ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ КАФЕДРА
ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Вебсистема управління завданнями для командної роботи»

Виконала студентка групи ІПЗ-22-2
ЛЕЩЕНКО Галина Олександрівна

Керівник роботи:
ЄФРЕМОВ Юрій Миколайович

Рецензент:
ЧИЖМОТРЯ Олена Геннадіївна

Житомир – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ ДЕРЖАВНИЙ
УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА» ФАКУЛЬТЕТ
ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ КАФЕДРА
ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«28» лютого 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ЛЕЩЕНКО Галина Олександрівна**

Керівник роботи: **ЄФРЕМОВ Юрій Миколайович**

Тема роботи: **«Вебсистема управління завданнями для командної роботи»**,
Затверджена Наказом закладу вищої освіти від **«27» лютого 2026 р., №92/с**

Вихідні дані для роботи: **Об'єктом дослідження** є процеси організації та координації спільної діяльності учасників командних проєктів у вебсередовищі.

Предметом дослідження є програмні засоби, методи та вебтехнології розробки системи для автоматизації управління завданнями та взаємодії між членами команди.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Єфремов Ю.М.	27.02.2026р	27.02.2026р
2	Єфремов Ю.М.	27.02.2026р	27.02.2026р
3	Єфремов Ю.М.	27.02.2026р	27.02.2026р

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з пояснювальної записки та вебсистеми управління завданнями для командної роботи. Пояснювальна записка містить 84 сторінок тексту, 47 рисунків, 25 таблиць та 3 сторінки додатків. Список використаних джерел складається з 30 позицій на 3 сторінках. Загальний обсяг роботи - 87 сторінок.

Головна мета полягає у комплексному проектуванні, програмній реалізації та тестуванні вебсистеми управління завданнями. Платформа створена для підвищення результативності командної роботи за рахунок зручного візуального представлення даних, налаштовуваної рольової моделі доступу та оптимізованої внутрішньої архітектури.

В основу розробки покладено клієнт-серверну архітектуру. Практична реалізація виконана з використанням такого технологічного стеку: Node.js + Express, React і TypeScript. Готовий програмний продукт орієнтований на автоматизацію процесів делегування завдань та забезпечення зручної комунікації між учасниками команди.

КЛЮЧОВІ СЛОВА: СТЕК, ВЕБСИСТЕМА, KANBAN, REACT, NODE, GITLAB.

					ІПЗ.КР.Б – 121 – 26 – ПЗ					
Змн.	Арк.	№ докум.	Підпис	Дата	Вебсистема управління завданнями для командної роботи			Лім.	Арк.	Аркушів
Розроб.		Лещенко Г.О.								
Керівник		Єфремов Ю.М..							3	87
								Житомирська політехніка, Група ІПЗ-22-2		
Рецензент		Чижмотря О.Г.								
Зав.каф.		Вакалюк Т.А.			Пояснювальна записка					

ABSTRACT

The bachelor's qualification thesis consists of an explanatory note and a web-based task management system for teamwork. The explanatory note contains 84 pages of text, 47 figures, 25 tables, and 3 pages of appendices. The list of references consists of 30 sources on 3 pages. The total volume of the work is 87 pages.

The main goal is the comprehensive design, software implementation, and testing of a web-based task management system. The platform is designed to increase teamwork productivity through a convenient visual data representation, a customizable role-based access model, and an optimized internal architecture.

The development is based on a client-server architecture. The practical implementation was carried out using the following technology stack: Node.js + Express, React, and TypeScript. The finished software product is aimed at automating task delegation processes and ensuring convenient communication between team members.

KEYWORDS: STACK, WEBSYSTEM, KANBAN, REACT, NODE, GITLAB.

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП	8
РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ПОБУДОВИ ВЕБСИСТЕМ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ.....	10
1.1. Аналіз предметної області та визначення завдань автоматизації командної взаємодії	10
1.2. Огляд та порівняльний аналіз існуючих програмних аналогів.....	12
1.3. Вибір архітектури та загальна структура майбутнього програмного продукту	16
1.4. Обґрунтування вибору інструментальних засобів розробки та вимоги до апаратного забезпечення	18
Висновки до першого розділу.....	23
РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБСИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ	24
2.1 Визначення варіантів використання.....	24
2.2. Аналіз системних та нефункціональних вимог, проєктування бази даних системи	30
2.3. Проєктування та реалізація алгоритмів роботи вебсистеми	39
2.4. Реалізація функціоналу вебсистеми управління завданнями для командної роботи	53
Висновки до другого розділу	59
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБСИСТЕМОЮ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ	61
3.1. Порядок встановлення та налаштування параметрів вебсистеми	61

3.2. Структура інтерфейсу та порядок використання вебсистеми управління завданнями для командної роботи	64
3.3. Тестування роботи вебсистеми.....	76
Висновки до третього розділу.....	79
ВИСНОВКИ.....	81
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	82
ДОДАТКИ.....	85
Додаток А. «Діаграма бази даних».....	86
Додаток В. «Проходження тестів у CI/CD pipeline»	87

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Інтерфейс прикладного програмування (Application Programming Interface);

SQL – Мова структурованих запитів (Structured Query Language);

ORM – Об'єктно-реляційне відображення (Object-Relational Mapping);

RTK – Набір інструментів Redux (Redux Toolkit);

JQL – Мова запитів Jira (Jira Query Language);

БД – База Даних;

ВСТУП

Актуальність теми. У сучасних умовах цифровізації бізнес-процесів та переходу до гібридних моделей роботи ефективне управління завданнями стає ключовим фактором успіху будь-якої команди. Зі зростанням складності проєктів та збільшенням обсягів інформації, традиційні методи координації (через месенджери або електронну пошту) виявляються неефективними, оскільки призводять до втрати контексту, порушення термінів та розмиття відповідальності.

Незважаючи на наявність на ринку великої кількості рішень для управління проєктами, питання вибору оптимального інструмента залишається відкритим. Популярні системи часто страждають або від надмірної складності, що потребує тривалого навчання персоналу, або від надмірної простоти, яка не дозволяє ефективно масштабувати робочі процеси. Це створює об'єктивну потребу в розробці гнучкої, швидкої та інтуїтивно зрозумілої вебсистеми, яка б поєднувала професійну глибину контролю з високою ергономікою інтерфейсу.

Об'єктом дослідження є процеси командної взаємодії та координації спільної діяльності в межах виконання професійних та навчальних проєктів.

Предметом дослідження є методи, архітектурні підходи та програмні засоби автоматизації управління завданнями у вебсередовищі.

Метою роботи є проєктування та розробка вебсистеми управління завданнями «Lesqor», яка забезпечує оптимальний баланс між функціональною насиченістю та зручністю використання, впроваджуючи механізми вбудованої автоматизації та ефективної візуалізації даних.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати предметну область та визначити ключові вимоги до систем командної взаємодії;
- провести порівняльний аналіз існуючих програмних аналогів та виявити їхні архітектурні обмеження;

- обґрунтувати вибір технологічного стеку та архітектури майбутнього програмного продукту;
- спроектувати логічну структуру бази даних та програмні інтерфейси системи;
- реалізувати функціональні модулі системи (управління проєктами, завданнями, ролями користувачів);
- провести тестування розробленого рішення та оцінити його ефективність.

Практичне значення отриманих результатів полягає у створенні працездатного програмного продукту, який може бути впроваджений у роботу малих ІТ-команд, освітніх груп або використаний як інструмент персональної продуктивності. Застосування розробленої системи дозволяє скоротити час на організаційну рутину та підвищити прозорість виконання різного характеру проєктів.

РОЗДІЛ 1. АНАЛІЗ МЕТОДІВ ТА ЗАСОБІВ ПОБУДОВИ ВЕБСИСТЕМ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ

1.1. Аналіз предметної області та визначення завдань автоматизації командної взаємодії

Командна робота є невіддільною складовою багатьох сфер діяльності, зокрема освітнього процесу, бізнесу та розробки програмного забезпечення. Ефективне виконання спільних проєктів вимагає чіткої організації, зрозумілого розподілу обов'язків та прозорості взаємодії між усіма учасниками. Проте організація цих процесів без використання спеціалізованих інформаційних інструментів часто супроводжується низкою труднощів.

Однією з найпоширеніших проблем є неефективний розподіл часу та недотримання термінів виконання (дедлайнів). Без централізованої системи управління учасники команд витрачають значний ресурс на з'ясування організаційних питань, пошук необхідної інформації та синхронізацію дій, замість того, щоб фокусуватися безпосередньо на виконанні завдань. Для керівників (наприклад, викладачів або менеджерів проєктів) процес моніторингу успішності та оцінювання індивідуального внеску кожного учасника суттєво ускладнюється. Виникає необхідність ведення ручного обліку та постійного опитування виконавців для актуалізації даних щодо поточного етапу робіт.

Впровадження вебсистеми управління завданнями дозволяє розв'язати ці проблеми, створюючи єдине цифрове середовище. Наявність візуалізованої панелі управління (Dashboard) забезпечує швидкий доступ до актуальної інформації про стан проєкту в режимі реального часу. Крім того, подібна система є універсальною: вона корисна не лише для колективів, а й для окремих користувачів, які потребують інструментів для самоорганізації. Можливість створення персонального робочого простору дозволяє декомпонувати власні цілі на дрібні кроки, встановлювати терміни та наочно відстежувати особистий прогрес.

Виходячи з проведеного аналізу предметної області, виникає об'єктивна потреба у розробці відповідного програмного забезпечення. Основними завданнями автоматизації командної та індивідуальної взаємодії у межах даної роботи є:

- централізація управління завданнями, що передбачає забезпечення можливості створення, редагування, видалення та розподілу доручень із зазначенням пріоритетів і термінів виконання;
- автоматизація контролю прогресу шляхом впровадження інтерактивної робочої панелі для візуалізації поточного стану проєкту та відстеження життєвого циклу окремих завдань;
- організація рольової моделі доступу, яка реалізується через створення інструментів для творців команд із можливістю управління складом учасників, призначення відповідальних та моніторингу їхньої ефективності;
- оптимізація комунікації за рахунок інтеграції багаторівневої системи спілкування, що охоплює автоматичне формування спільних чатів при реєстрації команди, підтримку приватних діалогів для комунікації вічна-віч, а також забезпечення контекстності обговорень завдяки прив'язці коментарів безпосередньо до робочих об'єктів;
- забезпечення персоналізації та безпеки робочого простору, що досягається реалізацією надійної авторизації через Google-акаунти, що додатково включає створення адаптивного інтерфейсу, побудованого на інтелектуальному сортуванні проєктів, команд та чатів за часом останнього доступу, разом із гнучкою системою фільтрації даних на сторінках системи;
- поглиблена візуалізація робочих процесів через впровадження інтерактивної Kanban-дошки, що дозволяє наочно відстежувати рух завдань та автоматизувати зміну їхніх статусів за допомогою перетягування (Drag-and-Drop), підтримуючи при цьому налаштування

кастомних колонок під специфічні потреби кожної команди для максимальної гнучкості;

- впровадження механізму глобального пошуку, який виступає універсальним інструментом для швидкого знаходження користувачів за нікнеймами, а також миттєвого доступу до необхідних проєктів, команд чи конкретних завдань у межах усієї системи.

1.2. Огляд та порівняльний аналіз існуючих програмних аналогів

У сучасному середовищі розробки програмного забезпечення та менеджменту вибір інструменту для управління проєктами перестав бути суто технічним питанням, перетворившись на стратегічне рішення, що визначає ефективність взаємодії всередині команди. Для детального обґрунтування розробки власної вебсистеми «Lesqor» необхідно провести критичний аналіз лідерів ринку, розглядаючи їх не лише як набір інтерфейсних рішень, а як реалізацію певних управлінських парадигм із їхніми архітектурними обмеженнями.

Яскравим представником візуально-орієнтованого підходу є Trello, що базується на методології Kanban [1]. Ця система стала еталоном завдяки концепції «низького порогу входження»: користувач здатний почати роботу без попередньої підготовки, оскільки метафора карток на дошці є інтуїтивно зрозумілою. Проте при масштабуванні проєкту ця простота перетворюється на серйозний недолік, відомий як проблема «горизонтального розростання». Коли кількість завдань перевищує критичну межу, візуальний простір дошки стає перевантаженим, що призводить до втрати фокуса та зростання когнітивного тиску на учасників команди. Крім того, архітектурна архітектоніка Trello має обмежену систему внутрішніх зв'язків. Відображення складних залежностей, де виконання одного завдання жорстко заблоковане іншим, реалізовано досить поверхнево. Це змушує команди інтегрувати сторонні розширення (Power-Ups), що не лише ускладнює загальну екосистему продукту, а й створює ризики для інформаційної безпеки, оскільки

конфіденційні корпоративні дані стають доступними для зовнішніх плагінів із неконтрольованою політикою конфіденційності (див. рис. 1.1).

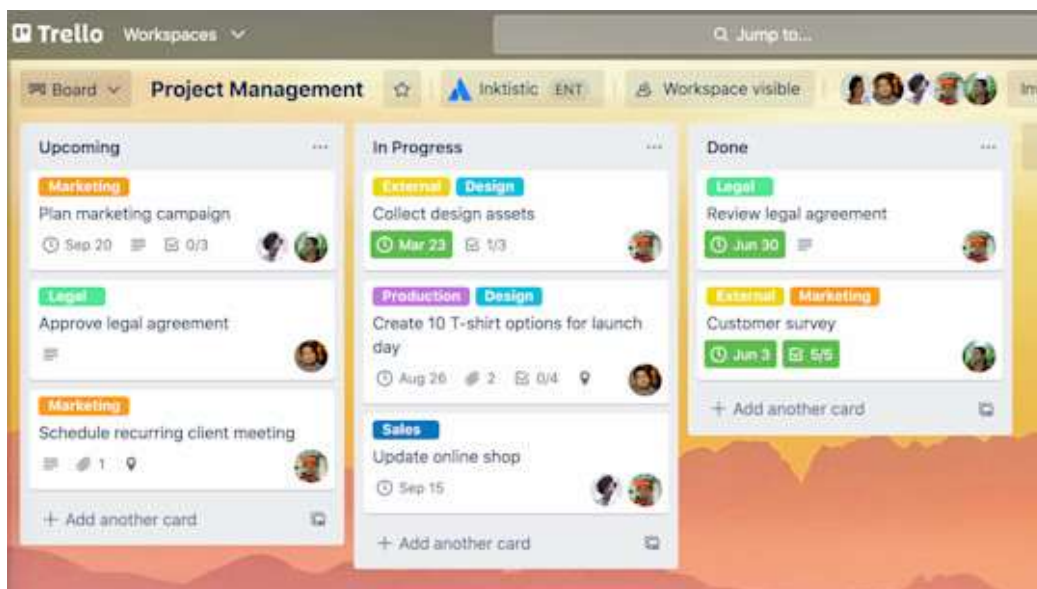


Рис. 1.1 – Інтерфейс Trello

Аналіз Trello дозволяє виділити надзвичайно успішну концепцію візуальної доступності. Її методологія Kanban є еталоном інтуїтивності, що забезпечує швидкий старт для будь-якого користувача. Проте при масштабуванні проєктів у Trello виникає проблема «горизонтального розростання», що веде до когнітивного перевантаження. У майбутній системі «Lesqor» планується запозичити принцип візуальних карток, але переосмислити їхню організацію, щоб уникнути хаосу при великій кількості завдань. Крім того, оскільки Trello змушує використовувати сторонні плагіни для базових автоматизацій, у «Lesqor» передбачається впровадження вбудованої логіки, що забезпечить вищий рівень безпеки та цілісності даних.

На протилежному боці спектра знаходиться Jira від компанії Atlassian - потужний індустріальний інструмент, орієнтований на професійну розробку за методологіями Agile, Scrum та DevOps [2]. Її архітектура дозволяє детально налаштовувати кожен крок життєвого циклу завдання, створювати багаторівневі ієрархії від епіків до підзавдань та оперувати беклогами. Проте Jira часто стає прикладом функціональної надлишковості. Для багатьох команд, особливо у сегменті малого та середнього бізнесу, її інтерфейс є занадто щільним і складним. Адміністративне налаштування робочих

процесів (workflows) часто перетворюється на окремий фаховий обов'язок, що забирає ресурси, які могли б бути спрямовані безпосередньо на розробку. Багато функцій, як-от складна мова запитів JQL чи глибинні аналітичні звіти, залишаються незадіяними, виконуючи роль «візуального шуму», що сповільнює роботу веб-інтерфейсу та відволікає користувача від поточних пріоритетів (див. рис. 1.2).

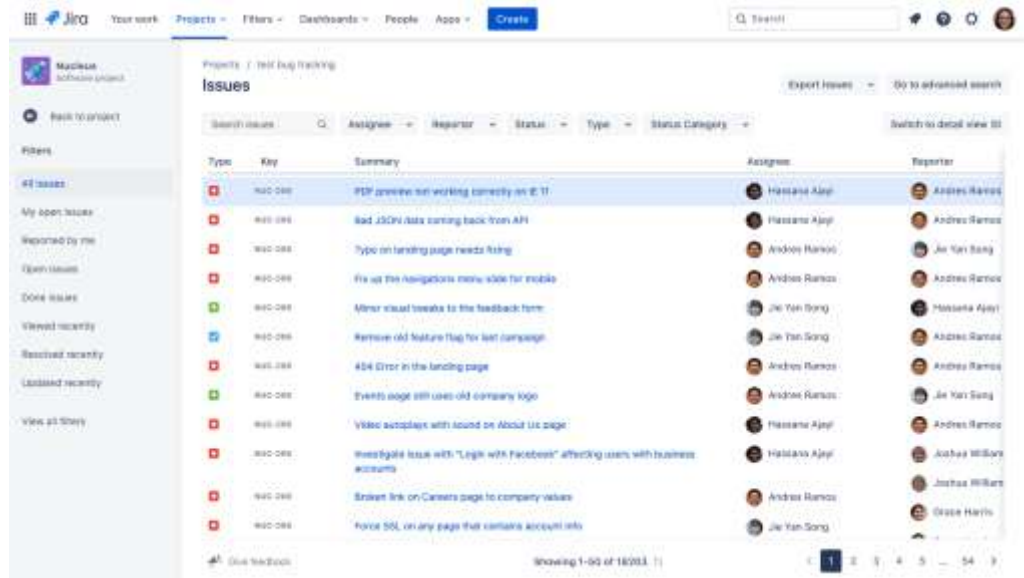


Рис.1.2 – Інтерфейс Jira

На противагу простоті, Jira пропонує потужну ієрархічну структуру та детальний контроль життєвого циклу завдань. Це професійний стандарт, проте він страждає від функціональної надлишковості. Для більшості команд інтерфейс Jira є занадто щільним, а поріг входження – надто високим. При розробці «Lesqor» за основу буде взято ідею строгої типізації станів завдань та ієрархічної підпорядкованості, проте архітектура буде спрощена до оптимальної трирівневої моделі. Це дозволить зберегти професійну глибину управління, притаманну Jira, але позбутися «візуального шуму» та необхідності в окремому адміністраторі системи.

Схожу нішу, але з іншим акцентом, займає Asana, вона пропонує концепцію «графа роботи», де користувач може самостійно обирати формат представлення даних: від класичних списків до діаграм Ганта [3]. Проте прагнення Asana бути універсальним рішенням для всіх типів пристроїв призвело до певних ергономічних дефектів у вебверсії. Дизайн системи часто

сприймається як розтягнута мобільна аплікація, де величезна кількість вільного простору (white space) використовується нераціонально. На великих моніторах це створює ефект «розмитості» контексту, змушуючи користувача виконувати зайві маніпуляції для отримання потрібних даних. Крім того, вартість розширених функцій автоматизації в Asana часто є невиправдано високою для команд, що лише починають свій шлях (див. рис. 1.3).

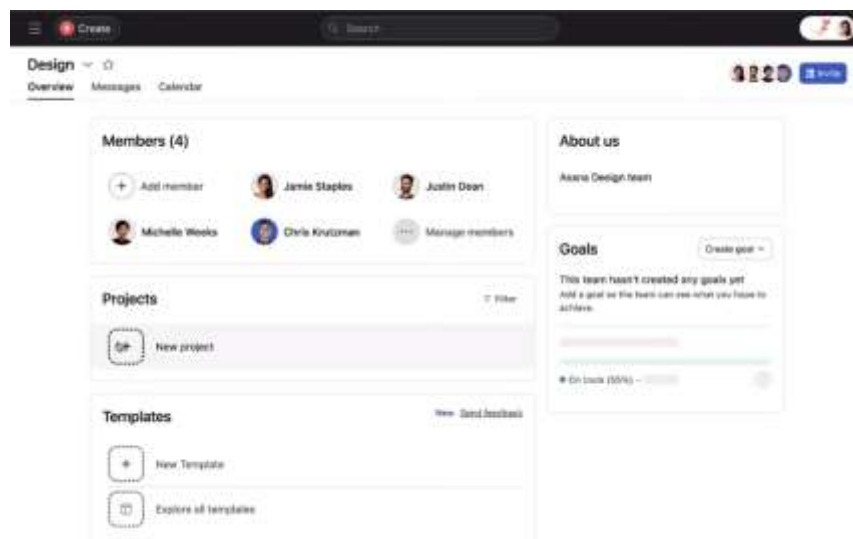


Рис. 1.3 – Інтерфейс Asana

Досвід використання Asana демонструє важливість мультимедійності – можливості перемикатися між списками та часовими шкалами. Однак ергономіка Asana часто критикується за нераціональне використання екранного простору (надлишок «білого місця» у вебверсії). У межах проєктування «Lesqor» обрано протилежний підхід, де вільний простір використовується цілеспрямовано для створення максимально дружнього до користувача інтерфейсу. Замість перевантаження екрана надмірними даними, система зосереджується на зниженні бар'єра входження, адже мінімалістичний дизайн та значна кількість вільного простору дозволяють новачкам інтуїтивно опанувати логіку роботи, не відволікаючись на складні та перенасичені елементами меню, що робить «Lesqor» доступнішим інструментом для команд, які прагнуть швидкого старту без тривалого навчання. Водночас «повітря» між елементами допомагає уникнути виникнення візуального шуму, дозволяючи користувачу повністю зосередитися на поточних діях, таких як обговорення завдань у коментарях, зміна статусів чи активна комунікація в

чаті, що суттєво підвищує загальний комфорт та ефективність повсякденної взаємодії з системою.

Таким чином, майбутня система «Lesqor» не є простим копіюванням аналогів, а виступає як синтез перевірених ринком рішень (див. табл. 1.1).

Таблиця 1.1

Характеристика ключових функціональних модулів вебсистеми «Lesqor» на основі аналізу аналогів

Функціональний модуль	Джерело концепції	Очікувана реалізація у «Lesqor»	Цільова перевага розробки
Візуальна модель завдань	Trello	Канбан-дошка з картками, що підтримують швидке редагування.	Інтуїтивний старт та мінімальний час на внесення змін.
Структура даних	Jira	Оптимізована трирівнева ієрархія (команда – проєкт – завдання).	Професійна глибина управління без надлишкової складності.
Автоматизація процесів	Власна ідея	Вбудована тригерна логіка: кастомні статуси та автозміна параметрів завдань.	Вища безпека та автономність без сторонніх плагінів.
Ергономіка інтерфейсу	Аналіз недоліків в Asana	Цілеспрямоване використання «білого простору» для зниження візуального шуму.	Зниження бар'єра входження та комфорт для швидкого навчання.
Пошукова система	Синтез аналогів	Глобальний пошук за ніками, назвами проєктів, команд та завдань.	Миттєвий доступ до даних у масштабованих проєктах.
Комунікаційна модель	Синтез аналогів	Інтеграція командних чатів, приватних повідомлень та контекстних коментарів.	Зменшення втрат інформації та синхронізація дій у реальному часі.

Узагальнюючи результати аналізу, наведено перелік ключових можливостей, які планується реалізувати у вебсистемі «Lesqor», та зазначено, які саме ідеї були адаптовані з існуючих платформ.

1.3. Вибір архітектури та загальна структура майбутнього програмного продукту

На етапі проєктування системи управління завданнями «Lesqor» пріоритетним завданням є вибір такої архітектурної моделі, яка забезпечить

гнучкість розробки, високу швидкість відгуку інтерфейсу та стабільність експлуатації. Оптимальним рішенням обрано клієнт-серверний підхід із повним розділенням інтерфейсу та логіки обробки даних, що дозволяє ізолювати процеси розробки: серверна частина фокусується на безпеці та обробці запитів, тоді як клієнтська - на забезпеченні якісної взаємодії з користувачем.

Для реалізації фронтенд-частини заплановано використання концепції Single Page Application (SPA) на базі бібліотеки React [4][5]. Такий вибір зумовлений специфікою роботи з Канбан-дошками, де користувачі постійно взаємодіють з елементами інтерфейсу. Для ефективного керування станом системи та автоматизації кешування даних застосовується стек Redux Toolkit та RTK Query [6][7]. Це дозволяє мінімізувати кількість запитів до сервера та забезпечує миттєве оновлення даних у межах інтерфейсу без потреби в постійному перезавантаженні сторінок, що робить роботу з «Lesqor» максимально плавною.

Взаємодія між компонентами проєктується через програмний інтерфейс REST API, що обмінюється даними у форматі JSON [8][9]. Це закладає фундамент для масштабованості: за потреби до розробленого API можна буде легко підключити сторонні сервіси. На серверній стороні передбачається впровадження механізмів Middleware для попередньої перевірки прав доступу та валідації вхідних даних [10].

Окремим етапом забезпечення стабільності є контейнеризація всього середовища розробки та розгортання за допомогою Docker, що гарантує ідентичність роботи системи на будь-яких етапах життєвого циклу продукту [11]. Основою для збереження інформації виступає реляційна база даних, структура якої базується на чітких зв'язках між сутностями «Користувач», «Проект», «Команда» та «Завдання». Для взаємодії з БД обрано технологію ORM, що дозволяє оперувати даними на рівні об'єктів коду, автоматизує створення таблиць та захищає систему від вразливостей, таких як SQL-ін'єкції [12]. Таке поєднання SPA-архітектури реактивного керування станом,

контейнеризації та структурованої моделі даних створює надійний фундамент для реалізації «Lesqor» [13]. Для забезпечення функціоналу чатів у реальному часі в систему інтегровано протокол WebSockets, що дозволяє миттєво доставляти повідомлення між користувачами без затримок, характерних для стандартних HTTP-запитів [14] (див. табл. 1.2).

Таблиця 1.2

Обґрунтування архітектурних рішень системи «Lesqor»

Компонент системи	Обрана концепція	Технічне обґрунтування та переваги
Клієнтська частина	React (SPA)	Забезпечує високу реактивність інтерфейсу та зручність роботи з елементами Drag-and-Drop.
Керування станом	Redux Toolkit / RTK Query	Оптимізує кешування даних та мінімізує кількість запитів до сервера, прискорюючи роботу інтерфейсу.
Протокол взаємодії	REST API (JSON)	Створює універсальний інтерфейс для передачі даних між фронтендом та бекендом.
Середовище розгортання	Docker	Забезпечує стабільність роботи системи через контейнеризацію середовища розробки.
Логіка безпеки	Middleware	Виконує попередню перевірку прав доступу (автентифікацію) та валідацію вхідних даних.
Збереження даних	Реляційна модель (SQL)	Гарантує цілісність даних та суворе дотримання зв'язків між об'єктами системи.
Взаємодія з БД	ORM	Абстрагує роботу з БД, захищає від SQL-ін'єкцій та прискорює процес розробки.
Комунікація в реальному часі	WebSockets (Socket.io)	Забезпечує миттєвий обмін повідомленнями в чатах та оновлення статусів завдань без оновлення сторінки.

Обґрунтування архітектурних рішень демонструє комплексний підхід до проєктування системи Lesqor. Вибір кожного технологічного стека продиктований необхідністю забезпечення продуктивності, масштабованості та безпеки, що є критично важливими вимогами для платформи спільного управління завданнями та командної роботи. Клієнтська частина базується на React, що дозволяє реалізувати підхід Single Page Application. Це є оптимальним вибором для створення динамічних інтерактивних інтерфейсів,

таких як канбан-дошки, оскільки забезпечує плавну роботу складних візуальних елементів, зокрема Drag-and-Drop для переміщення завдань, без необхідності повного перезавантаження сторінки. Управління глобальним станом системи здійснюється за допомогою Redux Toolkit та RTK Query. Це вирішує проблему синхронізації клієнта з сервером, а завдяки розширеним можливостям кешування даних суттєво зменшується кількість надлишкових мережових запитів, що розвантажує сервер та робить відгук інтерфейсу майже миттєвим.

Взаємодія між клієнтом та сервером побудована на стандарті REST API для виконання базових операцій зі створення, читання, оновлення та видалення даних у форматі JSON, що забезпечує стабільність та передбачуваність системи. Для забезпечення комунікації в реальному часі додатково впроваджено WebSockets за допомогою Socket.io, що гарантує миттєву двосторонню синхронізацію робочого простору, оновлення статусів завдань в усіх учасників команди та швидкий обмін повідомленнями.

Збереження складної ієрархічної бізнес-логіки, зв'язків між командами, проєктами та завданнями, а також транзакційна цілісність забезпечуються реляційною моделлю бази даних SQL. Використання ORM підходу для взаємодії з базою не лише підвищує швидкість розробки завдяки об'єктно-орієнтованому стилю, але й виступає додатковим рівнем безпеки, автоматично екрануючи вхідні дані та запобігаючи можливим вразливостям. Вся логіка безпеки винесена на рівень проміжних програмників Middleware для централізованої перевірки токенів авторизації, валідації вхідних даних та перевірки ролей до моменту виконання основних запитів.

Нарешті, вся інфраструктура системи ізолюється та розгортається за допомогою Docker-контейнерів, що повністю усуває проблему розбіжностей у версіях програмного забезпечення та гарантує ідентичну і стабільну роботу системи на будь-якому сервері. Загалом, така побудова архітектури створює надійний інженерний фундамент для подальшого масштабування проєкту та легкої інтеграції нового функціоналу без втрати загальної швидкодії.

1.4. Обґрунтування вибору інструментальних засобів розробки та вимоги до апаратного забезпечення

Проектування вебсистеми управління завданнями «Lesqor» вимагає вибору технологічного стеку, здатного забезпечити високу продуктивність, масштабованість та стабільність обробки запитів у режимі реального часу. На початковому етапі розробки було проведено критичний аналіз існуючих серверних екосистем, зокрема PHP (Laravel), Python (Django) та Java (Spring) [15][16][17]. Laravel забезпечує високу швидкість прототипування, Django вирізняється надійністю архітектурних рішень «з коробки», а Spring є індустріальним стандартом для побудови високонавантажених корпоративних систем. Проте в межах даного проєкту вказані стеки були визнані недостатньо ефективними через імовірні труднощі при реалізації асинхронної взаємодії, що є критично важливим для забезпечення інтерактивності інтерфейсу. Натомість було обрано екосистему Node.js у поєднанні з TypeScript (Full-stack TypeScript approach) [18][19]. Ключовим фактором цього вибору є забезпечення типобезпечності та однорідності кодової бази (Single Language Paradigm), що мінімізує ризики деградації структури даних при їх передачі між клієнтською та серверною частинами. Неблокуюча модель введення-виведення Node.js у поєднанні з фреймворком Express дозволяє ефективно обробляти велику кількість паралельних WebSocket-з'єднань, що забезпечує низьку затримку при оновленні статусів завдань та синхронізації чатів [20]. Вибір реляційної системи управління базами даних PostgreSQL зумовлений необхідністю підтримання цілісності складних зв'язків між сутностями «Команда», «Проект» та «Завдання» [21].

Для взаємодії з БД обрано ORM Sequelize, яка абстрагує SQL-запити на рівень об'єктно-орієнтованих моделей, що прискорює ітерації розробки. Безпека автентифікації забезпечується використанням JSON Web Token (jsonwebtoken) для управління сесіями, тоді як хешування паролів реалізовано через bcryptjs [22][23].

Додатково, для забезпечення надійної ідентифікації об'єктів у розподіленому середовищі впроваджено генерацію ідентифікаторів через бібліотеку `uuid`, а для обробки файлових завантажень – `middleware multer`. Інтерфейсна частина проєкту базується на компонентному підході `React`, з використанням іконографіки `Lucide-react` та інтеграцією `emojiPicker` для розширення інструментарію комунікації, що загалом оптимізує UX-контекст системи [24].

Таблиця 1.3

Порівняльна характеристика серверних платформ за ключовими архітектурними критеріями

Критерій порівняння	PHP (Laravel)	Python (Django)	Node.js (TypeScript)
Модель обробки	Синхронна (blocking)	Синхронна (WSGI)	Асинхронна (non-blocking)
Типобезпечність	Часткова (через PHP 8+)	Опціональна (hints)	Висока (TypeScript)
Real-time підтримка	Обмежена	Потребує сторонніх інструментів	Нативна (Socket.io)
Масштабованість	Середня	Висока	Висока (event-driven)

Продемонстровано порівняльний аналіз популярних серверних платформ за ключовими архітектурними критеріями (див. табл. 1.3). Цей аналіз дозволяє обґрунтувати вибір оптимального технологічного стека для розробки сучасних вебзастосунків, зокрема тих, що вимагають високої інтерактивності.

Аналізуючи модель обробки запитів, можна відзначити, що PHP та Python традиційно спираються на синхронну (блокуючу) архітектуру. У такій моделі кожен запит обробляється послідовно, що може створювати "вузькі місця" під час очікування операцій вводу-виводу (наприклад, звернень до бази даних). Натомість Node.js використовує асинхронну (неблокуючу) подієво-орієнтовану модель (Event Loop), яка здатна ефективно та паралельно обробляти тисячі одночасних з'єднань, мінімізуючи навантаження на апаратні ресурси сервера. Важливим аспектом проєктування надійної архітектури є типобезпечність. У той час як PHP та Python (Django) пропонують лише

часткову або опціональну типізацію, використання TypeScript у середовищі Node.js забезпечує високий рівень строгої статичної типізації. Це дозволяє виявляти більшість архітектурних та логічних помилок ще на етапі компіляції коду, що значно підвищує загальну стабільність системи та спрощує підтримку великих проєктів.

Критичною вимогою для систем командної роботи є підтримка комунікації в реальному часі. Node.js має беззаперечну архітектурну перевагу в цьому напрямку: його неблокуюча природа ідеально підходить для нативної та стабільної роботи з протоколом WebSockets (через бібліотеки на кшталт Socket.io). Реалізація аналогічного безперервного з'єднання у PHP або Python є менш ефективною і зазвичай вимагає налаштування додаткових брокерів повідомлень чи сторонніх інструментів.

З точки зору масштабованості, хоча Python і демонструє високі показники для багатьох задач, подієво-орієнтований підхід Node.js робить його максимально адаптивним до зростання навантажень. Підсумовуючи дані таблиці, можна зробити висновок, що зв'язка Node.js та TypeScript є найбільш виправданим та сучасним рішенням для проєктів, які потребують високої продуктивності, надійності коду та активного обміну даними в режимі реального часу.

Щодо технічної сторони питання, то система не вимагає надпотужних ресурсів, що робить її доступною для широкого кола користувачів. Для розгортання серверної частини, де будуть жити база даних та логіка Express, цілком вистачить віртуального сервера з двоядерним процесором та близько 4 ГБ оперативної пам'яті. Цього обсягу достатньо для швидкого опрацювання запитів від кількох команд одночасно. З боку звичайного користувача вимоги взагалі мінімальні: оскільки основне навантаження з рендерингу інтерфейсу лягає на браузер, вистачить будь-якого сучасного пристрою з 2 ГБ оперативної пам'яті. Головне – мати актуальну версію браузера (Chrome, Firefox чи Safari) та стабільний інтернет, щоб миттєво отримувати оновлення по своїх

завданнях. Такий підхід робить інструмент легким, але водночас достатньо потужним для щоденного використання в команді.

Висновки до першого розділу

У межах першого розділу обґрунтовано необхідність розробки вебсистеми «Lesqor» для автоматизації командної та індивідуальної взаємодії. Аналіз предметної області засвідчив, що відсутність централізованих інструментів спричиняє неефективний розподіл часу та складність моніторингу прогресу.

Визначено ключові вимоги до системи: централізація завдань, Канбан-візуалізація, рольова модель доступу, багаторівнева комунікація та інтелектуальний пошук. Проведений порівняльний аналіз аналогів (Trello, Jira, Asana) виявив їхні обмеження у вигляді функціональної надлишковості та ергономічних дефектів, що дозволило сформулювати концепцію «Lesqor» як синтез перевірених рішень із акцентом на мінімалістичний інтерфейс та зниження бар'єра входження.

Технічну реалізацію обґрунтовано вибором SPA-архітектури, REST API та WebSockets. Обрання технологічного стеку Full-stack TypeScript (React + Node.js/Express) забезпечує типобезпечність та ефективну обробку асинхронних подій. Використання PostgreSQL з ORM Sequelize гарантує цілісність даних, а контейнеризація в Docker – стабільність розгортання. Результати аналізу доводять, що запропонований підхід є оптимальним для створення масштабованої та зручної в експлуатації системи, яка відповідає сучасним стандартам продуктивності.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБСИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ

2.1 Визначення варіантів використання

Проектування будь-якого програмного комплексу починається з етапу формалізації вимог та моделювання поведінки системи. Для вебсистеми управління завданнями «Lesqor» цей процес базується на розробці сценаріїв взаємодії, які дозволяють трансформувати абстрактні потреби користувачів у конкретні функціональні модулі. Використання об'єктно-орієнтованого аналізу на початкових етапах дозволяє мінімізувати ризики розбіжностей між очікуваним функціоналом та технічною реалізацією.

Ключовим етапом проектування є ідентифікація акторів - зовнішніх сутностей, які взаємодіють із системою. У межах «Lesqor» актори представлені користувачами з різним рівнем адміністративного впливу та специфічними бізнес-цілями в межах команди.

Для забезпечення гнучкості та безпеки командної роботи в архітектуру системи було імплементовано ієрархічну модель ролей, характеристики яких наведено таблиці (див. табл. 2.1).

Таблиця 2.1.

Характеристика акторів системи «Lesqor»

Роль (Актор)	Опис діяльності	Ключові повноваження
Власник (Owner)	Повний контроль над екосистемою команди.	Видалення команди, додавання нових користувачів до команди, керування правами Менеджерів.
Менеджер (Manager)	Організатор робочого процесу та планування.	Створення та розподіл завдань між Виконавцями.
Виконавець (Executor)	Основний опрацювач завдань.	Перегляд завдань, зміна статусів у межах своєї зони відповідальності, коментування.

Характеристика акторів ілюструє чітку трирівневу рольову модель вебсистеми «Lesqor», розроблену для ефективного розподілу повноважень та оптимізації командної взаємодії. Найвищий рівень доступу має Власник, який наділений повним контролем над екосистемою команди, включно з

управління її складом, регулюванням прав Менеджерів та можливістю повного видалення робочого простору. Операційне управління делегується Менеджеру, чії компетенції зосереджені на безпосередній організації робочого процесу, плануванні, а також створенні та розподілі завдань. Своєю чергою, Виконавець виступає основною ланкою опрацювання доручень із базовим рівнем доступу, який дозволяє йому переглядати призначені картки, залишати коментарі для зворотного зв'язку та змінювати статуси виключно в межах власної зони відповідальності. Запропонована ієрархія забезпечує гнучкість управління, прозорість обов'язків та захист системи від несанкціонованого втручання в структуру проєктів.

Важливо зауважити, що в архітектурі системи реалізовано принцип ієрархічного успадкування повноважень: «Менеджер» автоматично отримує всі можливості «Виконавця», а «Власник» акумулює права обох нижчих рівнів у межах своєї команди. Такий підхід дозволяє уникнути дублювання логіки під час налаштування прав доступу на серверній частині (у проміжних шарах Middleware) та суттєво оптимізує процеси верифікації.

Для наочного відображення функціональних меж системи та поведінкових зв'язків між акторами розроблено діаграму варіантів використання (Use Case Diagram) за стандартом UML 2.0 [25]. Ця графоаналітична модель слугує базовим архітектурним планом, що розкриває закладену бізнес-логіку програмного продукту. На наведеній діаграмі (див. рис. 2.1) показано сценарії взаємодії для користувачів із різними рівнями доступу. Варто підкреслити, що розподіл повноважень є контекстозалежним і жорстко прив'язаним до конкретної команди. Завдяки такій гнучкості, користувач може одночасно поєднувати різні ролі в межах системи: бути «Власником» у власному просторі, «Менеджером» у команді партнера та «Виконавцем» у сторонніх завданнях. Подібна надійна ізоляція прав доступу гарантує безпеку даних та робить вебсистему зручною для роботи над крос-функціональними або масштабованими проєктами.

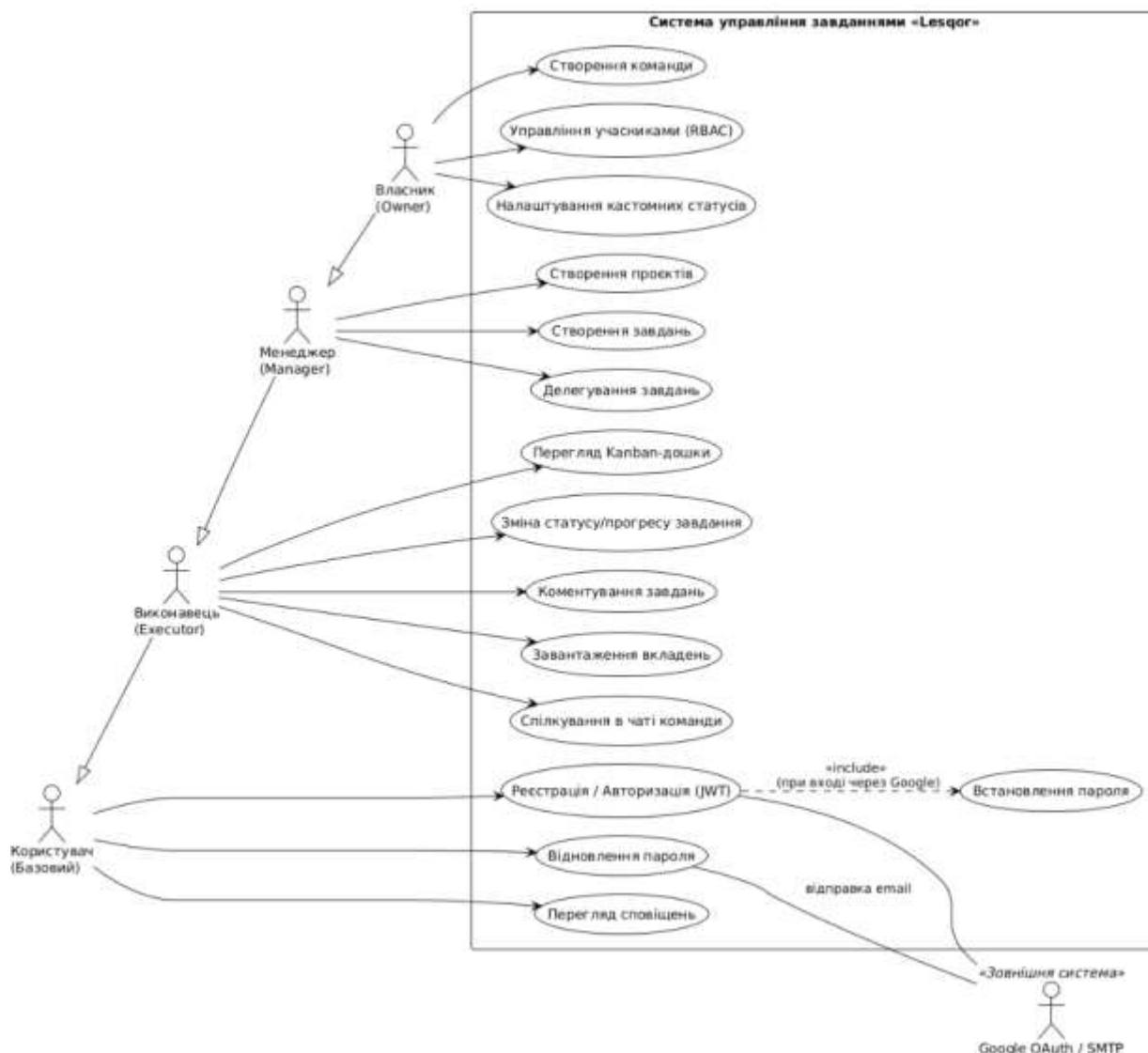


Рис. 2.1 – Діаграма Use case

Для системи «Lesqor» визначено наступні функціональні, що стали основою для подальшої технічної реалізації. Систематизація цих параметрів забезпечує логічний перехід від теоретичного аналізу предметної області до безпосереднього проектування архітектури бази даних та програмних компонентів. Нижче наведено перелік основних функціональних вимог (див. табл. 2.2).

Таблиця 2.2

Перелік основних функціональних вимог

Категорія	Назва вимоги	Деталізація функціоналу
Авторизація	Управління профілем	Регістрація, вхід (JWT), відновлення доступу, зміна аватара та fullName.
Проекти	Організація команд	Створення проєктів, додавання учасників за email, розмежування доступу.

Категорія	Назва вимоги	Деталізація функціоналу
Завдання	Життєвий цикл	Створення, редагування, видалення, встановлення пріоритетів, статусів та дедлайнів.
Ієрархія	Підзавдання	Можливість створення рекурсивних зв'язків для декомпозиції великих завдань.
Комунікація	Зворотний зв'язок	Система миттєвих коментарів, автоматична генерація сповіщень та чат для спілкування.

Наведений перелік функціональних вимог формує комплексне бачення розроблюваної системи, основний функціонал якої зосереджено навколо кількох ключових напрямків. Для забезпечення безпеки та персоналізації передбачено надійну авторизацію на базі JWT-токенів із можливістю повноцінного управління профілем користувача. Організація робочого простору охоплює створення проєктів, формування команд через запрошення електронною поштою та чітке розмежування прав доступу. Безпосереднє управління завданнями включає контроль їхнього повного життєвого циклу, зокрема створення, редагування, зміну статусів, пріоритетів та дедлайнів, а також підтримку ієрархічної структури через підзавдання для декомпозиції масштабних цілей на дрібні кроки. Крім того, невід'ємною частиною системи є командна взаємодія, що реалізується через систему миттєвих коментарів, автоматичні сповіщення та повноцінні чати. У комплексі реалізація цих вимог дозволить створити цілісне та автономне середовище для ефективного планування, відстеження прогресу та спільної роботи команд.

Для детального опису функціональних можливостей системи «Lesqor» розроблено специфікації сценаріїв використання (Use Cases). Нижче наведено сценарії, що охоплюють ключові процеси: від створення команди (включаючи соло-варіант) до управління ролями та відстеження прогресу. Кожен із представлених сценаріїв покроково розкриває логіку взаємодії акторів із платформою, що слугує надійною основою для безпомилкової програмної реалізації закладених бізнес-процесів.

Таблиця 2.3

Сценарій UC-1: Створення командного або персонального простору

Параметр сценарію	Опис
Назва сценарію	Створення нової команди (Team)
Головний актор	Користувач (User)
Передумова	Користувач авторизований у системі.
Основний потік	1. Користувач переходить на сторінку та команд та натискає на кнопку «Створити команду» 2. Система відображає поп-ап для введення назви, опису та додавання кастомних статусів. 3. Користувач вводить дані. 4. Система створює запис у таблиці Team та призначає користувачу роль Owner у TeamRole.
Результат	Створено ізольований робочий простір з повними правами доступу для автора.

Сценарій UC-1 Описує процедуру ініціалізації робочого середовища (див. табл. 2.3). Сценарій регламентує процес створення ізольованого простору (команди), під час якого автоматично формуються відповідні записи в таблицях бази даних та ініціалізуються права доступу для автора (роль Owner), що закладає основу для подальшої командної роботи.

Таблиця 2.4

Сценарій UC-2: Управління учасниками та ролями

Параметр сценарію	Опис
Назва сценарію	Запрошення учасників та призначення ролей
Головний актор	Власник (Owner)
Передумова	Існує створена команда; Власник знаходиться в її налаштуваннях.
Основний потік	1. Власник вводить email для запрошення. 2. Власник обирає роль (Manager або Executor) зі списку. 3. Система перевіряє наявність користувача в базі та створює зв'язок у TeamRole. 4. Система надсилає сповіщення (Notification) запрошеному учаснику.
Результат	Користувач стає членом команди з визначеними правами доступу.

Сценарій UC-2: Деталізує механізм адміністрування командної структури (див. табл. 2.4). Описує процес залучення нових учасників до робочого простору, верифікацію облікових записів та диференціацію рівнів доступу (Manager, Executor), що забезпечує реалізацію рольової моделі безпеки та прозорість взаємодії всередині команди. Крім того, такий підхід дозволяє власнику оперативно адаптувати ієрархію простору у разі масштабування проєкту або зміни складу учасників

Таблиця 2.5

Сценарій UC-3: Створення та призначення завдання

Параметр сценарію	Опис
Назва сценарію	Формування нового завдання (Task)
Головний актор	Менеджер (Manager), Власник (Owner)
Передумова	Актор має право <code>canAssignTasks</code> у межах обраного проєкту.
Основний потік	1. Актор відкриває форму створення завдання у проєкті. 2. Вводить назву, пріоритет, дедлайн, обирає виконавця (<code>assigneeId</code>), статус, додає якщо потрібно вкладення та посилання, а також може створити підзавдання. 3. Система валідує дані та створює запис у таблиці Task. 4. Виконавець отримує сповіщення про призначення завдання.
Результат	Завдання з'являється у списку завдань проєкту та у профілі виконавця.

Сценарій UC-3: Регламентує порядок постановки задач (див. табл. 2.5). Сценарій охоплює процес ініціалізації об'єкта «Завдання» у межах проєкту, включаючи визначення відповідальних осіб, часових рамок та пріоритетів, а також активацію системи нотифікацій для своєчасного інформування виконавця.

Таблиця 2.6

Сценарій UC-4: Відстеження прогресу та зміна стану

Параметр сценарію	Опис
Назва сценарію	Зміна статусу завдання та моніторинг продуктивності
Головний актор	Будь-який актор з доступом до проєкту
Передумова	Завдання призначене користувачу
Основний потік	1. Користувач змінює стан завдання (напр., з «To Do» на «Done») та прогресу. 2. Система оновлює <code>statusId</code> та <code>progress</code> у таблиці Task. 3. Система перевіряє зміну прогресу. 4. Оновлюється основний відсоток прогресу виконаних завдань по проєкту.
Результат	Актуалізовано стан проєкту; користувач бачить динаміку виконання робіт.

Сценарій UC-4: Визначає алгоритм моніторингу життєвого циклу завдань (див. табл. 2.6). Сценарій описує взаємодію користувача зі статусною та прогресною моделлю системи, автоматичну синхронізацію стану проєкту на основі статусу окремих задач та оновлення аналітичних даних на дашборді користувача в режимі реального часу. Подібна безперервна актуалізація інформації гарантує високу прозорість робочого процесу та сприяє своєчасному прийняттю управлінських рішень.

Сценарій UC-5: Комунікація у межах завдання

Параметр сценарію	Опис
Назва сценарію	Додавання коментарів до завдання
Головний актор	Будь-який актор з доступом до проєкту
Передумова	Користувач відкрив вікно деталей конкретного завдання.
Основний потік	1. Користувач вводить текст у поле коментаря. 2. Система створює запис у таблиці Comment із прив'язкою до taskId та userId. 3. Система оновлює стрічку коментарів у реальному часі через вебсокету або повторний запит.
Результат	Збережено контекст обговорення завдання безпосередньо у системі.

Сценарій UC-5: Окреслює функціонал контекстного обміну інформацією (див. табл. 2.7). Сценарій деталізує процес додавання коментарів, прив'язку обговорень до конкретних ідентифікаторів завдань (taskId) та використання протоколу WebSockets для миттєвого оновлення стрічки комунікацій між учасниками проєкту.

2.2. Аналіз системних та нефункціональних вимог, проєктування бази даних системи

Для гарантування стабільної експлуатації проєктованого веб-рішення на різноманітних апаратних платформах критично важливим є встановлення граничних параметрів системного середовища. Оскільки архітектура «Lesqor» передбачає активне використання ресурсів на стороні клієнта для рендерингу інтерфейсу, обчислювальна потужність пристрою безпосередньо впливає на динаміку взаємодії. Оптимальні та гранично допустимі характеристики обладнання систематизовано (див. табл. 2.8).

Таблиця 2.8

Системні вимоги

Параметр системи	Технічна специфікація
Рекомендовані характеристики:	
Програмне середовище	Актуальні версії Google Chrome, Microsoft Edge або Firefox
Канал зв'язку	Стабільне підключення зі швидкістю від 10 Мбіт/с
Обсяг ОЗП	4 ГБ або більше для багатозадачної роботи

Параметр системи	Технічна специфікація
Мінімально допустимі вимоги:	
Версія оглядача	Браузери на базі Chromium 100+ або Gecko 95+
Пропускна здатність мережі	Від 2 Мбіт/с (за умови низької затримки сигналу)
Обсяг ОЗП	Не менше 4 ГБ

Дотримання вказаних параметрів дозволяє користувачеві уникнути затримок при роботі з графічними компонентами та забезпечує миттєву синхронізацію даних через API.

Важливим аспектом проектування є формалізація нефункціональних вимог, що визначають якісні критерії стійкості та зручності програмного продукту. Ці показники описують експлуатаційні властивості «Lesqor», які безпосередньо впливають на рівень задоволеності кінцевого користувача. Ключові якісні параметри розробленого рішення представлені нижче (див. табл. 2.9).

Таблиця 2.9

Перелік нефункціональних вимог системи «Lesqor»

Категорія	Нефункціональна вимога	Метрика / Критерій приймання
Продуктивність (Performance)	Швидкість відповіді API	Час отримання списку завдань або оновлення статусу через RTK Query не повинен перевищувати 200-300 мс при нормальному навантаженні.
Продуктивність	Плавність UI (Drag-and-Drop)	Перетягування карток завдань між колонками Kanban-дошки має працювати плавно (на рівні 60 FPS) без "підвисань" браузера.
Безпека (Security)	Розмежування доступу (RBAC)	Строгий контроль на рівні API: лише Owner може додавати/змінювати кастомні статуси та команду. Виконавці (Executor) мають доступ лише до зміни статусу своїх завдань.
Безпека	Захист конфіденційних даних	Усі паролі повинні хешуватися (bcrypt) перед записом у базу даних. Авторизація виконується через захищені механізми (наприклад, JWT), а дані очищаються з localStorage при Logout.
Юзабіліті (Usability)	Адаптивна кольорова схема	Інтерфейс має підтримувати зміну тем (Default, Emerald, Gold) із забезпеченням високого контрасту.

Категорія	Нефункціональна вимога	Метрика / Критерій приймання
Юзабіліті	Збереження контексту	Обраний користувачем projectId (в URL або стейті) має зберігатися при навігації між сторінками поточного сеансу.
Підтримуваність (Maintainability)	Стандартизація системи контролю версій	Заголовки комітів у репозиторії суворо обмежуються лімітом у 50 символів для забезпечення чистоти історії змін.
Підтримуваність	Якість коду та документація	Критичний функціонал покривається тестами (Vitest для логіки).
Масштабованість (Scalability)	Рендеринг великих масивів даних	Фронтенд повинен стабільно відмальовувати проекти, що містять 500+ завдань, за необхідності використовуючи механізми віртуалізації списків для Kanban-дошки.
Надійність (Reliability)	Захист від збоїв рендерингу	Усі компоненти відображення статусів або дій повинні мати безпечний дефолтний випадок (default case), який повертає null або резервний UI у разі отримання невідомих даних від API.

Даний перелік регламентує комплекс нефункціональних вимог (НФВ), які визначають атрибути якості, системні обмеження та архітектурні характеристики розроблюваного програмного забезпечення для управління проєктами.

Специфікація вимог структурована за ключовими доменами якості відповідно до сучасних стандартів проєктування інформаційних систем. Вимоги до продуктивності та швидкодії встановлюють кількісні критерії ефективності роботи системи, де обмеження латентності API-запитів на рівні 200-300 мс забезпечує оптимальну синхронність взаємодії клієнтського застосунку із серверною частиною, а вимога щодо підтримки плавності інтерфейсу під час виконання інтенсивних операцій у DOM-дереві зумовлює необхідність застосування алгоритмів оптимізації рендерингу.

Архітектура безпеки системи базується на строгій ізоляції привілеїв та криптографічному захисті, при цьому реалізація моделі управління доступом на основі ролей (RBAC) гарантує, що модифікація критичних сутностей доступна виключно авторизованим суб'єктам, а захист конфіденційних даних реалізується шляхом незворотного хешування паролів та безпечного управління сесіями.

Вимоги до ергономіки та користувацького інтерфейсу передбачають імплементацію адаптивної системи тем оформлення та механізму збереження контексту, що дозволяє системі автоматично відновлювати навігаційний стан і суттєво оптимізує користувацький досвід.

Для забезпечення тривалого життєвого циклу програмного продукту та його підтримуваності регламентуються суворі правила до інфраструктури розробки, зокрема жорсткий ліміт для заголовків комітів у системі контролю версій, стандартизація документування коду та обов'язкове покриття критичної бізнес-логіки автоматизованими тестами.

Водночас проектування клієнтської частини враховує вимоги до масштабованості та відмовостійкості, що передбачає імплементацію патернів віртуалізації списків для роботи з великими масивами даних та інтеграцію механізмів безпечного рендерингу з обов'язковою обробкою дефолтних станів, що виключає можливість виникнення критичних помилок у разі отримання непередбачуваних структур даних від серверної частини.

Для забезпечення стабільного функціонування вебсистеми «Lesqor» було спроектовано реляційну модель даних, що складається з 9 взаємопов'язаних сутностей. Структура бази даних орієнтована на підтримку багатокористувацького доступу, ієрархічності завдань та гнучкого налаштування робочих процесів усередині команд. Логічну структуру бази даних представлено в додатку А.

Нижче наведено специфікації таблиць бази даних системи «Lesqor» на основі спроектованої діаграми.

Таблиця 2.10

Специфікація сутності User (Користувача)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор користувача (PK)
username	String	Логін (унікальне ім'я користувача)
password	String	Хешований пароль
email	String	Електронна пошта користувача
fullName	String	Повне ім'я та прізвище
avatar	String	URL-адреса фотографії профілю

Назва поля	Тип даних	Опис
role	Enum	Системна роль користувача (ADMIN, USER)
resetPasswordCode	String	Код для відновлення доступу
resetPasswordExpires	DateTime	Термін дії коду відновлення
mustChangePassword	Boolean	Прапорець обов'язкової зміни пароля (після OAuth)
createdAt	DateTime	Дата реєстрації в системі

Сутність User є базовою для підсистеми авторизації та керування акаунтами (див. табл. 2.10). Вона зберігає ключові персональні дані користувачів, їхні облікові дані (включно з безпечно хешованим паролем), системні ролі (наприклад, ADMIN чи USER), а також необхідну інформацію для безпечного відновлення доступу до системи.

Таблиця 2.11

Специфікація сутності Team (Команди)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор команди (PK)
name	String	Назва команди (робочого простору)
description	Text	Детальний опис цілей або напрямку команди
creatorId	String (UUID)	Творець команди (FK до User)
useStandardStatuses	Boolean	Прапорець генерації базових статусів за замовчуванням
createdAt	DateTime	Дата створення команди

Наведена сутність Team відповідає за організацію ізольованих робочих просторів у системі (див. табл. 2.11). Вона зберігає загальну метаінформацію про команду, визначає користувача-засновника (creatorId) та містить конфігураційні налаштування, як-от автоматичне використання стандартних статусів для нових проєктів.

Таблиця 2.12

Специфікація сутності TeamRole (Ролі в команді)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор запису ролі (PK)
userId	String (UUID)	Ідентифікатор користувача (FK до User)
teamId	String (UUID)	Ідентифікатор команди (FK до Team)

Назва поля	Тип даних	Опис
role	Enum	Рівень доступу: Owner, Manager, Executor, Member
canAssignTasks	Boolean	Дозвіл на делегування завдань іншим

Сутність TeamRole є зв'язковою (junction table), яка реалізує відношення «багато-до-багатьох» між користувачами та командами (див. табл. 2.12). Вона не лише фіксує членство користувача в робочому просторі, але й деталізує його рівень доступу (від звичайного учасника до власника) та специфічні права, такі як можливість делегувати завдання.

Таблиця 2.13

Специфікація сутності Project (Проекти)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор проекту (PK)
teamId	String (UUID)	Зв'язок із командою (FK до Team)
name	String	Назва проекту
description	String	Детальний опис проекту
createdAt	DateTime	Дата створення проекту

Дана сутність описує проекти, які створюються в межах певної команди (див. табл. 2.13). Вона слугує логічним контейнером для структурного групування завдань і містить базові ідентифікаційні дані: назву, текстовий опис та хронологію створення.

Таблиця 2.14

Специфікація сутності Status (Статуси завдань)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор статусу (PK)
teamId	String (UUID)	Приналежність статусу команді (FK до Team)
name	String	Назва статусу (напр., «В роботі»)
color	String	Код кольору для візуалізації в інтерфейсі
isDefault	Boolean	Чи є статус початковим для нових завдань

Дана сутність забезпечує гнучкість системи Kanban-дошок, дозволяючи командам налаштовувати власні етапи виконання завдань (див. табл. 2.14).

Зберігає назву статусу, кольорове маркування для зручності інтерфейсу та визначає, який статус автоматично присвоюється новим завданням.

Таблиця 2.15

Специфікація сутності Task (Завдання)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор завдання (PK)
projectId	String (UUID)	Зв'язок із проєктом (FK до Project)
creatorId	String (UUID)	Ідентифікатор автора завдання (FK до User)
assigneeId	String (UUID)	Відповідальний виконавець (FK до User)
parentTaskId	String (UUID)	Ідентифікатор батьківського завдання (Self-FK)
statusId	String (UUID)	Поточний стан на дошці (FK до Status)
title	String	Заголовок завдання
description	Text	Детальні інструкції та опис
deadline	DateTime	Кінцевий термін виконання (дедлайн)
priority	Enum	Пріоритетність: Low, Medium, High
progress	Integer	Відсоток виконання (від 0 до 100)
isRead	Boolean	Прапорець перегляду завдання виконавцем
createdAt	DateTime	Дата створення завдання

Сутність Task є центром бізнес-логіки системи управління завданнями (див. табл. 2.15). Зберігає вичерпну інформацію про завдання: прив'язку до проєкту, дані про автора та відповідального виконавця, поточний статус, дедлайни, пріоритет, відсоток виконання прогресу та ієрархічний зв'язок (якщо це підзавдання).

Таблиця 2.16

Специфікація сутності Comment (Коментарі)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор коментаря (PK)
taskId	String (UUID)	Завдання, до якого залишено коментар (FK)
userId	String (UUID)	Автор коментаря (FK до User)
content	String	Текст коментаря
createdAt	DateTime	Дата та час написання

Сутність Comment реалізує функціонал обговорення конкретних завдань (див. табл. 2.16). Вона пов'язує текстові повідомлення із завданнями та користувачами-авторами, фіксуючи точний час публікації для збереження хронології комунікації.

Таблиця 2.17

Специфікація сутності Message (Повідомлення)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор повідомлення (PK)
teamId	String (UUID)	Чат команди (FK до Team)
userId	String (UUID)	Відправник повідомлення (FK до User)
content	Text	Текст повідомлення
attachmentUrl	String	Посилання на медіа/файл у чаті
createdAt	DateTime	Дата відправки повідомлення

Сутність Message призначена для забезпечення внутрішньокмандного чату, поза контекстом окремих завдань (див. табл. 2.17). Вона зберігає текстовий контент повідомлень, прив'язку до конкретної команди та відправника, а також підтримує посилання на прикріплені медіафайли.

Таблиця 2.18

Специфікація сутності Notification (Сповідження)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор сповіщення (PK)
userId	String (UUID)	Ідентифікатор користувача, якому адресовано сповіщення (FK до User)
message	Text	Основний текст повідомлення
type	String	Категорія або тип сповіщення (напр., 'info', 'warning', 'invite'), за замовчуванням 'info'
link	String	Внутрішнє або зовнішнє посилання для переходу при кліку на сповіщення
relatedId	String (UUID)	Ідентифікатор пов'язаної сутності (наприклад, ID команди для відображення кнопок «Прийняти/Відхилити» запрошення)
isRead	Boolean	Прапорець стану сповіщення: прочитано чи ні (за замовчуванням false)

Назва поля	Тип даних	Опис
createdAt	DateTime	Точна дата та час генерації сповіщення

Сутність Notification відповідає за зберігання та управління системою внутрішніх сповіщень для користувачів (див. табл. 2.18). Вона фіксує не лише текстове повідомлення (message), але й розширений контекст дій: за допомогою полів link та relatedId система може формувати інтерактивні сповіщення (наприклад, запрошення до команди, де потрібна реакція користувача для прийняття чи відхилення запиту). Поле isRead дозволяє гнучко керувати станом інтерфейсу, підсвічуючи користувачу лише нові та непереглянуті події.

Таблиця 2.19

Специфікація сутності TaskAttachment (Вкладення)

Назва поля	Тип даних	Опис
id	String (UUID)	Унікальний ідентифікатор вкладення (PK)
taskId	String (UUID)	Зв'язок із завданням (FK до Task)
userId	String (UUID)	Користувач, який завантажив файл (FK до User)
url	String	Пряме посилання на завантажений файл
type	String	Формат вкладення: image, video, link, file
originalName	String	Початкова назва файлу перед завантаженням
createdAt	DateTime	Час завантаження

Специфікація сутності TaskAttachment відповідає за управління файловою системою завдань (див. табл. 2.19). Вона зберігає метадані завантажених файлів (їхній тип, оригінальну назву та пряме посилання на хмарне сховище), міцно пов'язуючи ці матеріали з відповідними завданнями та користувачами, які здійснили завантаження.

Наведені вище таблиці схеми бази даних «Lesqor» відображають структуру ключових сутностей та їх атрибутивний склад, створюючи каркас для зберігання даних про користувачів, команди, проєкти та завдання. На основі цих структурних елементів стає можливим аналіз логічної цілісності

системи, яка забезпечується через систему жорстких реляційних зв'язків, що визначають шлях проходження інформації від користувача до кінцевого завдання. В основі цієї архітектури лежить модель «багато-до-багатьох» (M:N), реалізована через проміжну таблицю TeamRole, що дозволяє одному користувачу бути членом кількох команд, зберігаючи при цьому унікальний рівень прав доступу (Owner, Manager або Executor) у кожній із них, забезпечуючи необхідну ізоляцію даних за збереження гнучкості користувацького досвіду. Логічне розширення цієї моделі вимагає чіткої ієрархічної підпорядкованості, яка реалізована через каскадні зв'язки «один-до-багатьох» (1:N), де команда виступає батьківським контейнером для проєктів (Project), завдань (Task), кастомних статусів (Status) та повідомлень (Message).

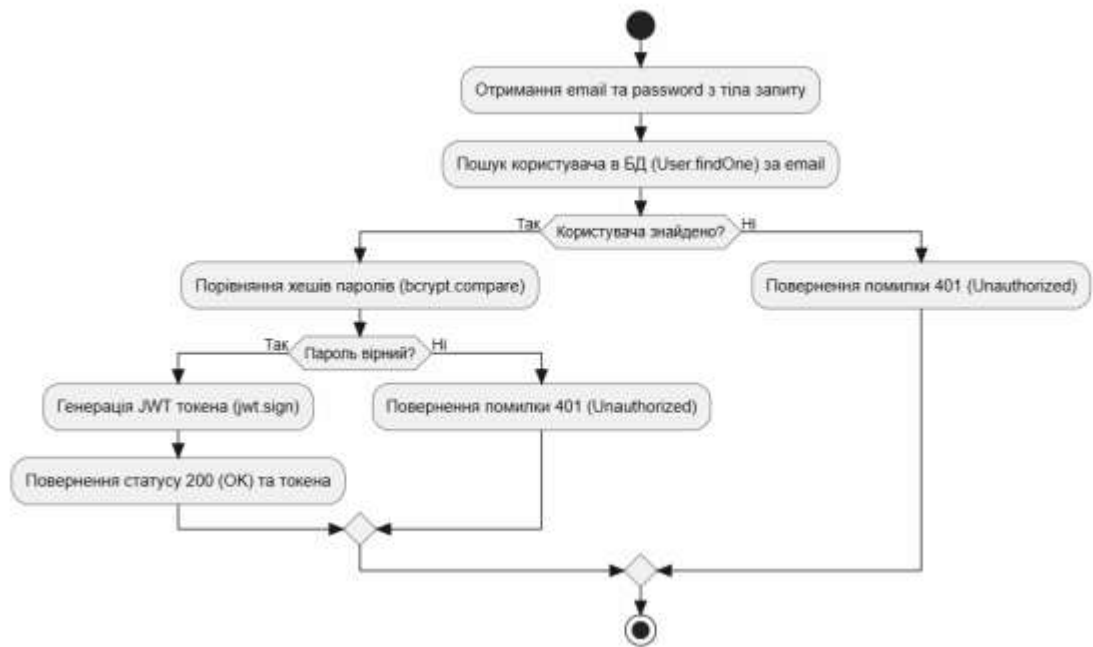
Своєю чергою, проєкти агрегують завдання, формуючи вертикаль управління, де рекурсивний зв'язок (self-reference) у межах таблиці Task через поле parentTaskId дозволяє реалізувати механізм декомпозиції, забезпечуючи можливість розбиття завдань на підзавдання будь-якої глибини вкладеності. Для гарантування стабільності цієї структури всі зв'язки підтримуються механізмами зовнішніх ключів (Foreign Keys) з налаштуванням каскадного видалення (ON DELETE CASCADE), що дозволяє автоматизувати підтримку чистоти даних: при видаленні проєкту система самостійно очищує пов'язані з ним завдання, коментарі та сповіщення, мінімізуючи ризик появи «інформаційного сміття». Завдяки тому, що ідентифікатори виконавця (assigneeId) та автора коментаря (userId) мають прямі зв'язки з таблицею користувачів, система отримує можливість миттєвої ідентифікації відповідальних осіб та динамічної генерації персоналізованих нотифікацій, що разом із гнучкою моделлю доступу та ієрархічною структурою створює стійкий фундамент для надійного функціонування «Lesqor».

2.3. Проектування та реалізація алгоритмів роботи вебсистеми

На етапі проектування програмного забезпечення критично важливим є формалізований опис бізнес-логіки. Для забезпечення стабільності, безпеки та

високої інтерактивності розроблюваної системи управління проєктами було спроектовано низку ключових алгоритмів. Варто зауважити, що у даному підрозділі наведено не вичерпний перелік усіх існуючих алгоритмів додатку. Проте саме ці механізми є фундаментальними, і їхнього опису повною мірою достатньо для розуміння загальної архітектури, руху даних та базової логіки функціонування більшої частини програмного продукту. Графічне представлення логіки їхньої роботи, маршрутизації даних та послідовності прийняття обчислювальних рішень продемонстровано за допомогою діаграм активностей.

Початковою точкою взаємодії користувача із системою є алгоритм автентифікації (Login), який формує базовий рівень безпеки вебзастосунку. Його проєктування зосереджено на надійній ідентифікації клієнта та встановленні stateless-сесії (сесії без збереження стану на сервері), що є сучасним стандартом для побудови масштабованих архітектур. Такий підхід дозволяє суттєво оптимізувати використання серверних ресурсів, оскільки зникає необхідність підтримувати активні сесії в оперативній пам'яті або базі даних. Натомість уся необхідна контекстна інформація про користувача надійно інкапсулюється та підписується безпосередньо в самому токени. Завдяки цьому архітектура стає гнучкою, а кожен наступний запит від клієнта обробляється як повністю самодостатній та незалежний процес. Відповідна діаграма детально демонструє послідовність обчислювальних процесів: ініціалізація запиту через отримання облікових даних, пошук відповідного запису в базі та проведення криптографічної перевірки хешу пароля. У разі успішної валідації сервер генерує унікальний криптографічно підписаний JWT-токен, який передається на клієнтську сторону для безперешкодної авторизації всіх наступних дій користувача. Або ж, у випадку невідповідності введених даних, алгоритм передбачає миттєве блокування доступу з поверненням клієнту відповідного HTTP-статусу помилки, що ефективно захищає систему від спроб несанкціонованого проникнення (див. рис. 2.2).



Після успішного входу та встановлення захищеної сесії, системі необхідно розмежувати права доступу до внутрішнього функціоналу. За цей етап відповідає алгоритм рольового розмежування доступу (RBAC Middleware), який спроектовано для забезпечення контекстної безпеки на рівні окремих робочих команд. На діаграмі проілюстровано процес перехоплення запиту, вилучення ідентифікаторів з контексту, перевірку наявності необхідних прав у базі даних та прийняття кінцевого рішення щодо дозволу на виконання операції (див. рис. 2.3).

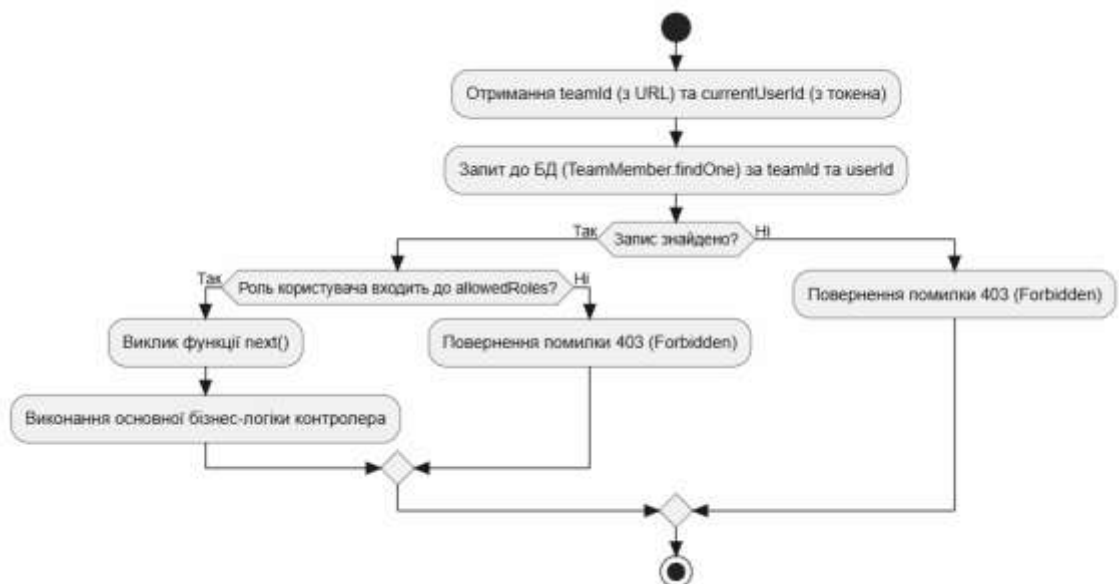


Рис. 2.3 – Діаграма активностей «алгоритм автентифікації (Login)»

Отримавши відповідні права авторизації, користувач переходить до безпосереднього управління проектами, де однією з найчастіших операцій є інтерактивна зміна станів завдань. Для цього розроблено алгоритм обробки Drag-and-Drop переміщення, який описує багатокроковий механізм взаємодії з Kanban-дошкою. Діаграма активностей деталізує етапи валідації кінцевої зони скидання, оптимістичного оновлення клієнтського інтерфейсу, асинхронної відправки мутації на сервер та ініціалізації логіки відкату (rollback) у разі виникнення помилки (див. рис. 2.4).

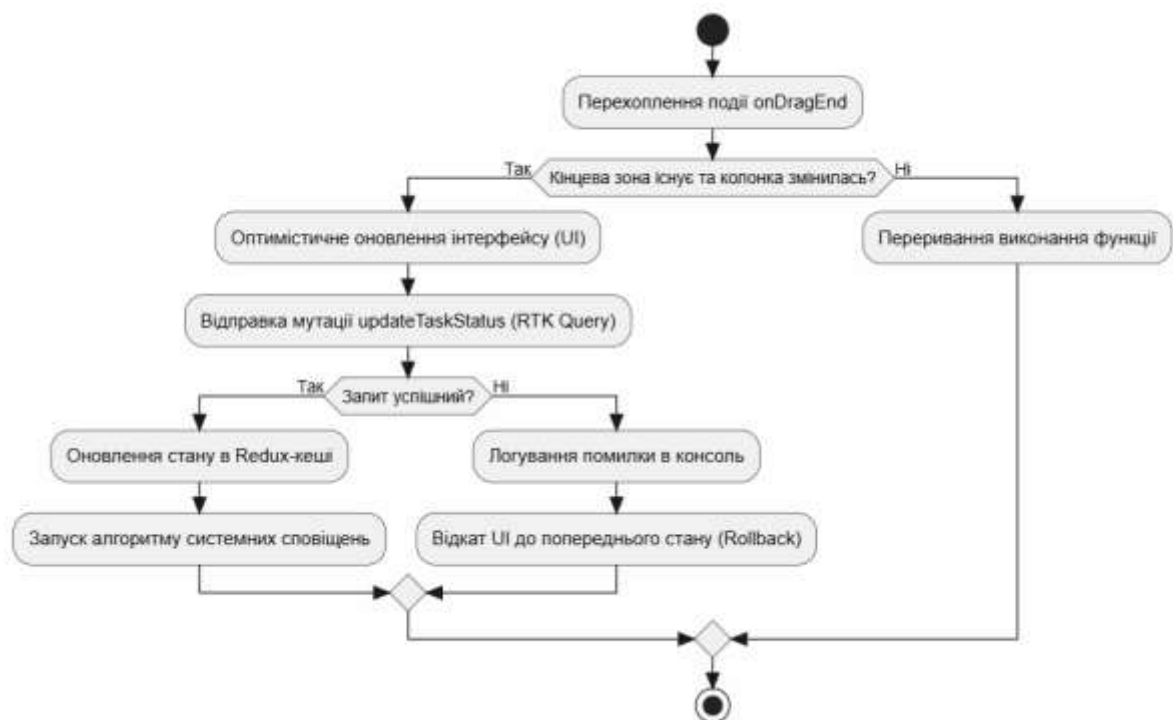


Рис. 2.4 – Діаграма активностей «алгоритм автентифікації (Login)»

Успішне виконання подібних транзакцій на клієнтській частині застосунку не обмежується лише локальним оновленням інтерфейсу - воно ініціює каскад асинхронних серверних процесів. Ключовим серед них є алгоритм маршрутизації цільових системних сповіщень, який вирішує завдання своєчасного інформування всіх зацікавлених учасників проекту про внесені зміни, гарантуючи безперервну синхронізацію командної роботи.

На наведеній діаграмі детально відображено логічну послідовність виконання цього алгоритму. Першим етапом є динамічний пошук цільової аудиторії: система аналізує зв'язки в базі даних для визначення кола осіб з

відповідними правами та ролями у робочому просторі (зокрема, керівників команди або призначених виконавців). Наступним, критично важливим кроком є обов'язкова контекстна фільтрація згенерованого списку користувачів. Цей механізм програмно виключає самого ініціатора події з переліку отримувачів, що дозволяє уникнути відправки надлишкових (спам) сповіщень про дії, які користувач щойно виконав самостійно.

Завершальним етапом роботи алгоритму є формування оптимізованого масиву об'єктів та виконання операції транзакційного масового запису даних (bulk insert) у відповідну таблицю (див. рис. 2.5). Використання пакетного збереження замість циклічних поштучних запитів дозволяє мінімізувати кількість звернень до бази даних, суттєво знизити навантаження на сервер та прискорити загальний час відгуку системи.

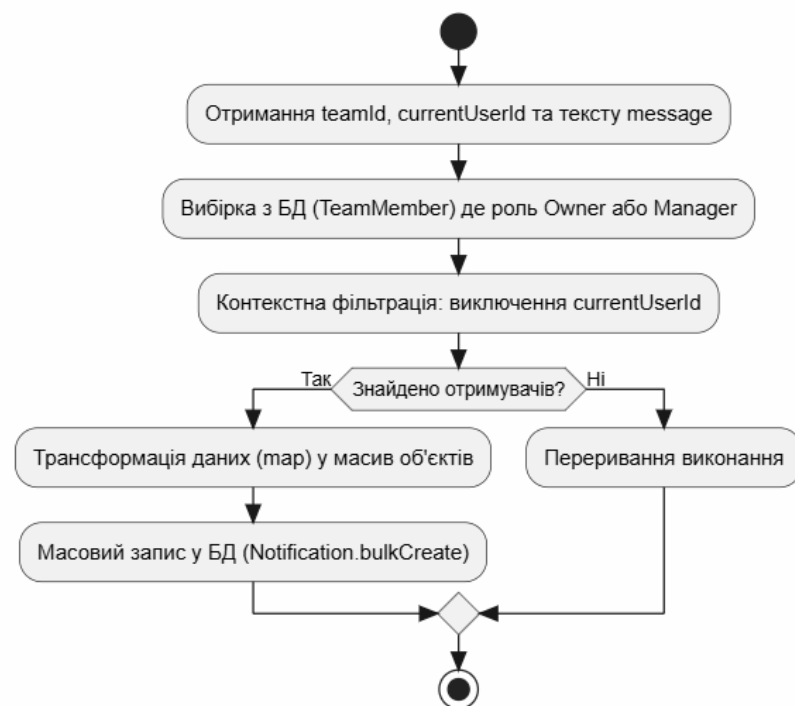


Рис. 2.5 – Діаграма активностей «алгоритм автентифікації (Login)»

Зрештою, паралельно з обробкою даних та сповіщень, система повинна забезпечувати ергономіку роботи з великими обсягами інформації. За цю персоналізацію користувацького досвіду відповідає алгоритм просторового сортування з персистенцією стану (Хронологічний стек). Діаграма ілюструє процес зчитування історії взаємодії з локального сховища браузера, оновлення масиву пріоритетів при виборі конкретного проєкту (переміщення елемента на

початок стеку) та реактивне пересортування відображеного списку (див. рис. 2.6). Такий підхід дозволяє інтуїтивно оптимізувати навігацію, оскільки найактуальніші для виконавця проєкти завжди автоматично закріплюються у верхній частині панелі. Крім того, перенесення цієї логіки на сторону клієнта гарантує миттєву адаптацію інтерфейсу без створення додаткового навантаження на серверну частину бази даних. У результаті, робочий простір динамічно підлаштовується під індивідуальні патерни поведінки користувача, зберігаючи високу стабільність системи незалежно від загальної кількості створених сутностей у команді.

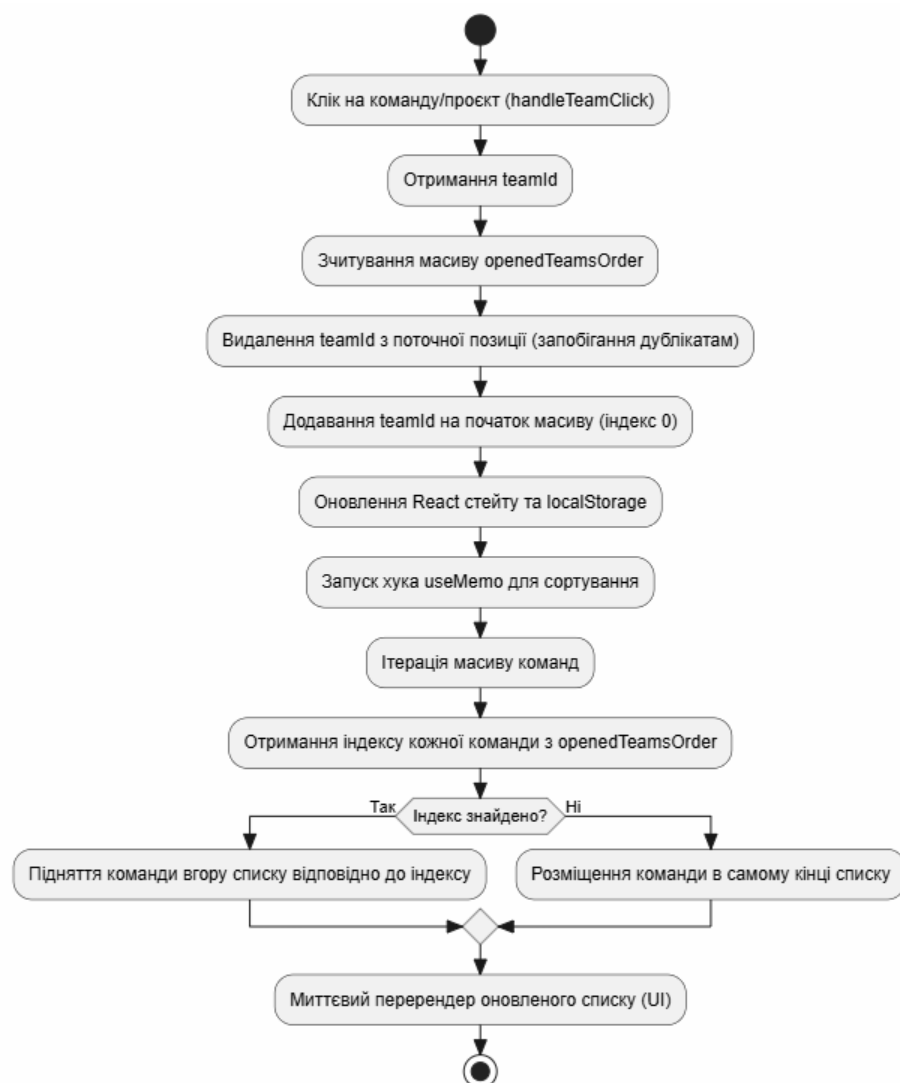


Рис. 2.6 – Діаграма активностей «алгоритм автентифікації (Login)»

На етапі реалізації бізнес-логіки системи «Lesqor» ключовим завданням є розробка алгоритмів, які забезпечують безперебійну взаємодію між клієнтською частиною, сервером додатків та реляційною базою даних.

Побудова алгоритмічного забезпечення базується на подієво-орієнтованій моделі Node.js, що дозволяє обробляти запити в асинхронному режимі, мінімізуючи час очікування відповіді для кінцевого користувача. Оскільки система підтримує як командну роботу, так і персональне планування, алгоритми розраховані на динамічну перевірку прав доступу при кожній операції.

Основою безпеки та розмежування доступу є алгоритм автентифікації на базі JSON Web Tokens (JWT). Нижче наведено лістинг коду, за допомогою якого генерується токен:

```
const token = jwt.sign(  
  { id: newUser.id, email: newUser.email, role: newUser.role },  
  JWT_SECRET,  
  { expiresIn: '24h' }  
);
```

Процес автентифікації та реєстрації в системі реалізовано за принципом відсутності стану (stateless) з використанням стандарту JSON Web Token (JWT). Після базової ідентифікації користувача в базі даних та криптографічної верифікації його пароля за допомогою бібліотеки bcrypt, алгоритм переходить до свого найважливішого етапу - генерації токена доступу. Генерація токена виконується методом jwt.sign, який формує захищений рядок даних. У корисне навантаження (payload) цього токена інкапсулюються ключові ідентифікаційні атрибути: унікальний ідентифікатор користувача, його електронна пошта та системна роль. Наявність цих даних у самому токені дозволяє бекенду при кожному наступному запиті миттєво розпізнавати клієнта та визначати його рівень доступу, повністю уникаючи повторних дорогих звернень до бази даних.

Криптографічне підписання токена здійснюється за допомогою унікального секретного ключа сервера, який ізольовано у змінних середовища. Цей алгоритм підпису гарантує абсолютну цілісність переданих даних: будь-яка спроба клієнта чи зловмисника змінити вміст токена (наприклад, самовільно підвищити собі права доступу) автоматично зробить цифровий підпис невалідним, і сервер відхилить такий запит. Додатково, для мінімізації

безпекових ризиків, алгоритм генерації встановлює жорсткий термін життя (TTL) токена, обмежуючи його валідність 24 годинами, після чого користувач має пройти автентифікацію повторно.

Після успішного підтвердження автентичності системою вступає в дію алгоритм рольового розмежування доступу (Role-Based Access Control) на рівні команд. Цей алгоритм відповідає за безпеку виконання транзакцій, таких як створення кастомних статусів або управління вкладеннями. Він працює як система перевірок (middleware) на сервері: при надходженні запиту система визначає ідентифікатор користувача, ідентифікує контекст (ідентифікатор команди або проєкту) та перевіряє поточну роль користувача (Owner, Manager, Executor) у цій конкретній команді. Рішення про виконання операції або її блокування приймається на основі матриці прав. Приклад лістингу наведено нижче:

```
export const requireTeamRole = (allowedRoles: string[]) => {
  return async (req: Request, res: Response, next: NextFunction) => {
    try {
      const { teamId } = req.params;
      const currentUserId = req.user.id;

      // Ідентифікація користувача в контексті команди
      const member = await TeamMember.findOne({
        where: { teamId, userId: currentUserId }
      });

      // Блокування доступу, якщо роль не відповідає вимогам
      if (!member || !allowedRoles.includes(member.role)) {
        res.status(403).json({ message: 'Forbidden: Insufficient role
permissions' });
        return;
      }
      next();
    } catch (error) {
      res.status(500).json({ message: 'Authorization error' });
    }
  };
};

// Приклад використання в роутері:
```

```
// router.post('/:teamId/statuses', requireTeamRole(['Owner']),
createCustomStatus);
```

Представлений фрагмент коду реалізує механізм проміжного програмного забезпечення (middleware) для фреймворку Express.js, який забезпечує контекстне рольове розмежування доступу на рівні конкретних робочих команд. Функція вищого порядку приймає масив дозволених ролей та повертає асинхронний обробник для захисту маршрутів API. Під час обробки запиту алгоритм вилучає ідентифікатор команди з URL-параметрів та ідентифікатор ініціатора, після чого звертається до бази даних через модель TeamMember для перевірки статусу користувача в ізолюваному середовищі цього проєкту. Якщо користувач не належить до команди або його роль не входить до масиву дозволених, система примусово блокує запит і повертає статус 403 Forbidden, унеможливаючи несанкціоновані транзакції. У разі успішної авторизаційної валідації викликається функція next(), яка передає керування контролеру для виконання цільової бізнес-логіки. Для забезпечення відмовостійкості весь алгоритм загорнуто в блок обробки винятків try-catch, який у випадку перебоїв із базою даних коректно повертає статус 500 Internal Server Error без аварійної зупинки серверного процесу.

Окрім механізмів безпеки та стабільності серверної частини, ефективна робота системи управління проєктами вимагає складної логіки на рівні користувацького інтерфейсу, ключовим компонентом якого є алгоритм динамічного формування Kanban-дошки та обробки переміщень. На відміну від статичних таблиць, цей алгоритм генерує структуру дошки на основі конфігурації. Спочатку він агрегує набір кастомних та стандартних статусів команди, формуючи з них динамічні колонки, а потім виконує розподіл масиву завдань відповідно до їхніх статусів. Друга частина алгоритму - це обробка подій перетягування (Drag-and-Drop): перехоплення зміни позиції, оптимістичне оновлення інтерфейсу на клієнті та ініціалізація запиту на оновлення статусу в базі даних із гарантованою обробкою дефолтних станів для запобігання помилкам рендерингу.

Приклад лістингу наведено нижче:

```
const onDragEnd = async (result: DropResult) => {
  const { destination, source, draggableId } = result;
  if (!destination || destination.droppableId === source.droppableId) return;

  // Відправка мутації на сервер через RTK Query
  try {
    await updateTaskStatus({
      id: draggableId,
      statusId: destination.droppableId
    }).unwrap();
  } catch (error) {
    console.error('Failed to update status', error);
    // Тут викликається логіка відкату UI у разі помилки (rollback)
  }
};

// Компонент статусу із гарантованим уникненням помилок рендерингу
const StatusIndicator = ({ status }: { status: string }) => {
  switch (status) {
    case 'new': return <span className="badge">Нове</span>;
    case 'in_progress': return <span className="badge">В роботі</span>;
    case 'done': return <span className="badge">Виконано</span>;
    case 'sent': return <span className="badge">Відправлено</span>;
    default: return null; // Дефолтний випадок для запобігання падінню UI
  }
};
```

Представлений фрагмент коду демонструє програмну реалізацію клієнтської логіки управління завданнями, що складається з двох ключових механізмів: обробника подій переміщення (Drag-and-Drop) та компонента безпечного візуального рендерингу. Асинхронна функція `onDragEnd` відповідає за перехоплення результатів перетягування картки завдання на Kanban-дошці. На початку виконання алгоритм здійснює валідацію кінцевої зони (`destination`): якщо картку скинуто поза межами допустимої області або її позиція не змінилася відносно початкової, функція негайно переривається для уникнення зайвих обчислень та мережевих запитів. У разі зміни колонки ініціюється асинхронна відправка мутації `updateTaskStatus` на сервер за допомогою інструментарію `RTK Query`, куди передається ідентифікатор

завдання та його новий статус. Для забезпечення відмовостійкості клієнтської частини цей процес обгорнуто у блок `try...catch`, що дозволяє перехоплювати мережеві помилки та ініціювати відкат користувацького інтерфейсу (rollback) до попереднього стану, зберігаючи консистентність даних на екрані.

Друга частина коду, функціональний компонент `StatusIndicator`, відповідає за відображення локалізованих індикаторів статусу завдання. Він використовує конструкцію `switch` для зіставлення внутрішніх системних ідентифікаторів (включно зі специфічним статусом `sent`) з відповідними візуальними елементами інтерфейсу. Критично важливою архітектурною особливістю цього компонента є імплементація безпечного дефолтного випадку (`default: return null`), який діє як програмний запобіжник: при отриманні невідомого, застарілого або некоректного значення статусу від API компонент просто нічого не відмальовує, тим самим гарантовано уникаючи виникнення критичних помилок рендерингу (unhandled runtime errors) та повного "падіння" вебдодатку.

Успішне виконання подібних транзакцій на клієнті (наприклад, переміщення картки завдання або зміна її стану) не обмежується лише локальним оновленням інтерфейсу, а ініціює подальші системні комунікаційні процеси, за які відповідає алгоритм генерації та маршрутизації цільових системних сповіщень. Цей процес запускається як реакція на зміну сутності (наприклад, переведення завдання в інший статус виконавцем). Алгоритм аналізує подію, визначає ідентифікатор команди, після чого виконує вибірку користувачів, які повинні отримати сповіщення (`Owner` та `Manager`). Важливим етапом є контекстна фільтрація вибірки, під час якої ініціатор події (поточний користувач) виключається з переліку отримувачів, після чого генерується масовий запис нових повідомлень у базу даних. Приклад лістингу наведено нижче:

```
async function notifyTeamManagers(teamId: string, currentUserId: string, message:
string) {
    // Вибірка керівників, виключаючи поточного користувача зі списку
    const targets = await TeamMember.findAll({
        where: {
```

```

        teamId,
        role: { [Op.in]: ['Owner', 'Manager'] },
        userId: { [Op.ne]: currentUserId }
    }
});

if (targets.length === 0) return;

// Формування масиву об'єктів для масового запису
const notifications = targets.map(target => ({
    userId: target.userId,
    message,
    isRead: false
}));

// Оптимізований масовий запис (bulk insert) в БД
await Notification.bulkCreate(notifications); }

```

Представлений фрагмент коду демонструє програмну реалізацію асинхронного алгоритму маршрутизації цільових системних сповіщень на стороні сервера. Головним завданням функції `notifyTeamManagers` є автоматичне інформування керівного складу команди про важливі події, ініційовані іншими користувачами. Алгоритм розпочинається із формування запиту до бази даних через ORM-модель `TeamMember` з використанням спеціальних операторів бази даних. Запит виконує комплексну вибірку цільової аудиторії: він шукає учасників конкретної команди (`teamId`), які мають управлінські привілеї (ролі `Owner` або `Manager`, що перевіряється оператором входження `Op.in`). При цьому критично важливим кроком є контекстна фільтрація, яка за допомогою оператора нерівності (`Op.ne`) примусово виключає з вибірки самого ініціатора дії (`currentUserId`), щоб запобігти надсиланню користувачеві сповіщень про його ж власні дії. Після завершення пошуку алгоритм перевіряє отриманий масив на порожнечу і, у разі відсутності валідних отримувачів, негайно перериває виконання для економії обчислювальних ресурсів сервера. Якщо ж адресатів знайдено, система за допомогою методу `map` трансформує отримані дані у новий масив об'єктів сповіщень, де кожному користувачеві призначається текст повідомлення та дефолтний статус `isRead: false`. Завершальним етапом роботи

алгоритму є використання методу `bulkCreate` моделі `Notification`, який замість виконання безлічі повільних поодиноких `INSERT`-запитів у циклі, здійснює оптимізований масовий запис усіх згенерованих об'єктів у базу даних за одну транзакцію, що мінімізує навантаження на базу даних та суттєво підвищує загальну продуктивність вебсистеми.

Поряд з ефективною обробкою даних на сервері, критично важливим для загальної швидкодії та зручності користувацького інтерфейсу є механізм управління великими масивами інформації безпосередньо на стороні клієнта, основою якого є алгоритм багатокритеріальної фільтрації та просторового сортування даних із персистенцією стану. Цей алгоритм описує логіку роботи інтерфейсу на сторінках завдань та команд. Він включає послідовне застосування фільтрів (за статусом, за виконавцем) та функцій сортування (за датою дедлайну, в алфавітному порядку або за принципом хронологічного стеку). Ключовою особливістю алгоритму є механізм персистенції: перед рендерингом система зчитує попередньо збережений стан із локального сховища (наприклад, `localStorage`) або параметрів `URL`, синхронізує його з реактивним станом додатку і лише потім відображає відсортований масив, забезпечуючи збереження користувацького контексту при навігації. Приклад лістингу наведено нижче:

```
const [openedTeamsOrder, setOpenedTeamsOrder] = useState<string[]>(() => {
  // Зчитування стану при першому завантаженні
  const saved = localStorage.getItem('openedTeamsOrder');
  return saved ? JSON.parse(saved) : [];
});

const handleTeamClick = (teamId: string) => {
  setOpenedTeamsOrder(prevOrder => {
    // Видалення дублікатів та додавання на початок масиву (індекс 0)
    const newOrder = [teamId, ...prevOrder.filter(id => id !== teamId)];
    localStorage.setItem('openedTeamsOrder', JSON.stringify(newOrder));
    return newOrder;
  });
};

const sortedTeams = useMemo(() => {
```

```

return [...teams].sort((a, b) => {
    const indexA = openedTeamsOrder.indexOf(a.id);
    const indexB = openedTeamsOrder.indexOf(b.id);

    // Команди, які не відкривались, залишаються в кінці
    if (indexA === -1 && indexB === -1) return 0;
    if (indexA === -1) return 1;
    if (indexB === -1) return -1;

    return indexA - indexB; // Сортування за індексом у стеку
});
}, [teams, openedTeamsOrder]);

```

Представлений фрагмент коду демонструє програмну реалізацію алгоритму просторового сортування даних із персистенцією стану, який вибудовує елементи (команди) за принципом хронологічного стеку. На початковому етапі алгоритм ініціалізує реактивний стан за допомогою хука `useState`, використовуючи механізм відкладеної (лінивої) ініціалізації: система зчитує попередньо збережену послідовність ідентифікаторів команд з локального сховища браузера (`localStorage`) та парсить її лише під час першого монтування компонента, що дозволяє уникнути зайвих ресурсоємних операцій при наступних рендерах.

Взаємодія користувача з інтерфейсом обробляється функцією `handleTeamClick`, яка при виборі команди оновлює масив історії: цільовий ідентифікатор спочатку фільтрується з його поточної позиції для уникнення дублікатів, а потім примусово розміщується на самому початку масиву (в індекс 0), після чого цей оновлений стан миттєво синхронізується як з `React`-стейтом, так і з пам'яттю браузера для збереження користувацького контексту між сесіями.

Безпосереднє сортування списку виконується за допомогою хука `useMemo`, який забезпечує кешування результатів обчислень.

Усередині нього створюється копія оригінального масиву команд, яка сортується на основі пошуку індексів кожного елемента у збереженому масиві історії: команди з меншим індексом (тобто ті, що були відкриті найпізніше) підіймаються у верхню частину списку, тоді як команди, з якими користувач

ще жодного разу не взаємодіяв (їхній індекс дорівнює -1), автоматично відправляються в самий кінець видачі. Наявність масиву залежностей у хуку гарантує, що це сортування реактивно переобчислюється та миттєво оновлює візуальний інтерфейс виключно при зміні вхідних даних або історії відкриттів, без необхідності повного перезавантаження сторінки. В результаті користувач отримує персоналізований та адаптивний інтерфейс, який автоматично підлаштовується під його робочі патерни, забезпечуючи швидкий доступ до найактуальніших просторів.

2.4. Реалізація функціоналу вебсистеми управління завданнями для командної роботи

Розробка програмного продукту розпочалася з ініціалізації серверної частини на базі платформи Node.js. В якості ORM-системи (Object-Relational Mapping) для взаємодії з реляційною базою даних PostgreSQL було обрано Sequelize. Створено відповідні моделі даних, контролери для обробки бізнес-логіки та маршрутизатори (роутери) для API. Підключення до бази даних реалізовано з використанням змінних оточення (див. рис. 2.7) з метою дотримання стандартів безпеки та приховування чутливої інформації.

```
PORT=
DB_NAME=
DB_USER=
DB_PASSWORD=
DB_HOST=
DB_DIALECT=
NODE_ENV=
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
```

Рис. 2.7 – Приклад файлу .env

Для забезпечення автоматичної синхронізації створених моделей з таблицями бази даних під час запуску сервера застосовано параметр `alter: true`. Це дозволяє оновлювати схему бази даних відповідно до поточного коду без втрати вже існуючих записів. Код ініціалізації сервера має наступний вигляд:

```
const start = async () => {
  try {
    await sequelize.sync({ alter: true });
    console.log('Database connected and synchronized');

    setupSwagger(app);
    console.log('Swagger docs available at http://localhost:' + PORT + '/api-docs');

    httpServer.listen(PORT, () => {
      console.log(`Server started on http://localhost:${PORT}`);
    });
  } catch (err: any) {
    console.error('Error starting server:', err.message);
  }
}
```

Після успішної генерації таблиць (див. рис. 2.8), працездатність та коректність відповідей створених ендпоінтів перевірялася за допомогою інструменту тестування API Postman.

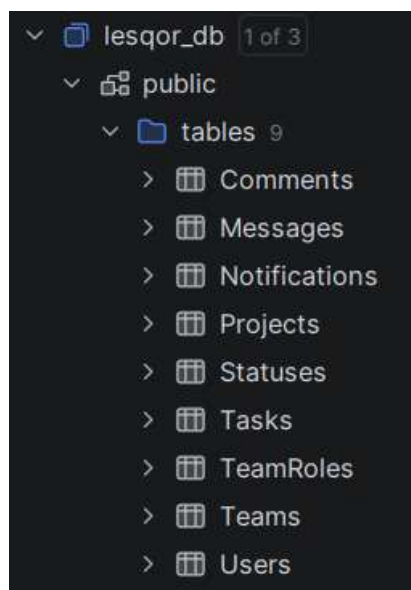


Рис. 2.8 – Створені таблиці у БД

Для автоматизації тестування та зручності подальшої підтримки клієнт-серверної взаємодії було згенеровано документацію за стандартом Swagger (див. рис. 2.9). Автоматично сформована специфікація містить вичерпний опис усіх доступних маршрутів, необхідних параметрів авторизації та HTTP-статусів відповідей. Впровадження цього стандарту значно знижує поріг входження для нових розробників і пришвидшує процес розширення функціоналу системи.



Рис. 2.9 – Відображення api-docs

Для оптимізації процесу розробки та тестування було налаштовано механізм сідерів (Seeders). Це дозволяє автоматично наповнювати локальну базу даних початковими та тестовими даними після її очищення, значно економлячи час розробника (див. рис. 2.10).

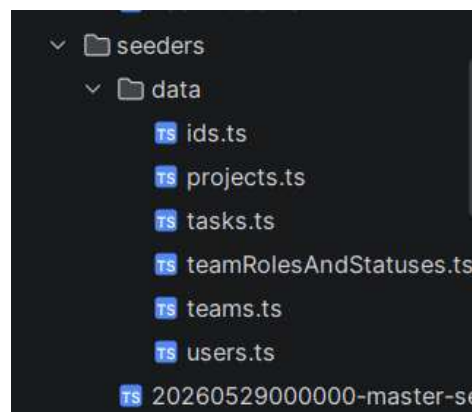


Рис. 2.10 – Відображення структури файлів сідерів

Однією з ключових архітектурних особливостей системи є імплементація обміну даними в реальному часі за допомогою бібліотеки Socket.io. Цей механізм автоматично активується під час створення нового командного простору, а також забезпечує роботу приватних чатів між двома користувачами. Наведений нижче код демонструє ініціалізацію WebSockets,

що створює двосторонній канал зв'язку для миттєвого обміну повідомленнями:

```
export const initSocket = (httpServer: HttpServer) => {
  io = new Server(httpServer, {
    cors: {
      origin: "*",
      methods: ["GET", "POST"]
    }
  });

  io.on('connection', (socket) => {
    console.log('User connected:', socket.id);

    socket.on('joinRoom', (teamId) => {
      socket.join(teamId);
      console.log(`Socket ${socket.id} joined team room: ${teamId}`);
    });

    socket.on('joinUser', (userId) => {
      socket.join(userId);
      console.log(`Socket ${socket.id} joined user room: ${userId}`);
    });

    socket.on('disconnect', () => {
      console.log('User disconnected:', socket.id);
    });
  });

  return io;
};
```

Важливим етапом життєвого циклу розробки стало налаштування процесу безперервної інтеграції та розгортання (CI/CD) [26]. Для ізоляції середовищ виконання використано платформу контейнеризації Docker. У системі контролю версій було налаштовано Runner (див. рис. 2.11) та успішно підключено до віддаленого репозиторію (див. рис. 2.12). Таке рішення дозволило повною мірою реалізувати конвеєр безперервної інтеграції та доставки (CI/CD), автоматизувавши процеси збірки і подальшого розгортання застосунку на робочому сервері.

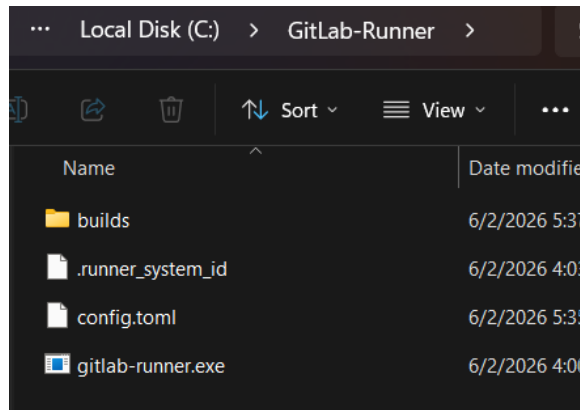


Рис. 2.11 – Вигляд розміщення ранера

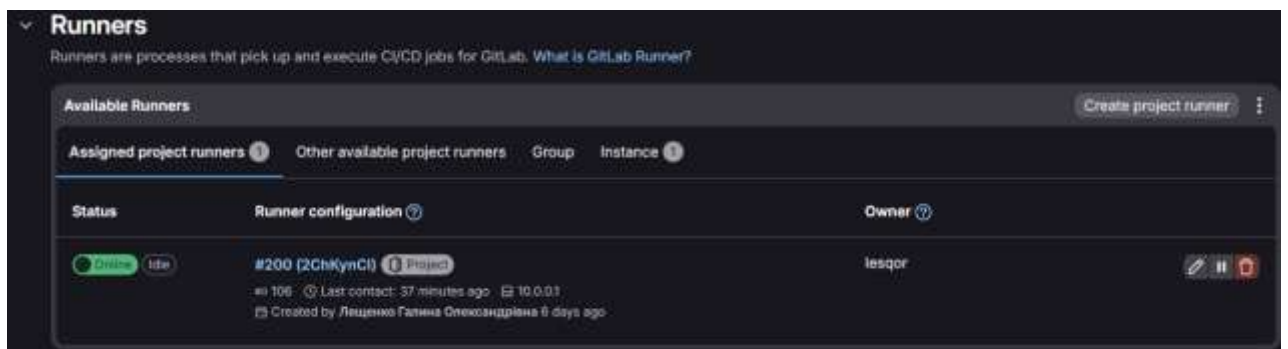


Рис. 2.12 – Відображення ранера в репозиторії

Для обох частин проекту розроблено конфігураційні файли Dockerfile. Докерфайл для бекенду використовує оптимізовану багатоетапну збірку (multi-stage build), що суттєво зменшує розмір кінцевого образу:

```
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci
COPY . .
RUN npm run build
FROM node:20-alpine
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
COPY --from=builder /app/dist ./dist
USER node
EXPOSE 3000
CMD ["node", "dist/index.js"]
```

З метою виключення непотрібних файлів із процесу збірки додано файл .dockerignore (див. рис. 2.13).

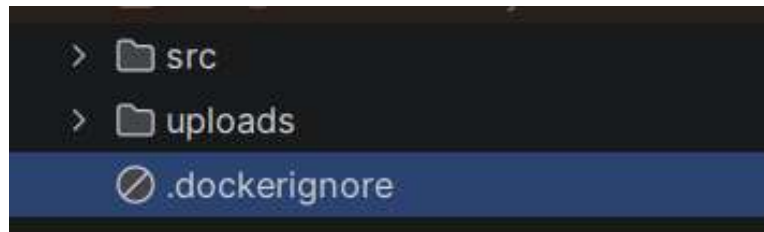


Рис. 2.13 – Файл .dockerignore

Для автоматизації деплою на орендований віртуальний сервер створено конфігураційний файл .gitlab-ci.yml. Для безпечного зберігання чутливих даних (IP сервера, SSH ключі) налаштовано секретні змінні середовища (див. рис. 2.14).

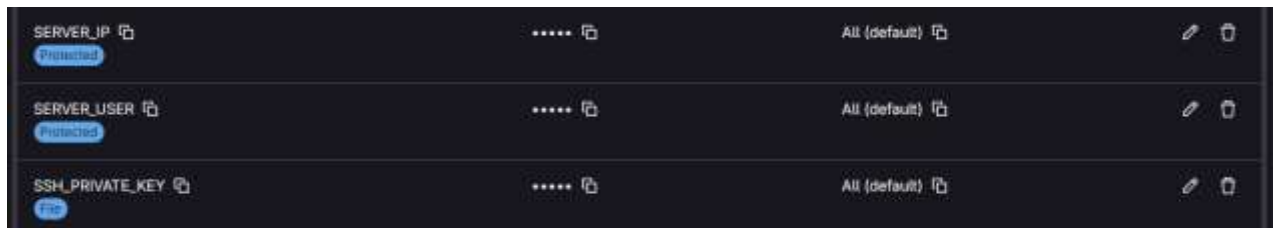


Рис. 2.14 – Відображення чутливих даних

Після завершення налаштувань pipeline пройшов успішно, і програмний продукт був автоматично розгорнутий на робочому сервері (див. рис. 2.15).

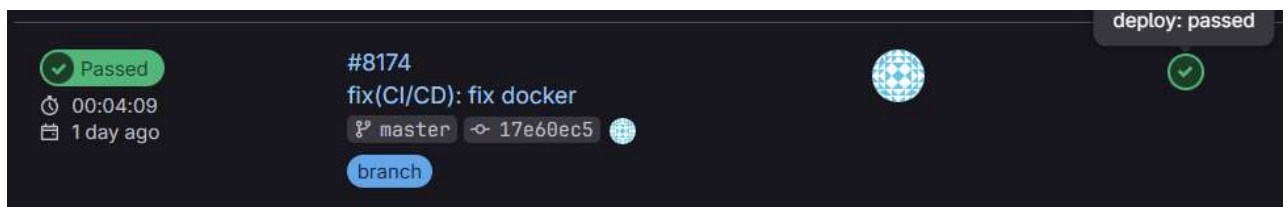


Рис. 2.15 – Зображення деплою

Наступний крок передбачав розширення користувацького досвіду (UX). Було імплементовано механізм перемикання кольорових тем інтерфейсу, що підвищує загальну доступність вебсистеми:

```
const themes: { id: Theme; name: string; primary: string; secondary: string }[] = [
  { id: 'default', name: 'Смарагдова', primary: '#004d39', secondary: '#008060' },
  { id: 'teal', name: 'Морська', primary: '#334d4d', secondary: '#85adad' },
  { id: 'gold', name: 'Золота', primary: '#664400', secondary: '#ffe6b3' },
  { id: 'purple', name: 'Фіолетова', primary: '#400080', secondary: '#e6ccff' },
];
```

Для гнучкого управління робочими процесами реалізовано можливість створення кастомних статусів завдань. Візуалізація робочого процесу побудована у вигляді Kanban-дошки, що дозволяє інтуїтивно переміщувати

завдання між колонками. Також інтегровано розширену систему фільтрації даних для зручного пошуку необхідної інформації.

На фінальному етапі розробки було виконано прив'язку зареєстрованого доменного імені до IP-адреси сервера шляхом налаштування DNS-записів. (див. рис. 2.16)



Рис. 2.16 – Відображення default record group

Наявність домену з підтримкою безпечного протоколу дозволила інтегрувати систему авторизації OAuth 2.0 через акаунти Google. Відповідні ідентифікатори клієнта були створені в панелі Google Cloud Console (див. рис. 2.17). Такий підхід позбавляє систему необхідності зберігати та обробляти додаткові паролі, ефективно знижуючи ризики компрометації конфіденційних даних користувачів.

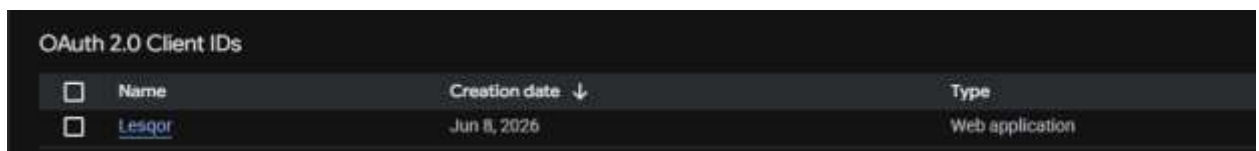


Рис. 2.17 – Відображення ідентифікаторів клієнта (Google Cloud Console)

Наостанок було розроблено підсистему безпеки для скидання пароля через email, впроваджено інтернаціоналізацію (зміна мови інтерфейсу на англійську) та проведено комплексне тестування, за результатами якого було виправлено виявлені програмні дефекти, які успішно виправлені.

Висновки до другого розділу

У результаті виконання другого розділу здійснено комплексне проєктування та програмну реалізацію вебсистеми «Lesqor». Визначено ключових акторів, розроблено діаграму варіантів використання та спроектовано реляційну базу даних PostgreSQL із дев'яти сутностей, що підтримує рольову модель доступу (RBAC) та рекурсивну декомпозицію

завдань. Реалізовано фундаментальні алгоритми бізнес-логіки: stateless-автентифікацію через JWT, динамічне розмежування доступу, просторове сортування та обробку Drag-and-Drop операцій. Серверну частину розроблено на базі Node.js і Sequelize з інтеграцією WebSockets для обміну даними в реальному часі, а клієнтську - доповнено адаптивними темами та безпечним рендерингом інтерактивної Kanban-дошки. Налаштовано автоматизований CI/CD конвеєр через GitLab Runner і Docker для розгортання на віддаленому сервері. Окрім цього, прив'язано домен, налаштовано Swagger-документацію, відновлення паролів через email та авторизацію Google OAuth, що дозволило створити безпечний програмний продукт і успішно вивести його в робоче середовище.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБСИСТЕМОЮ УПРАВЛІННЯ ЗАВДАННЯМИ ДЛЯ КОМАНДНОЇ РОБОТИ

3.1. Порядок встановлення та налаштування параметрів вебсистеми

Порядок встановлення та налаштування вебсистеми управління завданнями «Lesqor» базується на сучасних методологіях DevOps та принципах інженерії програмного забезпечення. Процес передбачає послідовне розгортання серверної інфраструктури, конфігурацію ізольованого середовища виконання та ініціалізацію сховищ даних. Базовим середовищем розгортання виступає віртуальний виділений сервер (VPS) під управлінням операційної системи сімейства Linux, що гарантує високу продуктивність, розширені можливості системного адміністрування та стійкість до мережевих навантажень.

Фундаментальним архітектурним рішенням на цьому етапі є впровадження платформи контейнеризації Docker спільно з оркестратором Docker Compose. Це забезпечує надійну ізоляцію програмних компонентів, зокрема клієнтської частини, серверного API та бази даних, повністю нівелює проблему конфліктів версій системних залежностей та гарантує абсолютну ідентичність середовищ локальної розробки та робочого сервера.

У ролі єдиної точки входу налаштовується високопродуктивний вебсервер Nginx. Він виконує функції зворотного проксі-сервера, забезпечуючи безпечну маршрутизацію HTTP-запитів та постійних WebSocket-з'єднань до відповідних ізольованих контейнерів. Гнучке керування параметрами системи реалізується через механізм змінних оточення (файли .env), де зберігаються критичні облікові параметри підключення до БД, криптографічні секрети JWT, налаштування SMTP-серверів та ідентифікатори Google OAuth 2.0.

Для комплексного розуміння фізичної архітектури та взаємодії всіх описаних інфраструктурних вузлів було побудовано відповідну візуальну модель. На діаграмі розгортання (Deployment Diagram) детально

проілюстровано топологію мережі, розташування контейнерів на сервері та потоки даних між внутрішніми і зовнішніми компонентами системи.

Як видно з наведеної діаграми, центральним вузлом інфраструктури є віртуальний виділений сервер (VPS) із доменним ім'ям lesqor.cfd. Усі вхідні клієнтські запити від браузера, де виконується SPA-застосунок на базі React/Vite, проходять через вебсервер Nginx по захищених протоколах HTTPS та WSS. Nginx виконує подвійну функцію: він віддає завантажені статичні файли клієнтського застосунку та працює як зворотний проксі, перенаправляючи динамічні запити до серверної частини.

Серверна бізнес-логіка повністю ізольована всередині середовища Docker. Вона складається з двох основних контейнерів: Backend-додатка на базі Node.js API та бази даних PostgreSQL, які комунікують між собою по внутрішньому порту 5432. Така конфігурація мережі надійно захищає базу даних від прямого доступу з глобальної мережі.

Окремим блоком на діаграмі виділено зовнішні сервіси (Third-party Services), з якими взаємодіє система. Клієнтська частина безпосередньо звертається до провайдера Google OAuth для забезпечення процесу безпечної авторизації, тоді як Backend-контейнер асинхронно комунікує із зовнішнім SMTP-сервером для відправки транзакційних електронних листів користувачам.

Процес автоматизації та оновлення системи візуалізовано через компонент GitLab CI/CD. Під час спрацьовування конвеєра, GitLab генерує нові артефакти: директорію зі статичними файлами фронтенду (dist) та зібрані Docker-образи. Далі відбувається автоматизований деплой цих артефактів на VPS-сервер, після чого ініціюється оновлення та перезапуск відповідних контейнерів, що замикає цикл безперервного розгортання програмного забезпечення.

Для наочного пояснення фізичної архітектури, схеми розподілу апаратних ресурсів, розташування виконуваних артефактів та протоколів

комунікації між вузлами мережі розроблено відповідну діаграму розгортання (Deployment Diagram) у стандарті UML (див. рис. 3.1).

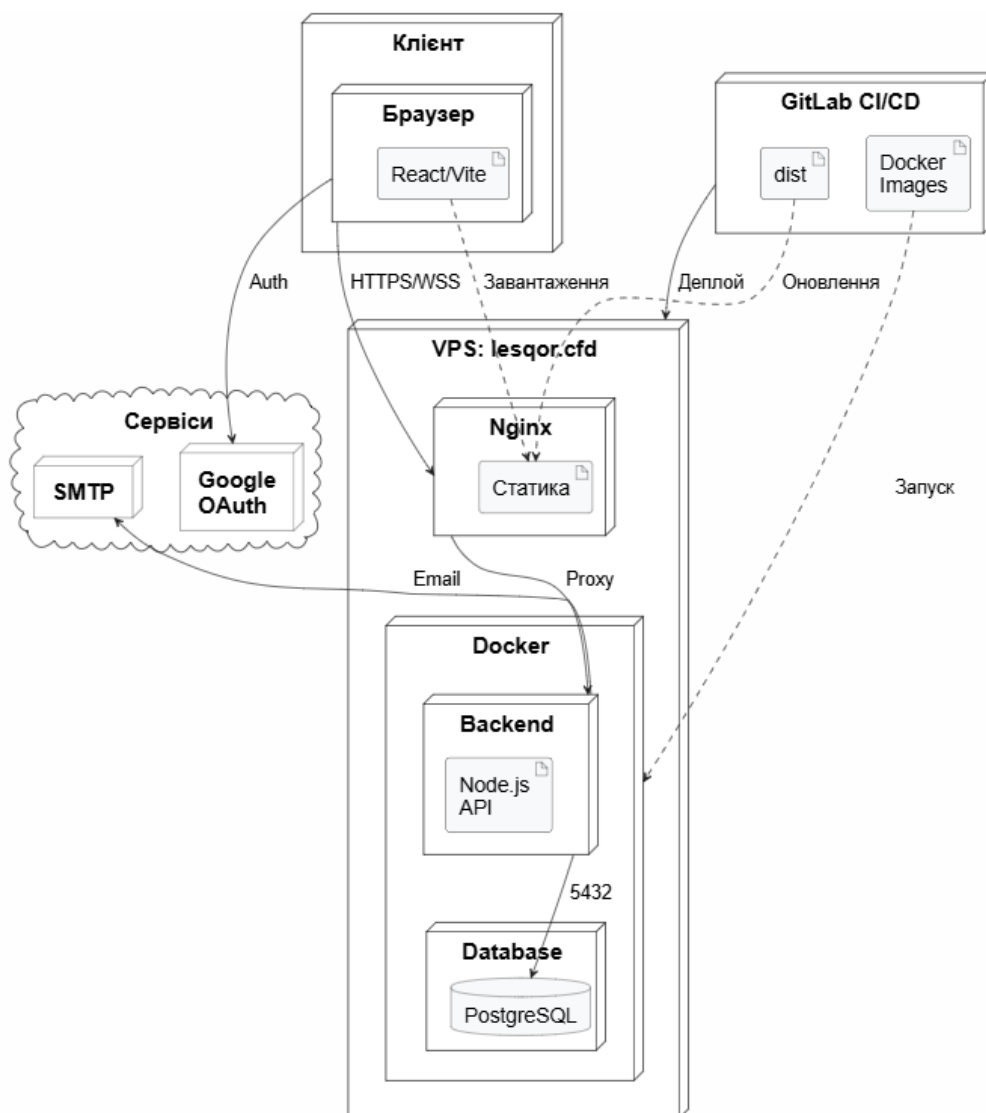


Рис. 3.1 – Діаграма розгортання

Схема ресурсів системи, що візуалізується за допомогою діаграми розгортання, демонструє конфігурацію програмних та апаратних засобів з точки зору зберігання і обробки даних [27]. Фізичний рівень інфраструктури представлений клієнтськими пристроями з веббраузерами та центральним сервером, на якому функціонує вебсервер Nginx у ролі зворотного проксі для балансування та маршрутизації мережових запитів [28]. Основні потужності для обробки та зберігання інформації локалізовані у середовищі Docker, де функціонують два ключові незалежні контейнери: контейнер серверної частини на базі Node.js, що відповідає за бізнес-логіку, та контейнер бази

даних PostgreSQL, який забезпечує постійне зберігання інформації. Додатково схема ресурсів охоплює зовнішні інтеграційні вузли, зокрема хмарні сервіси Google Identity, поштовий сервер для транзакційних розсилок та репозиторій артефактів, що формує цілісну екосистему проєкту. Для комплексного розуміння динаміки взаємодії компонентів розглядається схема роботи програмної системи, яка відображає процес управління операціями та потоками даних. Ініціатором інформаційного потоку виступає користувач, який через графічний інтерфейс клієнтського додатка формує авторизовані запити. Цей вхідний потік даних перехоплюється проксі-сервером і направляється до маршрутизаторів серверної частини, де система управління операціями ініціює перевірку прав доступу на основі рольової моделі. У разі успішної валідації потік даних передається до контролерів бізнес-логіки, які через систему об'єктно-реляційного відображення формують SQL-запити до сховища даних. Після виконання операцій читання або запису зворотний потік інформації трансформується у стандартизовану відповідь або генерує події в реальному часі через протокол WebSockets, що забезпечує миттєве оновлення стану інтерфейсів в інших учасників команди та гарантує синхронність спільної роботи.

3.2. Структура інтерфейсу та порядок використання вебсистеми управління завданнями для командної роботи

Розроблена вебсистема має повністю адаптивний користувацький інтерфейс, що гарантує коректне відображення та стабільну роботу на різних типах пристроїв (десктопних, планшетних та мобільних). Взаємодія з програмним продуктом розпочинається з головної сторінки (Landing Page), яка доступна за основним доменним іменем `lesqor.cfd` у веббраузері. На цій сторінці презентовано ключові функціональні можливості системи, а також розміщено елементи навігації для переходу до модулів авторизації та створення нового облікового запису (див. рис. 3.2).

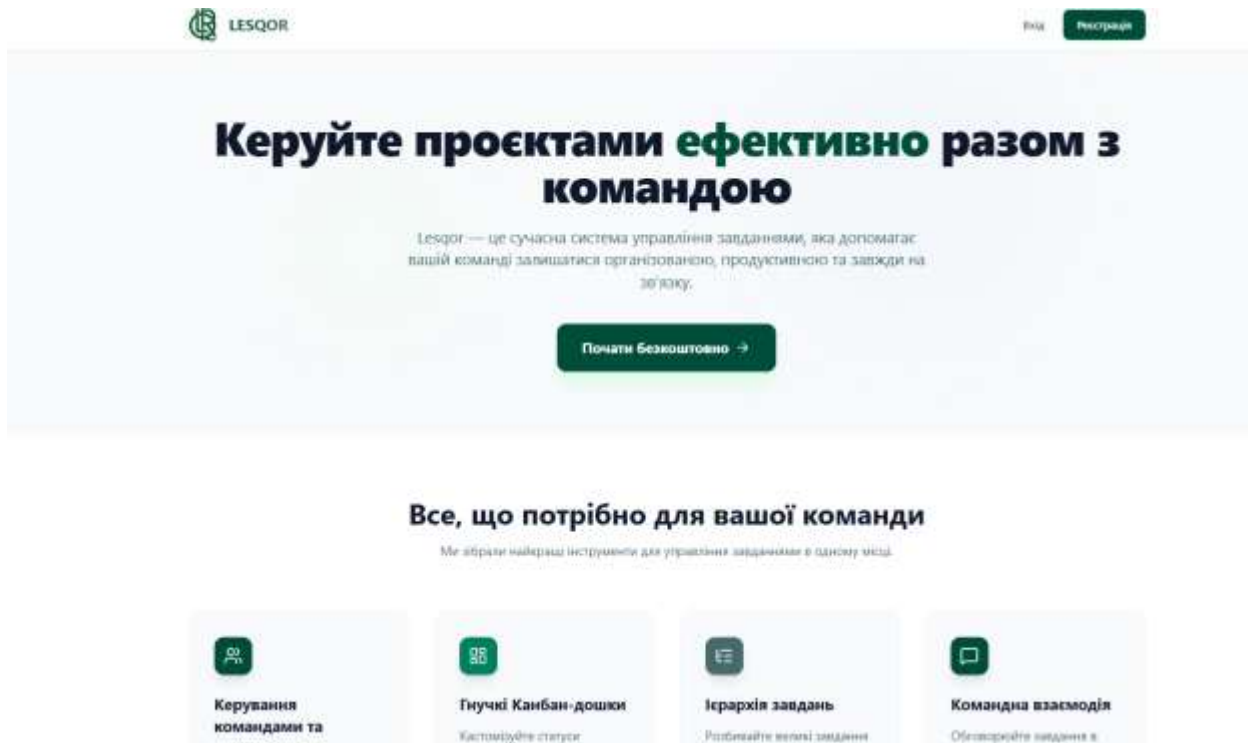


Рис. 3.2 – Головна сторінка вебсистеми

Процес ідентифікації та аутентифікації користувачів реалізовано у двох варіантах для забезпечення гнучкості користувацького досвіду. Клієнтам доступна як класична форма реєстрації за допомогою електронної пошти, так і інтегрована система швидкого входу (Single Sign-On) за протоколом OAuth 2.0 через обліковий запис Google (див. рис. 3.3-3.4).

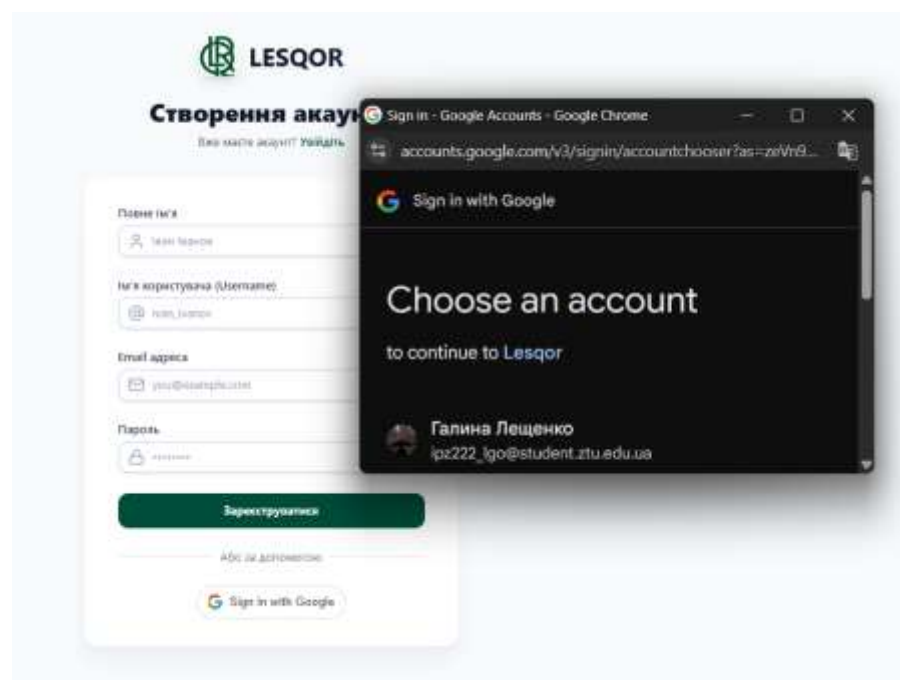


Рис. 3.3 – Реєстрація через Google OAuth

Рис. 3.4 – Стандартна форма реєстрації

З метою дотримання стандартів безпеки, при первинній реєстрації через сервіси Google система автоматично генерує тимчасовий пароль. Одразу після успішного входу користувачу відображається обов'язкове модальне вікно (Pop-up) для встановлення власного постійного пароля. Інтерфейс спроектовано таким чином, що закрити це вікно і продовжити роботу в системі без оновлення облікових даних неможливо (див. рис. 3.5).

Рис. 3.5 – Модальне вікно обов'язкового встановлення пароля

Завершивши процес авторизації, користувач отримує доступ до особистого кабінету. У модулі профілю реалізовано функціонал управління персональними даними (див. рис. 3.6): завантаження чи зміна аватара, редагування імені, а також персоналізація інтерфейсу шляхом вибору однієї із запропонованих кольорових тем оформлення (наприклад, "Смарагдова", "Морська" тощо).

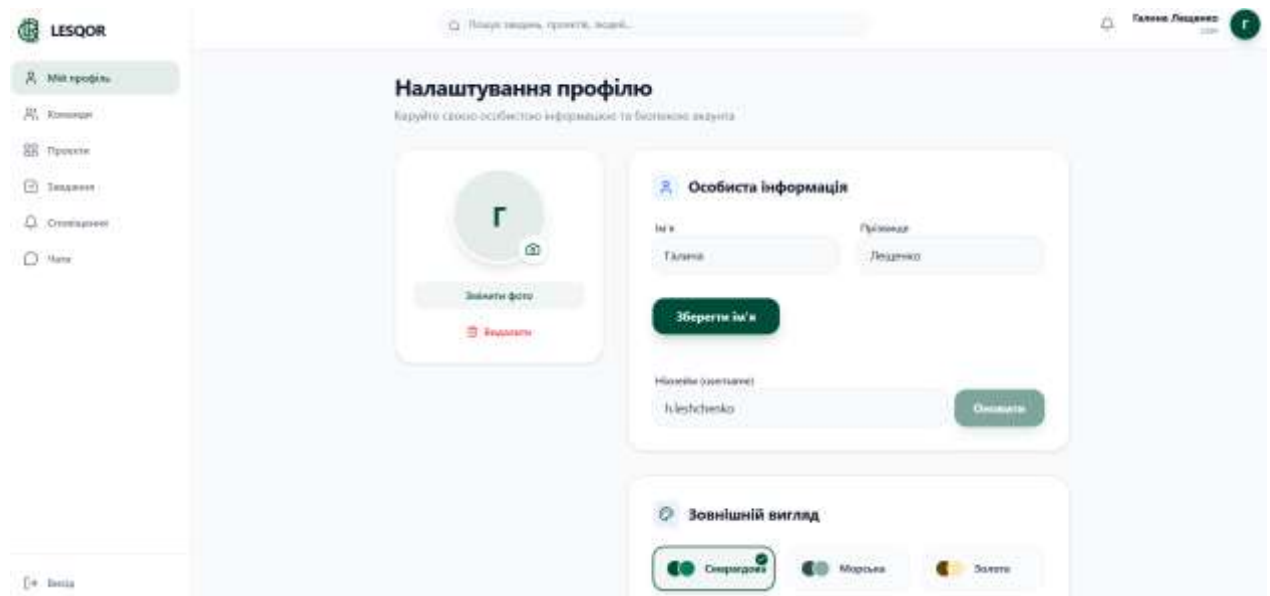


Рис. 3.6 – Сторінка профілю користувача

Основний робочий процес розпочинається у розділі управління командами. Користувач із роллю «Власник» має можливість ініціювати створення нового ізольованого робочого простору. Під час його формування передбачено не лише введення базової інформації, але й гнучке налаштування користувацьких статусів для завдань. Власник може визначити унікальні назви та кольорові ідентифікатори для кожного етапу, а також за потреби повністю вилучити стандартні статуси. У такому разі учасники команди взаємодіятимуть виключно зі створеним кастомним набором, переміщуючи картки завдань між колонками на інтерактивній Kanban-дошці для оновлення їхнього стану. Також є можливість редагувати вже наявні статуси. Це дозволяє максимально гнучко адаптувати життєвий цикл завдань під специфіку та методологію роботи конкретної команди (див. рис. 3.7-3.8).

Рис. 3.7 – Модальне вікно для створення команди

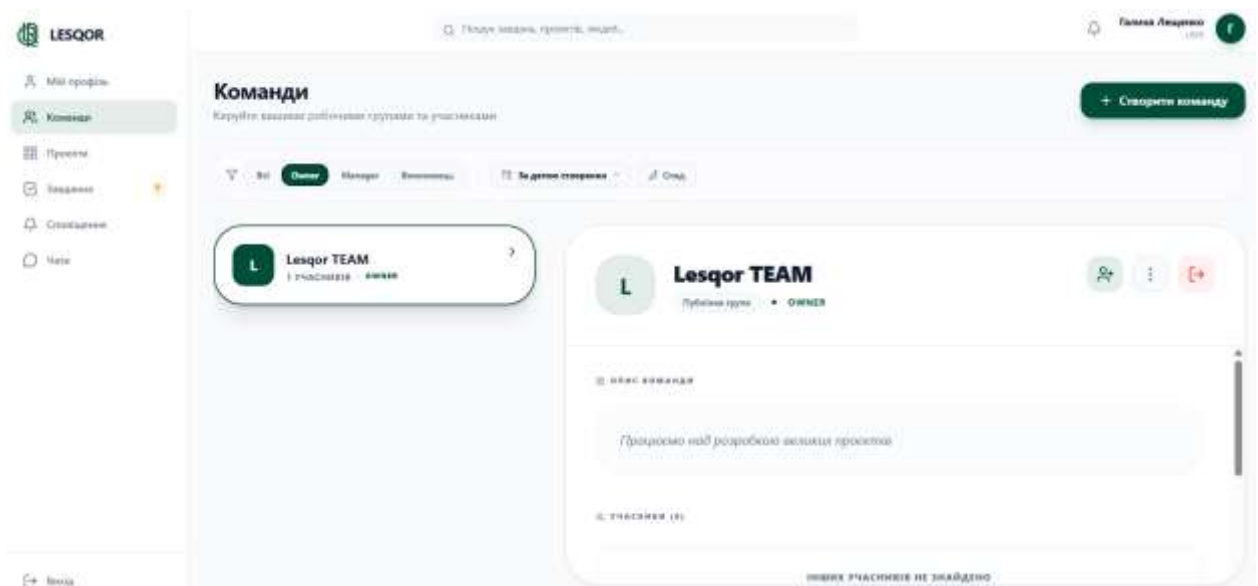


Рис. 3.8 – Відображення створеної команди на сторінці

Наступним кроком є формування складу команди. Власник простору може надсилати запрошення іншим учасникам (які вже мають обліковий запис у системі), призначаючи їм відповідні ролі згідно з ієрархічною моделлю доступу (RBAC). Наприклад, надання ролі «Менеджер» делегує учаснику розширені права, зокрема можливість генерувати нові завдання для інших членів команди (див. рис. 3.9-3.10).

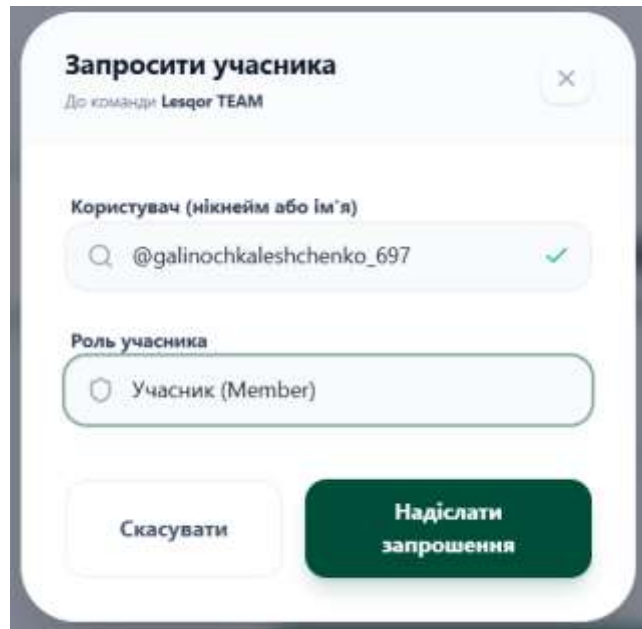


Рис. 3.9 – Модальне вікно для додавання користувача

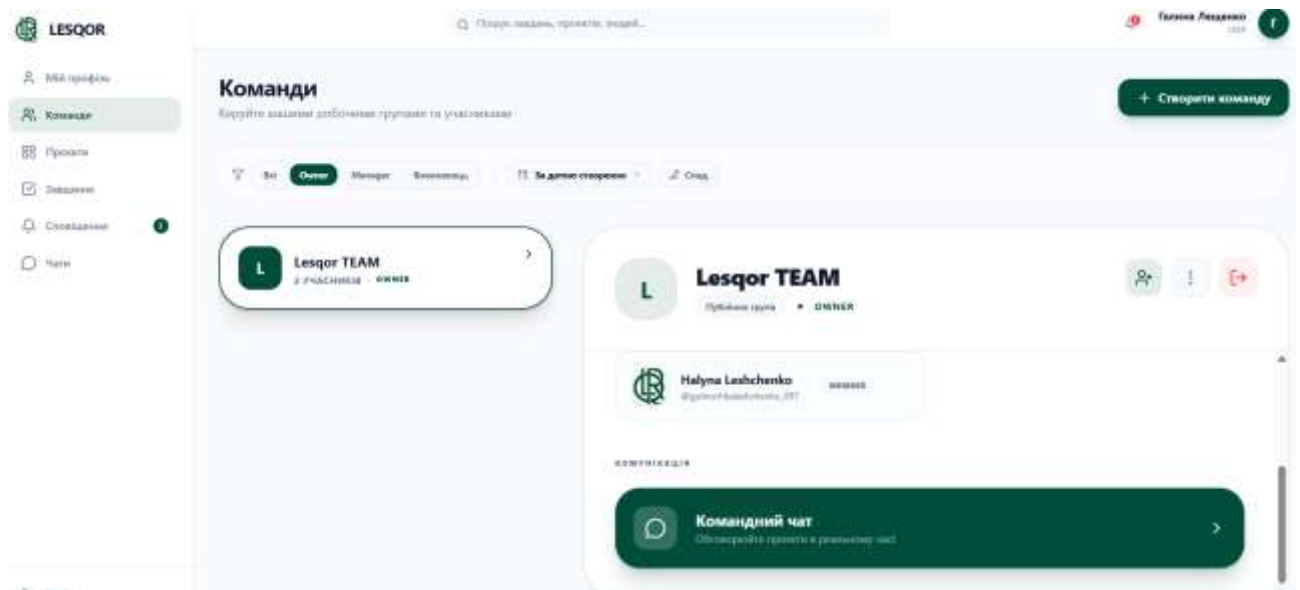


Рис. 3.10 – Відображення списку учасників команди

Для ефективного управління великою кількістю командних просторів та їх учасників, у системі передбачено ергономічні інструменти багатокритеріальної фільтрації та просторового сортування даних (див. рис. 3.11).

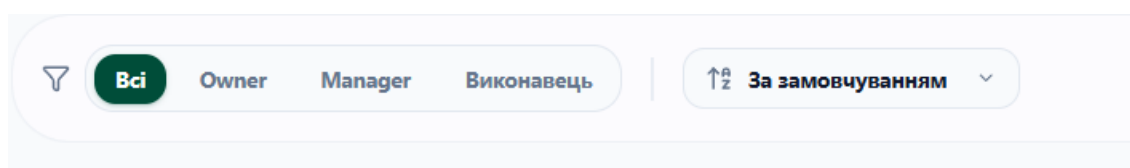


Рис. 3.11 – Панель інструментів фільтрації та сортування

У межах створеної команди ініціюється розробка проєктів. Використовуючи відповідний функціонал на сторінці команд, менеджер може створити новий проєкт, який виступатиме логічним контейнером для групування майбутніх завдань (див. рис. 3.12).

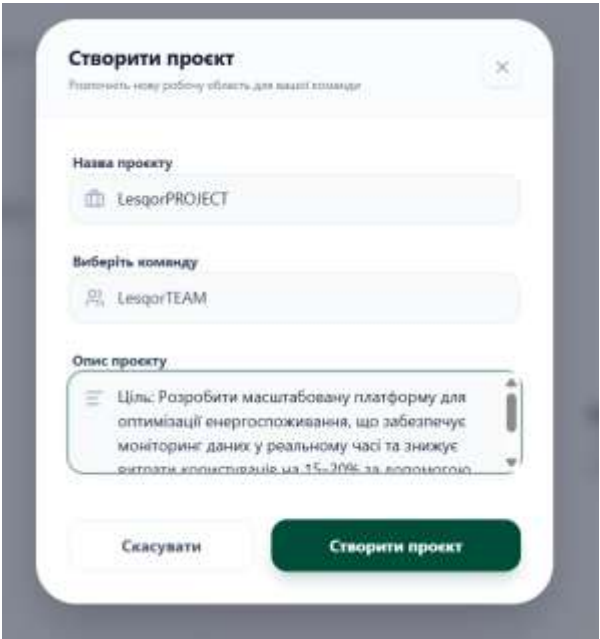


Рис. 3.12 – Модальне вікно для створення проєкту

Архітектура інтерфейсу дозволяє оперативно створювати нові задачі для учасників безпосередньо зі сторінки переліку проєктів, оптимізуючи процес планування (див. рис. 3.13-3.15).

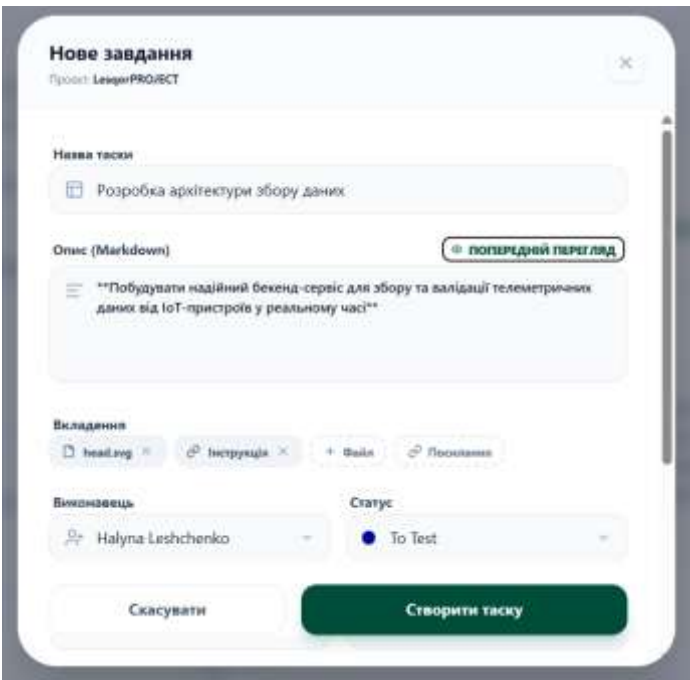


Рис. 3.13 – Модальне вікно швидкого створення завдання

Нове завдання

Проект: LesqorPROJECT

Вкладення

head.svg

Інструкція

+ Файл

Посилання

Виконавець

Halyna Leshchenko

Статус

To Test

Пріоритет

Середній

Дедлайн

06/10/2026

Підзавдання (1)

+ Додати підзавдання

Скиньте звітність

Halyna Leshchenko

Скасувати

Створити задачу

Рис. 3.14 – Модальне вікно швидкого створення завдання (продовження)

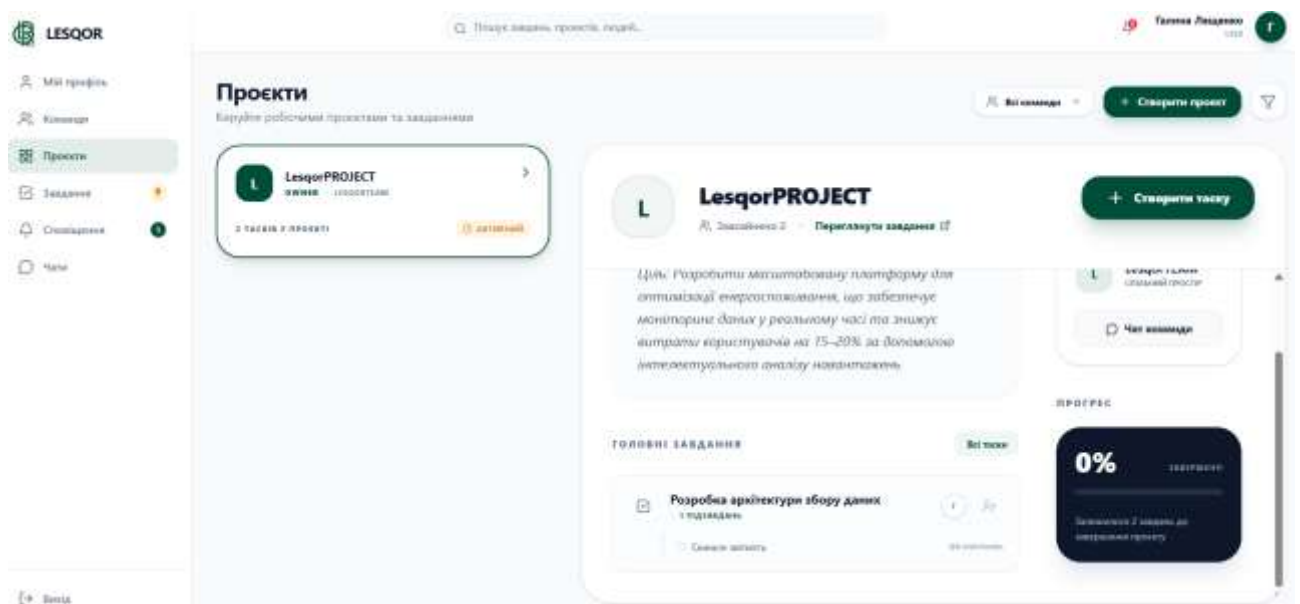


Рис. 3.15 – Сторінка огляду списку проєктів

Під час створення завдання, ініціатор призначає відповідального виконавця (assignee) з числа учасників команди. Щойно завдання делеговано, обраний користувач миттєво отримує відповідне системне сповіщення про нове завдання. Модуль сповіщень також акумулює інформацію про інші ключові події: долучення нових учасників або запрошення до інших команд, а також створення нових проєктів команди (див. рис. 3.16).

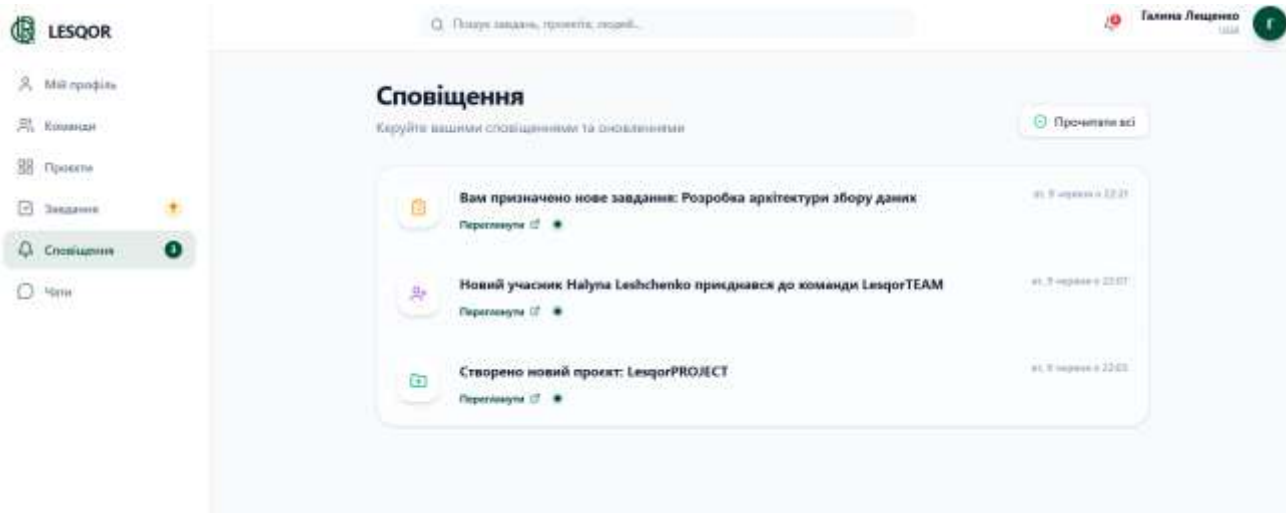


Рис. 3.16 – Сторінка системних сповіщень

Для безпосереднього опрацювання задач розроблено спеціалізований модуль. Навігація в ньому побудована ієрархічно: спочатку користувач бачить перелік активних проєктів і обирає той, з яким планує працювати (див. рис. 3.17).

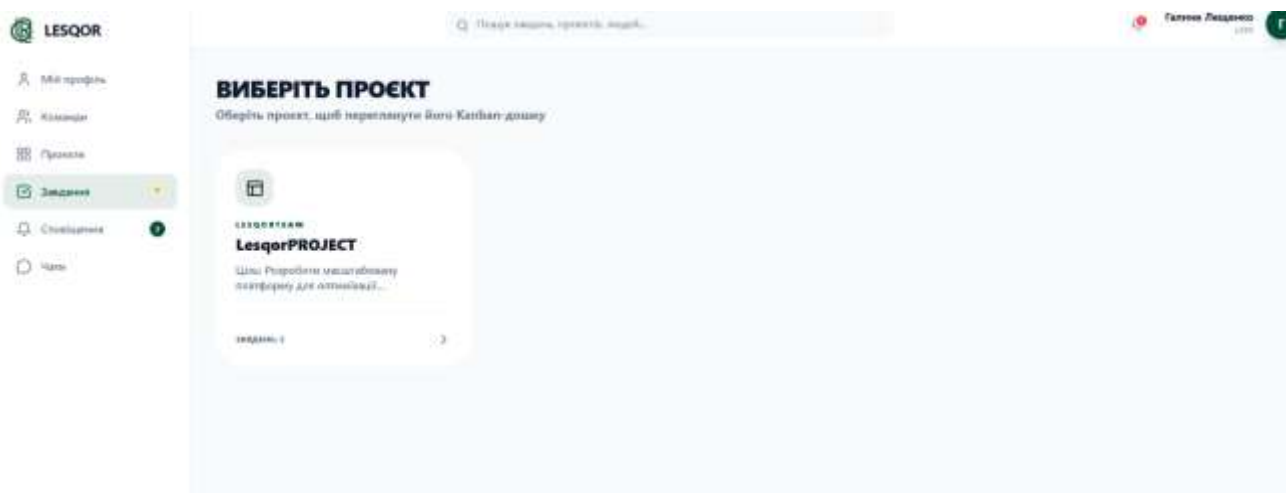


Рис. 3.17 – Інтерфейс вибору проєкту у модулі завдань

Робочий простір проєкту візуалізовано у форматі інтерактивної Kanban-дошки. Колонки дошки відповідають етапам виконання (статусам). Користувач може інтуїтивно змінювати стан завдання за допомогою механізму Drag-and-Drop (перетягування карток мишею). Для швидкої орієнтації серед великого масиву карток передбачено виклик додаткової панелі глибокого сортування та фільтрації. Також якщо в користувача є доступ на створення завдань в даному проєкті, то в нього відображається відповідна кнопка (див. рис. 3.18).

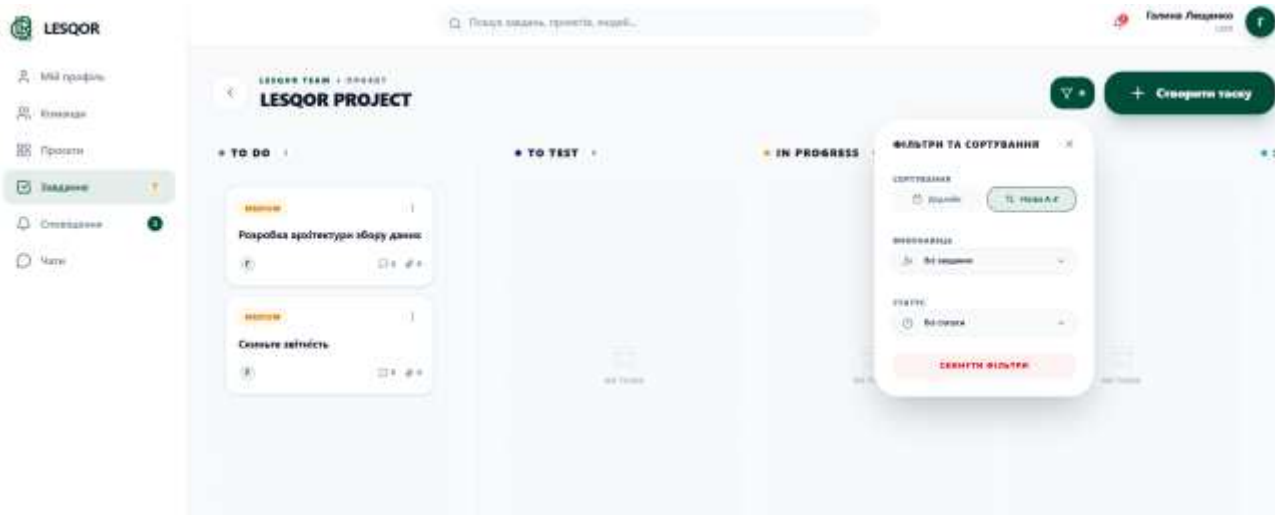


Рис. 3.18 – Kanban-дошка з відкритою панеллю фільтрації та сортування

На цій сторінці також відображено роботу системи обмеження прав доступу, що базується на встановленій рольовій моделі. Зокрема, елемент інтерфейсу для створення нових завдань відображається виключно для користувачів із ролями «Власник» та «Менеджер». Виконавці мають доступ лише до перегляду загальної дошки та зміни статусів призначених їм карток. Такий розподіл повноважень запобігає несанкціонованим змінам у списку завдань та адаптує інтерфейс до безпосередніх обов'язків користувача.

При виборі конкретної картки відкривається панель деталей завдання. Її інтерфейс містить інструменти для управління поточним завданням: редагування текстового опису, відстеження відсотка прогресу, а також додавання зовнішніх посилань та файлових вкладень. Зокрема, передбачено можливість ручного коригування показника прогресу виконавцем відповідно до фактично завершеного обсягу робіт, що забезпечує керівника актуальними даними щодо стану розв'язання задачі. Крім того, передбачено систему коментарів. Цей функціонал дозволяє учасникам команди обговорювати деталі процесу та фіксувати проміжні результати безпосередньо в картці відповідного завдання. Агрегація всіх робочих матеріалів та історії обговорень у межах однієї панелі спрощує подальший пошук необхідної інформації під час виконання чи аудиту завдання (див. рис. 3.19-3.20).

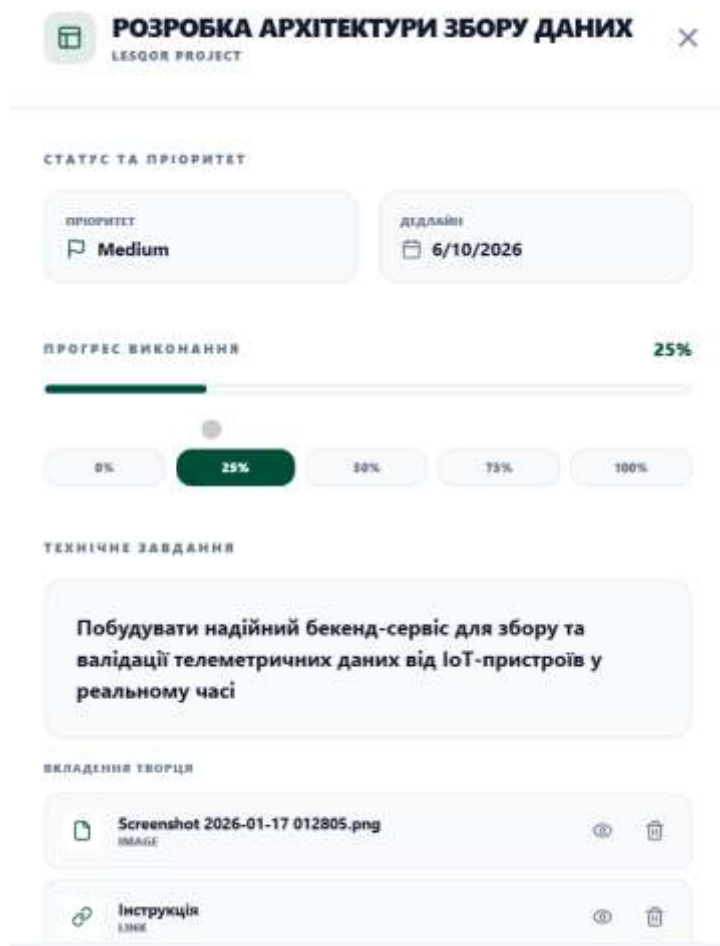


Рис. 3.19 – Детальна інформаційна панель завдання

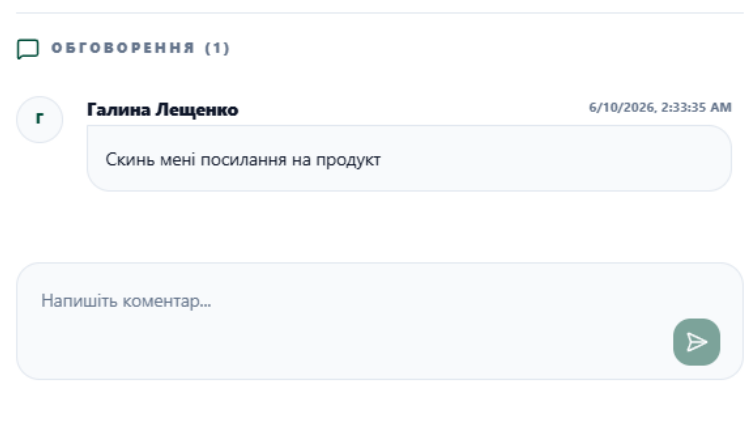


Рис. 3.20 – Детальна інформаційна панель завдання(коментарі)

Для забезпечення наскрізної навігації розроблено модуль глобального пошуку. Він дозволяє за ключовими словами швидко знаходити будь-яку сутність у системі: команди, проєкти, конкретні завдання чи профілі інших користувачів (див. рис. 3.21).

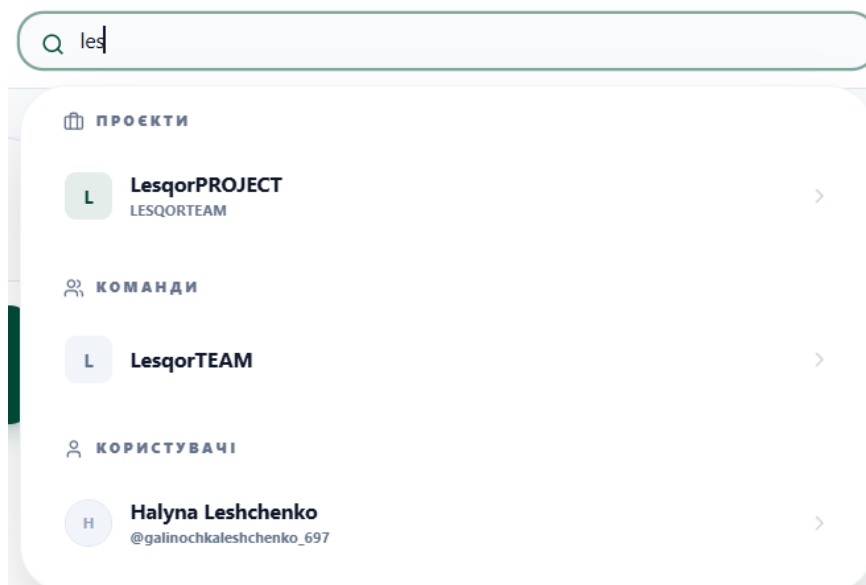


Рис. 3.21 – Інтерфейс глобального пошуку

Взаємодія між учасниками є додаткова, а саме можливість перегляду детальної інформації про профілі колег. З відповідної інформаційної панелі можна миттєво ініціювати комунікацію (див. рис. 3.22).

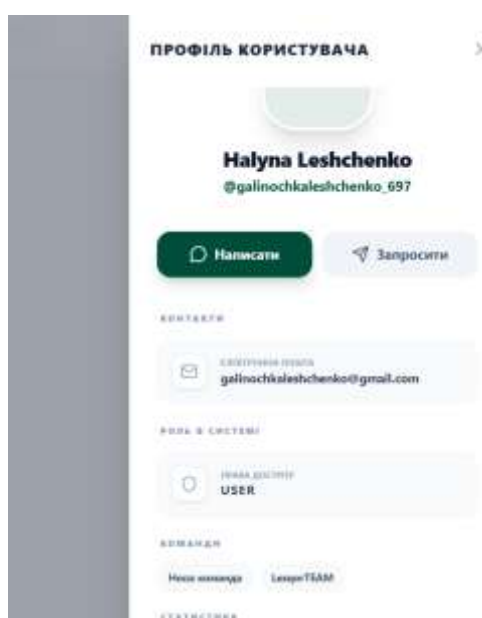


Рис. 3.22 – Бокова інформаційна панель профілю користувача

Комунікативний блок вебсистеми базується на технології WebSockets, що забезпечує обмін текстовими повідомленнями в режимі реального часу. Одразу після ініціалізації нової команди система автоматично створює загальний командний чат для всіх її учасників. Додатково реалізовано можливість створення приватних діалогів для спілкування між двома окремими особами (Direct Messages) (див. рис. 3.23).

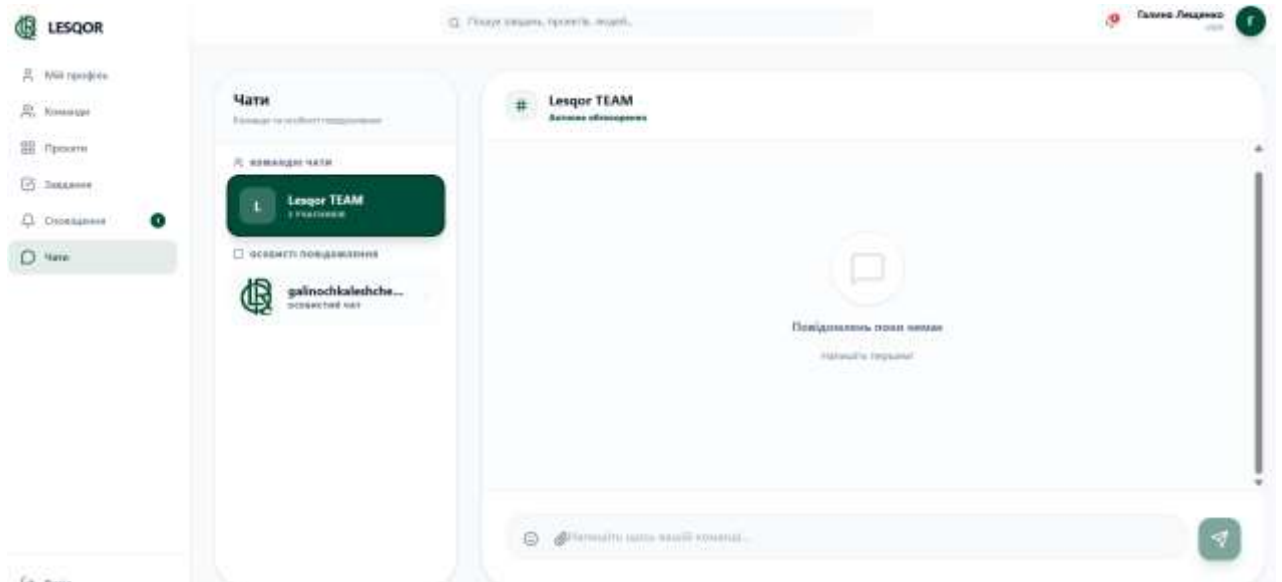


Рис. 3.23 – Інтерфейс модуля чатів та повідомлень

3.3. Тестування роботи вебсистеми

Тестування є одним із ключових етапів розробки програмного забезпечення, оскільки дозволяє перевірити коректність роботи функціоналу, виявити можливі помилки та забезпечити стабільність системи. У процесі розробки вебсистеми управління командними проєктами було проведено функціональне тестування основних можливостей системи за допомогою тест-кейсів. Нижче наведено приклади основних тест-кейсів, які використовувалися для перевірки роботи реалізованого функціоналу.

Таблиця 3.1

Test Case: Створення команди

Поле	Опис
Test Case ID	ТС-01
Функціонал	Створення команди
Передумови	Користувач авторизований
Кроки	1. Перейти в «Команди» 2. Натиснути «Створити команду» 3. Ввести назву, опис. Натиснути «+Додати» в полі «Кастомні статуси». Обрати колір, вписати назву статусу 4. Натиснути «Створити команду»»
Очікуваний результат	Команда створена; Є відображення нової команди зі всіма даними на сторінці команд; Є можливість додати проєкт до створеної команди; Створено чат команди;
Фактичний результат	Команда створена; Є відображення нової команди зі всіма даними на сторінці команд; Є можливість додати проєкт до створеної команди; Створено чат команди;

Поле	Опис
Статус	Pass
Фактичний результат	Команда створена; Є відображення нової команди зі всіма даними на сторінці команд; Є можливість додати проєкт до створеної команди; Створено чат команди;

Даний тест-кейс призначений для перевірки коректності роботи функціоналу створення нової команди (див. табл. 3.1). Під час тестування перевіряється можливість введення основної інформації про команду, додавання користувацьких статусів, створення командного простору та автоматичне формування чату для взаємодії учасників. Успішне проходження тесту підтверджує коректну роботу механізму створення команд та їх подальшого використання в системі.

Таблиця 3.2

Test Case: Створення завдання

Поле	Опис
Test Case ID	TC-01
Функціонал	Створення завдання
Передумови	Користувач авторизований
Кроки	1. Перейти в «Проекти». 2. Натиснути «Створити таску». 3. Ввести назву, опис. 4. Додати вкладення, посилання. 5. Обрати пріоритет, дедлайн. 6. Призначити завдання. 7. Натиснути «Створити таску».
Очікуваний результат	Завдання створене; Є відображення завдань на сторінці завдань з можливістю коментувати, змінювати дедлайн та додавати вкладення з посиланнями;
Фактичний результат	Завдання створене; Є відображення завдань на сторінці завдань з можливістю коментувати, змінювати дедлайн та додавати вкладення з посиланнями;
Статус	Pass

Тест-кейс спрямований на перевірку функціоналу створення нового завдання в межах проєкту (див. табл. 3.2). Під час виконання тесту перевіряється можливість додавання назви, опису, вкладень, посилань, пріоритету та дедлайну, а також призначення виконавця. Успішне виконання тесту підтверджує коректність роботи системи управління завданнями та їх відображення в інтерфейсі користувача.

Test Case: Створити окремий чат

Поле	Опис
Test Case ID	ТС-02
Функціонал	Створення окремого чату з пошуком користувача
Передумови	Користувач авторизований
Кроки	1. Ввести в пошук перші літери нікнейму користувача 2. Натиснути на користувача, який з'явився 3. Натиснути на кнопку «Написати».
Очікуваний результат	Створено новий чат з користувачем;
Фактичний результат	Створено новий чат з користувачем;
Статус	Pass

Даний тест-кейс детально перевіряє роботу функціоналу створення особистих чатів між користувачами системи (див. табл. 3.3). У процесі тестування оцінюється коректність роботи механізму пошуку користувачів за введеними символами (включаючи обробку затримок введення для оптимізації мережевих запитів), можливість безпомилкового вибору потрібного користувача з випадного списку та ініціалізація нового діалогу з подальшим встановленням з'єднання. Успішний результат цього тесту підтверджує стабільність модуля комунікації та гарантує надійну синхронізацію взаємодії між учасниками вебсистеми.

Окрім наскрізного функціонального тестування, для фронтенд-частини системи розроблено комплекс ізольованих UI-тестів [29]. Вони спрямовані на перевірку коректності рендерингу окремих компонентів інтерфейсу, їхньої реакції на зміну глобального стану та правильність імплементації дефолтних статус-кейсів (наприклад, коректна обробка станів завантаження або відсутності даних, що запобігає крашам інтерфейсу).

Важливим інженерним рішенням стала інтеграція написаних наборів тестів до автоматизованого конвеєра безперервної інтеграції та доставки (CI/CD). Це дозволяє виконувати автоматичну перевірку працездатності системи під час кожного збирання та розгортання нової версії застосунку, блокуючи перенесення коду з помилками на робочий сервер. Для об'єктивної оцінки надійності розробленого програмного забезпечення інструментарієм

платформи генерується детальний звіт, у якому наведено результати аналізу покриття коду тестами (Code Coverage) (див. рис. 3.24) [30].

File	% Stats	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	62.34	48.87	48.57	63.55	
entities/project/ui	45.83	46.15	33.33	45.45	
CreateProjectModel.tsx	45.83	46.15	33.33	45.45	22-32, 61-75, 93
entities/task/model	100	100	100	100	
taskDetailsSlice.ts	100	100	100	100	
entities/team/ui	70.23	70.9	36.36	74.66	
AddMemberModal.tsx	67.85	60	60	69.23	31-39, 94
CreateTeamModel.tsx	69.23	73.68	25	76.08	47-49, 68, 72, 118-182
LeaveTeamModel.tsx	100	100	100	100	
entities/user/api	25	100	10	25	
userApi.ts	25	100	10	25	33-74
entities/user/model	100	100	100	100	
userProfileDrawerSlice.ts	100	100	100	100	
features/auth/model	75.86	50	57.14	75.86	
authSlice.ts	75.86	50	57.14	75.86	55-56, 62-63, 69-71
features/notifications/ui	45.23	21.62	25	43.58	
NotificationDropdown.tsx	45.23	21.62	25	43.58	35-36, 46-52, 56-62, 67-75, 98-157
shared/api	18.75	0	33.33	18.75	
base.ts	18.75	0	33.33	18.75	6-10, 16-31
shared/context	100	100	100	100	
ThemeContext.tsx	100	100	100	100	

Рис. 3.24 – Відображення покриття системи тестами

Під час виконання процесу розгортання всі автоматизовані тести були успішно виконані, що підтверджує працездатність реалізованого функціоналу та відсутність критичних помилок. (див. Додаток Б)

Висновки до третього розділу

У даному розділі було розглянуто порядок встановлення, налаштування та використання розробленої вебсистеми управління завданнями для командної роботи. Описано процес розгортання програмного забезпечення із застосуванням Docker-контейнеризації, серверної інфраструктури на базі Linux та автоматизованого конвеєра GitLab CI/CD, що забезпечує стабільне розгортання та оновлення системи. Також було детально проаналізовано структуру користувацького інтерфейсу та порядок взаємодії користувача з основними функціональними модулями системи. Розглянуто процеси реєстрації та авторизації, управління профілем, створення команд і проєктів, постановки та контролю виконання завдань, використання системи сповіщень, глобального пошуку та модулів комунікації в режимі реального часу. Наведені інтерфейсні рішення демонструють орієнтованість системи на зручність

використання, ефективну організацію командної роботи та підтримку сучасних підходів до управління проєктами. Для перевірки працездатності реалізованого функціоналу було проведено функціональне тестування основних сценаріїв використання системи та розроблено набір UI-тестів для окремих компонентів клієнтської частини. Результати тестування підтвердили коректність роботи ключових модулів вебсистеми, а успішне проходження автоматизованих перевірок у CI/CD-конвеєрі засвідчило стабільність програмного продукту та готовність системи до практичного використання.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було спроектовано та реалізовано вебсистему «Lesqor» для управління завданнями та організації командної взаємодії. У процесі роботи обґрунтовано актуальність створення подібних рішень, проаналізовано існуючі аналоги та предметну область, що дозволило визначити основні вимоги до системи, зокрема централізоване управління завданнями, підтримку канбан-методології, рольову модель доступу, багаторівневу комунікацію та інтелектуальні механізми пошуку. На основі проведеного аналізу сформовано концепцію системи, орієнтованої на зручність використання, мінімалістичний інтерфейс та ефективну організацію робочих процесів у команді.

Під час розробки реалізовано сучасну клієнт-серверну архітектуру із використанням REST API та WebSockets, що забезпечує обмін даними в режимі реального часу. Використання технологічного стеку Full-stack TypeScript (React, Node.js) підвищило типобезпечність та спростило підтримку коду, а застосування PostgreSQL забезпечило надійне зберігання та цілісність даних. Система автентифікації та авторизації реалізована із застосуванням JWT та OAuth 2.0, що гарантує безпечний доступ до ресурсів. Додатково впроваджено механізми автоматизованого розгортання та оновлення через CI/CD і контейнеризацію Docker, що забезпечує стабільність роботи системи у різних середовищах. Реалізовано основні функціональні модулі для управління командами, проєктами та завданнями, включно з підтримкою Kanban-дошки, системою сповіщень, глобальним пошуком та інтерактивною взаємодією користувачів. Проведене тестування підтвердило коректність роботи основних функцій системи, а успішне проходження автоматизованих перевірок засвідчило її стабільність та готовність до практичного використання. Отримані результати підтверджують досягнення поставленої мети та виконання всіх завдань дослідження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Trello. URL: <https://trello.com/> (дата звернення: 05.03.2026).
2. Jira. URL: <https://www.atlassian.com/software/jira> (дата звернення: 08.03.2026).
3. Asana. URL: <https://asana.com/> (дата звернення: 12.03.2026).
4. SPA (Single-page application) - MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Glossary/SPA> (дата звернення: 15.03.2026).
5. React. URL: <https://react.dev/> (дата звернення: 18.03.2026).
6. Redux Toolkit. URL: <https://redux-toolkit.js.org/> (дата звернення: 20.03.2026).
7. RTK Query Overview. URL: <https://redux-toolkit.js.org/rtk-query/overview> (дата звернення: 24.03.2026).
8. REST API: що це, як працює, переваги і приклади - GoIT. URL: <https://goit.global/ua/articles/rest-api-shcho-tse-iak-pratsiuie-perevahy-i-pryklady/> (дата звернення: 27.03.2026).
9. JSON. URL: <https://www.json.org/json-en.html> (дата звернення: 30.03.2026).
10. Middleware - MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Glossary/Middleware> (дата звернення: 02.04.2026).
11. Docker. URL: <https://www.docker.com/> (дата звернення: 05.04.2026).
12. What is an ORM? The meaning of object-relational mapping database tools - freeCodeCamp. URL: <https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/> (дата звернення: 09.04.2026).
13. Що таке SPA додатки: огляд і переваги - Wezom. URL: <https://wezom.com.ua/ua/blog/chto-takoe-spa-prilozheniya> (дата звернення: 12.04.2026).

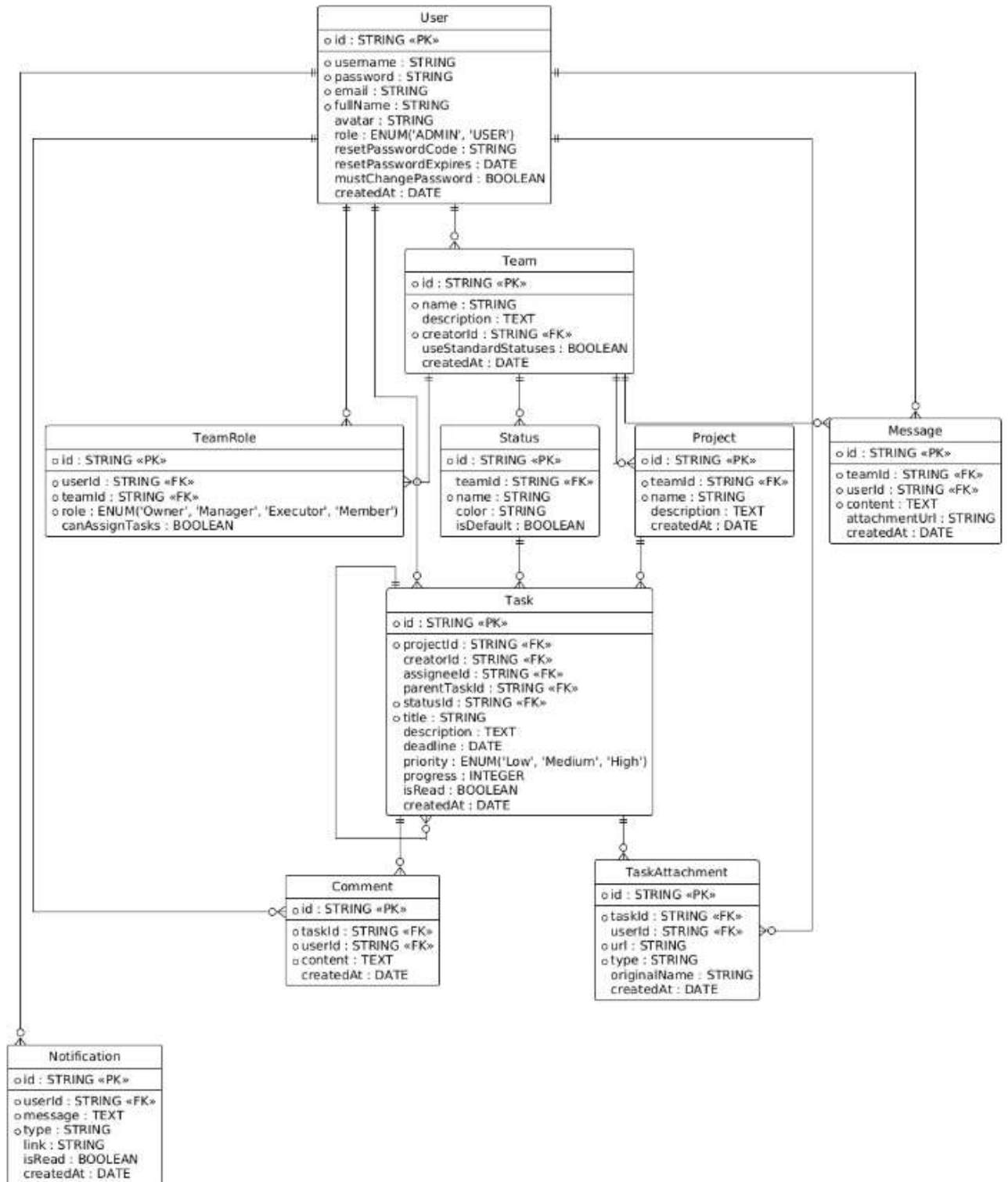
14. WebSockets API - MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API (дата звернення: 14.04.2026).
15. Laravel. URL: <https://laravel.com/> (дата звернення: 18.04.2026).
16. Django. URL: <https://www.djangoproject.com/> (дата звернення: 22.04.2026).
17. Spring. URL: <https://spring.io/> (дата звернення: 25.04.2026).
18. Node.js. URL: <https://nodejs.org/en> (дата звернення: 29.04.2026).
19. TypeScript. URL: <https://www.typescriptlang.org/> (дата звернення: 03.05.2026).
20. Express.js. URL: <https://expressjs.com/en/> (дата звернення: 06.05.2026).
21. PostgreSQL. URL: <https://www.postgresql.org/> (дата звернення: 10.05.2026).
22. jsonwebtoken - npm. URL: <https://www.npmjs.com/package/jsonwebtoken> (дата звернення: 13.05.2026).
23. bcryptjs - npm. URL: <https://www.npmjs.com/package/bcryptjs> (дата звернення: 16.05.2026).
24. Lucide React. URL: <https://lucide.dev/guide/react/> (дата звернення: 19.05.2026).
25. UML Use Case Diagram Tutorial - Lucidchart. URL: <https://www.lucidchart.com/pages/tutorial/uml-use-case-diagram> (дата звернення: 21.05.2026).
26. Документація GitLab CI/CD. URL: <https://docs.gitlab.com/ci/> (дата звернення: 24.05.2026).
27. Deployment Diagram UML Example - Guru99. URL: <https://www.guru99.com/uk/deployment-diagram-uml-example.html> (дата звернення: 27.05.2026).
28. Nginx. URL: <https://nginx.org/> (дата звернення: 29.05.2026).

29. Тестування інтерфейсу користувача (UI) - Wezom. URL: <https://wezom.com.ua/ua/blog/testing-ui-user-interface> (дата звернення: 01.06.2026).

30. Code coverage - SonarSource. URL: <https://www.sonarsource.com/resources/library/code-coverage/> (дата звернення: 03.06.2026).

ДОДАТКИ

					ПЗ.КР.Б – 121 – 26 – ПЗ	
						85



Додаток В. «Проходження тестів у CI/CD pipeline»

```
Log timestamps in UTC, Search visible log output
34 16:20:31 run 'npm fund' for details
35 16:20:31 4 vulnerabilities (1 moderate, 3 high)
36 16:20:31 To address all issues, run:
37 16:20:31   npm audit fix
38 16:20:31 Run 'npm audit' for details.
39 16:20:31 $ npm test
40 16:20:31 > frontend@0.0.0 test
41 16:20:31 > vitest run
42 16:20:32 [vite] v4.1.8 C:/GitLab-Runner/builds/2CHKynCIU/0/lpz/2022-2026/lpz-22-2/Leshchenko-halyna/Lesqor/fro
ntend
43 16:20:36 ✓ src/entities/task/model/taskDetailsSlice.test.ts (3 tests) 13ms
44 16:20:36 ✓ src/entities/user/model/userProfileDrawerSlice.test.ts (3 tests) 12ms
45 16:20:36 ✓ src/features/auth/model/authSlice.test.ts (8 tests) 26ms
46 16:20:36 ✓ src/shared/context/ThemeContext.test.tsx (7 tests) 155ms
47 16:20:37 ✓ src/features/notifications/ui/NotificationDropdown.test.tsx (4 tests) 184ms
48 16:20:37 ✓ src/entities/team/ui/LeaveTeamModal.test.tsx (4 tests) 367ms
49 16:20:37 ✓ src/entities/project/ui/CreateProjectModal.test.tsx (4 tests) 389ms
50 16:20:37 ✓ src/entities/team/ui/CreateTeamModal.test.tsx (6 tests) 746ms
51 16:20:37 ✓ src/entities/team/ui/AddMemberModal.test.tsx (5 tests) 771ms
52 16:20:37   ✓ calls addMember mutation on valid form submit 488ms
53 16:20:37 Test Files  9 passed (9)
54 16:20:37 Tests     44 passed (44)
55 16:20:37 Start at  19:20:32
56 16:20:37 Duration  4.99s (transform 1.55s, setup 2.98s, import 4.77s, tests 2.58s, environment 22.93s)
57 16:20:37 Cleaning up project directory and file-based variables
58 16:20:38 Job succeeded
```