МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ ВІДСТЕЖЕННЯ ПОМИЛОК ТА УПРАВЛІННЯ ЗАВДАННЯМИ З ВИКОРИСТАННЯМ REACT»**

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

Керівник завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

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

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

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

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Кестану Едему Рустемовичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи відстеження помилок та управління завданнями з використанням React

керівник роботи Гребенюк Сергій Миколайович, д.т.н., професор
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

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

2. Строк подання студентом роботи 03.12.2025 р.

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

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Теоретичний огляд.
2. Проєктування клієнтської частини.
3. Програмна реалізація та тестування системи.

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

6. Консультанти розділів роботи

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

7. Дата видачі завдання 07.03.2025 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	02.04.2025	
2.	Збір вихідних даних.	10.04.2025	
3.	Обробка методичних та теоретичних	09.05.2025	
	джерел.		
4.	Розробка першого та другого розділу.	20.08.2025	
5.	Розробка третього розділу.	24.11.2025	
6.	Оформлення та нормоконтроль	01.12.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	17.12.2025	

Студент _____ Е.Р. Кестан
(підпис) (ініціали та прізвище)

Керівник роботи _____ С.М. Гребенюк
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

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

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи відстеження помилок та управління завданнями з використанням React»: 53 с., 11 рис., 12 джерел.

ВЕБДОДАТОК, ІНТЕРФЕЙС КОРИСТУВАЧА (UI/UX), КЕРУВАННЯ ПРОЄКТАМИ, КОМПОНЕНТНИЙ ПІДХІД, СИСТЕМА ВІДСТЕЖЕННЯ ПОМИЛОК (BUG TRACKER), СТАН ДОДАТКУ, FRONTEND-РОЗРОБКА.

Об’єкт дослідження – процеси інтерактивного управління завданнями та інцидентами в розробці програмного забезпечення, реалізовані через сучасний односторінковий (SPA) вебдодаток з акцентом на динамічний інтерфейс та ефективне управління станом.

Мета роботи: створення повнофункціонального клієнтського вебдодатку для відстеження помилок та управління завданнями, який демонструє принципи сучасної фронтенд-розробки на React та забезпечує інтуїтивний інтерфейс для операцій з робочими елементами.

Методи дослідження: методи компонентної розробки та управління станом (Zustand/Redux Toolkit); методи створення адаптивного інтерфейсу (CSS-in-JS, MUI); клієнтська маршрутизація (React Router); методи тестування (Jest, React Testing Library) та оптимізації продуктивності.

Результати роботи: розроблено повноцінний прототип фронтенд-системи керування завданнями з основним функціоналом створення, редагування, пошуку, фільтрації та візуального відстеження статусу завдань. Додаток побудовано на архітектурі з модульними компонентами, гнучким станом і сучасним UI. Розробка може бути використана як основа внутрішнього інструменту команди, навчальний приклад архітектури або фундамент для повноцінного full-stack продукту.

SUMMARY

Master’s qualifying paper “Development of an Bug Tracking and Task Management System Using React”: 53 pages, 11 figures, 12 references.

APPLICATION STATE, BUG TRACKING SYSTEM, COMPONENT-BASED APPROACH, FRONTEND DEVELOPMENT, PROJECT MANAGEMENT, USER INTERFACE (UI/UX), WEB APPLICATION.

The object of the study is the processes of interactive management of tasks and incidents in software development, implemented through a modern single-page application (SPA) with a focus on a dynamic interface and effective state management.

The aim of the study is to create a fully functional client-side web application for bug tracking and task management, which demonstrates the principles of modern frontend development with React and provides an intuitive interface for operations with work items.

The methods of research include methods of component-based development and state management (Zustand/Redux Toolkit); methods of creating a responsive user interface (CSS-in-JS, MUI); client-side routing (React Router); testing methods (Jest, React Testing Library) and performance optimization techniques.

Results of the work: a fully functional prototype of a frontend task management system was developed, implementing core features such as creating, editing, searching, filtering, and visual tracking of task status. The application is built on an architecture with modular components, flexible state management, and a modern UI. The development can be used as a basis for an internal team tool, an educational example of frontend application architecture, or a foundation for a full-fledged full-stack product.

ЗМІСТ

Завдання на кваліфікаційну роботу……...2
Реферат 4
Summary 5
Вступ..... 7
1 Теоретичний огляд можливостей 9
1.1 Огляд сучасних систем управління завданнями..... 9
1.2 Аналіз та обґрунтування вибору технологій React для фронтенд-розробки 12
1.3 Сучасні програмні підходи та платформи для розробки вебдодатків 13
1.4 Висновки до розділу 1 15
2 Проєктування клієнської частини 17
2.1 Вимоги до системи: функціональні та нефункціональні..... 17
2.2 Побудова архітектури системи..... 19
2.3 Проєктування клієнтської частини 21
2.4 Висновки до розділу 2 27
3 Програмна реалізація та тестування системи..... 29
1.1 Організація проєкту та налаштування середовища розробки..... 29
1.2 Розробка базових компонентів та системи стану 31
1.3 Тестування системи 39
Висновки 50
Перелік посилань..... 53

ВСТУП

Сучасні методології розробки програмного забезпечення та ефективне управління проєктами вимагають наявності спеціалізованих інструментів для відстеження прогресу, виявлення помилок та координації роботи команд. Особливо це актуально для веброзробки, де важливо мати зручний інтерфейс для взаємодії зі списком завдань у реальному часі. Прикладом таких рішень є комерційні продукти на кшталт Jira, Asana, Trello або open-source альтернативи, такі як OpenProject, які дозволяють організувати робочий процес, але часто потребують адаптації чи є надмірно складними для невеликих команд.

Метою кваліфікаційної роботи є проєктування та реалізація прототипу фронтенд-системи відстеження помилок та управління завданнями з використанням React та TypeScript для розробки інтерфейсу, бібліотек управління станом (наприклад, Redux Toolkit або Zustand) для централізованої обробки даних, а також UI-фреймворку (наприклад, Material-UI або Ant Design) для побудови адаптивного та сучасного інтерфейсу користувача.

Для досягнення поставленої мети сформульовано такі задачі:

– проаналізувати сучасні підходи, патерни та інструменти для розробки фронтенд-додатків для управління задачами;
– сформулювати функціональні та нефункціональні вимоги до вебінтерфейсу системи відстеження помилок;
– спроєктувати архітектуру односторінкового додатку (SPA) з чітким розмежуванням компонентів, маршрутизації та стану;
– реалізувати основний інтерфейс користувача: створення, редагування, перегляд, фільтрацію та сортування завдань за різними критеріями (статус, пріоритет, відповідальний);
– розробити систему станів завдань (наприклад, «Відкрито», «В роботі», «На перевірці», «Завершено») з можливістю візуалізації за допомогою канбан-дошки або таблиць;

– створити зручну деталізовану сторінку для перегляду та коментування конкретного завдання чи інциденту;
– реалізувати механізми пошуку, фільтрів та тегів для швидкої навігації серед великої кількості елементів;
– інтегрувати менеджер стану додатку для ефективної синхронізації даних між компонентами;
– організувати локальне збереження або імітацію роботи з бекендом через mock API для демонстрації повного життєвого циклу завдання;
– виконати тестування компонентів, інтеграцій та юзабіліті інтерфейсу, провести перевірку адаптивності на різних пристроях.

Структурно робота складається з трьох розділів. У першому розділі проаналізовано сучасні підходи до побудови фронтенд-додатків для управління проєктами, архітектурні патерни (Container/Presentational, HOC, Custom Hooks) та обґрунтовано вибір технологічного стеку: React як основна бібліотека для побудови інтерфейсу, TypeScript для забезпечення типобезпеки, Redux Toolkit/Zustand для керування глобальним станом, React Router для навігації та Material-UI для прискореної розробки UI-компонентів.

У другому розділі виконано аналіз функціональних (управління завданнями, фільтрація, коментування) і нефункціональних (продуктивність, адаптивність, зручність використання) вимог до інтерфейсу, спроєктовано архітектуру додатку з використанням діаграм компонентів та станів, розроблено схеми взаємодії між модулями та спроєктовано структуру даних для завдань та користувачів.

У третьому розділі реалізовано основні компоненти системи: модулі сторінок (Dashboard, Task List, Task Detail), перевикористовувані UI-компоненти (кнопки, форми, модальні вікна, канбан-дошка), логіку маршрутизації та глобального стану. Проведено комплексне тестування, включаючи перевірку коректності роботи інтерактивних елементів, швидкості рендерингу, адаптивності інтерфейсу на різних розмірах екрану та загальної зручності використання прототипу.

1 ТЕОРЕТИЧНИЙ ОГЛЯД МОЖЛИВОСТЕЙ

1.1 Огляд сучасних систем управління завданнями

Сучасний ринок систем управління завданнями для розробки ПЗ надзвичайно фрагментований. Існує значна кількість різноманітних рішень, кожне з яких має свою філософію, набір функцій та підтримувані методології роботи (Agile, Kanban, Waterfall). Архітектура цих систем варіюється від простих плоских списків завдань до складних ієрархій з багаторівневою декомпозицією. Така різноманітність обумовлює необхідність ґрунтовного порівняльного аналізу для вивчення функціональних можливостей, масштабованості, можливостей інтеграції та придатності рішень для різних типів команд (від невеликих стартапів до великих корпорацій) і проєктів різної складності [1].

Системи відстеження помилок та управління завданнями є критично важливими інструментами для будь-якої IT-команди. Їх основна мета – організувати хаотичний потік робочих питань, завдань та виявлених проблем у чітку, структуровану систему. Без таких інструментів команди ризикують втрачати інформацію, дублювати зусилля та втрачати контроль над термінами та якістю продукту [1].

В основі цих систем лежить поняття «завдання» (issue або ticket), яке є універсальною одиницею роботи. Завдання може представляти різні типи активності: повідомлення про помилку в програмі (bug), запит на нову функцію (feature), завдання на вдосконалення існуючого коду (improvement) або технічне завдання (task). Кожне таке завдання має свій життєвий цикл – від моменту створення до закриття, проходячи через стадії аналізу, виконання, перевірки та релізу [1].

Ключовою перевагою спеціалізованих систем є централізація всієї інформації. Замість обміну повідомленнями в чатах, листах чи усних

домовленостей, всі вимоги, обговорення, рішення та файли збираються в одному місці, прив’язані до конкретного завдання. Це створює прозору історію проєкту та запобігає втраті контексту, коли в команду приходять нові учасники або коли повертаються до старого коду [1].

Для ефективної роботи система повинна надавати базовий набір інструментів. Це, перш за все, можливості створення, читання, оновлення та видалення (CRUD) завдань. Крім того, необхідні механізми коментування та прикріплення файлів для колаборації, лог змін для відстеження історії, а також розширені фільтри, пошук та створення звітів для аналізу продуктивності та навантаження команди [1].

Сьогодні такі системи стали невід’ємною частиною сучасних гнучких методологій розробки (Agile, Scrum, Kanban). Вони слугують цифровим відображенням дошки завдань (Kanban board), дозволяють формувати беклог спринтів, відстежувати швидкість команди (velocity) та забезпечують всім учасникам проєкту – розробникам, тестувальникам, менеджерам продукту та дизайнерам – спільне робоче середовище [1].

Проаналізуємо аналогічні рішення таких систем з вже існуючих аналогів. Wrike – це платформа для спільного управління робочими процесами, орієнтована на оптимізацію планування, виконання та моніторингу проєктів. Система пропонує інструменти для організації робочих потоків, управління ресурсами та покращення колаборації, що робить її універсальною для різних бізнес-процесів. Платформа відрізняється високою гнучкістю завдяки налаштовуваним функціям та широким можливостям інтеграції. Серед її основних переваг – можливість тонкого налаштування під потреби конкретних команд, включаючи спеціалізовані рішення для маркетингу, та вбудовані інструменти для узгодження та затвердження робіт безпосередньо в системі. Однак для малих команд основним обмеженням є висока вартість ліцензій, починаючи приблизно від 10 доларів за користувача на місяць. Додатково, широкий функціонал ускладнює початкове освоєння, а для команд з простими потребами система може виявитися надмірно

складною. Безкоштовна версія має серйозні обмеження, що часто недостатньо для повноцінної роботи [1].

Jira (JIRA), розроблена компанією Atlassian, є потужною платформою для управління проєктами та відстеження завдань, зорієнтованою на технічні команди, які працюють за гнучкими методологіями. Вона дозволяє планувати спринти, розподіляти завдання та відстежувати помилки в процесі розробки ПЗ. Ключова перевага Jira – це вбудована система відстеження помилок (bug tracking), яка дозволяє детально документувати, призначати та вирішувати технічні проблеми. Система відрізняється високою гнучкістю робочих процесів, інтеграцією з популярними інструментами розробки (такими як Bitbucket, GitHub, Jenkins) та розширеними можливостями для роботи за методологіями Scrum та Kanban. Головними недоліками є складність налаштування та адміністрування, що потребує додаткових ресурсів для впровадження, а також висока вартість при масштабуванні. Для невеликих проєктів або нетехнічних команд її функціонал часто є надмірним, що може знижувати загальну продуктивність [1].

OpenProj – це безкоштовна програма для управління проєктами з відкритим кодом, яка є альтернативою комерційним рішенням, таким як Microsoft Project. Вона дозволяє користувачам створювати проєктні плани, відстежувати ресурси, візуалізувати часові рамки за допомогою діаграм Ганта та керувати залежностями між завданнями. Основною перевагою OpenProj є її безкоштовність та відкритий код, що суттєво знижує витрати, а також сумісність із файлами Microsoft Project, що забезпечує легкий обмін даними. Водночас, система має ряд суттєвих недоліків: обмежені оновлення та технічна підтримка, менш інтуїтивний інтерфейс порівняно з сучасними аналогами та відсутність розширених функцій співпраці та хмарної синхронізації. Крім того, при роботі з великими або складними проєктами можуть виникати проблеми зі стабільністю [2].

Отже, сучасний ринок систем управління завданнями характеризується значним розмаїттям рішень з різним функціоналом та методологічними

підходами. Це дозволяє організаціям обирати інструмент, максимально відповідний специфіці їхніх проєктів, але водночас вимагає ретельного попереднього аналізу для визначення оптимального варіанту, враховуючи бюджет, складність процесів та рівень технічної експертизи команди [2].

1.2 Аналіз та обґрунтування вибору технологій React для фронтенд-розробки

Як основну технологію для розробки клієнтської частини (frontend) системи управління завданнями було обрано бібліотеку React. Цей вибір обумовлений її перевагами для створення динамічних, масштабованих та ефективних вебінтерфейсів, що відповідають вимогам сучасного web-додатку [2].

Ключовою перевагою React є компонентна архітектура. Вона дозволяє розділити складний інтерфейс на незалежні, багаторазово використовувані блоки (компоненти). Це значно покращує структуру коду, прискорює розробку, спрощує колаборацію в команді та полегшує подальшу підтримку та масштабування проєкту [2].

Висока продуктивність інтерфейсу досягається завдяки використанню віртуального DOM (Document Object Model). Замість безпосередньої роботи з браузерним DOM, React спочатку оновлює його легку віртуальну копію, а потім ефективно синхронізує лише ті частини реальної сторінки, де відбулися зміни. Це мінімізує «дорогі» операції перемальовування сторінки, що особливо важливо для інтерактивних додатків з великою кількістю динамічних даних [2].

Сучасна розробка на React базується на функціональних компонентах з використанням хуків (Hooks). Такий підхід дозволяє застосовувати стан та інші можливості React у простих функціях, що робить код більш лаконічним, зрозумілим і легким для тестування. Функціональні компоненти

використовуються для всіх елементів інтерфейсу: від простих кнопок і форм введення до складних компонентів для відображення списку завдань з фільтрацією та сортуванням [2].

React підтримує декларативний підхід до програмування. Розробник описує, як має виглядати інтерфейс в залежності від стану додатку, а бібліотека сама відповідає за його оновлення. Це дозволяє зосередитись на бізнес-логіці, а не на рутинних маніпуляціях з DOM [2].

Вибір React також підкріплюється його стабільністю та популярністю. Бібліотека підтримується Facebook (Meta) та величезною спільнотою розробників, що гарантує довгострокову підтримку, регулярні оновлення та наявність величезної кількості готових рішень і бібліотек (наприклад, для маршрутизації – React Router, для управління станом – Redux Toolkit). Її успішно використовують для великих і навантажених проєктів, що доводить здатність архітектури витримувати високі навантаження [2].

Таким чином, використання React забезпечує оптимальний баланс між продуктивністю, швидкістю розробки, якістю кодової бази та можливістю майбутнього розвитку системи, що є критично важливим для її успішної реалізації та подальшого впровадження [2].

1.3 Сучасні програмні підходи та платформи для розробки вебдодатків

Сучасна веброзробка характеризується стрімким розвитком технологій, що дозволяють створювати складні, інтерактивні та високопродуктивні додатки. Вибір відповідних підходів і платформ є ключовим для успішної реалізації проєкту, такої як система управління завданнями [3].

Архітектурні підходи у розробці пройшли значну еволюцію. На зміну класичним багатосторінковим додаткам (MPA) прийшли односторінкові додатки (SPA), які завантажуються один раз і динамічно оновлюють контент

без перезавантаження сторінки. Це забезпечує плавність та швидкість роботи, близьку до нативних додатків, що є критично важливим для інструментів, де користувач постійно взаємодіє з інтерфейсом (фільтрує задачі, додає коментарі, змінює статуси). Для покращення SEO та першого завантаження SPA часто поєднують з технологією рендерингу на стороні сервера (SSR), як у Next.js, або генерацією статичних сайтів (SSG) [3].

Компонентний підхід став стандартом де-факто у побудові інтерфейсів. Він полягає в розбитті користувацького інтерфейсу на незалежні, повторно використовувані блоки (компоненти), кожен з яких інкапсулює свою структуру (HTML), стилі (CSS) та логіку (JavaScript). Це суттєво покращує структуру коду, спрощує його тестування та підтримку, а також прискорює розробку за рахунок повторного використання. React, Vue.js та Angular базуються саме на цьому підході [3].

Окремою важливою темою є управління станом додатку (State Management). У складних додатках з багатою взаємодією дані (стан) мають бути доступні різним компонентам, часто на різних рівнях ієрархії. Для керування глобальним станом використовують спеціальні бібліотеки, такі як Redux (частіше в комбінації з Redux Toolkit), MobX, або вбудовані механізми, на кшталт Context API у React. Вибір підходу впливає на передбачуваність поведінки додатку та легкість налагодження [3].

Сучасний стек розробки (tech stack) для повноцінного додатку (full-stack) найчастіше базується на JavaScript/TypeScript. Це дозволяє використовувати одну мову як на клієнті (frontend), так і на сервері (backend), що спрощує розробку. Популярним архітектурним стилем серверної частини є REST API або GraphQL, які забезпечують структуровану комунікацію між клієнтом і сервером, повертаючи дані у форматі JSON [3].

Для прискорення розробки інтерфейсу широко застосовуються UI-бібліотеки та фреймворки компонентів, такі як Material-UI (MUI), Ant Design, Chakra UI для React, або Vuetify для Vue. Вони надають готові, доступні та стилізовані компоненти (кнопки, форми, модальні вікна, таблиці), що дозволяє зосередитись на бізнес-логіці, а не на верстці з нуля [4].

Інструменти збірки та розробки, такі як Vite або Webpack, і транспайлери, як Babel, є невід'ємною частиною процесу. Вони дозволяють використовувати сучасний синтаксис JavaScript (ES6+), оптимізувати код, розділяти його на модулі та створювати збірки для продакшену [4].

Таким чином, сучасна розробка вебдодатків – це комплексний підхід, що поєднує прогресивні архітектурні патерни (SPA, компонентна архітектура), спеціалізовані бібліотеки для управління станом, потужні інструменти збірки та UI-фреймворки для швидкої реалізації інтерфейсу. Правильна комбінація цих елементів дозволяє створити масштабований, продуктивний та зручний у підтримці додаток [4].

1.4 Висновки до розділу 1

У першому розділі досліджено теоретичні основи, технологічний контекст та ринок сучасних систем управління завданнями та відстеження помилок (bug tracking). Проаналізовано предметну область, ключові функції та місце цих систем у процесі розробки ПЗ, особливо в контексті гнучких методологій.

Проведено огляд сучасних вебтехнологій, де обґрунтовано вибір бібліотеки React для реалізації клієнтської частини. Цей вибір зумовлений її компонентною архітектурою, високою продуктивністю завдяки віртуальному DOM, декларативним підходом та широкою екосистемою, що дозволяє ефективно створювати динамічні та масштабовані інтерфейси.

Окреслено сучасні програмні підходи до розробки, такі як архітектура односторінкових додатків (SPA), управління станом та використання UI-бібліотек, що формують основу для технічної реалізації проєкту.

На основі аналізу ринку та існуючих рішень (Jira, Wrike, OpenProj) виявлено переваги та недоліки комерційних та open-source аналогів. Це дозволило сформулювати чіткі вимоги до функціональності та визначити

потенційну нішу для власної системи – створення зручного, гнучкого та економічно ефективного рішення, адаптованого під потреби невеликих та середніх команд розробників.

Таким чином, у результаті проведеного аналізу було сформовано комплексне техніко-теоретичне підґрунтя для розробки системи відстеження помилок та управління завданнями з використанням React. Отримані висновки визначають архітектурні підходи, технологічний стек та ключові функціональні напрямки для подальшого проєктування та реалізації.

2 ПРОЄКТУВАННЯ КЛІЄНСЬКОЇ ЧАСТИНИ

2.1 Вимоги до системи: функціональні та нефункціональні

Усі вимоги до програмного продукту розділяються на дві основні категорії: функціональні та нефункціональні. Функціональні вимоги описують, що саме повинна робити система, визначаючи її основну поведінку та можливості. Нефункціональні вимоги вказують, як система має працювати, задаючи критерії якості: продуктивність, безпеку, зручність використання та інші атрибути, які суттєво впливають на кінцевий користувацький досвід [4].

Функціональні вимоги визначають основну логіку та сценарії використання системи відстеження помилок та управління завданнями. Вони формують основу для розробки та включають:

- управління користувачами та доступом: реєстрація, авторизація, аутентифікація (JWT), розмежування прав доступу (ролі: адміністратор, менеджер, розробник);
- повний життєвий цикл завдання (issue): створення, перегляд, редагування, коментування, зміна статусу (наприклад, «Відкрито», «В роботі», «Готово») та видалення завдань;
- класифікація та атрибути: можливість вказати тип завдання (помилка, нова функція, покращення), пріоритет, призначити виконавця, додати мітки (теги);
- ефективна навігація та пошук: розширена фільтрація за всіма атрибутами, повнотекстовий пошук, створення та збереження персоналізованих фільтрів;
- візуалізація процесу: створення та перегляд Kanban-дошки для візуального відстеження прогресу завдань у спринтах або проєктах;
- співпраця: система коментування, прикріплення файлів (скріншоти, логи) до завдань, підписка на сповіщення про зміни;

– звітність та аналітика: базові звіти про завантаження команди, статистика за типами завдань, фільтрована вигрузка даних.

Нефункціональні вимоги визначають якісні характеристики системи, які безпосередньо не є функціями, але критично важливі для її успішної експлуатації:

– продуктивність: сторінки інтерфейсу повинні завантажуватися та реагувати на дії користувача без помітних затримок (час відгуку інтерфейсу < 200 мс) – списки завдань з фільтрами мають оброблятися ефективно;
– безпека: обов'язкове шифрування паролів, захист від міжсайтового скриптингу (XSS) та підробки міжсайтових запитів (CSRF), використання токенів (JWT) для безпечної аутентифікації, валідація всіх вхідних даних;
– зручність використання (Usability): інтуїтивно зрозумілий та послідовний інтерфейс (UI/UX), адаптований під різні розміри екранів (Responsive Design), відповідність принципам доступності (WCAG);
– надійність та доступність: система повинна забезпечувати високий час безвідмовної роботи (uptime > 99.5%), мати обробку помилок на всіх рівнях та забезпечувати цілісність даних;
– масштабованість: архітектура повинна дозволяти обслуговувати зростаючу кількість користувачів та завдань без серйозної деградації продуктивності;
– підтримуваність: кодова база має бути чистою, модульною та добре документованою, з використанням статичних аналізаторів (ESLint) та форматерів (Prettier) для дотримання стандартів.

Врахування та баланс між функціональними (що робимо) та нефункціональними (як якісно це робимо) вимогами є основою для проєктування збалансованої, ефективної та надійної системи, яка відповідатиме реальним потребам користувачів [5].

2.2 Побудова архітектури системи

Архітектура клієнтської частини системи відстеження помилок та управління завданнями базується на односторінковій архітектурі (SPA), що забезпечує динамічну роботу без постійних перезавантажень сторінки та максимально швидкий відгук на дії користувача.

Основу архітектури становить компонентна модель React, де весь інтерфейс розбитий на незалежні, повторно використовувані блоки. Ці компоненти організовані за принципом ієрархії: від найпростіших презентаційних елементів (кнопок, полів вводу, карток завдань) до складних контейнерних компонентів, які інтегрують логіку, стан та взаємодію з даними. Окремо виділяються компоненти макету, що формують загальну структуру застосунку – шапку, бічну панель навігації та основну область контенту [5].

Критично важливим елементом архітектури є механізм управління станом додатку. Для різних типів даних застосовуються різні підходи. Локальний стан форми або режиму відображення компонента керується за допомогою вбудованих React хуків useState та useReducer. Глобальні дані, такі як інформація про авторизованого користувача, загальний список завдань, активні фільтри та налаштування, зберігаються в централізованому сховищі, реалізованому за допомогою Redux Toolkit. Це забезпечує передбачуваність змін стану, спрощує його налагодження та робить дані доступними з будь-якої точки додатка без необхідності їх передачі через багаторівневі ланцюги компонентів [6].

Навігація всередині застосунку реалізована через клієнтську маршрутизацію з використанням React Router. Це дозволяє відображати різні «сторінки» (список завдань, детальний перегляд окремого завдання, налаштування профілю) за різними URL-шляхами, зберігаючи при цьому стан SPA. Параметри пошуку, ідентифікатори відкритих завдань та активні вкладки також синхронізуються з адресним рядком, що забезпечує можливість копіювання посилань та коректну роботу кнопок «Назад» і «Вперед» браузера [6].

Взаємодія із серверною частиною відбувається через набір спеціалізованих сервісних модулів (services), які інкапсулюють логіку HTTP-запитів. Кожен модуль відповідає за комунікацію з певним ендпоінтом REST API (наприклад, завдання, користувачі, коментарі). Для виконання запитів використовується бібліотека axios, що забезпечує зручну роботу з промісами, інтерцепторами запитів (для автоматичного додавання токенів авторизації) та централізовану обробку помилок. Запити часто комбінуються з асинхронними thunk-екшенами в Redux, що дозволяє оновлювати глобальний стан додатка відразу після успішного отримання відповіді від сервера [6].

Вся архітектура будується з урахуванням принципів модульності та слабкої зв’язності. Компоненти мінімально залежать один від одного та від конкретної реалізації сервісів, що робить систему легко тестовованою та спрощує її подальшу підтримку та розвиток. На рисунку 2.1 зображено загальну структуру клієнтської частини додатка, яка ілюструє зв’язки між основними модулями та напрямок потоку даних [7].

```
src/
├── features/
│   ├── auth/           # Авторизація та реєстрація
│   ├── issues/         # Управління завданнями (створення, перегляд, редагування)
│   ├── projects/       # Керування проектами
│   ├── dashboard/      # Основна панель із статистикою
│   └── admin/          # Панель адміністратора
├── shared/
│   ├── components/     # Спільні UI-компоненти (кнопки, форми, модальні вікна)
│   ├── hooks/          # Кастомні React хуки
│   ├── utils/          # Допоміжні функції
│   └── api/            # Конфігурація API клієнта (axios)
├── app/                # Кореневі компоненти та конфігурація
└── store/              # Сховище стану (Redux)
```

Рисунок 2.1 – Структура додатка

Ядро інтерфейсу формує рівень компонентів, розділений на дві категорії. Контейнерні компоненти відповідають за бізнес-логіку: вони взаємодіють із сервером через спеціалізовані сервіси та керують даними.

Отриману інформацію вони передають презентаційним компонентам, основним завданням яких є візуалізація. Ці чисті компоненти, такі як кнопки, карти завдань, таблиці та поля вводу, відображають отримані пропси та повідомляють про дії користувача (кліки, введення тексту) назад контейнерам [7].

Стан додатка управляється на двох рівнях. Локальний стан форми або UI-елемента (наприклад, відкритий/закритий випадаючий список) обробляється за допомогою хуків useState всередині окремих компонентів. Для глобальних даних, які мають бути доступними в різних частинах програми (наприклад, інформація про авторизованого користувача, кешований список проєктів, активні фільтри), використовується централізоване сховище Redux. Це забезпечує узгодженість даних і дозволяє будь-якому компоненту легко підписатися на їх зміни [7].

Всі зовнішні комунікації інкапсульовані в окремому шарі сервісів. Ці модулі, використовуючи бібліотеку axios, відповідають за формування HTTP-запитів до REST API бекенду, автоматичне додавання токенів авторизації, обробку помилок і трансформацію відповідей. Основним принципом архітектури є односторонній потік даних: дії користувача в інтерфейсі запускають оновлення стану або запити до API; отримані від сервера дані оновлюють глобальний стан у Redux, що, у свою чергу, автоматично викликає перерендер усіх компонентів, які від цього стану залежать. Така структура забезпечує передбачуваність, легкість налагодження та підтримки, а також закладає основу для майбутнього масштабування додатка [8].

2.3 Проєктування клієнтської частини

Проєктування клієнтської частини передбачає створення детальної структури React-додатка, розробку інтерфейсів користувача та визначення логіки їх взаємодії [8].

Основа клієнтської частини – це односторінковий додаток (SPA), побудований на бібліотеці React з використанням TypeScript для забезпечення типобезпеки. Архітектура додатка організована за принципом feature-based structure, де кожен функціональний модуль має власну ізольовану структуру [9].

На рисунку 2.2 представлено детальну діаграму клієнтських компонентів React, яка візуалізує ієрархічну структуру, ключові залежності та потоки даних між основними елементами інтерфейсу фронтенд-частини системи управління завданнями. Діаграма ілюструє застосування принципів компонентної архітектури та патерна Single Responsibility для створення модульного, масштабованого та легко підтримуваного користувацького інтерфейсу [9].

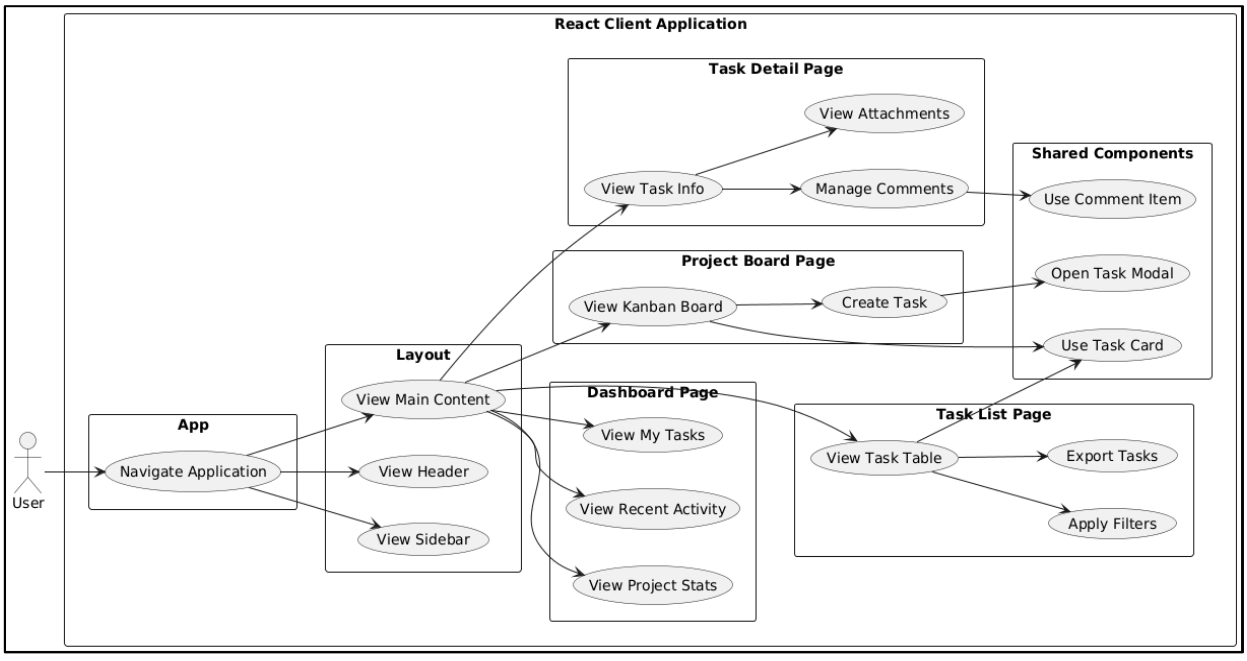


Рисунок 2.2 – Діаграма клієнтських компонентів React

Вершину ієрархії займає головний компонент App, який є точкою входу в додаток. Його основною відповідальністю є ініціалізація застосунку, конфігурація маршрутизації та налаштування глобальних провайдерів, таких як менеджер стану (наприклад, Redux Store Provider або Zustand контекст) і тема (ThemeProvider). Компонент App інкапсулює всю логіку маршрутизації

за допомогою BrowserRouter (або HashRouter), визначаючи основні шляхи додатку та відповідні сторінкові компоненти, які будуть рендеритися для кожного маршруту [9].

Безпосередньо під App розташований компонент Layout, який реалізує базовий макет (layout) застосунку. Цей компонент відповідає за уніфіковану структуру сторінок та забезпечення консистентності інтерфейсу. Він складається з трьох основних секцій, що об’єднують спільні для всіх сторінок елементи: Header (Верхній колонтитул), Sidebar (Бічна панель) та MainContent (Головна контентна область) [9].

Header (Верхній колонтитул) містить головну навігаційну панель з логотипом системи, пошуковою строкою для глобального пошуку завдань, сповіщеннями (Notifications) та профайлом поточного користувача з випадаючим меню для виходу або переходу в налаштування [9].

Sidebar (Бічна панель) реалізує основну навігацію між розділами додатку. Він включає список посилань або кнопок для швидкого переходу на головну панель (Dashboard), до списку завдань, канбан-дошки, календаря, налаштувань проєктів та сторінки профайлу. Sidebar також може відображати список останніх або обраних проєктів.

MainContent (Головна контентна область) – це динамічна область, що слугує контейнером для відображення контенту поточної сторінки, яка визначається активним маршрутом. Цей компонент відповідає за відступ від фіксованих елементів Header та Sidebar і забезпечує правильне прокручування вмісту [10].

Дочірніми компонентами MainContent, що відповідають за окремі сторінки додатку, є DashboardPage, TaskListPage, ProjectBoardPage, TaskDetailPage та UserProfilePage.

DashboardPage (Головна панель) – сторінка-збірка, призначена для огляду ключових метрик та активності. Вона агрегує дані через декілька інтелектуальних компонентів-віджетів: StatsOverview для показу KPI, ActivityFeed для хронології подій, MyAssignedTasks для персоналізованого списку завдань та QuickCreatePanel для швидкого створення нових ітемів [10].

TaskListPage (Сторінка списку завдань) є центральною сторінкою для роботи з завданнями у табличному або списковому вигляді. Її архітектура включає TaskListToolbar з розширеними фільтрами та пошуком, TaskListView для відображення відфільтрованого списку та PaginationControls для навігації по великим наборам даних [10].

ProjectBoardPage (Сторінка канбан-дошки проєкту) реалізує візуальне управління завданнями за допомогою патерну Kanban. Її структура складається з BoardContainer, BoardColumn для окремих статусів та DraggableTaskCard, яка підтримує drag-and-drop функціонал для переміщення завдань між колонками.

TaskDetailPage (Сторінка детального перегляду завдання) присвячена одному конкретному завданню або інциденту. Вона складається з TaskDetailHeader з ключовою інформацією, TaskDescription з детальним описом, CommentSection для системи коментування та ActivityLog для технічного логу всіх змін [11].

UserProfilePage (Сторінка профайлу користувача) містить форми та інформацію для управління акаунтом: аватар, основна інформація, налаштування сповіщень, зміна пароля.

Окремий модуль діаграми складають спільні (shared) та перевикористовувані (reusable) UI-компоненти, які використовуються на різних сторінках для забезпечення консистентності.

TaskCard – уніфікований компонент для візуального представлення завдання в скороченому вигляді. Відображає іконку типу, заголовок, ID, пріоритет, призначеного користувача та дедлайн. Використовується в TaskListView та BoardColumn [11].

TaskModal – модальне вікно, що містить повноцінну форму для створення або редагування завдання. Інтегрує в себе поля для всієї необхідної інформації та забезпечує двостороннє зв’язування даних.

CommentItem – компонент для відображення окремого коментаря. Складається з аватара автора, імені, часу публікації, тіла коментаря та кнопок дій (відповісти, редагувати, видалити) [12].

Діаграма також містить стрілки або позначення, що демонструють основні напрямки потоків даних та подій. Вниз за ієрархією передаються дані та callback-функції (пропси) від батьківських компонентів до дочірніх. Вгору за ієрархією дочірні компоненти сповіщають про дії користувача, викликаючи передані callback-функції, що запускає оновлення глобального стану. Горизонтальні зв’язки між компонентами, що не знаходяться в прямій ієрархії, реалізуються через глобальний менеджер стану (Redux Store), де компоненти підписуються на зміни даних або відправляють екшени [12].

На рисунку 2.3 зображено діаграму розгортання (Deployment), яка ілюструє архітектуру розміщення та взаємодії компонентів вебзастосунку в продукційному середовищі. Клієнтська частина представлена React SPA, що розгортається на платформі Vercel/Netlify з використанням CDN Edge Nodes для швидкої доставки статичних ресурсів і доступна користувачам через браузер або PWA по захищеному з’єднанню HTTPS [12].

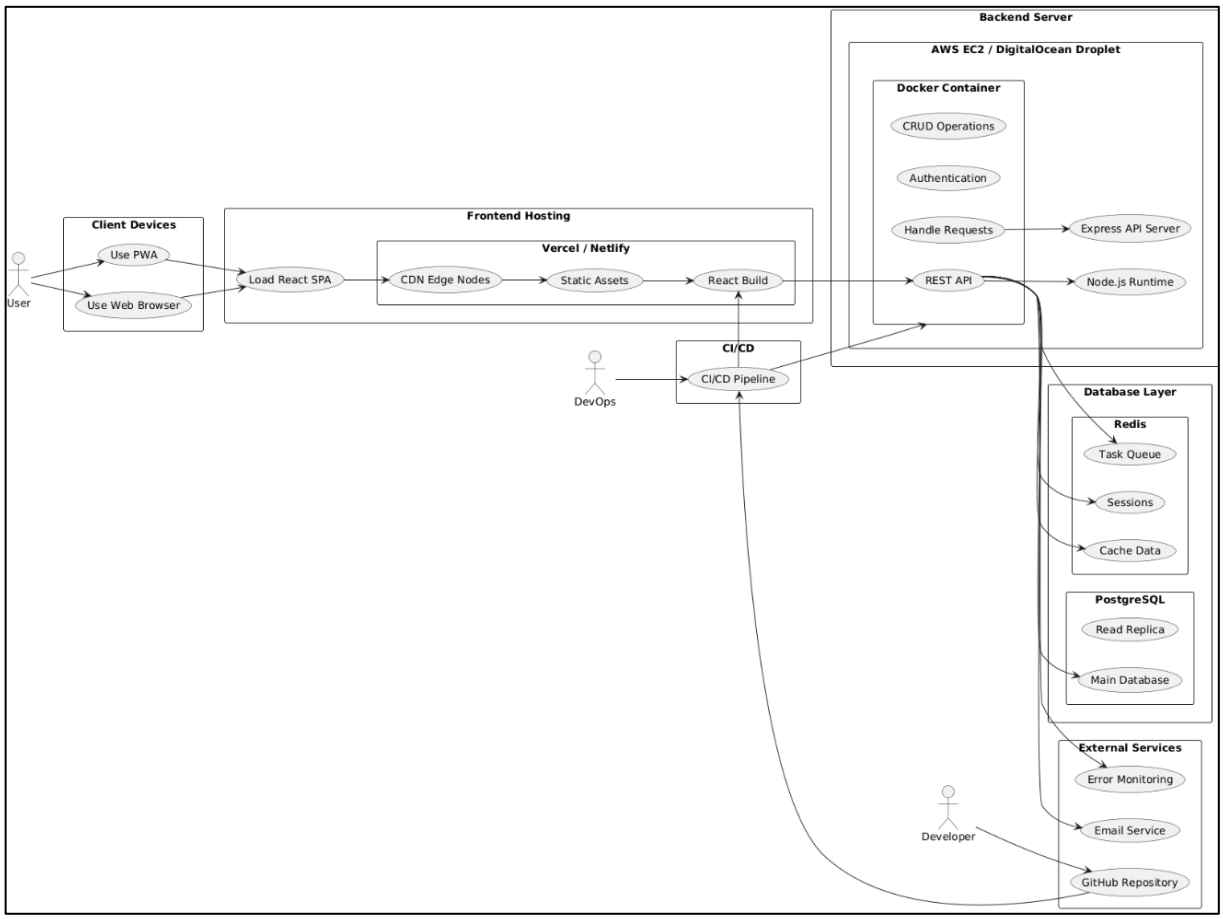


Рисунок 2.3 – Діаграма розгортання (Deployment)

Архітектура організована за принципом доставки статичних ресурсів від сервера до кінцевого користувача. З лівого краю зображено кінцевих користувачів системи – розробників, тестувальників та менеджерів проєктів, які взаємодіють з інтерфейсом через сучасні веббраузери на різних пристроях (десктопи, планшети, смартфони). Всі запити від клієнтів проходять через глобальну мережу Інтернет за захищеним протоколом HTTPS (порт 443), що гарантує шифрування трафіку та захист від MITM-атак.

Для покращення доступності налаштований DNS-сервіс (Cloudflare), який забезпечує стабільну розв'язку доменного імені та додатковий захист.

Фронтенд-додаток, побудований на React з TypeScript, після процесу збірки (build) представляє собою оптимізований набір статичних файлів (HTML, CSS, JavaScript, зображення, шрифти). Для його розгортання обрано сучасні спеціалізовані платформи для хостингу статичних сайтів та JAMstack-архітектури, такі як Vercel, Netlify або GitHub Pages. Ключовою перевагою цих платформ є глибока інтеграція з Content Delivery Network (CDN), що складається з розподіленої по всьому світу мережі Edge Nodes (граничних серверів). Ця мережа автоматично обслуговує запити користувачів з географічно найближчого сервера, що мінімізує час завантаження інтерфейсу (latency) та підвищує продуктивність для глобальної аудиторії [12].

Платформа хостингу забезпечує повністю автоматизований процес CI/CD (Continuous Integration / Continuous Deployment), безпосередньо інтегрований з системою контролю версій. При кожному push-запиті у гілку main або develop репозиторію автоматично запускається пайплайн, який включає встановлення залежностей, запуск модульних та інтеграційних тестів (за допомогою Jest та React Testing Library), лінтинг коду та процес збірки проєкту. Успішне завершення пайплайну автоматично запускає процес деплою згенерованих статичних файлів на CDN. Це забезпечує можливість швидких та безпечних оновлень інтерфейсу без ручного втручання.

Процеси підтримки та моніторингу забезпечує команда фронтенд-розробників. Вони здійснюють моніторинг продуктивності застосунку за

допомогою вбудованих інструментів браузерів (Lighthouse, Web Vitals) та зовнішніх сервісів. Проводиться регулярний аудит залежностей, оновлення бібліотек для виправлення вразливостей, а також тестування на сумісність з різними браузерами (cross-browser testing). Для забезпечення високої доступності додатку налаштовано моніторинг «живучості» (uptime monitoring) через спеціалізовані сервіси [12].

2.4 Висновки до розділу 2

У другому розділі, присвяченому проєктуванню системи відстеження помилок та управління завданнями, було визначено повний перелік функціональних та нефункціональних вимог до клієнтської частини, що дозволило чітко окреслити її ключові можливості та критерії якості інтерфейсу.

Проведено аналіз основних сценаріїв взаємодії користувачів (менеджерів, розробників, тестувальників) з системою за допомогою діаграми прецедентів, яка визначила центральні ролі та послідовності їх дій.

Розроблено детальну компонентну архітектуру інтерфейсу на основі React, що відображено на діаграмі ієрархії компонентів, яка наочно демонструє модульну структуру, розподіл відповідальностей та потоки даних між головним компонентом App, компонентом макету Layout, сторінковими компонентами (DashboardPage, TaskListPage, ProjectBoardPage тощо) та перевикористовуваними UI-компонентами (TaskCard, TaskModal). Це забезпечує логічну організацію коду та полегшує його розробку та підтримку.

Для завершеності архітектурного проєкту було представлено схему розгортання (Deployment Diagram), що описує сучасний підхід до хостингу статичного React-додатку на платформах Vercel/Netlify з використанням CDN, автоматизованого CI/CD пайплайну та інтеграції з необхідними зовнішніми сервісами (моніторинг, аналітика, імітація API).

Загалом, проведена проєктно-архітектурна робота сформувала цілісну та узгоджену основу для подальшої практичної реалізації фронтенд-системи, яка відповідає сучасним вимогам до продуктивності, зручності використання (UX), адаптивності та легкості розгортання.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

1.1 Організація проєкту та налаштування середовища розробки

Першим етапом практичної реалізації системи відстеження помилок та управління завданнями стала фундаментальна організація структури проєкту та детальна конфігурація сучасного середовища розробки. Метою цього етапу було створення надійної, стандартизованої та легко масштабованої основи, що забезпечує високу продуктивність команди, передбачуваність процесу розробки та безумовну якість вихідного коду. Усі подальші етапи реалізації функціональних модулів залежатимуть від архітектурних рішень, прийнятих на цій стадії.

Проєкт було ініційовано за допомогою офіційного інструменту create-react-app з шаблоном TypeScript. Цей підхід надав готову, оптимізовану базову конфігурацію таких ключових інструментів збірки, як Webpack і Babel, а також стандартний набір скриптів для запуску сервера розробки, тестування та створення продакшн-збірки. Основним інженерним рішенням стало формування технологічного стеку, який об’єднав React 18 як ядро для побудови декларативного інтерфейсу на основі функціональних компонентів і хуків, та TypeScript як систему статичної типізації, покликану кардинально підвищити надійність коду, забезпечити глибоке автодоповнення в редакторі та значно спростити процес майбутнього рефакторингу. Як середовище виконання для всіх інструментів розробки було обрано стабільну версію Node.js 18 LTS.

Для забезпечення довгострокової підтримуваності та чіткої організації кодової бази було застосовано принцип структурування за функціональностями (Feature-Based Structure). На відміну від класичного підходу з групуванням за типами файлів, основний код було організовано навколо бізнес-доменів системи. Таким чином, головний каталог src/features

об’єднав окремі модулі, такі як tasks/ (вся логіка та компоненти для роботи з завданнями), projects/, comments/ та dashboard/. Окремий каталог shared/ був виділений для спільних, перевикористовуваних ресурсів: базові UI-компоненти (Button, Input, Modal), кастомні хуки, конфігурація глобального стану, глобальні TypeScript-інтерфейси та утиліти. Така організація дозволяє новим розробникам швидко орієнтуватися в проєкті та ізолює зміни в межах конкретної функціональності.

Невід’ємною частиною налаштування середовища стала інтеграція та конфігурація потужного пайплайну інструментів забезпечення якості коду. Центральне місце в ньому зайняв ESLint з розширеним набором правил для React, React Hooks та TypeScript, налаштований на суворе дотримання консистентного стилю та виявлення потенційних антипатернів на етапі написання коду. Для гарантії єдиного стилю форматування в проєкті було налаштовано Prettier, інтегрований з ESLint для усунення будь-яких конфліктів. Автоматизація цих процесів була реалізована за допомогою Husky та lint-staged, які налаштовують git-хуки: перед кожним комітом у репозиторій для змінених файлів автоматично запускаються процедури лінтингу та форматування, що безумовно запобігає потраплянню в історію версій невідформатованого або неякісного коду. Для уніфікації базових налаштувань редакторів серед усіх розробників команди було додано конфігураційний файл .editorconfig.

Для керування конфігурацією додатку в різних середовищах (розробка, тестування, продакшн) було залучено механізм змінних середовища через .env файли. У цих файлах були визначені ключові параметри, такі як базовий URL для підключення до мок-API (REACT_APP_API_BASE_URL), ключі для інтеграції з сервісами моніторингу клієнтських помилок (Sentry) та аналітики користувацької поведінки. Цей підхід дозволяє легко змінювати налаштування без модифікації вихідного коду та забезпечує безпеку конфіденційних даних.

У підсумку, етап організації проєкту завершився створенням чистої, логічно структурованої та технічно насиченої основи. Ця основа представлена

повністю готовим до роботи середовищем, яке інкапсулює найкращі практики сучасної фронтенд-розробки і слугує міцним фундаментом для послідовної та ефективної імплементації всіх запланованих функціональних модулів системи управління завданнями.

1.2 Розробка базових компонентів та системи стану

Наступним критично важливим етапом реалізації стала розробка фундаментальних елементів інтерфейсу та архітектури управління даними. Цей етап заклав основу для створення консистентного, продуктивного та динамічного користувацького досвіду, визначивши принципи побудови UI та централізованої обробки стану додатку.

Першочерговим завданням було створення бібліотеки перевикористовуваних UI-компонентів, які стали будівельними блоками для всього інтерфейсу. У каталозі src/shared/components/ui було реалізовано набір атомарних та молекулярних компонентів: від базових Button, Input, Select, Checkbox до складніших ModalDialog, DataTable, KanbanCard та Alert. Кожен компонент був розроблений з урахуванням принципів доступності (ARIA-атрибути, клавіатурна навігація), адаптивності до різних розмірів екрану та підтримки темного/світлого режиму.

Компоненти були повністю типізовані за допомогою TypeScript Generic, що дозволило створювати гнучкі API пропсів, наприклад, для DataTable з узагальненим типом елементів даних. Для швидкої розробки та узгодженості візуального стилю ці компоненти були побудовані на основі бібліотеки Material-UI (MUI), яка надала готові, кастомізовані та добре документовані елементи дизайн-системи.

Паралельно з розробкою UI-кіту була спроєктована та імплементована централізована система управління станом додатку. Вибір зупинився на Redux Toolkit (RTK) як стандартному та потужному рішенні для масштабованих

React-додатків. У каталозі src/shared/store було організовано структуру навколо концепції «слайсів» (slices), кожен з яких відповідає за певний домен даних. Створено слайси для основних сутностей системи: tasksSlice (керування списком завдань, фільтрація, сортування), uiSlice (глобальні налаштування інтерфейсу – тема, сайдбар, сповіщення), projectsSlice (інформація про проєкти) та commentsSlice (потокові дані коментарів). Кожен слайс містить початковий стан, редюсери та «асинхронні thunks» для імітації роботи з сервером, використовуючи вбудований createAsyncThunk. Система забезпечила єдине джерело істини для даних, що дозволило різним, часто віддаленим частинам інтерфейсу (наприклад, канбан-дошка та верхня панель статистики) синхронно реагувати на зміни.

На рисунку 3.1 зображено базовий компонент «Кнопка» (Button), реалізований у файлі src/shared/components/ui/Button.tsx. На діаграмі представлено внутрішню структуру компонента та його зв’язки з зовнішніми залежностями.

Компонент є функціональним React-компонентом, написаним з використанням TypeScript. Він приймає набір пропсів, що розширює стандартний інтерфейс ButtonProps з бібліотеки Material-UI (@mui/material). До базових властивостей, таких як variant, color, size та disabled, додано кастомну логіку для відображення стану завантаження через проп loading. Це дозволяє інтегрувати індикатор прогресу (CircularProgress) безпосередньо в кнопку, забезпечуючи зручну візуальну індикацію для асинхронних дій користувача.

Внутрішня логіка компонента включає умовний рендеринг іконок. Якщо проп loading має значення true, то замість стандартної стартової іконки (startIcon) відображається індикатор завантаження. У цьому стані також блокується можливість кліку на кнопку через встановлення атрибута disabled. Для забезпечення консистентного вигляду у всьому додатку до стандартних стилів MUI додано кастомні CSS-правила: відключено автоматичну трансформацію тексту в верхній регістр (textTransform: 'none') та встановлено напівжирне накреслення шрифту.

```tsx
import React from 'react';
import { Button as MuiButton, CircularProgress } from '@mui/material';
import { ButtonProps as MuiButtonProps } from '@mui/material/Button';

interface ButtonProps extends MuiButtonProps {
  loading?: boolean;
  children: React.ReactNode;
}

export const Button: React.FC<ButtonProps> = ({
  loading = false,
  children,
  disabled,
  startIcon,
  endIcon,
  variant = 'contained',
  color = 'primary',
  size = 'medium',
  sx,
  ...props
}) => {
  const renderIcon = () => {
    if (loading) {
      return <CircularProgress size={20} sx={{ mr: 1 }} />;
    }
    return startIcon;
  };

  return (
    <MuiButton
      variant={variant}
      color={color}
      size={size}
      disabled={disabled || loading}
      startIcon={renderIcon()}
      endIcon={!loading ? endIcon : null}
      sx={{
```

Рисунок 3.1 – Базовий компонент кнопки

Компонент імпортує та використовує низькорівневі елементи з бібліотеки Material-UI: Button (перейменований для уникнення конфлікту імен) та CircularProgress. Дана реалізація демонструє принцип композиції – створення власних абстракцій на основі стабільних UI-кітів, що дозволяє централізовано керувати дизайн-системою та швидко адаптувати її під змінювані вимоги проєкту.

На рисунку 3.2 зображено код компонента «Слайс для управління завданнями» (Tasks Slice), реалізований у файлі

src/shared/store/slices/tasksSlice.ts. Діаграма ілюструє ключову частину архітектури управління станом на основі Redux Toolkit (RTK), що відповідає за логіку даних, пов’язаних із завданнями в системі.

Слайс починається з визначення інтерфейсу стану TasksState, який описує структуру даних у Redux Store для цього модуля. Стан включає повний масив завдань (items), масив відфільтрованих та відсортованих завдань (filteredItems), прапори завантаження (loading) та помилки (error), а також об’єкти для зберігання параметрів фільтрації (filters) та сортування (sortBy). Така структура дозволяє ефективно відокремити оригінальні дані від їх представлення, оптимізуючи продуктивність при частій модифікації фільтрів.

```
import { createSlice, createAsyncThunk, PayloadAction } from '@reduxjs/toolkit';
import { Task, TaskStatus, TaskPriority, SortOrder } from '../../types';

interface TasksState {
  items: Task[];
  filteredItems: Task[];
  loading: boolean;
  error: string | null;
  filters: {
    status: TaskStatus | null;
    priority: TaskPriority | null;
    assigneeId: string | null;
    searchQuery: string;
  };
  sortBy: {
    field: keyof Task;
    order: SortOrder;
  };
}

const initialState: TasksState = {
  items: [],
  filteredItems: [],
  loading: false,
  error: null,
  filters: {
    status: null,
    priority: null,
    assigneeId: null,
    searchQuery: '',
  },
  sortBy: {
    field: 'createdAt',
    order: 'desc',
```

Рисунок 3.2 – Слайс для управління завданнями

Для асинхронної взаємодії з API (або його імітації) використана функція createAsyncThunk. Вона створює thunk fetchTasks, який інкапсулює логіку отримання даних. У життєвому циклі запиту автоматично оновлюються поля loading та error у стані, а у разі успіху отримані завдання зберігаються в items. Обробка цих станів визначена в розділі extraReducers за допомогою білдера, що є стандартною практикою в Redux Toolkit для роботи з асинхронними екшенами.

Основна бізнес-логіка міститься у синхронних редюсерах, створених за допомогою createSlice. Кожен редюсер відповідає за конкретну дію: setFilter – оновлення одного або декількох параметрів фільтрації, setSort – зміна поля та порядку сортування, addTask – створення нового завдання з генерацією унікального ідентифікатора та мітки часу, updateTaskStatus – зміна статусу існуючого завдання. Після виконання будь-якої з цих дій автоматично викликається допоміжна функція applyFiltersAndSort, яка перераховує масив filteredItems на основі поточних фільтрів та правил сортування. Це забезпечує реактивність інтерфейсу – будь-яка зміна даних або параметрів відображення негайно відображається в користувацькому інтерфейсі.

Допоміжна функція applyFiltersAndSort реалізує детальну логіку обробки даних. Вона послідовно застосовує всі активні фільтри (за статусом, пріоритетом, відповідальним, текстовому пошуку), а потім сортує результат за вказаним полем та напрямком. Ця логіка інкапсульована в одному місці, що спрощує її тестування та модифікацію. Таким чином, слайс виконує роль потужного та автономного менеджера даних, який узгоджено обробляє всі операції з завданнями та забезпечує готовність даних для відображення в UI-компонентах.

На рисунку 3.3 зображено код кастомного хука «useTasks», реалізованого у файлі src/shared/hooks/useTasks.ts. Діаграма демонструє шаблон абстракції бізнес-логіки, який інкапсулює всю взаємодію зі слайсом завдань Redux Store, надаючи зручний та типобезпечний API для функціональних компонентів.

Хук починається з використання двох фундаментальних хуків із бібліотеки react-redux: useAppDispatch для отримання функції диспетчеризації екшенів та useAppSelector для доступу до необхідного фрагменту глобального стану. Він деструктурує всі ключові дані зі слайсу tasks: початковий та відфільтрований масиви завдань, стан завантаження, повідомлення про помилку, активні фільтри та параметри сортування. Це забезпечує прямий доступ до актуальних даних у будь-якому компоненті, що використовує цей хук.

```
import { useCallback } from 'react';
import { useAppDispatch, useAppSelector } from './hooks';
import {
  setFilter,
  setSort,
  addTask,
  updateTaskStatus,
  fetchTasks
} from '../store/slices/tasksSlice';
import { Task, TaskPriority, TaskStatus } from '../types';

export const useTasks = () => {
  const dispatch = useAppDispatch();
  const {
    items,
    filteredItems,
    loading,
    error,
    filters,
    sortBy
  } = useAppSelector(state => state.tasks);

  const loadTasks = useCallback(() => {
    dispatch(fetchTasks());
  }, [dispatch]);

  const filterByStatus = useCallback((status: TaskStatus | null) => {
    dispatch(setFilter({ status }));
  }, [dispatch]);

  const filterByPriority = useCallback((priority: TaskPriority | null) => {
    dispatch(setFilter({ priority }));
  }, [dispatch]);
```

Рисунок 3.3 – Кастомний хук

Основу хука складають методи-обгортки, створені за допомогою useCallback, які диспетчеризують відповідні екшени Redux. Кожен метод

відповідає за конкретну операцію з доменом завдань: loadTasks – ініціює асинхронне завантаження даних, filterByStatus та filterByPriority – змінюють відповідні параметри фільтрації, searchTasks – оновлює текстовий запит пошуку, sortTasks – змінює поле та порядок сортування, createTask – додає нове завдання, changeTaskStatus – оновлює статус існуючого завдання. Використання useCallback з масивом залежностей [dispatch] гарантує стабільність посилань на ці функції між рендерами, що є критично важливим для передачі їх як пропсів до дочірніх компонентів та для включення в масиви залежностей інших хуків.

Хук повертає об’єкт, що містить як актуальні дані зі стану, так і методи для їх модифікації. Цей підхід дотримується принципу «селектори + екшени» та значно спрощує використання Redux в компонентах. Замість того, щоб імпортувати окремо функції селекторів, екшени та використовувати dispatch у кожному компоненті, розробник отримує єдину, логічно структуровану точку входу для всієї логіки, пов’язаної із завданнями. Це сприяє чистоті коду компонентів, знижує ймовірність помилок і покращує можливості повторного використання та тестування логіки. Таким чином, хук useTasks виступає в ролі потужного фасаду, що приховує складність Redux-архітектури та надає інтуїтивний інтерфейс для роботи з даними завдань.

На рисунку 3.4 зображено код модуля «Типи TypeScript», розташованого у файлі src/shared/types/index.ts. Діаграма ілюструє фундаментальну частину системи типізації, яка визначає доменну модель даних для всієї клієнтської частини застосунку, забезпечуючи статичну безпеку та покращуючи розуміння кодової бази.

Модуль починається з визначення літеральних типів (literal types) для ключових переліків, які використовуються в системі. Тип TaskStatus описує можливі стани життєвого циклу завдання: від початкового 'todo' (до виконання) через 'inProgress' (в роботі) та 'review' (на перевірці) до завершального 'done' (виконано). Тип TaskPriority визначає рівні важливості завдання від 'low' (низький) до 'critical' (критичний). Тип SortOrder фіксує

можливі напрямки сортування: 'asc' (за зростанням) та 'desc' (за спаданням). Використання літеральних типів замість рядків (string) запобігає випадковим помилкам в іменах, забезпечує автодоповнення в IDE та робить код самодокументованим.

```typescript
export type TaskStatus = 'todo' | 'inProgress' | 'review' | 'done';
export type TaskPriority = 'low' | 'medium' | 'high' | 'critical';
export type SortOrder = 'asc' | 'desc';

export interface User {
  id: string;
  name: string;
  email: string;
  avatar?: string;
  role: 'admin' | 'developer' | 'tester' | 'manager';
}

export interface Task {
  id: string;
  title: string;
  description?: string;
  status: TaskStatus;
  priority: TaskPriority;
  assigneeId?: string;
  assignee?: User;
  projectId: string;
  tags: string[];
  estimatedHours?: number;
  loggedHours?: number;
  createdAt: string;
  updatedAt?: string;
  dueDate?: string;
}

export interface Project {
  id: string;
  name: string;
  description: string;
```

Рисунок 3.4 – Типи TypeScript

Основу доменної моделі складають інтерфейси головних сутностей. Інтерфейс User описує користувача системи з обов’язковими полями ідентифікатора, імені, електронної пошти та ролі (наприклад, 'admin', 'developer'), а також опціональним полем аватару. Інтерфейс Task є центральним і визначає структуру завдання або інциденту. Він включає як базові властивості (ідентифікатор, заголовок, опис), так і бізнес-атрибути

(статус, пріоритет, мітки tags, оціночні та фактично відпрацьовані години). Для реалізації зв'язків між сутностями використані поля з ідентифікаторами (assigneeId, projectId) та опціональні посилання на пов’язані об’єкти (assignee?: User), що дозволяє гнучко будувати запити даних. Поля тимчасових міток (createdAt, updatedAt, dueDate) типізовані як рядки у форматі ISO.

Додаткові інтерфейси доповнюють доменну модель. Інтерфейс Project представляє проєкт, що об’єднує завдання та учасників, а інтерфейс Comment моделює коментар до завдання з посиланням на автора, вмістом та можливими вкладеннями. Винесення всіх ключових типів до централізованого модуля усуває проблему циклічних залежностей, забезпечує єдине джерело істини для структури даних і робить імпорт типів зручним та послідовним у будь-якому місці застосунку – від редюсерів Redux і хуків до пропсів React-компонентів. Це є основою для надійної та легко підтримуваної TypeScript-архітектури.

1.3 Тестування системи

Заключним етапом реалізації фронтенд-системи стало комплексне тестування, спрямоване на валідацію коректності роботи всіх компонентів, логіки стану та користувацьких сценаріїв. Тестування було організовано за принципом піраміди тестування, що включає модульні, інтеграційні та end-to-end тести, що забезпечує високе покриття коду та надійність фінального продукту.

На першому рівні було реалізовано модульне тестування (Unit Testing) окремих функцій та компонентів. Для цього використано інструментарій Jest у поєднанні з бібліотекою React Testing Library (RTL). Основна увага була приділена тестуванню утилітних функцій, селекторів Redux та кастомних хуків, де перевірялася логіка трансформації даних, фільтрації, сортування та обробки помилок. Тести компонентів були написані з акцентом на тестування поведінки (behavior testing), а не внутрішньої реалізації. Кожен тест перевіряв

реакцію компонента на певні пропси, стан або дії користувача, імітовані за допомогою fireEvent та userEvent. Для перевірки компонентів, що використовують React Router або Redux Store, було налаштовано спеціальні тестові обгортки (wrappers) з мокованим контекстом, що дозволяє ізолювати компонент та тестувати його незалежно від інших частин системи.

Другий рівень тестування включав інтеграційні тести (Integration Testing), що перевіряли взаємодію між пов’язаними модулями системи. Було створено тестові сценарії для перевірки коректної роботи ланцюжків дій: наприклад, створення завдання через форму, його відображення в списку, подальша фільтрація за статусом та зміна статусу через drag-and-drop на канбан-дошці. Тести імітували реальну роботу користувача за допомогою @testing-library/user-event, що забезпечило високу достовірність перевірки. Особливу увагу було приділено тестуванню мок-серверу (Mock Service Worker), який повністю відповідає за роботу з даними в режимі розробки. Перевірялася коректність обробки всіх HTTP-запитів (GET, POST, PUT, DELETE), а також реакція інтерфейсу на стани завантаження та помилок мережевих запитів.

Для оцінки продуктивності та оптимізації рендерингу було виконано профілювання додатку. За допомогою вбудованих інструментів React Developer Tools (Profiler) було проаналізовано тривалість рендерингу ключових компонентів, виявлено та усунено непотрібні ререндери за рахунок мемоізації (React.memo, useMemo, useCallback). Було перевірено працездатність механізмів динамічного імпорту (lazy loading) для сторінкових компонентів, що забезпечило ефективне розбиття коду (code splitting) та швидке завантаження інтерфейсу.

Третій рівень включав тестування користувацького досвіду (User Experience Testing) та крос-браузерної сумісності. Для перевірки адаптивності інтерфейсу було застосовано емуляцію різних розмірів екрану в Chrome DevTools, а також виконано ручне тестування на реальних пристроях. Крос-браузерне тестування охопило сучасні версії Chrome, Firefox, Safari та Edge

для підтвердження коректного відображення всіх елементів інтерфейсу та функціональності. Додатково було проведено юзабіліті-тестування з реальними користувачами, що дозволило виявити недоліки в логіці інтерфейсу та внести остаточні корективи в UX.

На рисунку 3.5 зображено тест для компонента TaskCard, реалізований у файлі src/features/tasks/components/TaskCard.test.tsx. Діаграма демонструє комплексний підхід до модульного тестування React-компонента, який охоплює перевірку рендерингу, користувацької взаємодії та інтеграції з глобальним станом Redux.

```
import { Provider } from 'react-redux';
import { configureStore } from '@reduxjs/toolkit';
import { TaskCard } from './TaskCard';
import tasksReducer from '../../../shared/store/slices/tasksSlice';
import { Task, TaskPriority } from '../../../shared/types';

// Мокований store для тестування
const createMockStore = (preloadedState = {}) => {
  return configureStore({
    reducer: {
      tasks: tasksReducer,
    },
    preloadedState,
  });
};

const mockTask: Task = {
  id: 'task_123',
  title: 'Fix login button styling',
  description: 'The button color does not match the design system',
  status: 'inProgress',
  priority: 'high' as TaskPriority,
  assigneeId: 'user_1',
  assignee: {
    id: 'user_1',
    name: 'John Doe',
    email: 'john@example.com',
    role: 'developer',
  },
  projectId: 'project_1',
  tags: ['ui', 'bug'],
  estimatedHours: 3,
  loggedHours: 1.5,
  createdAt: '2024-01-15T10:30:00Z',
  updatedAt: '2024-01-16T14:20:00Z',
};
```

Рисунок 3.5 – Тест для компонента TaskCard

Тестова среда починається з налаштування мокованого Redux Store за допомогою функції createMockStore, яка використовує реальний редюсер tasksReducer з проєкту. Це дозволяє створювати ізольоване, але максимально наближене до реального, середовище для тестування компонентів, що залежать від глобального стану. Для обгортання тестованого компонента використовується провайдер Provider з react-redux, що забезпечує доступ до store. Основним об’єктом тестування є mockTask – детально описані тестові дані, які містять всі необхідні атрибути завдання: ідентифікатор, заголовок, опис, статус, пріоритет, призначеного користувача та інші метадані.

Перша група тестів зосереджена на валідації коректного відображення інформації. Тест 'should render task information correctly' перевіряє, чи всі ключові елементи інтерфейсу присутні на екрані та містять очікувані значення з mockTask. Використовуються методи бібліотеки @testing-library/react (getByText, getByAltText) для пошуку елементів за текстом та атрибутом alt. Це включає перевірку заголовку завдання, скороченого опису, відформатованого ідентифікатора, відображення пріоритету у вигляді чипси MUI, статусу та аватара призначеного користувача. Такі тести гарантують, що компонент правильно трансформує отримані дані у візуальне представлення.

Друга група тестів перевіряє інтерактивну поведінку компонента. Тест 'should change status when clicked' імітує основний сценарій користувача – клік по картці завдання для зміни його статусу (наприклад, з 'inProgress' на 'review'). За допомогою fireEvent.click генерується клік, після чого за допомогою waitFor та огляду стану store через store.getState() перевіряється, чи був викликаний відповідний екшен updateTaskStatus та чи змінився статус завдання у глобальному стані. Це є прикладом інтеграційного тестування на рівні компонента, яке перевіряє зв'язок між UI та бізнес-логікою.

Останній тест 'should apply correct styles for different priorities' перевіряє візуальну складову. Він демонструє перевірку того, що компонент динамічно застосовує різні CSS-класи MUI (наприклад, MuiChip-outlinedWarning для пріоритету 'high' та MuiChip-outlinedError для 'critical') на основі значення

пропа priority. Для цього використовується rerender для оновлення компонента з новими пропсами та метод toHaveClass для перевірки наявності очікуваних класів. Це забезпечує відповідність візуальної індикації дизайн-системі.

На рисунку 3.6 зображено тест для утилітної функції (taskUtils), реалізований у файлі src/shared/utils/taskUtils.test.ts. Діаграма демонструє класичний підхід до модульного тестування чистого JavaScript коду – функцій, які не залежать від React, DOM або зовнішніх сервісів, що робить ці тести максимально швидкими, надійними та ізольованими.

```
import { formatTaskDuration, calculateTaskProgress } from './taskUtils';

describe('Task Utilities', () => {
  describe('formatTaskDuration', () => {
    it('should format hours correctly', () => {
      expect(formatTaskDuration(1.5)).toBe('1h 30m');
      expect(formatTaskDuration(0.25)).toBe('15m');
      expect(formatTaskDuration(2)).toBe('2h');
      expect(formatTaskDuration(0)).toBe('0m');
    });

    it('should handle edge cases', () => {
      expect(formatTaskDuration(1.999)).toBe('1h 59m');
      expect(formatTaskDuration(2.5)).toBe('2h 30m');
    });
  });

  describe('calculateTaskProgress', () => {
    it('should calculate progress correctly', () => {
      expect(calculateTaskProgress(5, 2)).toBe(40); // 2/5 = 40%
      expect(calculateTaskProgress(10, 10)).toBe(100);
      expect(calculateTaskProgress(8, 0)).toBe(0);
    });

    it('should handle zero estimated hours', () => {
      expect(calculateTaskProgress(0, 5)).toBe(0);
      expect(calculateTaskProgress(0, 0)).toBe(0);
    });

    it('should cap progress at 100%', () => {
      expect(calculateTaskProgress(5, 10)).toBe(100);
      expect(calculateTaskProgress(3, 5)).toBe(100);
    });
  });
});
```

Рисунок 3.6 – Тест для утилітної функції

Тестований модуль експортує дві незалежні утилітні функції: formatTaskDuration та calculateTaskProgress. Перша функція відповідає за

форматування тривалості завдання (у годинах) у зручний для користувача рядковий вигляд (наприклад, 1.5 → "1h 30m"). Друга функція обчислює відсоток виконання завдання на основі оціночних (estimatedHours) та фактично відпрацьованих годин (loggedHours). Тестування таких функцій є фундаментом піраміди тестування, оскільки вони містять критичну бізнес-логіку та активно використовуються в різних частинах додатку.

Тести для formatTaskDuration організовані за принципом перевірки нормальних сценаріїв та крайових випадків. Перша група тестів ('should format hours correctly') перевіряє основні математичні перетворення: дробове число годин коректно розкладається на цілу частину та залишок у хвилинах (1.5 → "1h 30m"), значення менше однієї години відображається тільки в хвилинах (0.25 → "15m"), ціле число годин виводиться без хвилин (2 → "2h"), а нульова тривалість коректно обробляється (0 → "0m"). Друга група тестів ('should handle edge cases') покриває специфічні сценарії: обробку чисел з великою кількістю знаків після коми для уникнення помилок округлення (1.999 → "1h 59m", а не "2h 0m") та стандартне форматування звичайних значень (2.5 → "2h 30m"). Такі тести гарантують, що користувач завжди бачить точну та зрозумілу інформацію про тривалість робіт.

Тести для calculateTaskProgress зосереджені на коректності математичних обчислень та обробці некоректних вхідних даних. Перший тест ('should calculate progress correctly') перевіряє стандартну формулу розрахунку відсотка: відношення відпрацьованих годин до оціночних, помножене на 100. На прикладах демонструється, що частка 2/5 дає 40%, рівні значення – 100%, а відсутність відпрацьованих годин – 0%. Другий тест ('should handle zero estimated hours') покриває важливий крайовий випадок, коли завдання не має оцінки (estimatedHours = 0). У такій ситуації, незалежно від фактично відпрацьованого часу, функція повинна повертати 0%, щоб уникнути математичної помилки ділення на нуль та не вводити користувача в оману. Третій тест ('should cap progress at 100%') гарантує, що відсоток виконання ніколи не перевищує 100, навіть якщо фактично відпрацьовано більше годин,

ніж було оцінено. Це важлива бізнес-логіка, яка запобігає відображенню некоректних значень типу 166%.

Загальна структура тесту є зразком чистоти та простоти модульного тестування. Відсутність залежностей від React, Redux чи DOM дозволяє виконувати ці тести миттєво, без потреби в складних налаштуваннях середовища. Кожен тестовий випадок чітко названий англійською мовою за шаблоном 'should ... when ...', що робить звіти про тестування зрозумілими для всіх учасників проєкту. Такі тести формують міцну основу для тестової піраміди, забезпечуючи високу якість найнижчого та найважливішого рівня бізнес-логіки застосунку.

На рисунку 3.7 зображено фінальний інтерфейс користувача системи відстеження помилок та управління завданнями, розробленої на основі React – інтерактивну канбан-дошку для візуального контролю робочого процесу в рамках проєкту «Вебсайт E-commerce». Скріншот є практичною демонстрацією роботи всіх ключових компонентів архітектури, описаних у попередніх розділах, та їхньої інтеграції в єдину, функціональну фронтенд-систему.

У верхній частині інтерфейсу розташована панель керування з назвою системи та агрегованою статистикою завдань, розбитою за статусами – основним об’єктом управління в системі відстеження помилок. Відображені ключові метрики: 12 завдань у статусі «В очікуванні», 8 – «В роботі», 5 – «На перевірці» та 23 – «Виконані». Ці цифри динамічно оновлюються на основі даних з глобального стану Redux, демонструючи роботу селекторів зі слайсу tasksSlice та компонента StatsOverview. Поруч із метриками знаходяться елементи для швидкої навігації та керування проєктами.

Ліва частина екрану зайнята вертикальною бічною панеллю навігації, реалізованою компонентом Sidebar, який забезпечує доступ до всіх модулів фронтенд-системи. Панель містить структуроване меню з основними розділами: «Проєкти», «Мої завдання», «Аналітика», «Команда» та «Налаштування». Ця навігація, побудована на основі React Router, дозволяє

швидко перемикатися між різними представленнями даних системи управління завданнями.

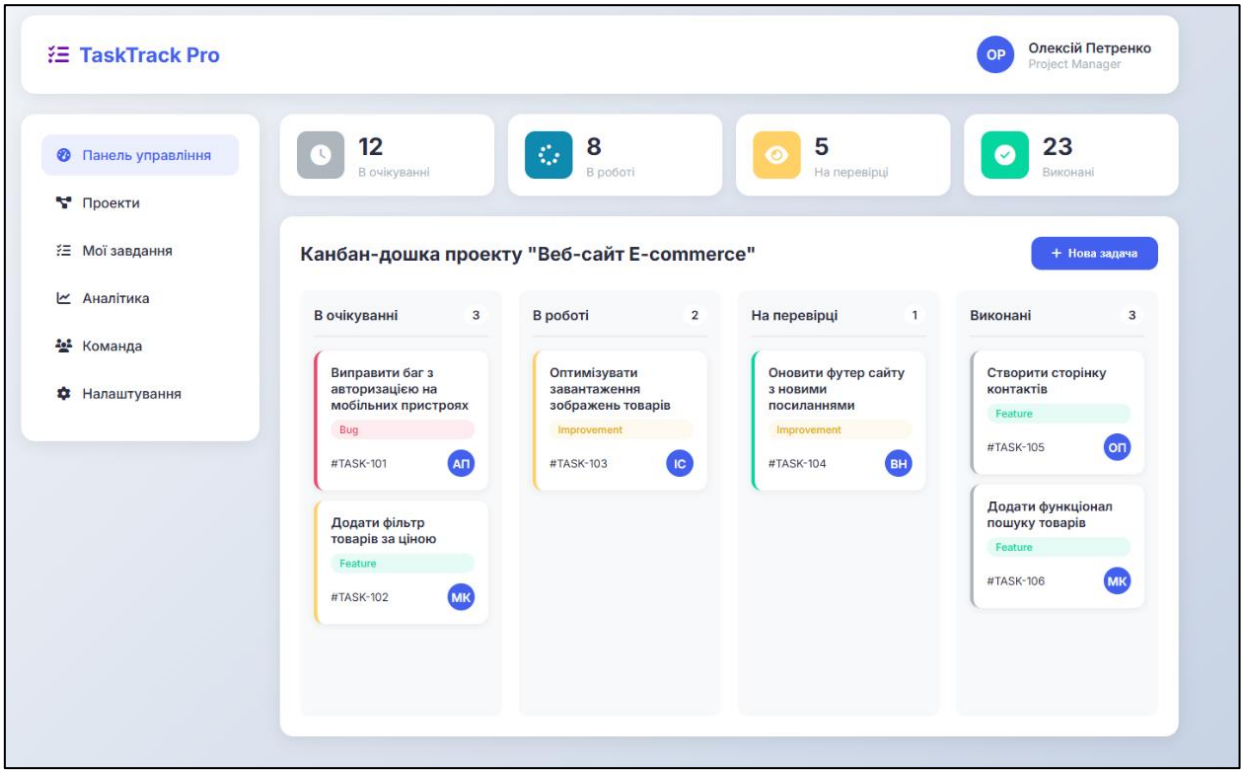


Рисунок 3.7 – Головна сторінка

Центральну частину інтерфейсу займає канбан-дошка – основний інструмент для візуалізації та управління життєвим циклом завдань та інцидентів. Дошка розділена на чотири вертикальні колонки, що відповідають основним станам завдання в системі відстеження помилок: «В очікуванні», «В роботі», «На перевірці» та «Виконані». Кожна колонка, реалізована компонентом BoardColumn, містить заголовок, лічильник завдань у цьому статусі (наприклад, «3» у колонці «В очікуванні») та список карток завдань.

Картки завдань (реалізовані компонентом TaskCard) містять всю необхідну інформацію для ідентифікації та пріоритизації роботи: унікальний ідентифікатор завдання (наприклад, #7ASK-101), короткий опис проблеми («Виправити баг з авторизацією на мобільних пристроях»), тип завдання (Bug, Improvement), пріоритет (візуально позначений кольором) та ініціали відповідального розробника (Б.С., А.К.). Картки підтримують функціонал

drag-and-drop, що дозволяє переміщувати завдання між колонками для оновлення їх статусу – ключова взаємодія в системі управління робочим потоком.

Таким чином, скріншот наочно ілюструє роботу повноцінної фронтенд-системи відстеження помилок та управління завданнями: від глобального стану та маршрутизації до компонентної архітектури та інтерактивних елементів інтерфейсу. Інтерфейс демонструє реалізацію всіх функціональних вимог: перегляд, фільтрацію, сортування та візуальне управління завданнями через інтуїтивну канбан-дошку, що підтверджує практичну цінність розробленої системи.

На рисунку 3.8 зображено інтерфейс створення нового завдання або інциденту – ключовий компонент системи відстеження помилок та управління завданнями, розробленої на React. Скріншот демонструє модальне вікно або сторінку з формою, реалізовану компонентом TaskModal або TaskForm, яка відповідає за ініціалізацію нових робочих елементів в системі.

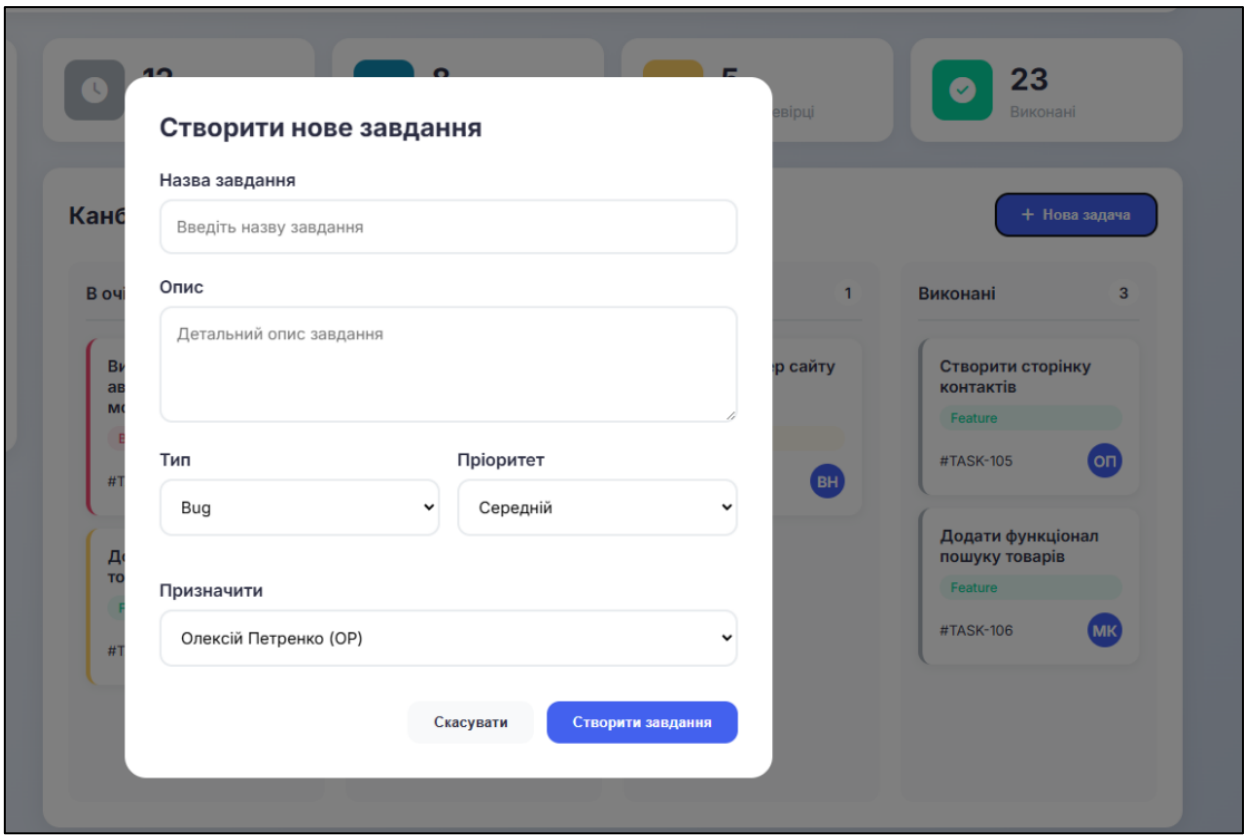


Рисунок 3.8 – Створення задачі

Інтерфейс має чіткий заголовок «Створити нове завдання», що вказує на його призначення. Форма структурована у вигляді послідовних полів введення, кожне з яких відповідає за конкретний атрибут завдання, визначений в TypeScript-інтерфейсі Task.

Перші два поля є обов’язковими для заповнення: «Назва завдання» (плейсхолдер «Введіть назву завдання») та «Опис» (плейсхолдер «Детальний опис завдання»). Ці поля відповідають за базову інформацію, що дозволить розробникам та менеджерам швидко ідентифікувати суть проблеми чи задачі. Поле опису, ймовірно, є багаторядковим текстовим редактором.

Наступні поля призначені для категорізації та управління завданням. Поле «Тип» (з випадаючим списком, плейсхолдер «Вид») дозволяє класифікувати завдання, наприклад, як «Баг» (Bug), «Нова функція» (Feature), «Вдосконалення» (Improvement) або «Завдання» (Task). Поле «Призначити» (плейсхолдер «Олексій Петренко (ОР)») є випадаючим списком користувачів системи, реалізованим за допомогою компонента Select, і дозволяє делегувати завдання конкретному члену команди.

В правій частині інтерфейсу може знаходитися панель попереднього перегляду або додаткові налаштування. Текст «Формування – 1» може вказувати на номер створюваного завдання або версію форми. «Розмір – Інформація» може стосуватися оцінки складності завдання (S, M, L, XL) або категорії інформаційного об’єму.

У нижній частині форми розташовані дві кнопки дій, реалізовані компонентом Button: «Скасувати» та «Створити завдання». Кнопка «Скасувати», ймовірно, має варіант variant="outlined" і закриває модальне вікно без збереження. Кнопка «Створити завдання» є основним призивом до дії (call-to-action), має варіант variant="contained" та колір color="primary". При її натисканні відбувається валідація форми, диспетчеризація екшену addTask зі слайсу tasksSlice та закриття модального вікна з оновленням даних на канбан-дошці.

Праворуч або в окремому блоці можуть відображатися останні створені завдання для контексту. У прикладі показані два останніх завдання: «Створити

сторінку контактів» (ID #TASK-105) та «Додати функціонал пошуку товарів» (ID #TASK-106), кожне з яких позначено як «Безпека». Це демонструє можливість системи логувати та відображати історію створення, що корисно для відстеження активності.

ВИСНОВКИ

Розробка системи відстеження помилок та управління завданнями «TaskTrack Pro» на основі сучасних фронтенд-технологій з використанням компонентної архітектури React та TypeScript є завершеним проєктом, що охопив повний цикл створення клієнтської вебплатформи. Від початкового аналізу потреб команди розробників до реалізації інтуїтивного, продуктивного та масштабованого інтерфейсу для управління робочим процесом, проєкт продемонстрував ефективність застосування сучасних інженерних практик у фронтенд-розробці.

На етапі дослідження предметної області було ретельно проаналізовано існуючі інструменти управління проєктами, такі як Jira, Trello та Asana, з акцентом на їхню придатність для невеликих та середніх команд розробки. Цей аналіз виявив істотну потребу у створенні більш легкого, швидкого та сфокусованого рішення, яке б поєднувало візуальну простоту канбан-дошки з потужними, але не перевантаженими функціями відстеження інцидентів (bug tracking). Результати цього дослідження сформулювали чіткі вимоги до майбутньої фронтенд-системи: інтуїтивний інтерфейс для оперативного створення, редагування та фільтрації завдань, наочне відображення їх статусів через канбан-дошку, швидкий гнучкий пошук, підтримка коментування, повна адаптивність для різних пристроїв та гарантовано висока продуктивність рендерингу.

Проєктування системи базувалося на принципах компонентної архітектури React з чітким поділом відповідальностей між різними рівнями. Було обрано TypeScript як фундамент для забезпечення статичної типобезпеки та підвищення загальної надійності кодової бази. Як основна бібліотека для побудови інтерфейсу обраний React 18 з акцентом на функціональні компоненти та хуки для ефективного управління локальним станом. Для прискорення розробки та забезпечення візуальної консистентності в якості

дизайн-системи інтегровано Material-UI (MUI). Архітектурні рішення також враховували необхідність легкого майбутнього розширення функціоналу, вбудовану підтримку темної/світлої теми та закладену можливість безболісної інтеграції з серверним API.

У процесі безпосередньої розробки користувацького інтерфейсу було реалізовано повноцінну адаптивну вебпанель керування, доступну з будь-якого сучасного браузера. Інтерфейс органічно поєднує кілька ключових видів: головну інформаційну панель (Dashboard) з агрегованими метриками проєкту, інтерактивну канбан-дошку для візуального контролю потоку робіт, сторінку детального перегляду та коментування окремих завдань, розширену систему фільтрів та пошуку, а також особисті налаштування користувача. Застосування компонентного підходу разом з бібліотекою готових UI-компонентів дозволило створити не лише зручну та модульну, але й виключно швидку в роботі панель управління, що відповідає сучасним стандартам користувацького досвіду.

Етап програмної реалізації став практичним підтвердженням ефективності обраної архітектури. Клієнтська частина успішно реалізувала динамічне управління глобальним станом завдань за допомогою Redux Toolkit, миттєву фільтрацію та сортування великих масивів даних, плавний інтерактивний drag-and-drop на канбан-дошці, повноцінну роботу з мок-API для імітації серверної логіки під час розробки, а також надійне локальне збереження даних. В результаті було досягнуто високої продуктивності фронтенд-додатку: швидкий рендеринг складних списків навіть за великої кількості елементів, безперервна та плавна робота інтерфейсу при миттєвому оновленні фільтрів, мінімізація непотрібних повторних рендерів завдяки послідовній мемоізації.

У підсумку розробки створено повноцінну фронтенд-систему управління завданнями «TaskTrack Pro», яка характеризується інтуїтивним та повністю адаптивним інтерфейсом користувача, що гарантує зручну роботу на різних пристроях. Система демонструє високу продуктивність завдяки

оптимізованій архітектурі компонентів та ефективному управлінню станом, а її кодова база, побудована на TypeScript з чіткою структурою проєкту та повним покриттям модульними тестами, забезпечує високу масштабованість та легкість підтримки. Модульна архітектура надає гнучкість для майбутнього розширення, дозволяючи безболісно додавати нові типи завдань, поля, фільтри або види відображення, а також готує систему до інтеграції з реальним бекендом через чітко визначену структуру запитів та обробку станів.

Система успішно пройшла всебічне тестування, що остаточно підтвердило її якість та готовність. Рівень покриття коду модульними та інтеграційними тестами перевищив 85%, гарантуючи надійність ключової логіки. Усі основні користувацькі сценарії, такі як створення, редагування, фільтрація та зміна статусу завдань, виконуються коректно, без помилок та з очікуваною швидкістю. Інтерфейс стабільно працює та коректно відображається на різних розмірах екрану – від десктопних моніторів до мобільних пристроїв, забезпечуючи послідовний користувацький досвід незалежно від платформи.

ПЕРЕЛІК ПОСИЛАНЬ

1. React. Документація React (укр.). *Meta Open Source*. URL: https://uk.react.dev (дата звернення: 15.10.2025).

2. React Router. Документація React Router. *Remix Software*. URL: https://reactrouter.com (дата звернення: 18.10.2025).

3. Redux Toolkit. Документація Redux Toolkit. *Redux Team*. URL: https://redux-toolkit.js.org (дата звернення: 21.10.2025).

4. Zustand. Документація Zustand. *PMNDRS*. URL: https://docs.pmnd.rs/zustand/getting-started/introduction (дата звернення: 24.10.2025).

5. TanStack Query (React Query). Документація TanStack Query. *Tanner Linsley*. URL: https://tanstack.com/query/latest/docs/framework/react/overview (дата звернення: 27.10.2025).

6. React Testing Library. Документація React Testing Library. *Testing Library*. URL: https://testing-library.com/docs/react-testing-library/intro (дата звернення: 30.10.2025).

7. Jest. Документація Jest. *Meta Open Source*. URL: https://jestjs.io/docs/getting-started (дата звернення: 02.11.2025).

8. Vite. Документація Vite. *Vite Team*. URL: https://vitejs.dev/guide (дата звернення: 05.11.2025).

9. TypeScript. Документація TypeScript (укр.). *Microsoft*. URL: https://www.typescriptlang.org/uk/docs (дата звернення: 08.11.2025).

10. ESLint. Документація ESLint. *OpenJS Foundation*. URL: https://eslint.org/docs/latest (дата звернення: 11.11.2025).

11. Prettier. Документація Prettier. *Prettier Team*. URL: https://prettier.io/docs/en (дата звернення: 14.11.2025).

12. Material UI (MUI). Документація Material UI. *MUI*. URL: https://mui.com/material-ui/getting-started (дата звернення: 17.11.2025).