

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 «Веб технології»)

на тему:

«Платформа для управління ІТ-проектами»

Виконав студент групи ВТ-21-1
МАНЬКІВСЬКИЙ Владислав Вячеславович

Керівник роботи:
САВІЦЬКИЙ Роман Святославович

Рецензент:
ДУНЄВ Сергій Сергійович

Житомир – 2025

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

«14» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **МАНЬКІВСЬКИЙ Владислав Вячеславович**

Керівник роботи: **САВІЦЬКИЙ Роман Святославович**

Тема роботи: **«Платформа для управління ІТ-проектами»**,

затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №61/с**

Вихідні дані для роботи: **Об'єктом є автоматизація роботи управління ІТ-проектами з метою підвищення ефективності командної роботи та спрощення процесів планування, відстеження й координації завдань.**

Предмет дослідження є процеси проектування, розробки та реалізації платформи для управління ІТ-проектами.

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

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Савіцький Р. С.	17.02.2025	17.02.2025
2	Савіцький Р. С.	19.02.2025	19.02.2025
3	Савіцький Р. С.	21.02.2025	21.02.2025

РЕФЕРАТ

Випускна кваліфікаційна робота бакалавра складається з вебдодатку платформи для управління ІТ-проєктами та пояснювальної записки.

Текстова частина викладена на 67 сторінках друкованого тексту. Список використаних джерел містить 30 найменувань і займає 3 сторінки. В роботі наведено 61 рисунок та 24 таблиць. Загальний обсяг роботи – 72 сторінок.

Метою роботи є розробка платформи для управління ІТ-проєктів, яка забезпечує ефективну координацію командної роботи, контроль за виконанням завдань і оптимізацію робочих процесів. Система передбачає інтеграцію з GitHub для синхронізації статусів завдань та написання документації.

У роботі поставлені основні завдання на розробку системи. Також було проведено порівняльний аналіз аналогічних систем. Було обрано монолітну архітектуру з додатковим сервісом. Приведені основні сценарії використання системи, логіка взаємодії між модулями системи та алгоритми роботи ключових модулів, спроектована база даних з використанням PostgreSQL. Клієнтська та серверна частина розроблена за допомогою фреймворку Next.js, та додатково для відокремленого сервера використано Express.js.

Ключові слова: ПЛАТФОРМА, РОБОЧИЙ ПРОСТІР, ЗАВДАННЯ, СТАТУС, УПРАВЛІННЯ.

					ІПЗ.КР.Б – 121 – 25 – ПЗ				
Змн.	Арк.	№ докум.	Підпис	Дата					
Розроб.		В.В. Маньківський			Платформа для управління ІТ-проектами	Літ.	Арк.	Аркушів	
Керівник		Р.С. Савіцький					3	72	
						Житомирська політехніка, група ВТ-21-1			
Рецензент.		С.С. Дунєв							
Зав. каф.		Т.А. Вакалюк							

ABSTRACT

The bachelor's final qualification work consists of a web application of an IT project management platform and an explanatory note.

The text part is presented on 67 pages of printed text. The list of used sources contains 30 names and occupies 3 pages. The work contains 61 figures and 24 tables. The total volume of the work is 72 pages.

The goal of the work is to develop a platform for managing IT projects that provides effective coordination of teamwork, task execution control, and workflow optimization. The system provides integration with GitHub for task status synchronization and documentation writing.

The work sets the main tasks for the development of the system. A comparative analysis of similar systems was also conducted. A monolithic architecture with an additional service was chosen. The main scenarios of using the system, the logic of interaction between the system modules and the algorithms of the key modules are presented, a database is designed using PostgreSQL. The client and server parts are developed using the Next.js framework, and additionally Express.js is used for a separate server.

Keywords: PLATFORM, WORKSPACE, TASKS, STATUS, MANAGEMENT.

					ИПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						4

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ	9
1.1. Визначення та постановка задачі.....	9
1.2. Аналіз аналогів програмного продукту	10
1.3. Визначення архітектури ПЗ	14
1.4. Обґрунтування вибору інструментальних засобів.....	16
Висновки до першого розділу.....	18
РОЗДІЛ 2. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ	20
2.1. Визначення варіантів використання.....	20
2.2. Розробка бази даних системи	26
2.3. Проектування та реалізація алгоритмів роботи системи.....	34
2.4. Реалізація функціоналу системи	38
Висновки до другого розділу	47
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ПЛАТФОРМОЮ	48
3.1. Порядок встановлення та налаштування параметрів платформи.....	48
3.2. Структура інтерфейсу платформи	49
3.3. Тестування роботи програмного продукту	59
Висновки до третього розділу	61
ВИСНОВКИ.....	63
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	65
Додатки.....	68
Додаток А.....	69

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ACID – Atomicity, Consistency, Isolation, and Durability

API – application programming interface

СКБД – система керування базами даних

HTTP – Hypertext Transfer Protocol

IT – information technologies

ORM – Object-Relational Mapping

ПЗ – програмне забезпечення

SEO – Search Engine Optimization

UI – User Interface

UX – User Experience

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						6

ВСТУП

Платформа для управління ІТ-проектів орієнтована на швидку та зручну взаємодію з робочими просторами, їх завданнями та документацією у GitHub репозиторії.

Актуальність теми зумовлена величезною кількістю ІТ-проектів, що потребують ефективного менеджменту та координації. Під час розробки проектів можуть виникнути проблеми з управлінням чи контролем, тому використання зручної платформи для управління ІТ-проектами набуває важливості, оскільки дозволяє централізувати відстеження завдань, покращити комунікацію між учасниками та автоматизувати рутинні процеси, що в результаті сприятиме підвищенню продуктивності, забезпеченню злагодженої роботи команди та успішному досягненню цілей проекту.

Метою випускної кваліфікаційної роботи бакалавра є розробка платформи для управління ІТ-проектів, яка забезпечує ефективну координацію командної роботи, контроль за виконанням завдань і оптимізацію робочих процесів. Система передбачає інтеграцію з GitHub для синхронізації статусів завдань та написання документації.

Завданням випускної кваліфікаційної роботи бакалавра є виконання таких задач:

- постановка основних задач системи та узагальнення її структури;
- аналіз аналогів;
- побудова архітектури, що буде задовольняти вимоги системи;
- вибір та обґрунтування засобів реалізації системи;
- визначення варіантів використання платформи;
- розробка функціоналу та реалізація алгоритмів платформи;
- розробка UI/UX для клієнтської частини платформи;
- розгортання платформи;
- мануальне та юніт тестування розробленого функціоналу.

Об’єктом дослідження є автоматизація роботи управління ІТ-проектами з метою підвищення ефективності командної роботи та спрощення процесів планування, відстеження й координації завдань.

Предмет дослідження є процеси проектування, розробки та реалізації платформи для управління ІТ-проектами.

За темою випускної кваліфікаційної роботи бакалавра було опубліковано тезу [1].

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						8

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

1.1. Визначення та постановка задачі

Метою кваліфікаційної роботи бакалавра є розробка платформи для управління IT-проектами для підвищення ефективності роботи у команді та спрощення управління завданнями.

Платформи для управління IT-проектами – це онлайн-сервіси, які допомагають командам планувати, відстежувати та координувати роботу над програмними продуктами. Вони забезпечують інструменти для постановки задач, розподілу відповідальності, контролю прогресу та спільної роботи.

Для досягнення мети платформа повинна мати певні базові модулі. А саме, модуль проєктів, який забезпечує можливість створення, редагування та управління проєктами. Модуль користувачів дозволяє керувати ролями та правами доступу учасників проєкту, створювати облікові записи та налаштовувати рівні доступу до даних і функцій системи. Модуль завдань є основним інструментом для створення, відстеження та управління завданнями, з підтримкою встановлення пріоритетів, призначення виконавців і коментування. Модуль документації дозволяє створювати, редагувати та зберігати технічну й проєктну документацію.

Окрім базових модулів, необхідно мати ті, які спростять взаємодію з платформою. Варто передбачити модуль реального часу, який відповідає за синхронізацію даних між користувачами, оновлення статусу завдань та коментарів у режимі реального часу з використанням WebSockets. Модуль сповіщень надсилає повідомлення про зміни у проєкті, оновлення завдань, нові коментарі або важливі події через внутрішню систему оповіщень. Модуль інтеграцій забезпечує безперебійну взаємодію з системою контролю версій GitHub, що дозволяє автоматично пов'язувати гілки та запити на злиття з конкретними завданнями у проєкті та забезпечувати централізоване та синхронізоване ведення документації у GitHub Wiki.

Розробка платформи розпочинається з визначення її загальної структури. На цьому етапі формулюється основна концепція та ключові функціональні можливості майбутнього застосунку. Також здійснюється проектування моделі даних, яка детально описує всі сутності платформи та зв'язки між ними, що є фундаментом для подальшої розробки.

Наступним етапом є побудова архітектури як клієнтської, так і серверної частини платформи. Це передбачає визначення загальної архітектурної моделі системи, планування способів обробки запитів від клієнтів та організації зберігання даних на серверній стороні.

Третій етап – реалізація базового функціоналу платформи. Розробляється система аутентифікації та авторизації користувачів, що включає визначення різних ролей та надання їм відповідних прав доступу до функціоналу та даних. Також створюються механізми для управління проєктами та реалізується основна функціональність для роботи з робочими елементами.

Останній етап є ретельне тестування та налагодження розробленої платформи. Проводиться мануальне та юніт тестування. У процесі цього виявляються програмні помилки та недоліки, які одразу усуваються.

Результатом реалізації поставленого завдання є вебплатформа для управління проєктами, яка включає багатокористувацький клієнтський, сервер для обробки та зберігання даних у базі та сервіс для змін у реальному часі.

1.2. Аналіз аналогів програмного продукту

Існує багато платформ для управління проєктами, що дозволяють командам ефективно організовувати робочі процеси, відстежувати завдання та співпрацювати в реальному часі. Для порівняння ми розглянемо три популярні сервіси: Linear [2], GitHub Projects [3] та Jira [4]. Ці платформи відрізняються своєю цільовою аудиторією, рівнем гнучкості, інтеграцією з іншими інструментами та можливостями налаштування процесів управління проєктами.

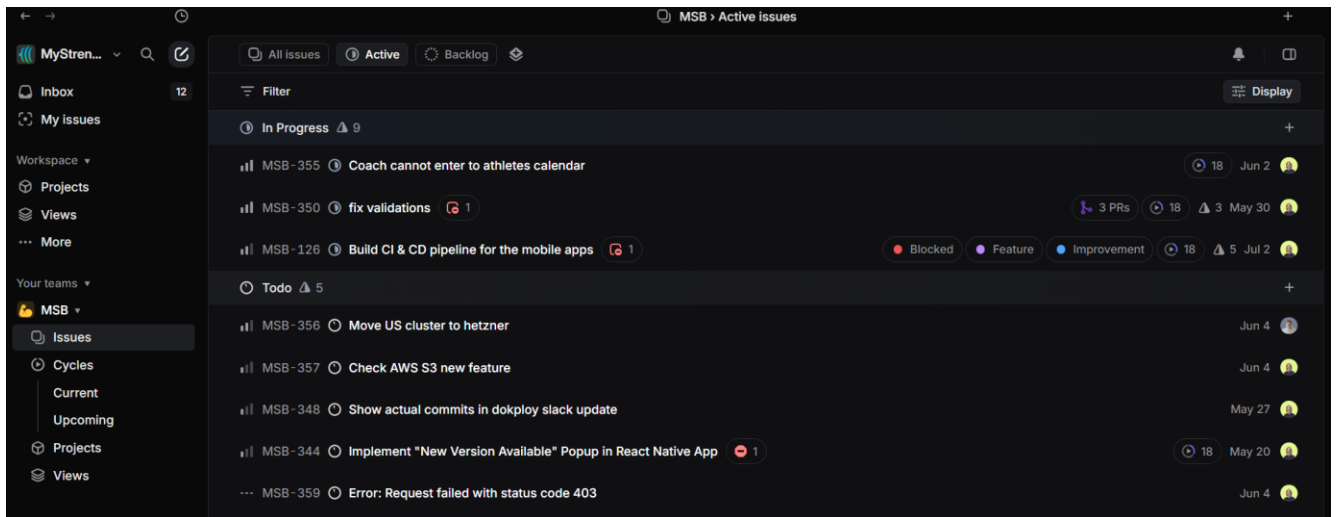


Рис. 1.1 – Linear

Linear (рис.1.1) – це спеціалізований інструмент для сучасного управління розробкою продуктів. Він забезпечує оптимізацію процесів роботи із завданнями, проектами та дорожніми картами продукту, спрощуючи їхнє ведення та контроль.

Основними перевагами Linear є:

- висока продуктивність і швидкість;
- зручний, інтуїтивно зрозумілий інтерфейс;
- підходить для малих та середніх команд.

Основними недоліками Linear є:

- мало додаткових функцій для великих підприємств.

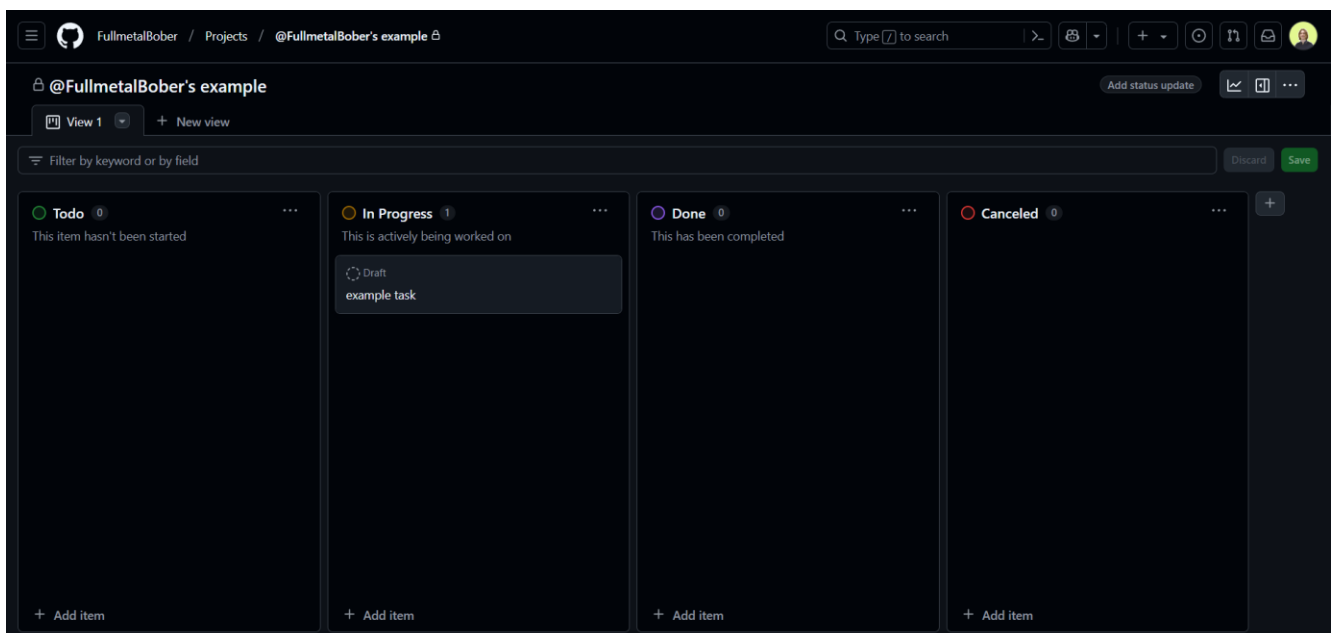


Рис. 1.2 – GitHub Project

GitHub Projects (рис.1.2) – це гнучкий інструмент для управління завданнями, що поєднує можливості електронних таблиць, канбан-дошки та дорожньої карти. Він інтегрується із завданнями та pull requests на GitHub, допомагаючи ефективно планувати та відстежувати робочі процеси.

Основними перевагами GitHub Projects є:

- глибока інтеграція з GitHub (issues та pull requests);
- підходить для проєктів, що знаходяться на платформі GitHub.

Основними недоліками GitHub Projects є:

- тісна прив'язка до GitHub, тому не підходить для тих, хто його не використовує.

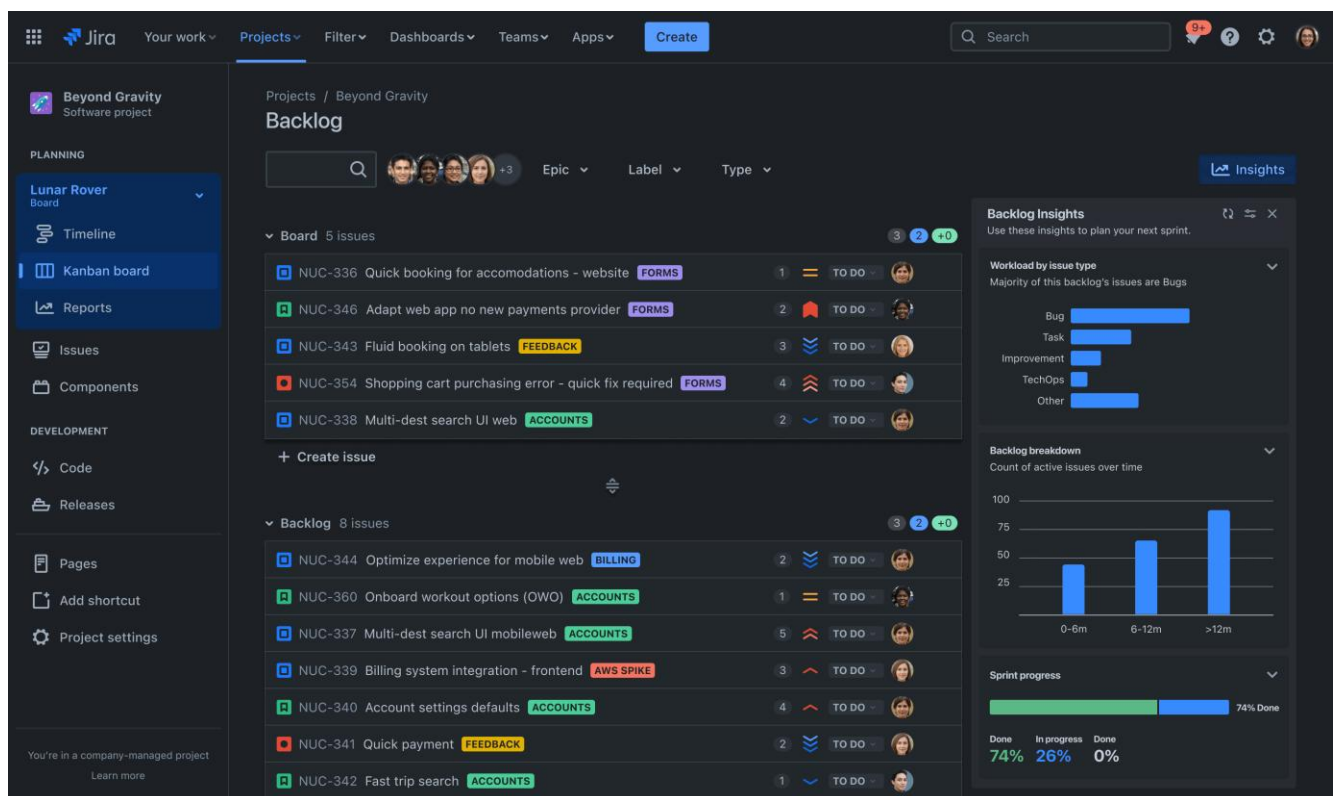


Рис. 1.3 – Jira

Jira (рис.1.3) – це потужний інструмент для управління проєктами, призначений для планування, відстеження та координації завдань у командах. Він підтримує гнучкі методології розробки, такі як Scrum і Kanban, забезпечуючи глибоку кастомізацію та інтеграцію з іншими сервісами.

Основними перевагами Jira є:

- можливості інтеграції з низкою популярних ресурсів;
- широкий спектр функцій;
- підходить для великих і складних проєктів;
- висока можливість налаштування під потреби команди.

Основними недоліками Jira є:

- менш інтуїтивно зрозумілий інтерфейс, що означає важкість входження;
- для невеликих команд може бути важко налаштувати робочий простір;
- продуктивність може бути низькою при великій кількості завдань.

Порівняння функціональних можливостей можна переглянути в таблиці 1.1.

Таблиця 1.1

Функціональні можливості аналогів

Функціональні можливості	Linear	GitHub Projects	Jira	Розроблювана платформа
Підтримка кількох проєктів	так	так	так	так
Коментарі до завдань	так	так	так	так
Пріоритети та статуси завдань	так	так	так	так
Сповіщення	так	так	так	так
Підтримка кількох користувачів	так	так	так	так
Співпраця в реальному часі	так	ні	ні	так
Простота використання	легко	важко	важко	легко
Інтеграція GitHub	так	так	так	так
Блоковий редактор тексту	так	ні	ні	так
Можливість ведення документації	ні	так	ні	так

Кожна з розглянутих платформ має свої особливості та унікальні можливості, які підходять для різних команд і типів проєктів. Linear підходить для невеликих та середніх команд, завдяки своїй швидкості та мінімалістичному дизайну. GitHub Projects чудово підходить для тих, хто бажає все тримати у одному сервісі: код та завдання. Jira пропонує широкий спектр налаштувань.

Отже, під час аналізу аналогів було виявлено, яким саме має бути базовий функціонал платформи для управління IT-проєктами. Основою функціоналу стане можливість працювати у режимі відображення даних у реальному часі. Окрім цього користувачі мають мати змогу користуватися зручним блоковим редактором тексту. Сама ж платформа має бути інтуїтивно зрозумілою та мінімалістичною. Таким чином, завдяки аналізу аналогів було визначено ключові функції та пріоритети розробки платформи.

1.3. Визначення архітектури ПЗ

Для розробки платформи для управління IT-проєктами було обрано монолітну архітектуру. Також був доданий допоміжний сервіс для підтримки вебсокетів (рис.1.4). Такий підхід дозволив зберегти цілісність системи, водночас делегуючи частину функціональностей окремим процесам.

Монолітна архітектура – це модель, у якій програма є єдиною уніфікованою системою програмного забезпечення, де інтерфейс, серверна частина та база даних тісно пов'язані. Уся бізнес-логіка, інтерфейс користувача та обробка даних перебувають в одній кодовій базі.

Проте, незважаючи на уніфікацію програмного продукту, уся логіка платформи чітко розділена на фронтенд та бекенд, що дозволяє краще організувати роботу над різними частинами застосунку. На фронтенді реалізовано інтерфейс користувача та логіку взаємодії, тоді як у бекенді зосереджено обробку даних, бізнес-правила та взаємодію з базою даних.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						14

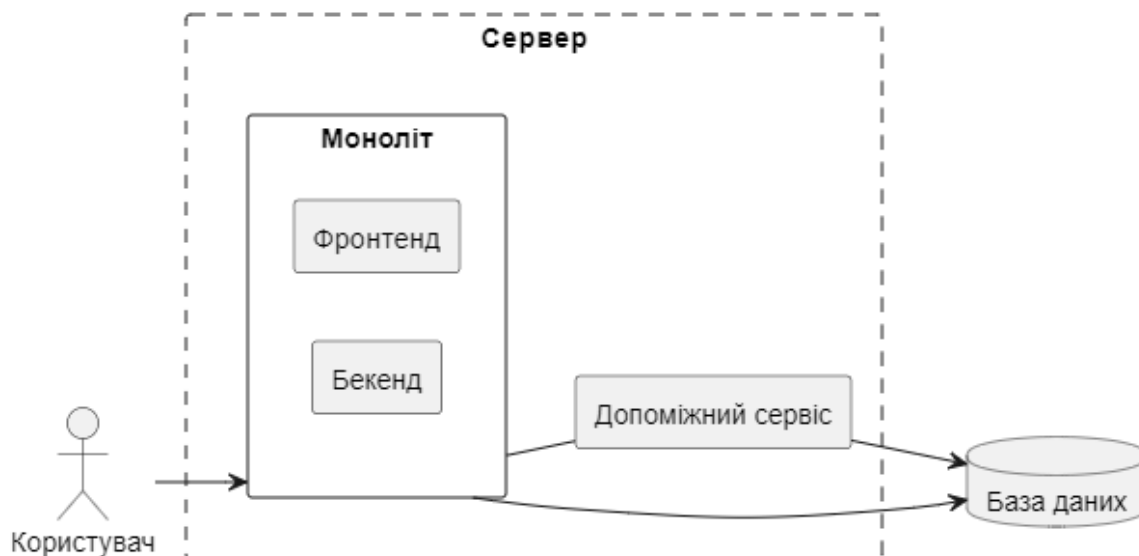


Рис. 1.4 – Монолітна архітектура з допоміжним сервісом

Особливе місце в бекенд частині займають server actions – функції, які виконуються безпосередньо на сервері у відповідь на дії користувача (див. рис.1.5). Вони дозволяють ефективно обробляти запити, працювати з базою даних або сторонніми API, не ускладнюючи клієнтський код і знижуючи навантаження на фронтенд. Server actions є важливим елементом розмежування обов’язків між частинами застосунку, забезпечуючи централізовану логіку обробки подій і високий рівень безпеки даних.

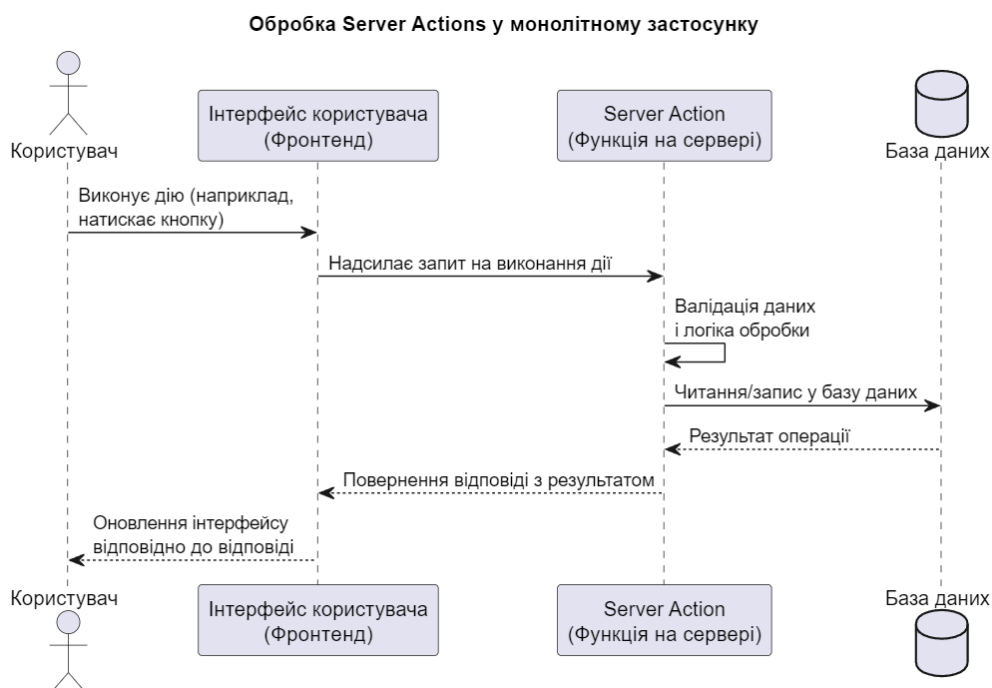


Рис. 1.5 – Принцип роботи Server Actions

На діаграм можна побачити, що користувач ініціює дію на фронтенді, яка надсилає запит до серверної функції (server action). На сервері виконується валідація та бізнес-логіка, після чого дані зчитуються або записуються в базу. Результат повертається на фронтенд, і інтерфейс оновлюється відповідно до відповіді.

Допоміжний сервіс створено для обробки WebSocket-з'єднань і вебхуків. Це допомагає відокремити асинхронну комунікацію та обробку подій за межі основного монолітного застосунку. Такий підхід підвищує масштабованість і стійкість системи до навантажень, спрощує розробку й обслуговування спеціалізованих функцій, не ускладнюючи основну кодову базу.

Отже, обрана монолітна архітектура з інтеграцією допоміжного сервісу й використанням server actions — це компромісний, але ефективний підхід, який поєднує цілісність і простоту моноліту з гнучкістю та розширюваністю модульної структури. Така архітектура дозволяє платформі залишатися керованою, розширюваною та придатною до подальшого масштабування в міру зростання функціональності.

1.4. Обґрунтування вибору інструментальних засобів

Додаток написаний на мові програмування TypeScript [5]. Для імплементації монолітної архітектури обрано фреймворк Next.js, що дає змогу працювати як з клієнтською, так і серверною частинами застосунку [6], а щодо допоміжного сервісу підійде серверний фреймворк Express.js [7]. Як базу даних обрано PostgreSQL [8].

TypeScript – це мова програмування, побудована на JavaScript, що додає статичну типізацію, що сприяє виявленню помилок на етапі розробки, покращує читабельність та підтримуваність кодової бази.

Next.js – це фреймворк, який базується на React, популярній бібліотеці для створення UI на основі компонентів, додаючи серверний та статичний рендеринг. Ці типи рендерингу дозволяють створювати швидкі, оптимізовані і SEO-дружні

вебдодатки. Server Actions – це асинхроні функції, які хоч і можуть бути викликані на клієнті, але виконуються на сервері, вони спрощує обробку форм та мутацію даних зберігаючи TypeScript типізацію, зменшуючи необхідність у відокремленому API.

Express.js обраний для взаємодії користувачів з платформою у реальному часі та обробки вебхуків. Завдяки своїй гнучкості він дозволяє швидко створювати масштабовані API та обробляти HTTP-запити з мінімальними затримками. Сумісність з великим числом middleware та підтримкою WebSocket-підключень робить його ефективним інструментом для поставлених цілей.

Для визначення бази, яка відповідає потребам системи, проведемо порівняння декількох СКБД, а саме MongoDB, MSSQL, та PostgreSQL (табл.1.2). Оскільки необхідна база даних, яка зможе забезпечити сувору типізацію, оберемо реляційну базу. MongoDB нереляційна, тому потреби системи не закриє. База даних MSSQL хоч і багата на різноманітний функціонал, проте можуть виникнути проблеми з її розгортанням у середовищі Linux, тому вона не може бути використана в проєкті. Наступною розглянемо PostgreSQL, яка найкраще підходить під потреби системи. База даних забезпечує відповідність ACID-транзакціям та підтримує складні запити, що важливо для роботи зі складно пов'язаними даними. Наприклад, створення вибірок задачі, користувачі та коментарі в одному запиті. Крім того, через популярність PostgreSQL, вона має широку екосистему інструментів та плагінів для оптимізації продуктивності.

Таблиця 1.2

Порівняльний аналіз баз даних

Показник	PostgreSQL	MSSQL	MongoDB
Модель даних	Об'єктно–реляційна	Реляційна	Нереляційна
Схема	Жорстка	Жорстка	Гнучка
Транзакції	ACID	ACID	Частковий ACID
Підтримка складних зв'язків	Так	Так	Немає вбудованих

Ліцензія та оплата	PostgreSQL License (відкрита, безкоштовна)	Пропрієтарна (є безкоштовна Express, але повноцінні версії платні)	SSPL (відкрита, але з обмеженнями для комерційного використання, платна Enterprise версія)
Підтримувані платформи ОС	Windows, Linux, macOS, BSD	Windows, Linux (з обмеженнями)	Windows, Linux, macOS
Область застосування	Середній та великий бізнес	Великий бізнес	Малий та середній бізнес

Для зручності розгортання бази даних використовується Docker, що забезпечує швидке розгортання, спрощує керування конфігураціями та гарантує стабільну роботу незалежно від платформи.

Під час розробки було використано Visual Studio Code. Це легке, але потужне середовище, яке підтримує широкий спектр розширень для різних мов програмування та технологій. Завдяки вбудованим, та доповненим, функціям підсвічування синтаксису, доповнення коду та інтеграції з системами контролю версій, Visual Studio Code значно підвищує швидкість і зручність роботи розробників.

Висновки до першого розділу

У першому розділі було визначено мету створення платформи для управління ІТ-проектами. Описано ключові модулі системи, такі як управління проектами, користувачами, завданнями, а також модулі для роботи в реальному часі, сповіщень та інтеграцій. Визначено етапи розробки.

Під час аналізу аналогів програмного продукту було досліджено особливості та функціональні можливості популярних платформ для управління проектами, таких як Linear, GitHub Projects та Jira виявивши їхні сильні та слабкі сторони. Порівняння рішень дало змогу визначити, які саме функції є критично важливими для ефективної роботи платформи, а також виявити аспекти, де існуючим рішенням бракує зручності, гнучкості чи функціональності. Це стало основою для

формування вимог до власної платформи та допомогло окреслити її ключові переваги.

Визначено архітектуру програмного забезпечення та було обґрунтовано вибір монолітної архітектури з допоміжним сервісом, як оптимального підходу для реалізації платформи.

На основі сформульованих вимог було обрано інструментальні засоби для розробки платформи. Обрані технології, TypeScript, Next.js, Express.js та PostgreSQL, забезпечують надійність, масштабованість та гнучкість під час реалізації як монолітної частини, так і допоміжного сервісу.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						19

РОЗДІЛ 2. АНАЛІЗ НАПРЯМКІВ ВИКОРИСТАННЯ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

2.1. Визначення варіантів використання

Визначення варіантів використання системи дає змогу окреслити основні сценарії взаємодії користувачів із платформою, відповідно до їх ролей та доступних дій. Для візуалізації таких сценаріїв створено діаграми варіантів використання, що охоплюють ключові функціональні області платформи: роботу із завданнями, документацією, коментарями, сповіщеннями, управління Github–інсталяціями, учасниками та робочими просторами.

Кожен користувач робочого простору наділений однаковими правами у керуванні завданнями, а саме їх перегляд, створення та редагування (рис.2.1).



Рис. 2.1 – Діаграма варіантів використання завдань

Кожен користувач робочого простору наділений однаковими правами у керуванні документацією, а саме перегляд, створення, редагування, видалення та синхронізація змін файлу з Github Wiki (рис.2.2).

Кожен користувач робочого простору наділений однаковими правами у коментуванні завдань, а саме перегляд, створення, редагування та видалення (рис.2.3).



Рис. 2.2 – Діаграма варіантів використання документації

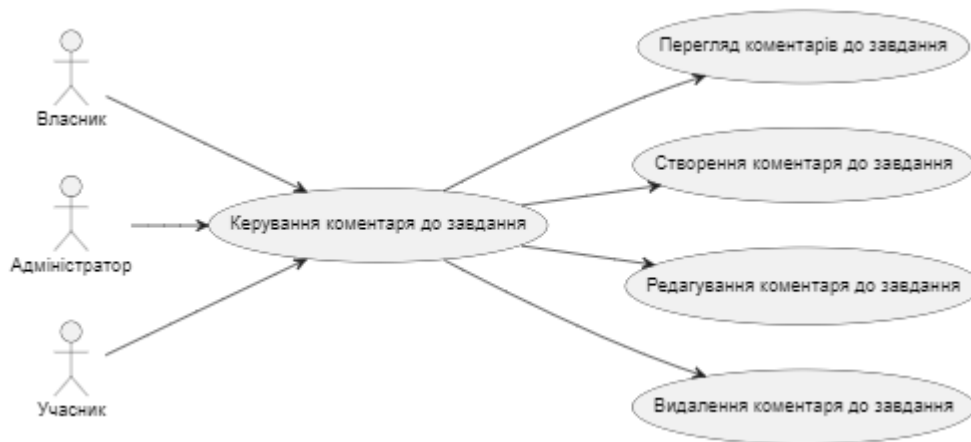


Рис. 2.3 – Діаграма варіантів використання коментарів завдань

Кожен користувач робочого простору наділений однаковими правами в управлінні Github інсталяціями, а саме їх перегляд, створення та видалення (рис.2.4).



Рис. 2.4 – Діаграма варіантів використання Github інсталяцій

Користувач робочого простору може переглядати сповіщення завдання, та відмітити його як прочитане (рис.2.5).

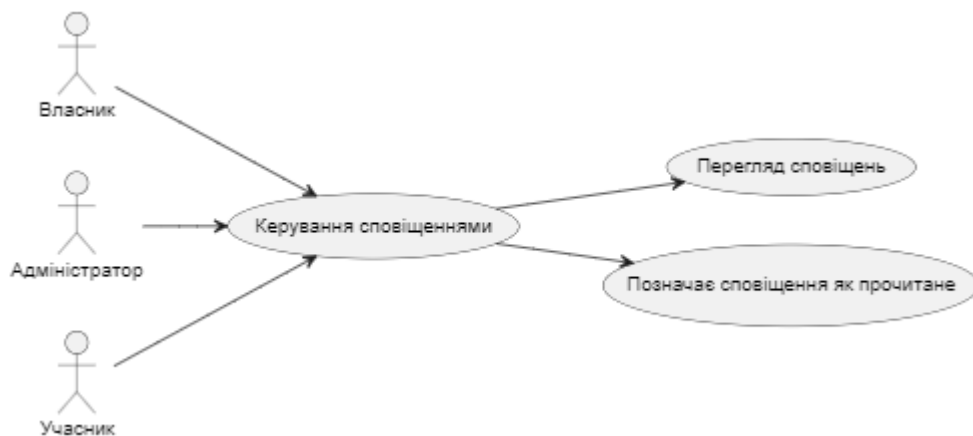


Рис. 2.5 – Діаграма варіантів використання сповіщень

Кожен користувач робочого простору може переглядати його учасників, але лише Власник та Адміністратори мають змогу додавати, видаляти чи змінювати роль учасників (рис.2.6).

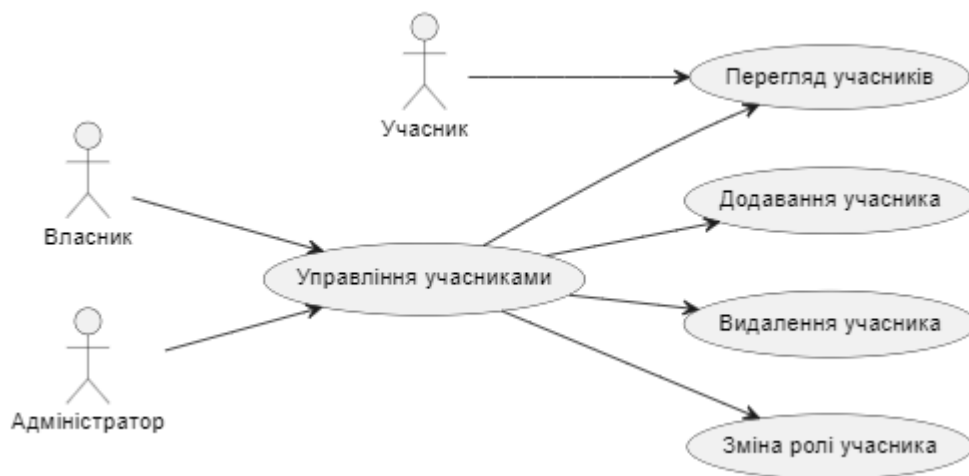


Рис. 2.6 – Діаграма варіантів використання управління учасниками робочого простору

Власник має змогу редагувати робочий простір та видаляти його. Адміністратор також може редагувати простір, та може покинути його. Учасник має змогу лише покинути робочий простір (рис.2.7).

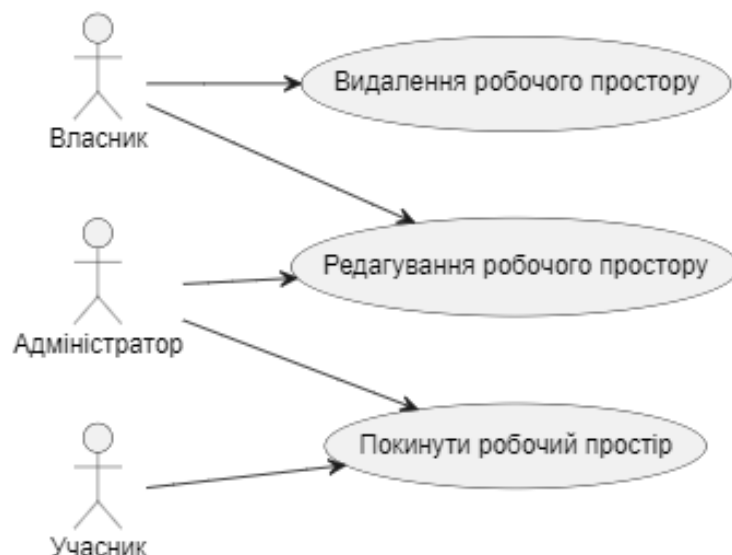


Рис. 2.7 – Діаграма варіантів використання взаємодії з робочим простором

Користувач може як увійти, так і вийти зі свого акаунту, також він може переглядати та редагувати профіль (рис.2.8).

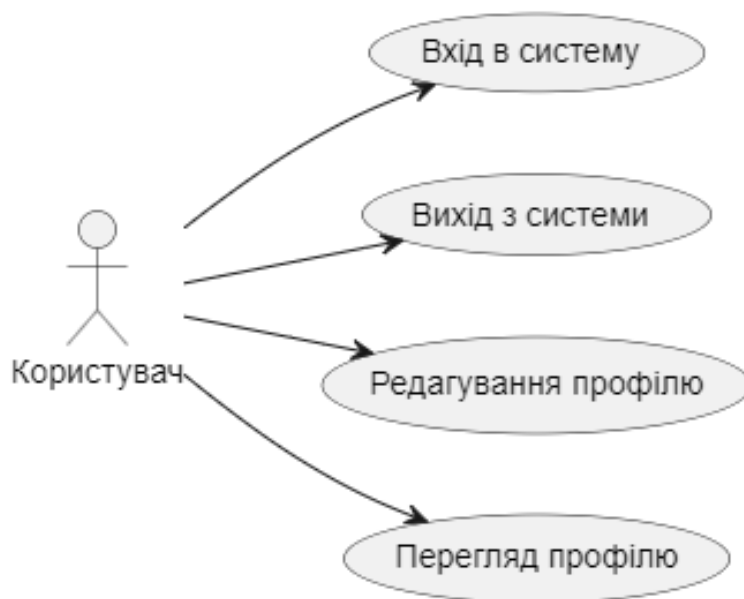


Рис. 2.8 – Діаграма варіантів використання взаємодії з профілем

Описи ключових прецедентів наведено в таблицях 2.1 – 2.10.

Таблиця 2.1

Опис прецеденту «Вхід/Реєстрація»

Назва варіанту використання	Вхід/Реєстрація
Основні актори	Користувач
Короткий опис	Користувачі можуть увійти через email або Github для отримання доступу до робочих просторів. Якщо користувача не існує, він буде створений автоматично

Таблиця 2.2

Опис прецеденту «Створення робочого простору»

Назва варіанту використання	Створення робочого простору
Основні актори	Користувач
Короткий опис	Користувач може створити робочий простір, автоматично здобувши у ньому роль власника

Таблиця 2.3

Опис прецеденту «Запрошення користувача до робочого простору»

Назва варіанту використання	Запрошення користувача до робочого простору
Основні актори	Власник, адміністратор
Короткий опис	Користувачі з відповідними правами можуть надсилати запрошення іншим користувачам для приєднання до робочого простору

Таблиця 2.4

Опис прецеденту «Надання ролі учаснику»

Назва варіанту використання	Надання ролі учаснику
Основні актори	Власник, адміністратор

Короткий опис	Адміністратор або власник можуть або дати іншому користувачу роль адміна, або забрати її
---------------	--

Таблиця 2.5

Опис прецеденту «Створення завдання»

Назва варіанту використання	Створення завдання
Основні актори	Власник, адміністратор, учасник
Короткий опис	Користувачі можуть додавати нові задачі, виставивши пріоритет, статус, та виконавця. Виконавець отримує сповіщення

Таблиця 2.6

Опис прецеденту «Редагування завдання»

Назва варіанту використання	Редагування завдання
Основні актори	Власник, адміністратор, учасник
Короткий опис	Користувачі можуть редагувати завдання, змінивши опис, пріоритет, статус чи виконавця. Виконавець отримує відповідне сповіщення

Таблиця 2.7

Опис прецеденту «Коментування завдання»

Назва варіанту використання	Коментування завдання
Основні актори	Власник, адміністратор, учасник
Короткий опис	Коментарі прив'язані до завдань для обговорення, зворотного зв'язку чи уточнення деталей. Користувачі можуть додавати нові або відповідати на старі.

Таблиця 2.8

Опис прецеденту «Інтеграція з GitHub»

Назва варіанту використання	Інтеграція з GitHub
Основні актори	Власник, адміністратор, учасник
Короткий опис	Користувачі можуть підключити GitHub App до робочого простору для динамічного відстеження завдань та GitHub wiki документації

Таблиця 2.9

Опис прецеденту «Редагування документації»

Назва варіанту використання	Редагування документації
Основні актори	Власник, адміністратор, учасник
Короткий опис	Користувачі можуть додавати, редагувати, видаляти чи синхронізувати з GitHub wiki файли документації з обраного репозиторія.

Таблиця 2.10

Опис прецеденту «Видалення робочого простору»

Назва варіанту використання	Видалення робочого простору
Основні актори	Власник
Короткий опис	Лише власник має можливість повністю видалити робочий простір разом з усіма завданнями та учасниками

2.2. Розробка бази даних системи

Для збереження даних використовується реляційна база даних PostgreSQL, яка складається з 13 таблиць, які забезпечують виконання основної системи.

Усього присутньо чотири переліка, а саме “WorkspaceRole” – ролі користувачів у робочому просторі, “WorkspaceInviteStatus” – статуси запрошення у робочий простір, “IssueStatus” – статуси роботи завдання, “IssuePriority” – пріоритети завдання.

На рисунку 2.9 продемонстровано візуальну взаємодію таблиць та переліків.

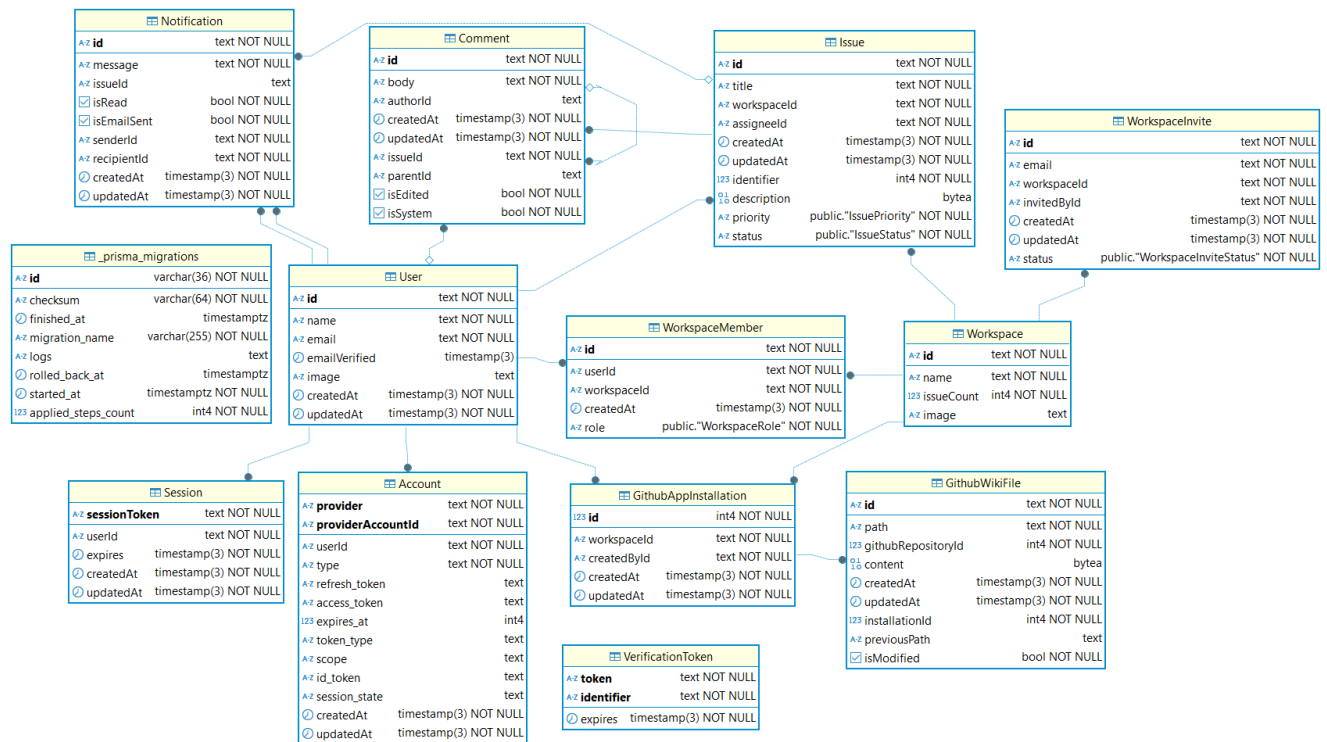


Рис. 2.9 – Візуаліція бази даних

Таблиця “User” зберігає дані користувача (табл. 2.11)

Таблиця 2.11

Опис таблиці User

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
name	text	Ім'я користувача
email	text	Email користувача
emailVerified	timestamp null	Чи верифікований email
image	text null	Аватар користувача
createdAt	timestamp	Дата створення

updatedAt	timestamp	Остання дата оновлення
-----------	-----------	------------------------

Таблиця “Account” зберігає дані провайдера користувача, наприклад GitHub, для автентифікації (табл. 2.12).

Таблиця 2.12

Опис таблиці Account

Поле	Тип	Призначення
provider (PK)	text	Назва провайдера
providerAccountId (PK)	text	Унікальний ідентифікатор облікового запису провайдера
userId	text	Ідентифікатор користувача
type	text	Тип облікового запису
refresh_token	text null	Токен для оновлення access_token
access_token	text null	Токен доступу до провайдера
expires_at	int null	Час закінчення дії токена доступу (відраховується у UNIX)
token_type	text null	Тип токена
scope	text null	Сфера дії токена
id_token	text null	Ідентифікаційний токен
session_state	text null	Стан сесії
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “Session” містить токени автентифікованих користувачів (табл. 2.13).

Таблиця 2.13

Опис таблиці Session

Поле	Тип	Призначення
sessionToken (PK)	text	Токен сесії
userId	text	Ідентифікатор користувача
expires	timestamp	Час закінчення дії токена доступу
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “VerificationToken” містить токени для підтвердження пошти (табл. 2.14).

Таблиця 2.14

Опис таблиці VerificationToken

Поле	Тип	Призначення
identifier (PK)	text	Ідентифікатор користувача
token (PK)	text	Токен
expires	timestamp	Час закінчення дії токена доступу

Таблиця “Workspace” містить інформацію про робочий простір (табл. 2.15).

Таблиця 2.15

Опис таблиці Workspace

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
name	text	Назва робочого простору
image	text null	Аватар робочого простору

issueCount	int	Кількість створених завдань у робочому місці
------------	-----	--

Таблиця “WorkspaceMember” містить користувачів робочих просторів та їх ролі (табл. 2.16).

Таблиця 2.16

Опис таблиці WorkspaceMember

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
role	text	Роль користувача у робочому місці
userId	text	Ідентифікатор користувача
workspaceId	text	Ідентифікатор робочого простору
createdAt	timestamp	Дата створення

Таблиця “WorkspaceInvite” містить пошти користувачі, яких запросили у робочі простори (табл. 2.17).

Таблиця 2.17

Опис таблиці WorkspaceInvite

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
email	text	Пошта запрошеного користувача
workspaceId	text	Ідентифікатор робочого простору
invitedById	text	Ідентифікатор користувача, який запросив

Продовження таблиці 2.17

status	WorkspaceInviteStatus	Статус запрошення
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “Issue” містить завдання, та дані про них (табл. 2.18).

Таблиця 2.18

Опис таблиці Issue

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
identifier	int	Ідентифікатор у робочому просторі
title	text	Назва завдання
description	bytea null	Опис завдання, у форматі зрозумілому для колобораційного редактору текста
status	IssueStatus	Статус завдання
priority	IssuePriority	Пріоритет завдання
workspaceId	text	Ідентифікатор робочого простору
assigneeId	text	Ідентифікатор користувача, якому призначено завдання
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “Comment” містить коментарі до завдань (табл. 2.19).

Опис таблиці Comment

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
body	text	Текст коментаря
issueId	text	Ідентифікатор завдання
authorId	text null	Ідентифікатор автора
parentId	text null	Ідентифікатор батьківського коментаря
isEdited	bool	Чи редагувався коментар
isSystem	bool	Чи це системний коментар
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “Notification” містить сповіщення до завдань (табл. 2.20).

Таблиця 2.20

Опис таблиці Notification

нПоле	Тип	Призначення
id (PK)	text	Ідентифікатор
message	text	Повідомлення
issueId	text null	Ідентифікатор завдання
isRead	bool	Чи прочитане
senderId	text	Ідентифікатор відправника
recipientId	text	Ідентифікатор отримувача
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “GithubAppInstallation” містить ідентифікатори завантажених GitHub App на GitHub акаунти (табл. 2.21).

Таблиця 2.21

Опис таблиці GithubAppInstallation

Поле	Тип	Призначення
id (PK)	int	Ідентифікатор
workspaceId	text	Ідентифікатор робочого простору
createdById	text	Ідентифікатор користувача, який підключив GitHub App
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Таблиця “GithubWikiFile” містить змінні файли документації (табл. 2.22).

Таблиця 2.22

Опис таблиці GithubWikiFile

Поле	Тип	Призначення
id (PK)	text	Ідентифікатор
path	text	Назва файлу
previousPath	text null	Назва файлу до змін
installationId	int	Ідентифікатор GithubAppInstallation
githubRepositoryId	int	Ідентифікатор GitHub репозиторію
content	bytea null	Вміст файлу, у форматі зрозумілому для колобораційного редактору текста

isModified	bool	Чи змінювався вміст файлу
createdAt	timestamp	Дата створення
updatedAt	timestamp	Остання дата оновлення

Використовуючи такі таблиці, як User, Account, Session та VerificationToken, реалізовано механізм автентифікації користувачів. Таблиці Workspace, WorkspaceMember та WorkspaceInvite були використані для реалізації логіки робочих просторів. Таблиці Issue, Comment та Notification необхідні для роботи над завданнями, обговорення та сповіщення в межах робочого простору. Таблиці GithubAppInstallation та GithubWikiFile необхідні для роботи над документацією. Остання таблиця яка не була перерахована, це _prisma_migrations, яка була автоматично згенерована для міграцій.

2.3. Проектування та реалізація алгоритмів роботи системи

Для отримання доступу до основного функціоналу платформи, користувачам необхідно автентифікуватися. Цей процес зображений на рисунку 2.10.

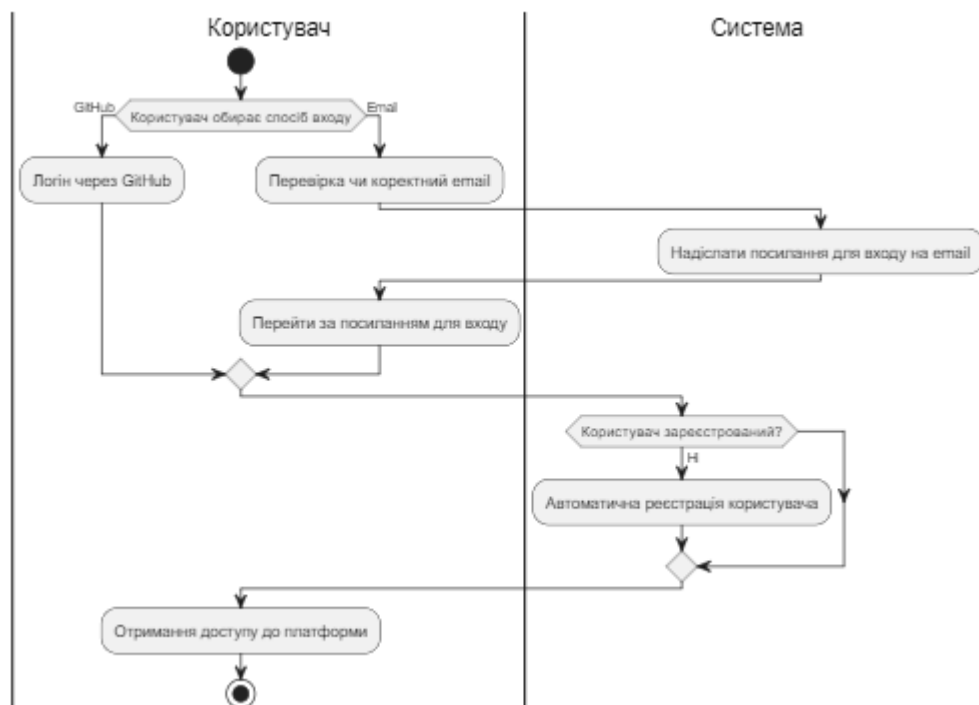


Рис. 2.10 – Діаграма активності для авторизації

Під час автентифікації користувачу необхідно обрати, чи бажає він виконати цей процес за допомогою електронної пошти, чи за допомогою GitHub. Якщо ж було обрано email, необхідно підтвердити його, натиснувши на посилання, яке надійде на пошту. У випадку GitHub, потрібно обрати обліковий запис, з якого користувач бажає увійти. Якщо ж користувач уперше авторизувався з такою поштою, то він буде автоматично зареєстрований.

Наступна діаграма активності на рисунку 2.11 демонструє процес створення нового робочого простору користувачем. Користувачу необхідно ввести назву майбутнього робочого простору та надіслати дані. Система, у свою чергу, додає цей робочий простір та надає користувачу роль власника у ньому. Користувач отримує підтвердження та потрапляє у новостворений робочий простір.

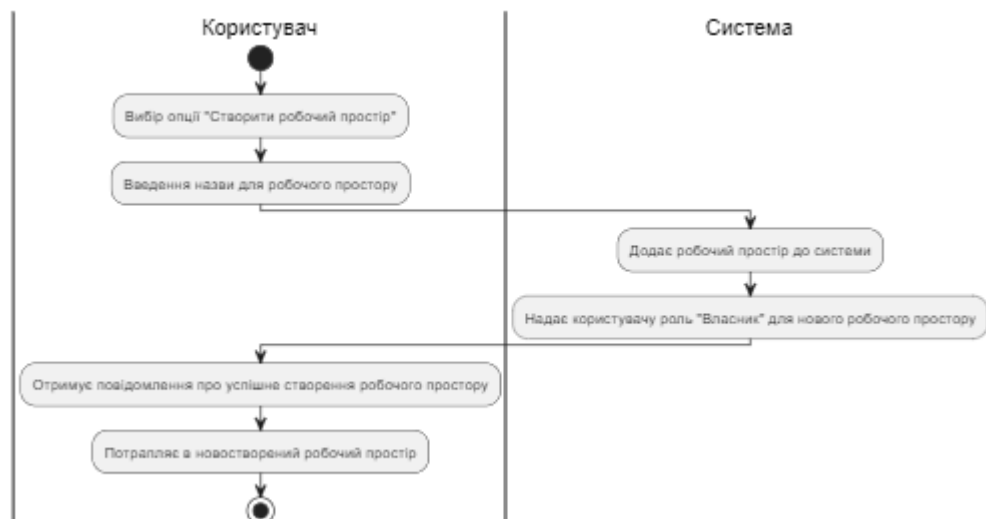


Рис. 2.11 – Діаграма активності для створення робочого простору

На рисунку 2.12 зображено діаграму активності для процесу редагування завдання. Користувач натискає на необхідне завдання зі списку, отримуючи змогу редагувати його. Він може як змінити пріоритет, статус та виконавця завдання, так і у колабораційному редакторі оновити опис. Система автоматично зберігає зміни, створює сповіщення про зміну виконавця, та по вебсокетах повідомляє інших користувачів про цю зміну.



Рис. 2.12 – Діаграма активності для редагування завдання

Наступна діаграма активності на рисунку 2.13 демонструє процес додавання нового учасника у команду. Для того, щоб працювати над одним проєктом у команді, адміністратору чи власнику необхідно додати користувачів у команду. Для цього необхідно ввести пошту нового учасника, якщо учасник із цією поштою вже у команді чи у запрошених, повертається помилка, у іншому випадку система зберігає запрошення, надсилає email запрошення на надану пошту.

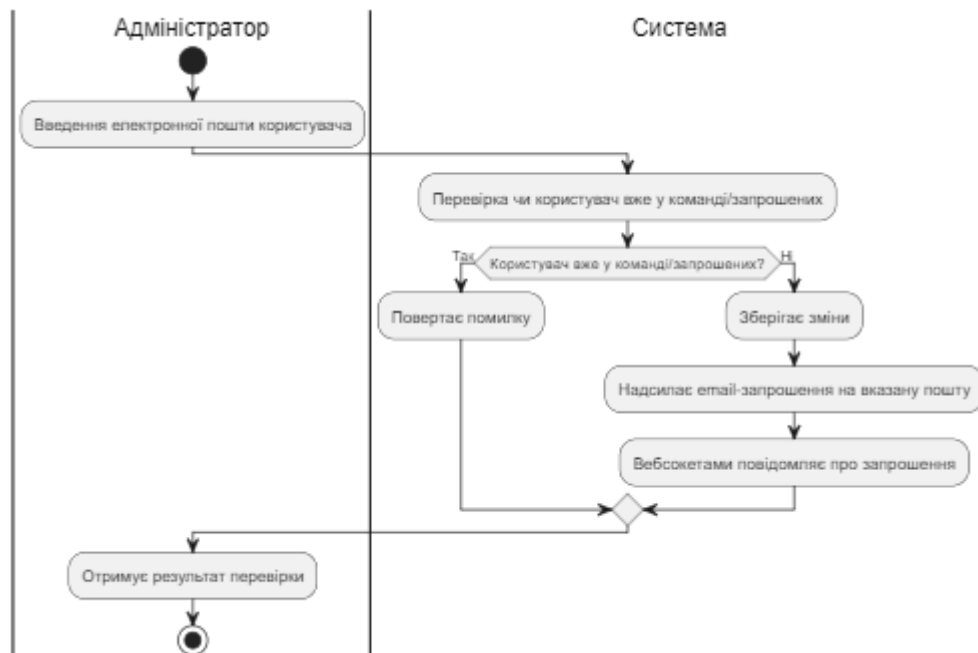


Рис. 2.13 – Діаграма активності для додавання нового користувача у проєкт

Діаграма на рисунку 2.14 ілюструє процес отримання доступу до документації. Щоб відображати вміст, робочий простір повинен мати хоча б одну інтеграцію з GitHub і прив'язані до неї репозиторії. Далі користувач має обрати репозиторій. Якщо ж у репозиторії вимкнена GitHub Wiki, буде виведене відповідне повідомлення.

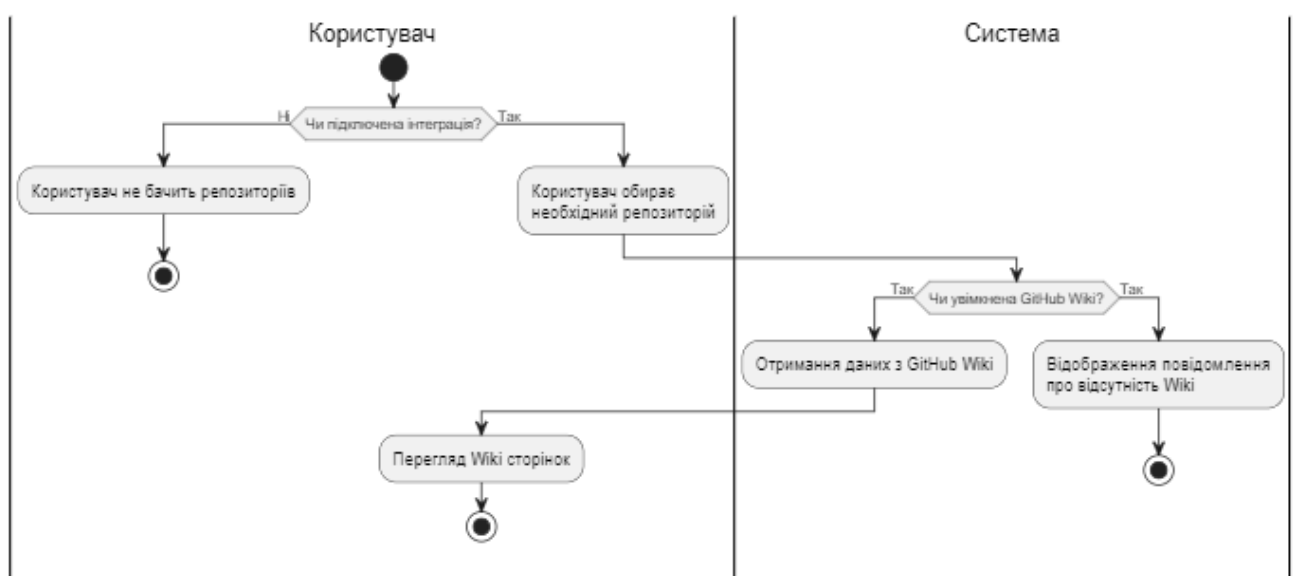


Рис. 2.14 – Діаграма активності для перегляду документації

На рисунку 2.15 зображено процеси взаємодії з документацією. Усі зміни відбуваються у реальному часі. Будь-яка дія користувача, створення, редагування та видалення, відбувається у реальному часі. Після створення або редагування сторінки необхідно натиснути кнопку синхронізації, щоб зміни з'явилися у GitHub Wiki. Лише видалення синхронізується автоматично.

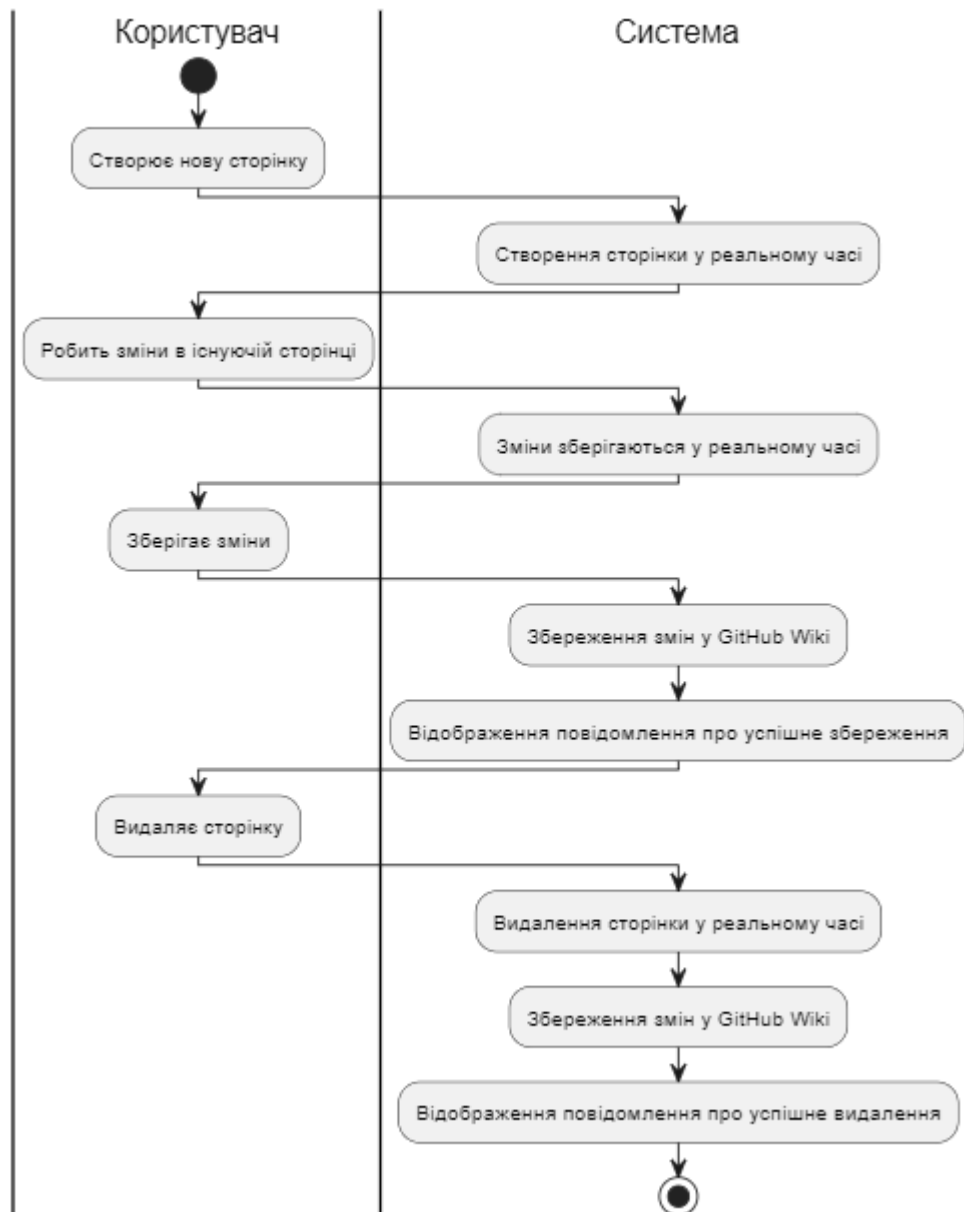


Рис. 2.15 – Діаграма активності для документації

2.4. Реалізація функціоналу системи

Оскільки у проєкті використовується Next.js, використаємо його найбільшу перевагу, а саме серверний рендеринг. У прикладі серверного компонента

DashboardPage дані про завдання завантажуються безпосередньо на сервері. Отримані дані одразу передаються до клієнтського компонента IssuesTable, що дозволяє зменшити час завантаження сторінки.

```
export default async function DashboardPage(
  props: Types) {
  const params = await props.params;
  ...
  return <IssuesTable issues={issues} />;
}
```

У проєкті для взаємодії з сервером використовуються Server Actions – функції, які виконуються безпосередньо на сервері.

```
'use server';
export async function deleteWorkspace(id: string) {
  ...
  return response;
}
```

ORM Prisma [9] дозволила взаємодіяти з базою даних у типобезпечний спосіб. Основою для опису структури таблиць є Prisma schema, в якій визначаються моделі. Наприклад, модель Workspace містить поля з типами, а також зв'язки з іншими моделями, що дозволяє формувати реляційні зв'язки між сутностями. Для автоматичної генерації унікального ідентифікатора використовується @default(cuid()), а кількість задач у робочому просторі за замовчуванням дорівнює нулю (@default(0)).

```
/// @zod.strip()
model Workspace {
  id          String          @id @default(cuid())
  /// @zod.string.trim().min(1).max(255)
  name        String
  image        String?
  issueCount   Int            @default(0)
  members      WorkspaceMember[]
  issues       Issue[]
  workspaceInvite WorkspaceInvite[]
  githubAppInstallation GithubAppInstallation[]
}
```

Для підключення Prisma в проєкті використовується функція prismaClientSingleton, яка створює екземпляр PrismaClient та розширює його поведінку. Наприклад, при створенні нового користувача, якщо ім'я не було передане, воно автоматично генерується на основі email-адреси.

```
const prismaClientSingleton = () => {
  return new PrismaClient().$extends({
    query: {
      user: {
        async create({ args, query }) {
          args.data.name ??= args.data.email.split('@')[0];
          return query(args);
        },
      },
    },
  });
};
```

Генератор `zod-prisma-types` [10] дозволив автоматично створює схеми `Zod` [11] на основі моделей `Prisma`. Він додається до `Prisma schema` через блок `generator zod`, де вказується провайдер, директорія для згенерованих файлів (`./zod`), та додаткові опції.

```
generator zod {
  provider           = "zod-prisma-types"
  output             = "./zod"
  createRelationValuesTypes = true
  createPartialTypes   = true
  useTypeAssertions    = true
}
```

Була реалізована універсальна функція `zodValidate`, яка застосовується для валідації даних у всіх `Server Actions`. Вона приймає схему, згенеровану на основі моделей `Prisma`, і перевіряє отримані дані через `safeParse`. У разі помилки функція генерує виняток з описом усіх повідомлень про помилки, а у випадку успішної перевірки повертає валідовані дані.

```
export function zodValidate<T>(schema: ZodType<T>, data: unknown) {
  const validatedFields = schema.safeParse(data);
  if (!validatedFields.success)
    throw new Error(
      validatedFields.error.errors.map(e => e.message).join(', ')
    );
  const { data: validatedDate } = validatedFields;

  return validatedDate;
}
```

Для отримання та поділу на клієнт/сервер змінних середовища використано `@t3-oss/env-nextjs` [12] бібліотеку, а для валідації вхідних змінних `Zod`.

```
import { createEnv } from '@t3-oss/env-nextjs';
import { z } from 'zod';
```

```

export const env = createEnv({
  clientPrefix: 'NEXT_PUBLIC_',
  server: {
    NODE_ENV: z
      .enum(['development', 'test', 'production'])
      .default('development'),
    DATABASE_URL: z.string().min(1),
    ...
  },
  client: {
    ...
    NEXT_PUBLIC_GITHUB_APP_NAME: z.string().min(1).default('progresshubapp'),
  },
  runtimeEnv: {
    NODE_ENV: process.env.NODE_ENV,
    DATABASE_URL: process.env.DATABASE_URL,
    ...
    NEXT_PUBLIC_GITHUB_APP_NAME: process.env.NEXT_PUBLIC_GITHUB_APP_NAME,
  },
});

```

Обробка зображень відбувається за допомог бібліотеки sharp [13], яка дозволяє змінювати розмір файлу та конвертувати його у формат WebP. У функції uploadImage зображення спочатку зчитується як масив байтів, після чого зменшується до 256×256 пікселів і перетворюється у WebP. Після оптимізації файл передається в uploadthing [14] за допомогою utapi.uploadFiles, що відповідає за збереження файлу.

```

export async function uploadImage(name: string, file: File) {
  const optimizedFile = await sharp(await file.arrayBuffer())
    .resize(256, 256)
    .webp()
    .toArray();

  const utFile = new UTFile(optimizedFile, name);
  const response = await utapi.uploadFiles(utFile);

  return response.data?.key;
}

```

Для UI компонентів було використано бібліотеку shadcn/ui [15], яка на відміну від інших бібліотек, не встановлюється у залежності проєкту, а на вимогу копіюється файлом у код, що дозволяє повне редагування компонента (рис.2.16).

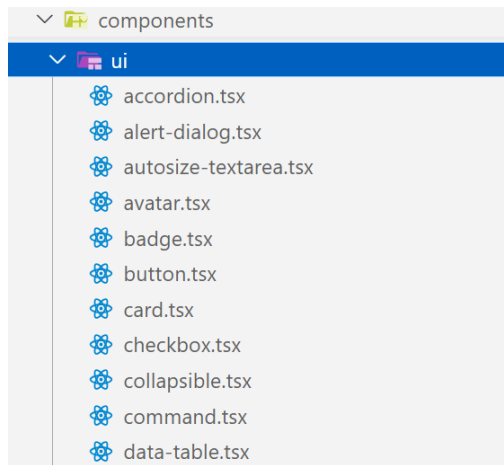


Рис. 2.16 – Структура компонентів shadcn/ui

Автентифікація відбувається через бібліотеку next-auth [16]. Ця бібліотека може автоматично зберігати дані через Prisma, та підтримує автентифікацію через різних провайдерів, серед них Nodemailer, необхідний для автентифікації за посиланням на пошту, та GithubProvider. У проєкті було використано Session Based authentication, яка забезпечує зберігання стану автентифікації на сервері.

```
NextAuth({
  adapter: PrismaAdapter(prisma),
  providers: [
    Nodemailer({
      server: {
        ...
      }
    }),
    GithubProvider({
      clientId: env.GITHUB_CLIENT_ID,
      clientSecret: env.GITHUB_CLIENT_SECRET,
    }),
  ],
});
```

У додатку А описано логіку для авторизації на серверних функціях, де відбувається отримання автентифікованого користувача через next-auth. Далі через Prisma відбувається перевірка, чи є користувач у необхідному робочому просторі, для отримання необхідного ресурсу. Після цього відбувається перевірка, чи підходить роль, необхідна для серверної функції.

Як редактор тексту був підібраний фреймворк Tiptar [17]. Цей фреймворк вирізняється високим рівнем кастомізації, а саме можна змінювати функціональність, стилі та навіть самі компоненти відповідно до потреб (рис.2.17).

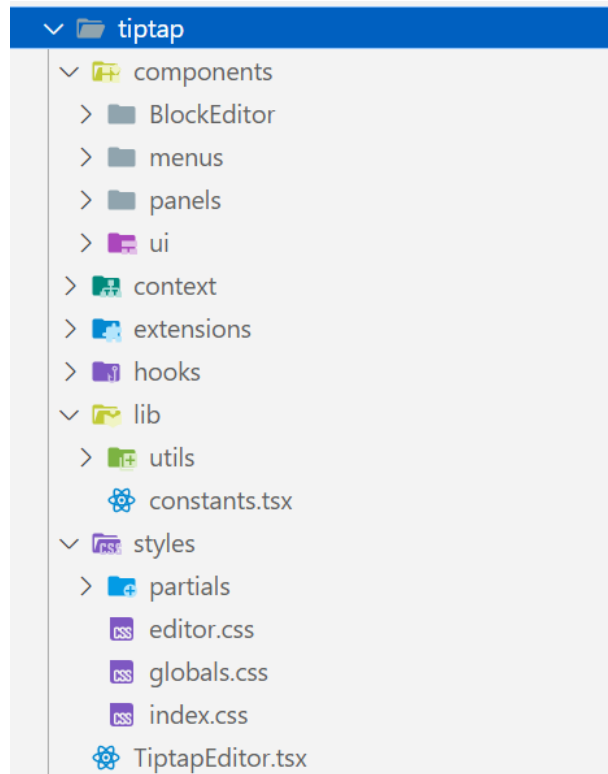


Рис. 2.17 – Tiptap редактор

Для інтеграції колабораційності у Tiptap було додано бібліотеку Hocuspocus [18], яка поєднує WebSocket обробку, для роботи у реальному часі, та Yjs [19], для об'єднання змін. Для зберігання змін використано Prisma.

```
const hocuspocusServer = Server.configure({
  extensions: [
    new Logger(),
    new Database({
      fetch: async ({ documentName }) => {
        ...
      },
      store: async ({ documentName, state }) => {
        ...
      },
    }),
  ],
  async onAuthenticate({ token }) {
    jwt.verify(token, env.TIPTAP_COLLAB_SECRET ?? 'secret');
  },
});
```

Оновлення змін у реальному часі відбувається у кілька етапів. Перш за все було обрано Socket.IO [20], як основний обробник змін. Перш за все, аби отримувати зміни, необхідно під'єднатися до необхідної кімнати.

```
io.on('connection', socket => {
```

```

socket.on('join', (room: unknown) => {
  if (typeof room !== 'string') throw new Error('Room must be a string');
  socket.join(room);
  console.log(`Socket ${socket.id} joined room ${room}`);
});
...
});

```

Зміни для поширення отримуються на сервері express, де валідується структура, та передається далі підключеним до цієї кімнати користувачам.

```

router.post('/notify', (req, res) => {
  const { room, entity, event, payload } = notifySchema.parse(req.body);

  emitToRoomChanges(
    room,
    entity,
    event,
    payload as typeof payload & { id: string }
  );

  res.send('OK');
});

```

На клієнті зміни у реальному часі оброблюються через хук useSocketObserver, де потрібна назва події, за якою відслідковувати, та масив, де у кожного елемента має бути id, за яким можна робити оновлення та видалення.

```

export function useSocketObserver<T extends { id: string }>(
  eventName: entities,
  initValues: T[],
  onDelete?: () => void
) {
  const { socket } = useSocket();
  const [state, setState] = useState<T[]>(initValues);

  useEffect(() => {
    ...

    socket.on(`${eventName}/update`, handleUpdate);
    socket.on(`${eventName}/create`, handleCreate);
    socket.on(`${eventName}/delete`, handleDelete);

    return () => {
      socket.off(`${eventName}/update`, handleUpdate);
      socket.off(`${eventName}/create`, handleCreate);
      socket.off(`${eventName}/delete`, handleDelete);
    };
  }, [socket, eventName, onDelete, router]);
}

```

```

return state;
}

```

Для відстеження змін у репозиторії (створення гілок, відкриття чи закриття Pull request) або для додавання змін було створено Github App [21], яку користувачу потрібно встановити у необхідний GitHub акаунт чи організацію (рис.2.18).

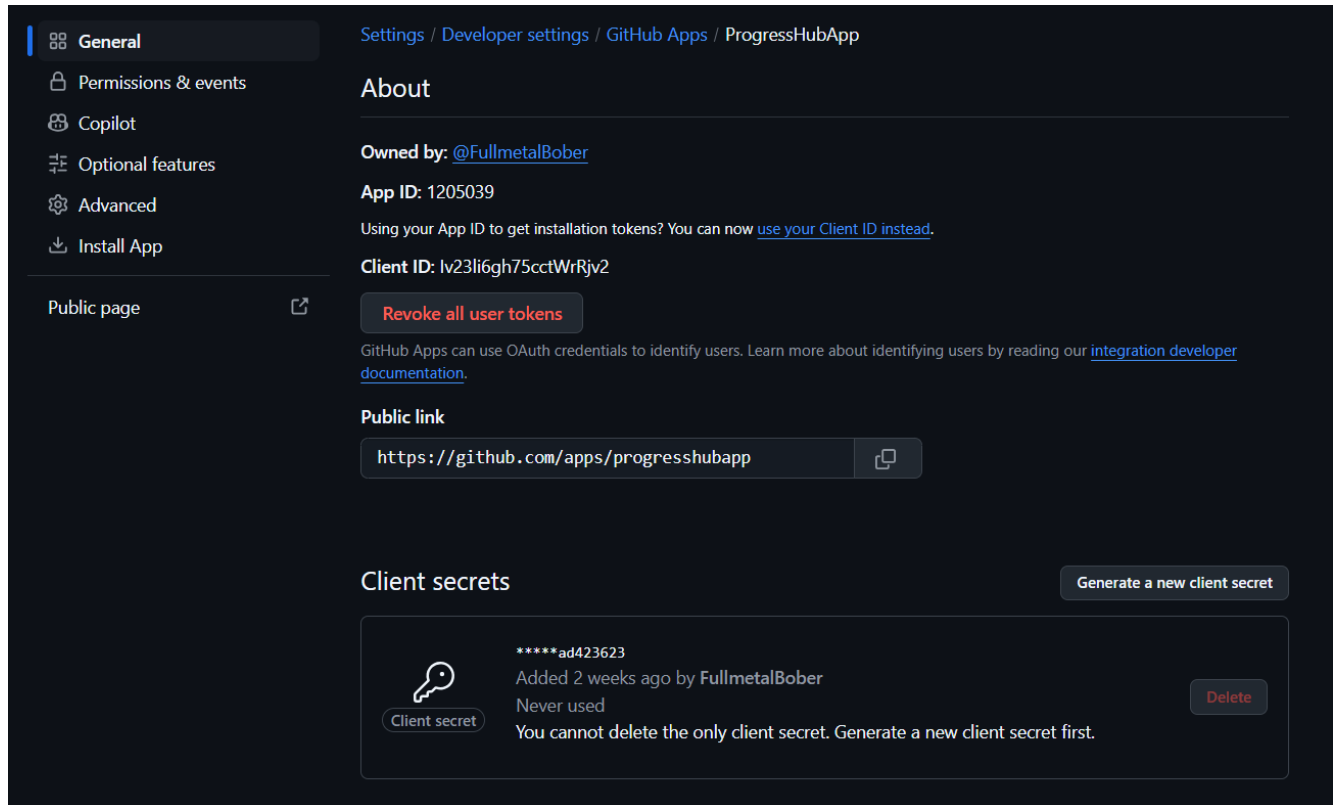


Рис. 2.18 – Створена GitHub App

Зміни з GitHub App передаються по вебхукам, для їх знаходження було використано фреймворк Probot [22], який спрощує взаємодію з GitHub вебхуками. На приклад, ми підписалися на подію створення, і якщо було створенно гілку, ми переходимо у функцію, яка відповідає за отримання завдання, якщо таке є, та зміни його статусу.

```

function handle(app: Probot) {
  app.on('create', async context => {
    const { ref_type, ref, installation } = context.payload;
    if (!installation) return;

    if (ref_type === 'branch') {
      const branchName = ref;

      moveIssueTo(
        installation.id,
        branchName,

```

```

        `Branch ${branchName} created`,
        'IN_PROGRESS'
    );
}
});
}

```

Octokit [23] – це бібліотека, створена для комунікації з GitHub, оскільки вона вбудована у Probot, було прийнято рішення використовувати саме її. Octokit була корисна як для відображення встановлених GitHub App у робочий простір, їх видалення звідти, перегляду репозиторіїв користувача та отримання токена для pull/push.

```

const localOctokit = new ProbotOctokit({
  auth: {
    appId: env.GITHUB_APP_ID,
    privateKey: env.GITHUB_PRIVATE_KEY,
    clientId: env.GITHUB_CLIENT_ID,
    clientSecret: env.GITHUB_CLIENT_SECRET,
    type: 'installation',
    installationId,
  },
});

```

Оскільки до GitHub Wiki не можна отримати доступ через Octokit, чи GitHub Api, було прийнято рішення використовувати Git [24] систему. Через Git перевіряється, чи такий репозиторій існує, виконується клонування, отримання та відправлення змін.

```

const gitClone = simpleGit(basePath, gitBaseConfig);
await gitClone.clone(
  `https://x-access-token:${token}@github.com/${repoFullName}.wiki.git`,
  path
);

```

З часом велика кількість сповіщень почне переповнювати базу даних та інтерфейс користувачів. Для вирішення цієї проблеми було додано бібліотеку cron, що періодично очищатиме застарілі сповіщення [25]. Бібліотека запускати скрипт у певні моменти часу, чи через певний період. Таким чином система видалятиме повідомлення, які прочитані більш 3 днів тому чи коли у користувача у робочому просторі більше 150 сповіщень.

```

const cleanTooManyNotifications = CronJob.from({
  cronTime: '0 */12 * * *', // Every 6 hours
  onTick: async () => {

```

```

    console.log('Cleaning too many notifications');
...
  },
  runOnInit: true,
});

```

Висновки до другого розділу

Було визначено варіанти використання платформи відповідно до ролей користувачів, побудовано діаграми варіантів використання для ключових функцій системи та наведено описи основних прецедентів.

Структура бази даних платформи побудована на реляційній моделі PostgreSQL і охоплює ключові сутності, необхідні для повноцінної реалізації функціоналу.

Було спроектовано ключові алгоритми роботи системи, що охоплюють ключові процеси. Для кожного з цих процесів побудовано відповідні діаграми активності.

Було реалізовано основний функціонал системи з урахуванням особливостей фреймворку Next.js, зокрема використано серверний рендеринг та Server Actions для оптимізації взаємодії з сервером. Для роботи з базою даних застосовано Prisma, яка автоматично генерує Zod схеми. Здійснено автентифікації через next-auth, реалізовано перевірку прав доступу, а також додано підтримку реального часу через socket.io. Виконано обробку зображень, спрощено керування змінними середовища та підключено Shadcn/ui. Забезпечено колаборативне редагування через редактор Tiptap та його взаємодію з GitHub App, що дозволяє синхронізувати зміни в репозиторії документації. Були налаштовані вебхуки GitHub App для змін статусів завдань відповідно до роботи у репозиторії.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ПЛАТФОРМОЮ

3.1. Порядок встановлення та налаштування параметрів платформи

Для встановлення та налаштування платформи необхідно розгорнути компоненти та підключити зовнішні сервіси, необхідні для коректного функціонування системи.

На рисунку 3.1 зображена діаграма розгортання, на якій візуалізовано взаємодію користувача, серверних частин та зовнішніх сервісів.

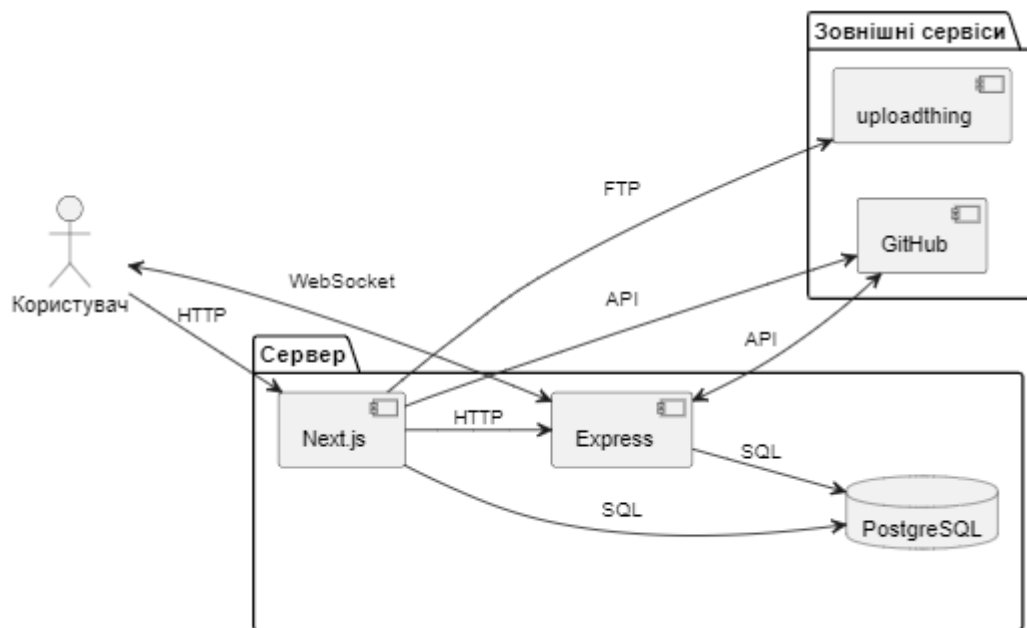


Рис. 3.1 – Діаграма розгортання платформи

У більш детальному огляді діаграми розгортання можемо побачити, що дії, які виконує користувач, це HTTP запити до Next.js, як то отримання сторінок. Користувач також під'єднаний по WebSocket до Express, де у двосторонньому режимі передаються зміни редактора, а у односторонньому з Express до користувача інші зміни у реальному часі, які у свою чергу отримуються від Next.js.

Next.js та Express мають доступ до PostgreSQL та GitHub. GitHub має змогу надсилати запити лише у Express. Лише Next.js має змогу надсилати файли у uploadthing.

Розгортання серверних компонентів платформи було здійснено за допомогою Coolify [26] – платформи для автоматизованого управління інфраструктурою, яка підтримує розміщення вебдодатків у контейнерах. У даному випадку за допомогою

Coolify було реалізовано розгортання бази даних PostgreSQL (рис.3.2) та обох основних серверних частин: Next.js [27] та Express [28], кожна з яких була контейнеризована через Docker.

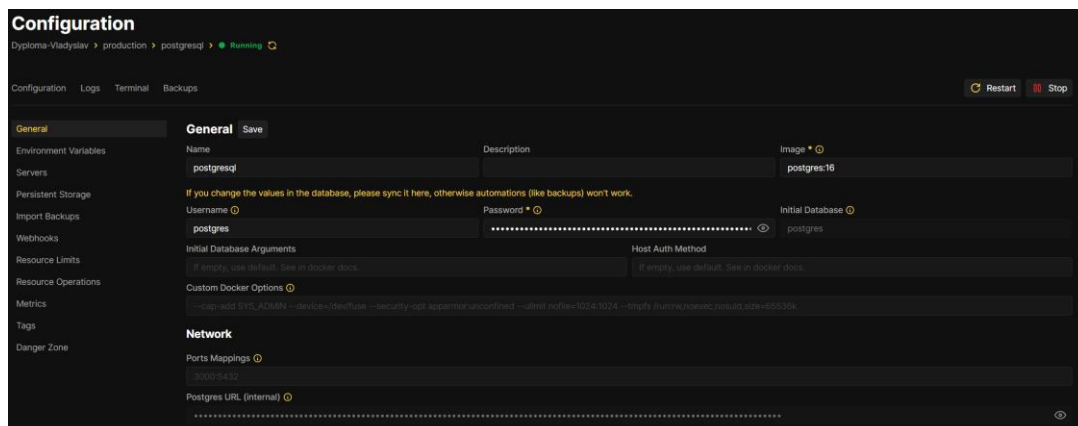


Рис. 3.2 – Розгорнутий PostgreSQL у Coolify

Для зручного доступу, до серверних частин було підключено доменне ім'я. З'єднання до серверних частин було захищене по SSL-сертифікату через інтеграцію Let's Encrypt, що дозволило застосунку працювати по HTTPS. Платформа Coolify самостійно генерує та оновлює сертифікати (рис.3.3).

```
Container Labels
1  traefik.enable=true
2  traefik.http.middlewares.gzip.compress=true
3  traefik.http.middlewares.redirect-to-https.redirectscheme.scheme=https
4  traefik.http.routers.http-0-zww0o0g8w044ogw8owsk0k0s.entryPoints=http
5  traefik.http.routers.http-0-zww0o0g8w044ogw8owsk0k0s.middlewares=redirect-to-https
6  traefik.http.routers.http-0-zww0o0g8w044ogw8owsk0k0s.rule=Host(`progresshub.mazepost.com`) && PathPrefix(`/`)
7  traefik.http.routers.http-0-zww0o0g8w044ogw8owsk0k0s.service=http-0-zww0o0g8w044ogw8owsk0k0s
8  traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.entryPoints=https
9  traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.middlewares=gzip
10 traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.rule=Host(`progresshub.mazepost.com`) && PathPrefix(`/`)
11 traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.service=https-0-zww0o0g8w044ogw8owsk0k0s
12 traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.tls.certresolver=letsencrypt
13 traefik.http.routers.https-0-zww0o0g8w044ogw8owsk0k0s.tls=true
14 traefik.http.services.http-0-zww0o0g8w044ogw8owsk0k0s.loadbalancer.server.port=3000
15 traefik.http.services.https-0-zww0o0g8w044ogw8owsk0k0s.loadbalancer.server.port=3000
```

Рис. 3.3 – Детальні налаштування розгорнутого сервісу

3.2. Структура інтерфейсу платформи

Взаємодія користувача відбувається через вебдодаток, і при відвідуванні користувача зустрічає головна сторінка (рис.3.4).

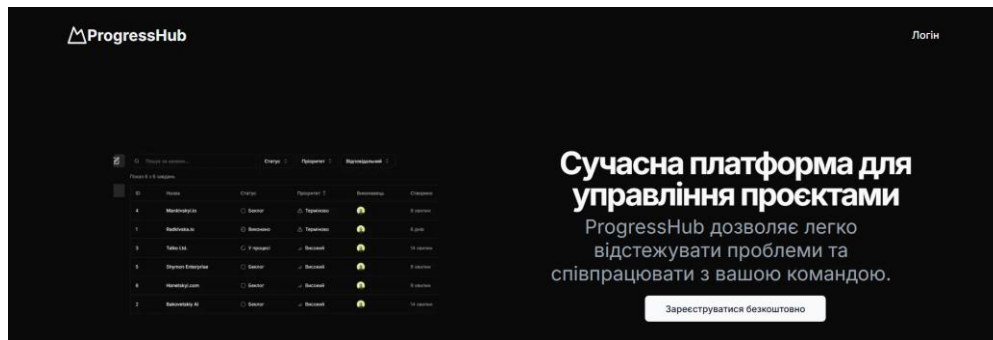


Рис. 3.4 – Головна сторінка платформи

При натисканні кнопки “Логін”, користувач потрапляє на відповідну сторінку з формою, де може або автентифікуватися через пошту, або через GitHub (рис.3.5).

Рис. 3.5 – Форма авторизації

При обранні автентифікації через пошту, користувач отримуємо електронного листа, де при натисканні на кнопку користувач потрапляє в систему (рис.3.6).

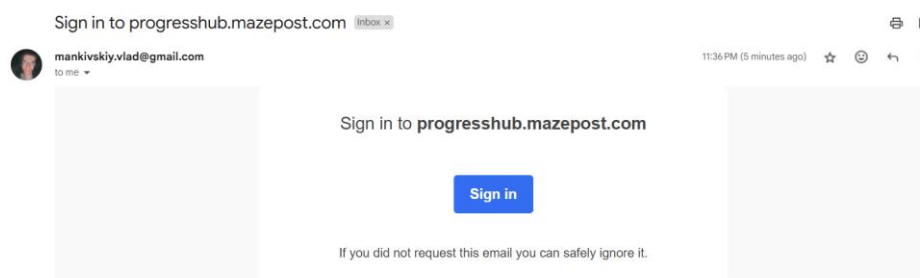


Рис. 3.6 – Електронний лист автентифікації

При обранні автентифікації через GitHub, користувач потрапляє на сторінку, де потрібно підтвердити цю дію (рис.3.7).

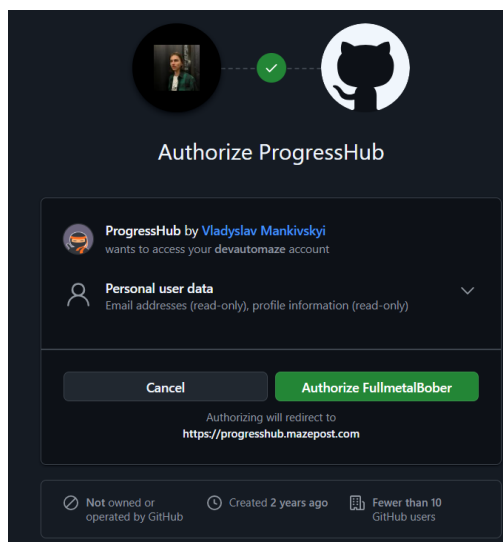


Рис. 3.7 – Підтвердження авторизації на сторінці GitHub

Після успішної автентифікації користувач потрапляє на сторінку з формою створення робочого простору (рис.3.8).

Рис. 3.8 – Форма створення нового робочого простору

Після створення нового робочого простору, користувач бачить сторінку з порожніми завданнями (рис.3.9).

My Workspace

Сповіщення

Завдання

Вікі

Пошук за назвою...

Статус

Пріоритет

Відповідальний

Показ 5 з 5 завдань

ID	Назва	Статус ↑	Пріоритет	Виконавець	Створено	Оновлено
2	My issue 2	○ Тудушка	▲ Високий		2 хвилини	2 хвилини
3	My issue 3	○ Тудушка	▲ Середній		2 хвилини	2 хвилини
4	My issue 4	○ У процесі	▲ Терміново		1 хвилина	1 хвилина
5	My issue 5	○ У процесі	▲ Низький		менше хвилини	менше хвилини
1	My issue 1	○ Беклог	--- Без пріоритету		4 хвилини	4 хвилини

Рис. 3.9 – Сторінка новоствореного робочого простору

При кліку на назву робочого простору, користувач бачить меню, де може перейти чи в налаштування, чи на сторінки створення та приєднання до робочих просторів (рис.3.10).

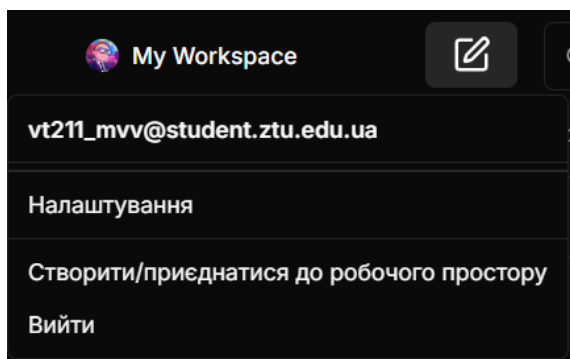


Рис. 3.10 – Меню робочого простору

При натисканні на кнопку “Налаштування” користувач перейде на сторінку налаштувань робочого простору, де далі можемо перейти до управління учасників. На поточній сторінці можна змінити логотип та назву робочого простору (рис.3.11).

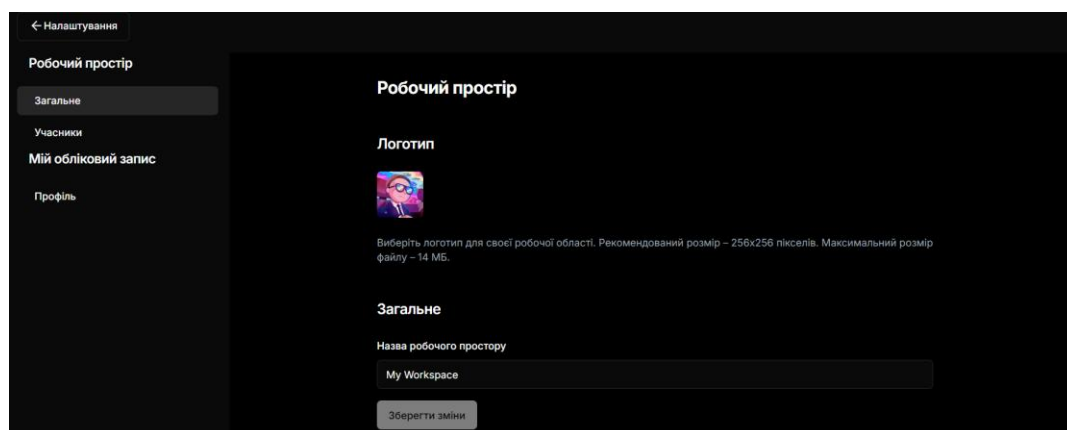


Рис. 3.11 – Сторінка налаштування робочого простору

Якщо на сторінці налаштування робочого простору прогорнути нижче, можна побачити решту налаштувань, де одне це GitHub інтеграція, необхідна для документації, а інше це видалення робочого простору (рис.3.12).

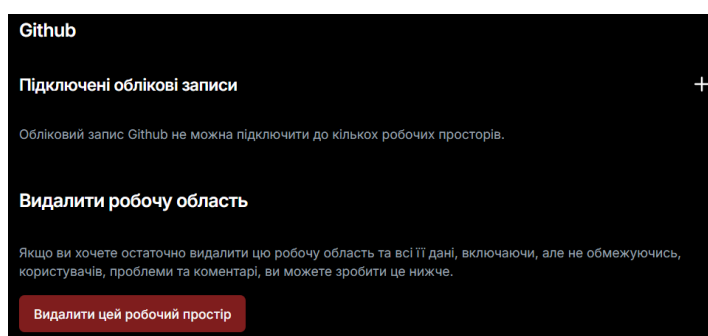


Рис. 3.12 – Решта налаштувань робочого простору

Спробувавши підключити обліковий запис GitHub користувач потрапить на сторінку встановлення (рис.3.13).

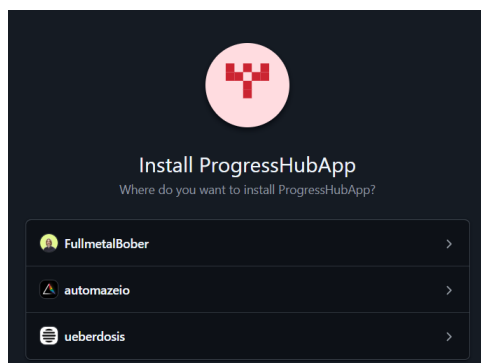


Рис. 3.13 – Інсталяція GitHub App на акаунт GitHub

Встановивши GitHub App користувач потрапить на сторінку налаштувань робочого простору, де побачить доданий обліковий запис. Далі ми можемо або натиснути налаштувати його, тим самим потрапивши на стрінку GitHub з встановленою GitHub App, або ж видалити цю інтеграцію (рис.3.14).

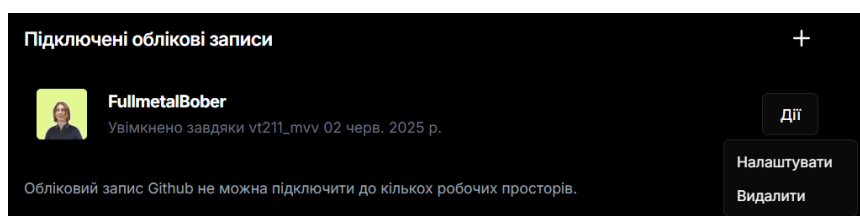


Рис. 3.14 – Підключений обліковий запис GitHub

На сторінці управління учасниками, користувач може запросити нових учасників за допомогою email, а також змінювати ролі чи видаляти під'єднаних користувачів (рис.3.15).

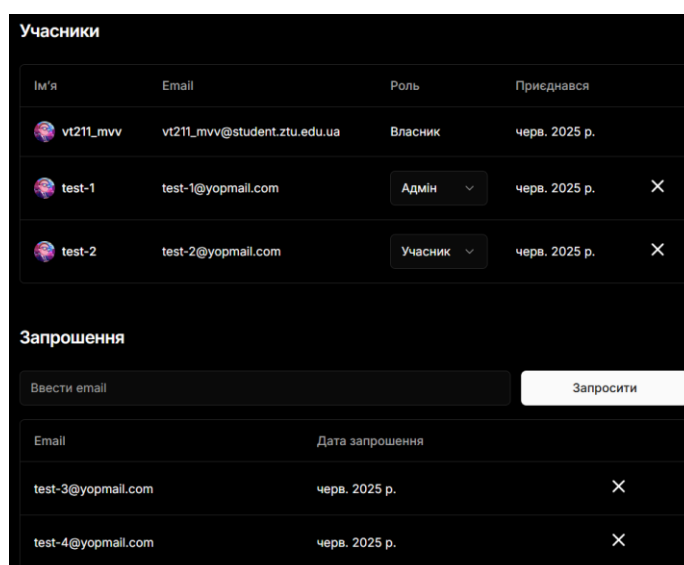


Рис. 3.15 – Сторінка управління користувачами у робочому просторі

На сторінці налаштування профілю, користувач може змінити аватар чи ім'я (рис.3.16).

Рис. 3.16 – Сторінка налаштувань користувача

Також, якщо користувач не власник, він має можливість покинути поточний робочий простір на сторінці налаштування профілю (рис.3.17).

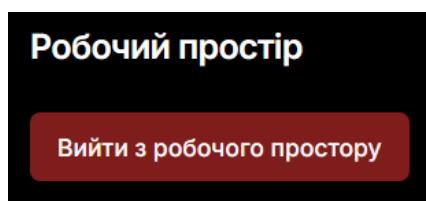


Рис. 3.17 – Кнопка виходу з робочого простору

На сторінці із списком завдань, користувач може створити нове завдання, ввівши назву завдання та обравши статус, пріоритет та виконавця (рис.3.18–3.21).

Рис. 3.18 – Форма створення завдання

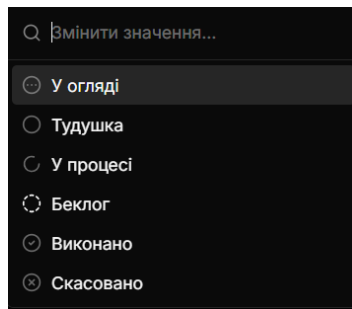


Рис. 3.19 – Меню обрання статусу завдання

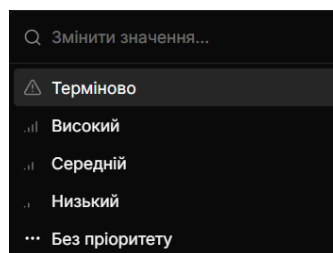


Рис. 3.20 – Меню обрання пріоритету завдання

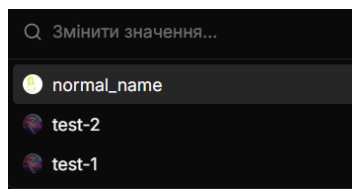


Рис. 3.21 – Меню обрання виконавця завдання

При переході на сторінку завдання, користувач може редагувати дані, які були введені при створенні. Також є можливість редагування опису завдання через блоковий колаборативний редактор. Всі зміни на сторінці відбуваються у реальному часі (рис.3.22).



Рис. 3.22 – Форма створення завдання

Натиснувши у редакторі на “/” відкриється меню швидкого додавання елементу, як то наприклад заголовкучи списку(рис.3.23).

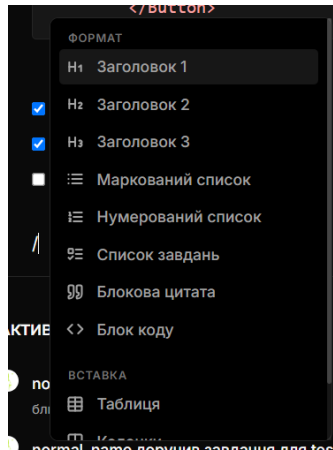


Рис. 3.23 – Меню швидкого додання елемента у редакторі

Якщо виділити текст у редакторі, з'явиться меню форматування тексту (рис.3.24).

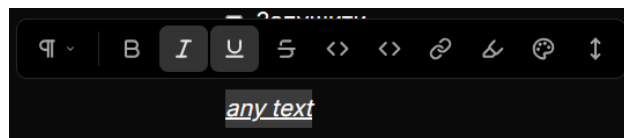


Рис. 3.24 – Меню швидкого додання елемента у редакторі

При зміні статусу, пріоритету чи виконавця, виводиться відповідне системне повідомлення в секції “Активність”. У цій секції користувач може коментувати, редагувати, видаляти та відповідати на коментарі. Також, якщо користувач користується гілкою на GitHub, яка містить у собі ідентифікатор завдання, який починається із символу “#”, то деякі події з гілкою будуть змінювати статус завдання, та повідомляти про це у секції “Активність” (рис.3.25).

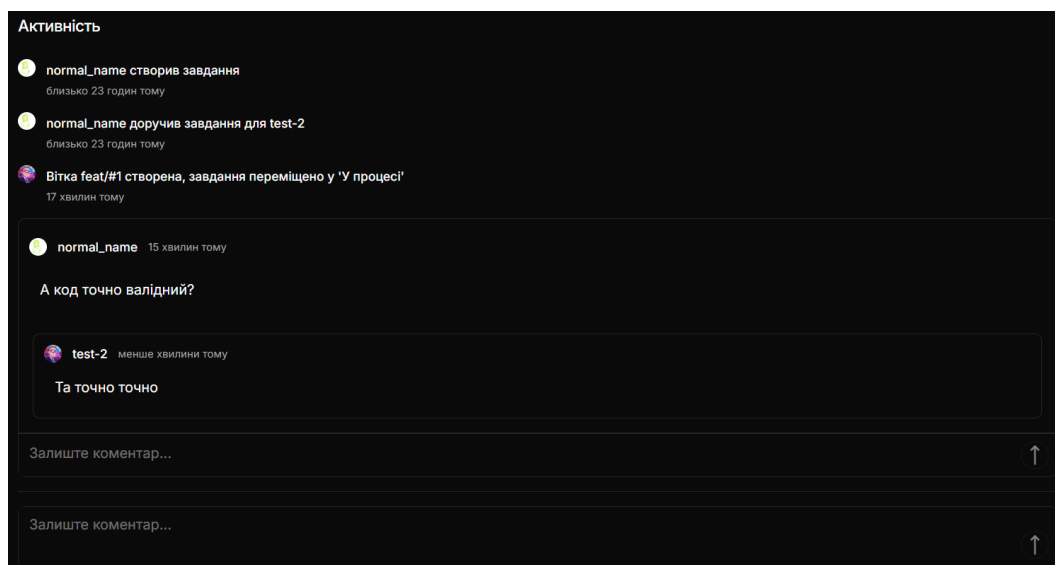


Рис. 3.25 – Активність завдання

Після інтегрування GitHub у робочий простір, на сторінці вікі ми можемо переглянути репозиторії, до яких інтеграція має доступ (рис.3.26).

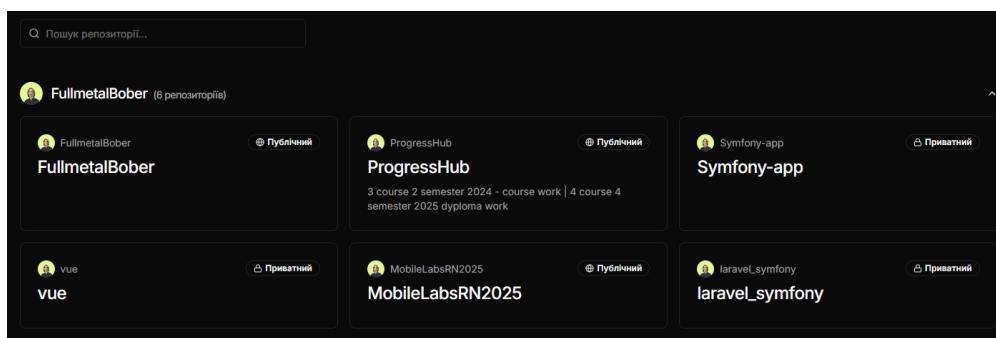


Рис. 3.26 – Сторінка з GitHub репозиторіями для документації

На GitHub репозиторії вже має бути створений хоча б один файл документації, аби мати можливість потрапити на сторінку редагування (рис.3.27–3.28).

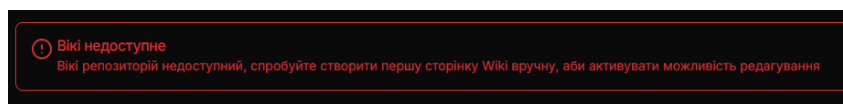


Рис. 3.27 – Повідомлення про помилку доступу GitHub Wiki

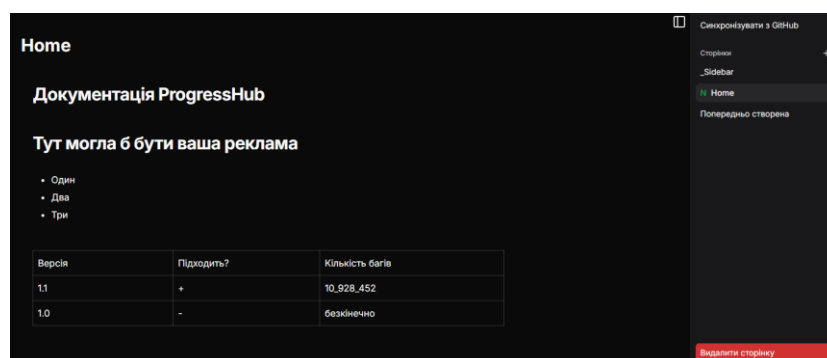


Рис. 3.28 – Сторінка GitHub Wiki

На сторінці редагування документації ми можемо створювати нові файли, редагувати їх через колаборативний редактор чи видаляти. Видалення потребує додаткового підтвердження (рис.3.29–3.30).

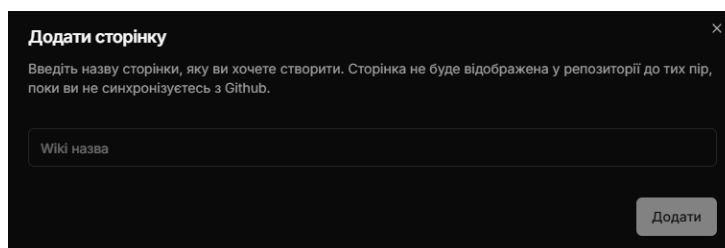


Рис. 3.29 – Модальне вікно додання сторінки документації

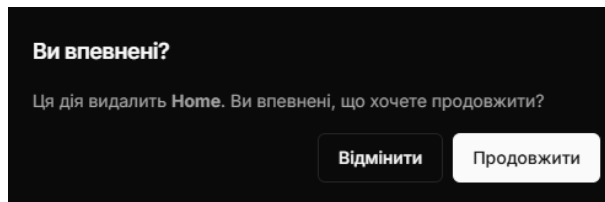


Рис. 3.30 – Підтвердження видалення сторінки документації

Також на сторінці документації можна синхронізувати сторінку, на якій знаходиться користувач, з GitHub, лише видалення синхронізується автоматично (рис.3.31).

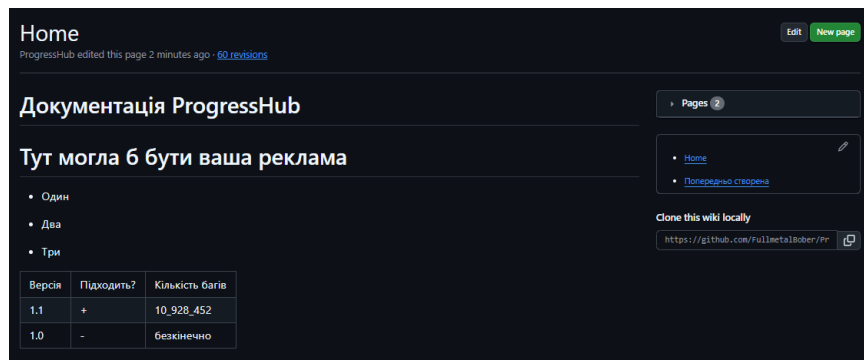


Рис. 3.31 – Сторінка документації на GitHub

З головної сторінки робочого простору користувач може перейти на сторінку зі сповіщеннями, які надходять користувачу із завдань, де він є виконавцем при зміні статусу, пріоритету, чи опису (рис.3.32).

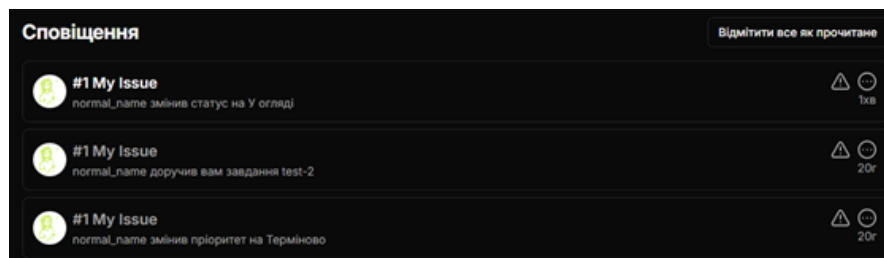


Рис. 3.32 – Сторінка зі сповіщеннями

Платформа має адаптивність (рис.3.33).

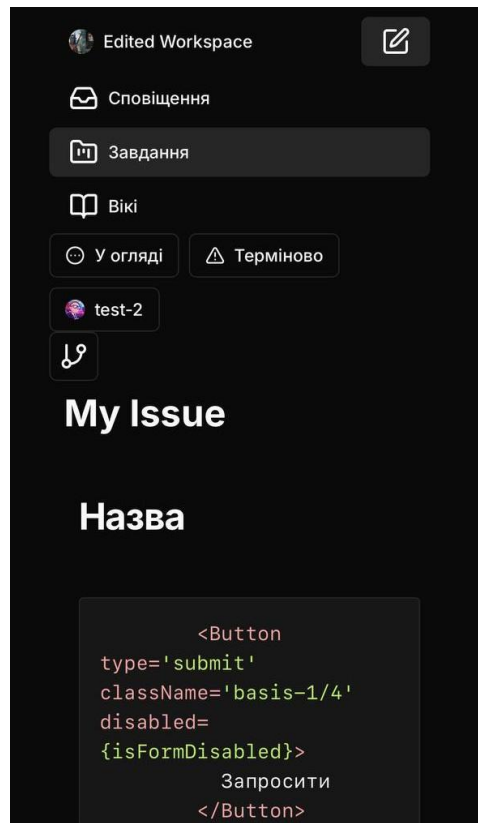


Рис. 3.33 – Сторінка із завданням з мобільного пристрою

3.3. Тестування роботи програмного продукту

Для тестування платформи було застосовано різні методи. Зокрема, допоміжний сервіс перевірили як мануально за допомогою інструмента Postman [29], так і за допомогою unit-тестів на Jest [30] (рис.3.34–3.35).

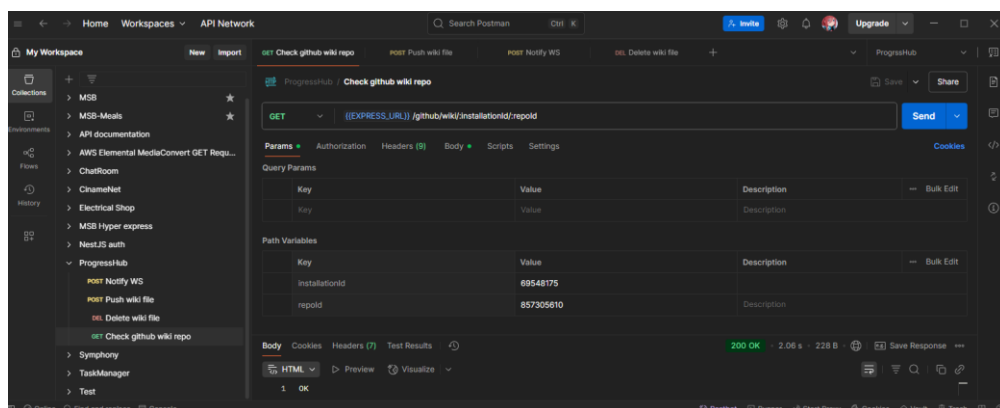


Рис. 3.34 – Запит у Postman

```

PASS express/___tests___/router.test.ts
Express Router Tests
GET /
  ✓ should return version information (23 ms)
POST /notify
  ✓ should successfully emit room changes with valid data (20 ms)
  ✓ should return 400 for invalid data (8 ms)
GET /github/setup
  ✓ should create GitHub app installation and redirect (7 ms)
  ✓ should return 400 for invalid parameters (6 ms)
GET /github/wiki/:installationId:repoId
  ✓ should return OK when wiki exists (6 ms)
  ✓ should handle wiki errors appropriately (6 ms)
POST /github/wiki/:installationId:repoId/file
  ✓ should handle file operations successfully (5 ms)
  ✓ should return 400 for invalid requests (5 ms)
DELETE /github/wiki/:installationId:repoId/file
  ✓ should delete a wiki file successfully (5 ms)
  ✓ should return 400 for invalid requests (4 ms)

Test Suites: 1 passed, 1 total
Tests: 11 passed, 11 total
Snapshots: 0 total
Time: 0.871 s, estimated 1 s

```

Рис. 3.35 – Результати тестування допоміжного сервісу

Unit-тести допоміжного сервісу комплексно перевіряють усі його маршрути, як успішні так і помилкові сценарії, а саме /notify тестує WebSocket повідомлення та валідацію даних, /github/setup перевіряє створення GitHub App та перенаправлення на клієнтську частину, а /github/wiki/* тестують повний цикл роботи з файлами включаючи перевірку існування створення, оновлення та видалення файлів через Git операції, отримання GitHub токенів та інформації про репозиторії.

Основну частину платформи було перевірено шляхом мануального тестування. Особливу увагу було приділено таким ключовим функціональностям, як: автентифікація, робота в редакторах, перевірка подій, пов'язаних із WebSocket, та надсилення електронних листів.

Форми користувачів валідуються на клієнті, завдяки цьому користувач ще до відправки форми буде бачити помилки через валідацію (рис.3.36).

Ви впевнені, що email вірний?

Рис. 3.36 – Помилка валідації форми

Було перевірено роботу сайту на швидкість та ефективність за допомогою інструмента Lighthouse, і як бачимо на рисунках 3.37–3.38 було досягнуто гарну продуктивність.

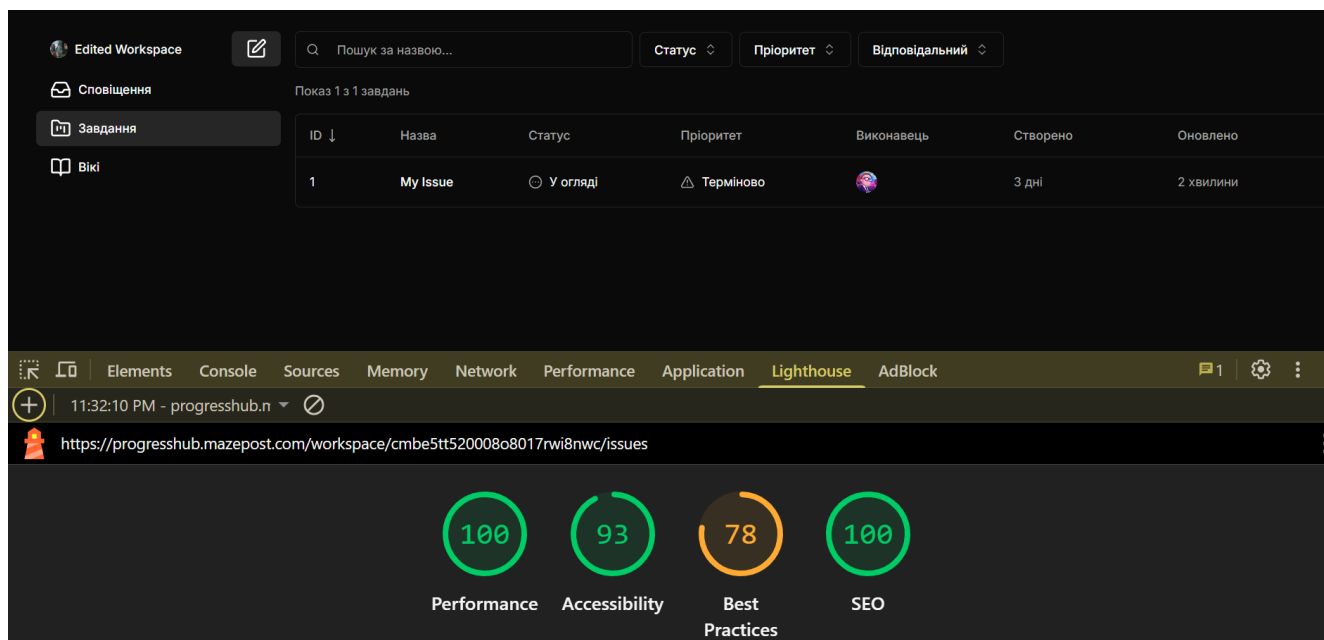


Рис. 3.37 – Перевірка сторінки зі списком завдань з Lighthouse

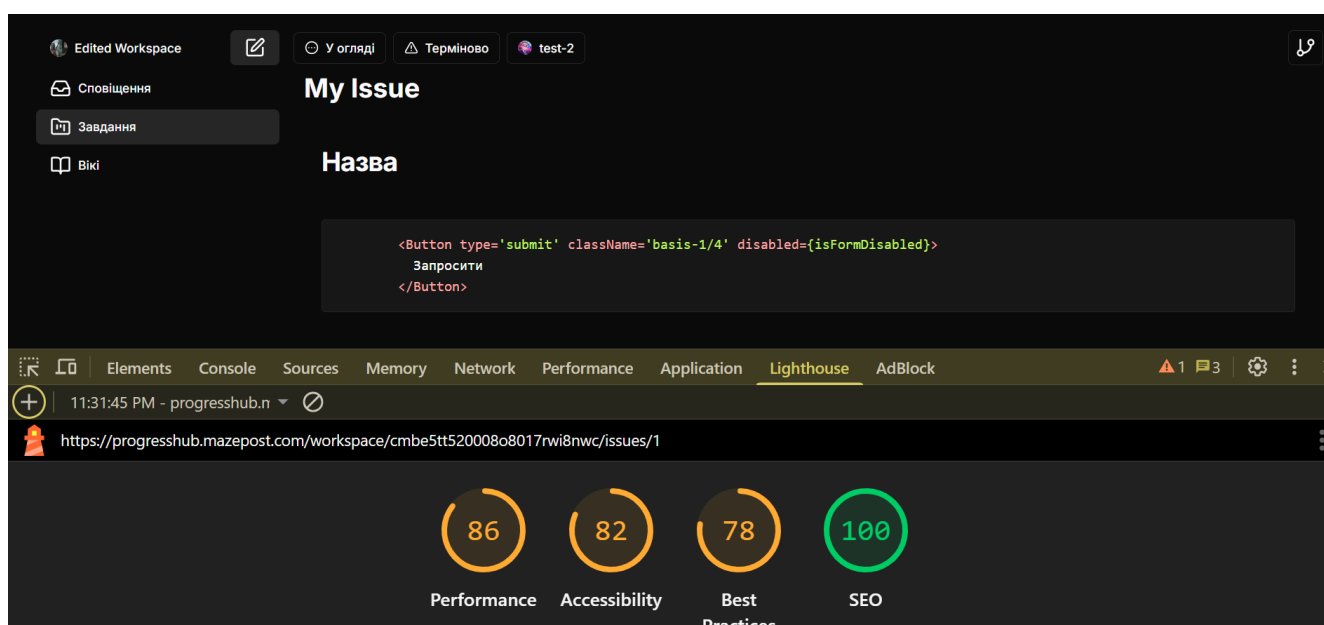


Рис. 3.38 – Перевірка сторінки завдання з Lighthouse

Висновки до третього розділу

Було представлено архітектуру розгортання платформи, що забезпечує її коректне функціонування та взаємодію між компонентами. На основі діаграми розгортання проаналізовано зв'язки між користувачем, серверною частиною та зовнішніми сервісами. У результаті були створені окремі Docker-контейнери, які розгорнуто за допомогою платформи Coolify.

Було проаналізовано структуру інтерфейсу платформи, яка реалізована у вигляді вебдодатку. Визначено основні етапи взаємодії користувача. Описано інтеграцію з GitHub та підтримку реального часу.

Платформу було протестовано з використанням ручного та автоматизованого тестування. Допоміжний сервіс перевірено unit-тестами на Jest, що охоплюють як успішні, так і помилкові сценарії роботи. Основну частину системи протестовано вручну, а саме автентифікацію, редактори, обробку подій та надсилання листів. Реалізовано клієнтську валідацію форм. Тестування продуктивності через Lighthouse показало високі результати.

ВИСНОВКИ

В результаті написання випускної кваліфікаційної роботи бакалавра було виконано комплекс робіт, що охоплюють усі основні етапи створення платформи для управління ІТ-проектами. Цю мету було досягнуто.

У першому розділі було сформульовано мету створення платформи та визначено ключові модулі системи, включаючи керування проектами, користувачами, завданнями, а також модулі реального часу, сповіщень і інтеграцій. Проведено аналіз аналогічних рішень, що дозволило виявити переваги. Та і обмеження. Обґрунтовано вибір монолітної архітектури з допоміжним сервісом. Також обрано стек технологій – TypeScript, Next.js, Express.js та PostgreSQL.

У другому розділі було визначено варіанти використання платформи відповідно до ролей користувачів, побудовано діаграми варіантів використання та описано ключові прецеденти. Структура бази даних спроектована на основі PostgreSQL, що охоплює основні сутності платформи. Також розроблено алгоритми роботи для основних процесів системи, які ілюстровано у вигляді діаграм активності. Реалізація функціоналу базується на можливостях фреймворку Next.js із використанням Server Actions та серверного рендерингу. Для взаємодії з базою даних застосовано Prisma, а для перевірки даних – Zod. Забезпечено автентифікацію через next-auth, контроль доступу, підтримку реального часу через socket.io, а також колаборативне редагування з Tiptap, як окремо, та і у зв'язці з git та GitHub. Реалізовано обробку зображень, підключено бібліотеку компонентів Shadcn/ui, а також налаштовано GitHub вебхуки для автоматизації зміни статусів завдань.

У третьому розділі було представлено архітектуру розгортання платформи, що базується на використанні Docker-контейнерів та системи керування розгортанням Coolify. На основі діаграми розгортання було проаналізовано взаємодію між клієнтом, серверною частиною та зовнішніми сервісами. Також проведено аналіз структури інтерфейсу вебдодатку, окреслено основні сценарії взаємодії користувача з платформою, описано інтеграцію з GitHub і забезпечення

роботи в реальному часі. Функціональність платформи була перевірена за допомогою ручного та автоматизованого тестування. Для допоміжного сервісу реалізовано unit–тести на Jest, що охоплюють як успішні, так і помилкові сценарії. Основні процеси – автентифікація, редагування, обробка подій, надсилання електронної пошти – протестовано вручну. Оцінка продуктивності за допомогою Lighthouse підтвердила високу ефективність системи.

					ІПЗ.КР.Б – 121 – 25 – ПЗ	Арк.
						64

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Маньківський В.В., Савіцький Р.С. ПРОЄКТУВАННЯ ТЕХНОЛОГІЧНОГО СТЕКУ КОЛАБОРАТИВНОГО РЕДАКТОРУ. Тези доповідей XV Міжнародної науково-технічної конференції “Інформаційно-комп’ютерні технології”, 28–29 березня 2025. С. 43–44. [Електронний ресурс] – Режим доступу до ресурсу: <https://conf.ztu.edu.ua/informatsiyno-komp%CA%BCyuterni-tekhnohohiyi-28-29-bereznaya-2025-r>
2. Linear [Електронний ресурс] – Режим доступу до ресурсу: <https://linear.app>
3. GitHub Projects [Електронний ресурс] – Режим доступу до ресурсу: <https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>
4. Jira [Електронний ресурс] – Режим доступу до ресурсу: <https://www.atlassian.com/software/jira>
5. TypeScript [Електронний ресурс] – Режим доступу до ресурсу: <https://www.typescriptlang.org>
6. Next.js [Електронний ресурс] – Режим доступу до ресурсу: <https://nextjs.org>
7. Express.js [Електронний ресурс] – Режим доступу до ресурсу: <https://expressjs.com>
8. PostgreSQL [Електронний ресурс] – Режим доступу до ресурсу: <https://www.postgresql.org>
9. Prisma [Електронний ресурс] – Режим доступу до ресурсу: <https://www.prisma.io/docs>
10. Zod-prisma-types [Електронний ресурс] – Режим доступу до ресурсу: <https://github.com/chrishoermann/zod-prisma-types>
11. Zod [Електронний ресурс] – Режим доступу до ресурсу: <https://v3.zod.dev>
12. @t3-oss/env-nextjs [Електронний ресурс] – Режим доступу до ресурсу: <https://env.t3.gg/docs/nextjs>
13. sharp [Електронний ресурс] – Режим доступу до ресурсу: <https://sharp.pixelplumbing.com>

14. uploadthing [Электронный ресурс] – Режим доступа до ресурсу:
<https://uploadthing.com>
15. shadcn/ui [Электронный ресурс] – Режим доступа до ресурсу:
<https://ui.shadcn.com>
16. next-auth [Электронный ресурс] – Режим доступа до ресурсу:
<https://authjs.dev>
17. Tiptap [Электронный ресурс] – Режим доступа до ресурсу:
<https://tiptap.dev/docs>
18. Hocuspocus [Электронный ресурс] – Режим доступа до ресурсу:
<https://tiptap.dev/docs/hocuspocus/introduction>
19. Yjs [Электронный ресурс] – Режим доступа до ресурсу: <https://docs.yjs.dev>
20. Socket.IO [Электронный ресурс] – Режим доступа до ресурсу: <https://socket.io>
21. Github App [Электронный ресурс] – Режим доступа до ресурсу:
<https://docs.github.com/en/apps/creating-github-apps/about-creating-github-apps/about-creating-github-apps>
22. Probot [Электронный ресурс] – Режим доступа до ресурсу:
<https://probot.github.io>
23. Octokit [Электронный ресурс] – Режим доступа до ресурсу:
<https://octokit.github.io/rest.js/v22>
24. Git [Электронный ресурс] – Режим доступа до ресурсу: <https://git-scm.com/doc>
25. Cron [Электронный ресурс] – Режим доступа до ресурсу:
<https://github.com/kelektiv/node-cron>
26. Coolify [Электронный ресурс] – Режим доступа до ресурсу:
<https://coolify.io/docs/get-started/introduction>
27. Progresshub [Электронный ресурс] – Режим доступа до ресурсу:
<http://progresshub.mazepost.com>
28. Progresshub-API [Электронный ресурс] – Режим доступа до ресурсу:
<http://progresshub-api.mazepost.com>

29. Postman [Электронный ресурс] – Режим доступа до ресурсу: <https://www.postman.com>
30. Jest [Электронный ресурс] – Режим доступа до ресурсу: <https://jestjs.io>

Додатки

```

import { WorkspaceMember } from '@prisma/zod';
import { auth } from '../auth/utils';
import prisma from '../db';

type TRoles = WorkspaceMember['role'][];

type TAvailableEntities = {
  userId: string;
  workspaceId: string;
  issueId: string;
  workspaceInviteId: string;
  workspaceMemberId: string;
  githubAppInstallationId: number;
  githubWikiFileId: string;
};

type TCheckEntity =
  | Pick<TAvailableEntities, 'userId'>
  | Pick<TAvailableEntities, 'workspaceId'>
  | Pick<TAvailableEntities, 'issueId'>
  | Pick<TAvailableEntities, 'workspaceInviteId'>
  | Pick<TAvailableEntities, 'workspaceMemberId'>
  | Pick<TAvailableEntities, 'githubAppInstallationId'>
  | Pick<TAvailableEntities, 'githubWikiFileId'>;

const workspaceAvailability = (userId: string, workspaceId: string) =>
  prisma.workspaceMember.findUnique({
    where: {
      userId_workspaceId: {
        userId,
        workspaceId,
      },
    },
  });

const issueAvailability = (userId: string, issueId: string) =>
  prisma.workspaceMember.findFirst({
    where: {
      userId,
      workspace: {
        issues: {
          some: {
            id: issueId,
          },
        },
      },
    },
    select: {
      role: true,
    },
  });

const workspaceInviteAvailability = (

```

```

    userId: string,
    workspaceInviteId: string
  ) =>
    prisma.workspaceMember.findFirst({
      where: {
        userId,
        workspace: {
          workspaceInvite: {
            some: {
              id: workspaceInviteId,
            },
          },
        },
      },
      select: {
        role: true,
      },
    });

const workspaceMemberAvailability = (
  userId: string,
  workspaceMemberId: string
) =>
  prisma.workspaceMember.findFirst({
    where: {
      userId,
      workspace: {
        members: {
          some: {
            id: workspaceMemberId,
          },
        },
      },
    },
    select: {
      role: true,
    },
  });

const githubAppInstallationAvailability = (
  userId: string,
  githubAppInstallationId: number
) =>
  prisma.workspaceMember.findFirst({
    where: {
      userId,
      workspace: {
        githubAppInstallation: {
          some: {
            id: githubAppInstallationId,
          },
        },
      },
    },
    select: {
      role: true,
    },
  });

```

```

const githubWikiFileAvailability = (userId: string, githubWikiFileId: string) =>
  prisma.workspaceMember.findFirst({
    where: {
      userId,
      workspace: {
        githubAppInstallation: {
          some: {
            githubWikiFile: {
              some: {
                id: githubWikiFileId,
              },
            },
          },
        },
      },
    },
    select: {
      role: true,
    },
  });

function getWorkspaceMember(userId: string, entity: TCheckEntity) {
  if ('workspaceId' in entity)
    return workspaceAvailability(userId, entity.workspaceId);
  if ('issueId' in entity) return issueAvailability(userId, entity.issueId);
  if ('workspaceInviteId' in entity)
    return workspaceInviteAvailability(userId, entity.workspaceInviteId);
  if ('workspaceMemberId' in entity)
    return workspaceMemberAvailability(userId, entity.workspaceMemberId);
  if ('githubAppInstallationId' in entity)
    return githubAppInstallationAvailability(
      userId,
      entity.githubAppInstallationId
    );
  if ('githubWikiFileId' in entity)
    return githubWikiFileAvailability(userId, entity.githubWikiFileId);

  throw new Error('Invalid options provided in getWorkspaceMember');
}

async function protect(entity?: TCheckEntity, workspaceRoles?: TRoles) {
  const session = await auth();
  const user = session?.user;

  if (!user?.id || !user?.email) {
    return {
      error: 'You must be logged in to access this resource',
    };
  } else if (!entity) {
    return {
      user,
    };
  }

  if ('userId' in entity) {
    if (entity.userId !== user.id)
      return {

```

```

        error: 'You do not have access to this resource',
    };
    else
        return {
            user,
        };
    }

    const workspaceMember = await getWorkspaceMember(user.id, entity);

    if (!workspaceMember) {
        return {
            error: 'You are not a member of this workspace',
        };
    } else if (workspaceRoles && !workspaceRoles.includes(workspaceMember.role)) {
        return {
            error: 'You do not have the required role to access this resource',
        };
    } else {
        return {
            user,
        };
    }
}

export async function protectAction(
    entity?: TCheckEntity,
    workspaceRoles?: TRoles
) {
    const { user, error } = await protect(entity, workspaceRoles);

    if (error) {
        throw new Error(error);
    }
    if (!user?.id || !user?.email) {
        throw new Error('Something went wrong, user not found in `protectAction`');
    }

    return {
        id: user.id,
        email: user.email,
        name: user.name,
    };
}

```