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

ПОЯСНЮВАЛЬНА ЗАПИСКА

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

«Вебсистема управління завданнями та командної взаємодії в ІТ-проєктах»

Виконав студент групи ІПЗ-22-4
ВАРНАВСЬКИЙ В'ячеслав Олексійович

Керівник роботи:
МОРОЗОВ Андрій Васильович

Рецензент:
САВІЦЬКИЙ Роман Святославович

Житомир – 2026

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

«ЗАТВЕРДЖУЮ»
Зав. кафедри інженерії програмного забезпечення
Тетяна ВАКАЛЮК
«28» лютого 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ВАРНАВСЬКИЙ В'ячеслав Олексійович**
Керівник роботи: **МОРОЗОВ Андрій Васильович**
Тема роботи: **«Вебсистема управління завданнями та командної взаємодії в ІТ-проєктах»**,
затверджена Наказом закладу вищої освіти від **«17» лютого 2026р., №92/с**
Вихідні дані для роботи: **розробка вебдодатку для управління завданнями.**
Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	**Консультант**	Дата	
		Завдання видав	Завдання прийняв
1	МОРОЗОВ А.В.	27.02.2026 р.	27.02.2026 р.
2	МОРОЗОВ А.В.	27.02.2026 р.	27.02.2026 р.
3	МОРОЗОВ А.В.	27.02.2026 р.	27.02.2026 р.

РЕФЕРАТ

Випускна робота бакалавра складається з реалізації веб-системи управління завданнями та командної взаємодії в ІТ-проєктах, а також пояснювальної записки. Пояснювальна записка до випускної роботи містить 98 сторінок, включає 75 ілюстрацій, 2 таблиці, 1 додаток та 30 використаних літературних джерел.

Основною метою роботи є побудова веб-системи для управління завданнями та командної взаємодії в ІТ-проєктах малих розмірів. Вебсистема підтримує декілька видів комунікації через коментарі та чати. Також реалізовані базові процеси AGILE планування та управління.

Під час роботи використані методи включають в себе: аналіз аналогів для виявлення областей для покращення, розробка веб-системи, використовуючи сучасні та надійні технології, аналіз потреб майбутніх користувачів та адміністраторів системи.

Результати кваліфікаційної роботи полягають у створенні веб-системи для управління та планування проєктів для невеликих команд з мінімальною втратою контексту роботи.

Ключові слова: AGILE, ПЛАНУВАННЯ, КОМУНІКАЦІЯ, ВЕБ, PHP.

ABSTRACT

The bachelor's thesis consists of the implementation of a web-based task management and team interaction system in IT projects, and an explanatory note. The explanatory note to the thesis contains 98 pages, includes 75 illustrations, 2 tables, 1 appendix and 30 used literature sources.

The main goal of the work is to build a web-based system for task management and team interaction in small-sized IT projects. The web-based system supports several types of communication through comments and chats. The basic AGILE planning and management processes are also implemented.

The methods used during the work include: analysis of analogues to identify areas for improvement, development of a web-based system using well-established technologies, analysis of the needs of future users and system administrators.

The results of the qualification work are the creation of a web application for project management and planning for small teams with minimal loss of work context.

Keywords: AGILE, PLANNING, COMMUNICATION, WEB, PHP.

Зміст

РЕФЕРАТ 4

ABSTRACT 4

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ 7

ВСТУП 8

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ 10

1.1 Актуальність та постановка задачі 10

1.2 Аналіз аналогів програмного продукту 13

1.3 Вибір архітектури 22

1.4 Обґрунтування вибору інструментальних засобів та вимоги до програмного забезпечення 26

Висновки до першого розділу 28

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ТА КОМАНДНОЇ ВЗАЄМОДІЇ В ІТ-ПРОЄКТАХ 30

2.1 Визначення варіантів використання та об’єктно-орієнтованої структури системи 30

2.2 Розробка бази даних системи 35

2.3 Проєктування та реалізація алгоритмів роботи системи 39

2.4 Реалізація вебсистеми управління завданнями та командної взаємодії в ІТ-проєктах 46

Висновки до другого розділу 50

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК ВИКОРИСТАННЯ ДОДАТКУ 52

3.1 Порядок та опис роботи з програмним додатком (Опис інтерфейсу). 52

3.2 Тестування роботи програмного комплексу 78

3.3 Порядок налаштування та встановлення системи 89

Висновки до третього розділу 92
ВИСНОВКИ 94
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 96
ДОДАТКИ 99

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ПЗ - програмне забезпечення.

БД - база даних.

СУБД - система управління базами даних.

CRUD - create, read, update, delete.

SQL - Structured Query Language.

HTML - HyperText Markup Language.

CSS - Cascading Style Sheets.

JS – JavaScript.

IOPS - Input/Output Operations Per Second.

ВСТУП

Актуальність теми. Сучасна сфера інформаційних технологій характеризується високою динамікою розробки, складністю архітектурних рішень та необхідністю постійної координації фахівців. Ефективна командна взаємодія є критичним фактором успіху будь-якого IT-проєкту, проте на практиці команди часто стикаються з проблемою розпорошення інформації. Використання великої кількості розрізнених додатків для спілкування, відстеження завдань та документування призводить до втрати контексту, дублювання даних та виникнення системного безладу. Більшість існуючих сервісів фокусуються на розв’язанні лише однієї конкретної задачі, не забезпечуючи належної інтеграції між комунікацією та процесом виконання завдань. Це створює бар'єри для своєчасного реагування на ризики та може негативно впливати на моральний стан учасників команди через вигорання та необхідність подвійної роботи. Таким чином, розробка інтегрованої вебсистеми, що виступає єдиним цифровим простором для проєктної діяльності, є актуальним завданням, спрямованим на підвищення прозорості та ефективності роботи команд.

Мета роботи. Метою кваліфікаційної роботи є розробка алгоритмів функціонування та програмна реалізація веб-системи управління завданнями та командної взаємодії, яка забезпечує усунення інформаційних розривів та збереження повного контексту роботи через глибоку інтеграцію комунікаційних засобів і засобів трекінгу завдань.

Завдання роботи. Для досягнення визначеної мети необхідно вирішити наступні завдання:

- Провести аналіз вимог до систем командної взаємодії та дослідити функціональні можливості сучасних аналогів;
- Визначити та обґрунтувати вибір гібридної сервісно-орієнтованої архітектури для забезпечення масштабованості та відмовостійкості системи;
- Провести вибір та обґрунтування технологічного стеку, що включає сучасні фреймворки та системи управління базами даних;

- Спроєктувати об’єктно-орієнтовану структуру системи та розробити схему бази даних, що підтримує складні реляції між завданнями, подіями та чатами.
- Розробити та реалізувати алгоритми роботи ключових модулів системи, включаючи систему багатоканальних сповіщень у реальному часі та механізми ведення журналу змін.
- Реалізувати програмний комплекс та провести тестування основних сценаріїв використання, таких як управління спринтами та контекстне обговорення завдань.

Об’єктом дослідження є процеси автоматизації управління завданнями та координації комунікації всередині команд у сфері інформаційних технологій.

Предметом дослідження є методи, архітектурні патерни та вебтехнології для створення інтегрованого середовища командної взаємодії з підтримкою реального часу.

У роботі використано методи системного аналізу для дослідження предметної області, принципи об’єктно-орієнтованого проєктування для побудови структури системи, методи вебпрограмування та моделювання реляційних баз даних.

РОЗДІЛ 1. АНАЛІЗ ПРОБЛЕМАТИКИ, МЕТОДІВ ТА ЗАСОБІВ ВИРІШЕННЯ ЗАДАЧІ

1.1 Актуальність та постановка задачі

Сучасні рішення для командної взаємодії у сфері інформаційних технологій не завжди бувають зручними для організації обговорень, слідкування за змінами в завданнях, та термінового інформування учасників. Більшість подібних сервісів може виконувати добре лише одну певну задачу не концентруючись на інших аспектах управління життєвим циклом задач. Тому користувачі повинні постійно перемикатися між великою кількістю додатків, що призводить до втрати контексту, наприклад, 2 людини можуть одночасно спілкуватись і через пошту, окремий додаток чат, веб-сервіс де знаходиться інформація про проєкт, або задачі. Це неодмінно призводить до необхідності дублювати важливу інформацію зі сподіваннями, що випадкове важливе зауваження не загубиться. За даними досліджень, після перемикання між різними додатками (наприклад, поштою та таск-трекером), розробнику потрібно в середньому близько 23 хвилин, щоб повернутися до стану глибокої концентрації [9].

Це розпорошення блокує можливість прогнозувати, аналізувати поточний стан проєктів, бо якась частина інформації може бути просто відсутня. Системний безлад виникає там де він не очікувався, бо логічні зв’язки між робочими процесами залишаються невидимими для системи. Без явного механізму документування усіх змін, що роблять розробники, або інші члени команди, та потенційного блокування однієї задачі іншою, команда втрачає можливість для своєчасного реагування на ризики, що можуть виникати.

Проблеми з комунікацією всередині проєктів можуть впливати не лише на фінальний результат, але й на моральний стан учасників. Звичайними є випадки коли частина роботи втрачається, через необачність або велике навантаження хтось забуває передати важливу новину, або попередити про

якісь зміни, що може призводить до конфліктів, напруженості всередині колективу, тощо. Такі ситуації можуть призводити до необхідності перепрацювань, необхідність подвійної роботи, та знову необхідності повторної комунікації, додаткових уточнень тощо. Та у висновку залишиться лише людина, що отримала негативний досвід роботи та можливе вигорання. Розрізнена комунікація та непрозорість процесів є одними з головних причин зриву дедлайнів та емоційного вигорання в IT-командах [4]. Таким чином, присутність уніфікованої платформи для роботи може допомогти підвищити ефективність виконання завдань, обговорення важливих питань, та своєчасного реагування на зміни.

Розробка та використання інтегрованої інформаційної системи, що буде виступати як єдиний цифровий простір для всієї проєктної діяльності, є необхідним рішенням цього переліку проблем. Головна ідея подібної платформи полягає в усуненні бар'єрів для комунікації, управлінні для збереження контексту роботи. Подібний підхід дозволить сприймати управління проєктами комплексно, зважаючи на повідомлення, завдання, події не як окремі частинки, що розрізнені між собою, а один великий пласт інформації, що структурований та зручний для роботи. Перехід від простих дошок до повноцінних ALM-систем дозволяє об'єднати планування, розробку та комунікацію в єдиний процес, усуваючи «сліпі зони» між етапами проєкту [14]. Має бути забезпечена така глибина інтеграції, при якій спілкування буде сприйматися не як другорядне, необов'язкове джерело інформації, а як повноцінна органічна частина робочого процесу. Тільки автоматизувавши подібні процеси можливо буде впевнитись в мінімізації інформаційних втрат.

Отже, перехід до інтегрованого середовища дозволить перетворити розрізнені дії окремих учасників на злагоджену систему, де кожний член буде мати повний контекст для прийняття швидких та ефективних рішень.

Тому для управління проєктами повинна бути реалізована система для зручного пошуку необхідних задач, їх зміна, додавання пов’язаних матеріалів, та адресація на певних учасників. Важливою складовою є реляції між завданнями, вони повинні бути простими та відображати реальність залежностей одних завдань до одних. Правильна побудова реляцій в системі є ключем до управління ризиками та уникнення блокувань [4].

Комунікаційна функціональність системи повинна бути розподілена на 2 складових: звичайні чати та контекстні коментарі. Вони мають бути розділені на швидкі повідомлення для прояснення нескладних питань, що потребують оперативності, та коментарі можна буде прив’язати до якогось певного об’єкта як завдання або події та які будуть підтримувати ієрархічну структуру та систему реакцій. Водночас, для миттєвої координації та швидкого прояснення нескладних питань будуть виокремлені повідомлення, що будуть існувати в контексті чату, з демонстрацією усіх членів обговорення, індикацією хто в поточний момент набирає текст, та коли прочитав повідомлення. Такий підхід дозволить розділити спілкування на контекстне де важливість та вміст окремих коментарів має велике значення, та високу пропускну здатність повідомлень в чатах, водночас не втрачаючи їх з поля зору.

Для ефективної нотифікації учасників важливо розробити уніфіковану систему сповіщень, що повинна реагувати на події в усіх модулях та доставляти повідомлення через розподілену сітку каналів, таких як: внутрішньосистемні (на самому вебсайті), push-сповіщення у браузері (якщо користувач не знаходиться на вебсайті) та листи на електронну пошту (для 100 відсоткового надійного оповіщення). Важливою частиною завдання є реалізація асинхронної роботи цих потоків даних, щоб мінімізувати навантаження на основну систему [2].

Таким чином, імплементація цієї системи вимагає не просто автоматизації окремих процесів, а створення архітектурно цілісного комплексу,

де кожний робочий елемент і частина комунікації існують у глибокому, наскрізному контексті. Реалізація цих вимог забезпечить підвищення прозорості, координації та загальної ефективності роботи команди.

1.2 Аналіз аналогів програмного продукту

Сучасний ринок програмного забезпечення для управління проєктами та командної роботи пропонує широкий спектр інструментів, що фокусуються на окремих аспектах даної діяльності. За даними IEEE, часті перемикання між ізольованими інструментами є однією з головних причин зниження продуктивності розробників та втрати робочого контексту [18]. Для глибшого розуміння контексту та більш точного аналізу інтегрованої системи керування завданнями необхідно провести детальний аналіз найбільш розповсюджених аналогів, таких як Jira, Trello, Slack та Microsoft Teams.

Jira (Atlassian): Промисловий стандарт для Agile-управління

Jira є безумовним лідером у сегменті професійного управління життєвим циклом розробки та відстеженням помилок [5]. Його функціонал ґрунтується на концепції гнучких методів планування, таких як Scrum [17], Kanban, та забезпечує максимальну свободу у моделюванні робочих процесів. Тут можливе створення складних Workflow-схем, що дозволяє встановити нескінченну кількість станів та логіку переходів між ними. Також доступна глибока кастомізація сутностей та присутні потужні інструменти аналітики та звітності. Проте, незважаючи на це, Jira є досить складною для невеликих або академічних команд. Його архітектура сфокусована лише на задачах де є коментарі та через це немає чату в реальному часі. Крім того, налаштування й підтримка складних зв’язків між сутностями часто вимагає використання додаткових плагінів, що збільшує витрати ресурсів на управління. Також частина функціоналу обмежена підпискою, що потрібна якщо кількість

користувачів в одній команді більше 10 та потрібно більше місця для файлів. Інтерфейс сервісу виглядає наступним чином (див. рис. 1.1).

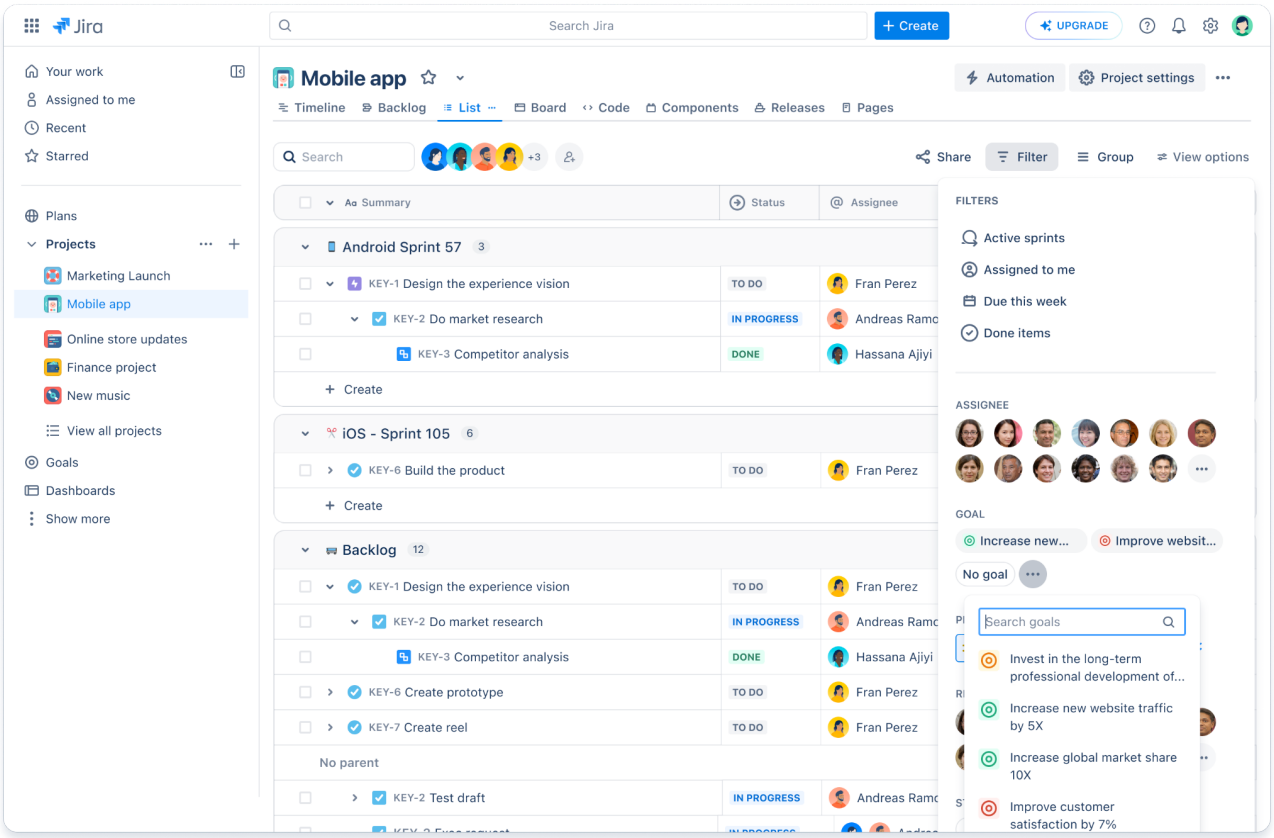


Рис. 1.1 Інтерфейс Jira

Trello: Візуальна простота та Kanban-модель

Trello пропонує альтернативний, візуально орієнтований підхід до управління [19]. Його філософія ґрунтується на мінімалізмі та гнучкості, що ідеально підходить для швидких, неформальних або академічних проєктів. Завдання у Trello представлені у вигляді карток, що розташовані у візуалізації Kanban, тому можна переміщувати їх між списками, що відповідають за різні етапи роботи. Головною перевагою є інтуїтивно зрозумілий інтерфейс та можливість розширення функціоналу використовуючи плагіни. Та в його простоті і є проблема для середніх за обсягом проєктів, що мають певну кількість зв'язків. Наприклад, відсутня нативна підтримка залежностей типу «Задача А блокує Задачу Б», та повинна бути реалізована через сторонні

інтеграції. Комунікація, схоже до Jira, обмежена простими, неієрархічними коментарями, що швидко перетворюються на лінійний потік, де важко відслідковувати гілки обговорень. Інтерфейс сервісу виглядає наступним чином (див. рис. 1.2).

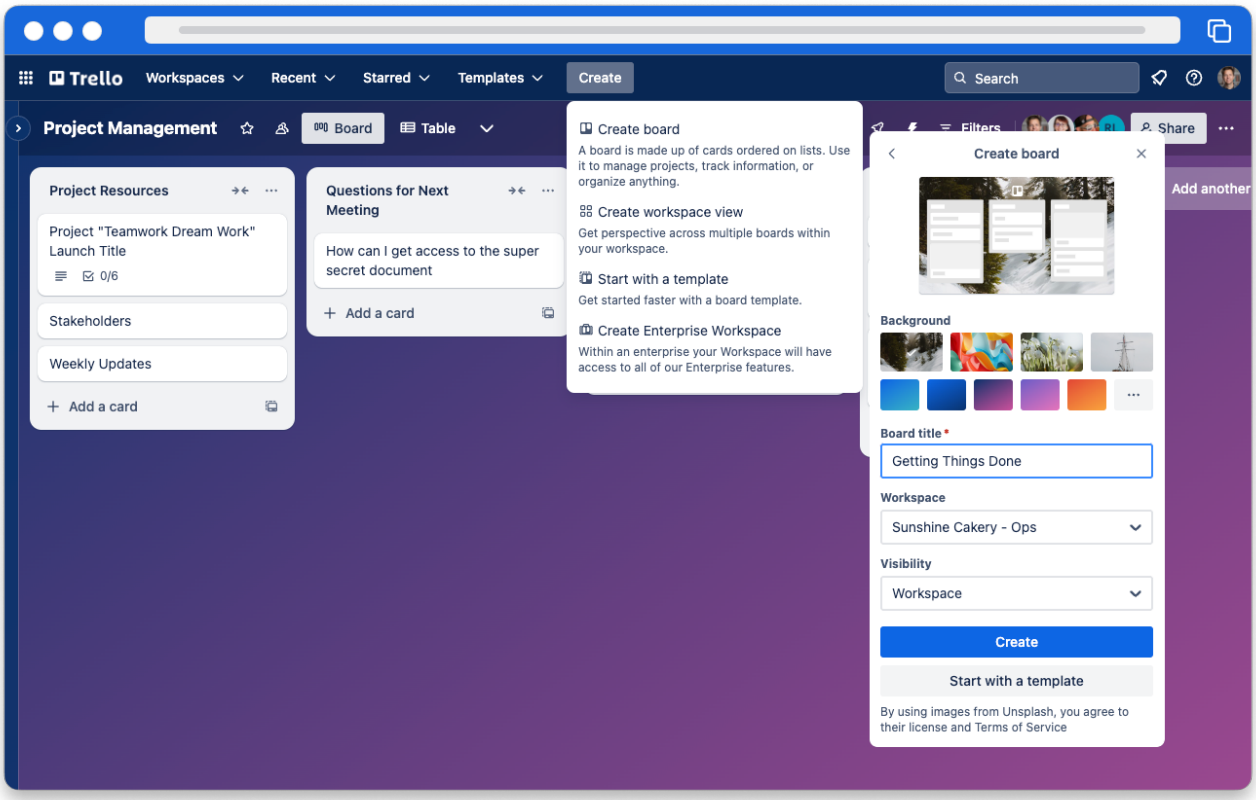


Рис. 1.2 Інтерфейс Trello

Slack: Лідер у корпоративних комунікаціях реального часу

Slack фокусується на швидкій, багаторівневій комунікації позиціонучи себе не як інструмента управління проєктами, а як потужний «хаб» для обміну повідомленнями та інтеграції зовнішніх сервісів. Основний функціонал складається з організації каналів для тематично розподілених обговорень, миттєве спілкування через приватні повідомлення та групові чати, а також підтримку гілок обговорень для структурованих діалогів. Головна перевага Slack - у глибоких інтеграціях з тисячами додатків. Проте основний недолік, що був виявлений у постановці задачі, це втрата контексту. Рішення що були прийняті у Slack там і залишаються, і єдиний варіант їх не втратити та мати

доступ з основної системи де відбувається робота - вручну переносити, бо повідомлення не має жодного стосунку до сутності завдання або івенту, та не видиме в трекері. Повідомлення в чатах є ефемерними та їх цінність швидко втрачається. Хоча Slack і прискорює вирішення дрібних питань, він створює інформаційні «силоси», де критичні архітектурні рішення губляться в потоці неструктурованих повідомлень [20]. Slack фокусується на швидкості комунікації, але поглиблює проблему фрагментації даних. Інтерфейс сервісу виглядає наступним чином (див. рис. 1.3).

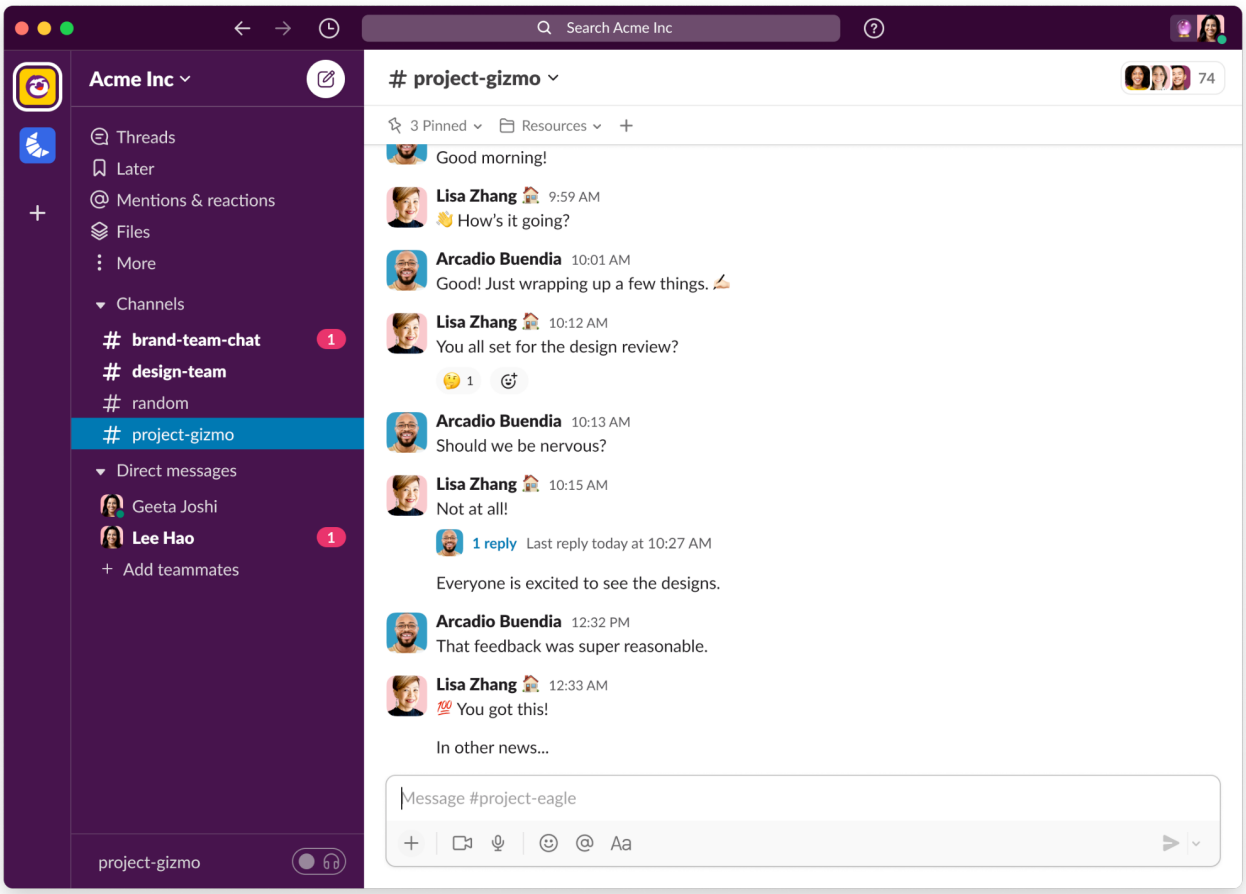


Рис. 1.3 Інтерфейс Slack

Azure DevOps: Комплексна екосистема для управління життєвим циклом розробки.

Azure DevOps від Microsoft є однією з найпотужніших корпоративних платформ, що містить у собі інструменти для управління усім життєвим циклом

розробки програмного забезпечення [3]. Її функціонал включає інструменти для планування та трекінгу завдань, управління репозиторіями коду, а також налаштування безперервної інтеграції та розгортання. Також там присутня підтримка гнучких методологій управління, наприклад Agile, Scrum, Kanban, що забезпечує глибоку кастомізацію робочих процесів.

Проте, незважаючи на подібні потужні інструменти для трекінгу та розробки Azure DevOps має помітні обмеження у сфері командної комунікації. Цей сервіс фокусується переважно на асинхронному спілкуванні через коментарі до задач (Work Items), або під час перевірки коду (Pull Requests). Через те що відсутня нативна підтримка спілкування у реальному часі, команди вимушені використовувати сторонні комунікаційні інструменти, що все одно не мають глибокої інтеграції з основною платформою, наприклад, Microsoft Teams або Slack, для швидкого обміну інформацією.

Необхідність постійного перемикання між середовищем трекінгу завдань та системами комунікації незворотно призводить до втрати контексту. Важливі термінові рішення та обговорення відбуваються в закритих чатах та повинні бути окремо зафіксовані в завданнях, що буде потребувати подвійної роботи з перенесення рішень в Azure DevOps. Це створює інформаційні розриви та ускладнює швидкий пошук історії прийняття рішень та блокує можливість комплексного розуміння стану проєкту. Окрім того, загальна архітектура, інтерфейс та процеси конфігурації Azure DevOps можуть бути занадто перевантаженими та складними для впровадження у невеликих або академічних командах, вимагаючи окремого спеціаліста для адміністрування платформи. Інтерфейс сервісу виглядає наступним чином (див. рис. 1.4).

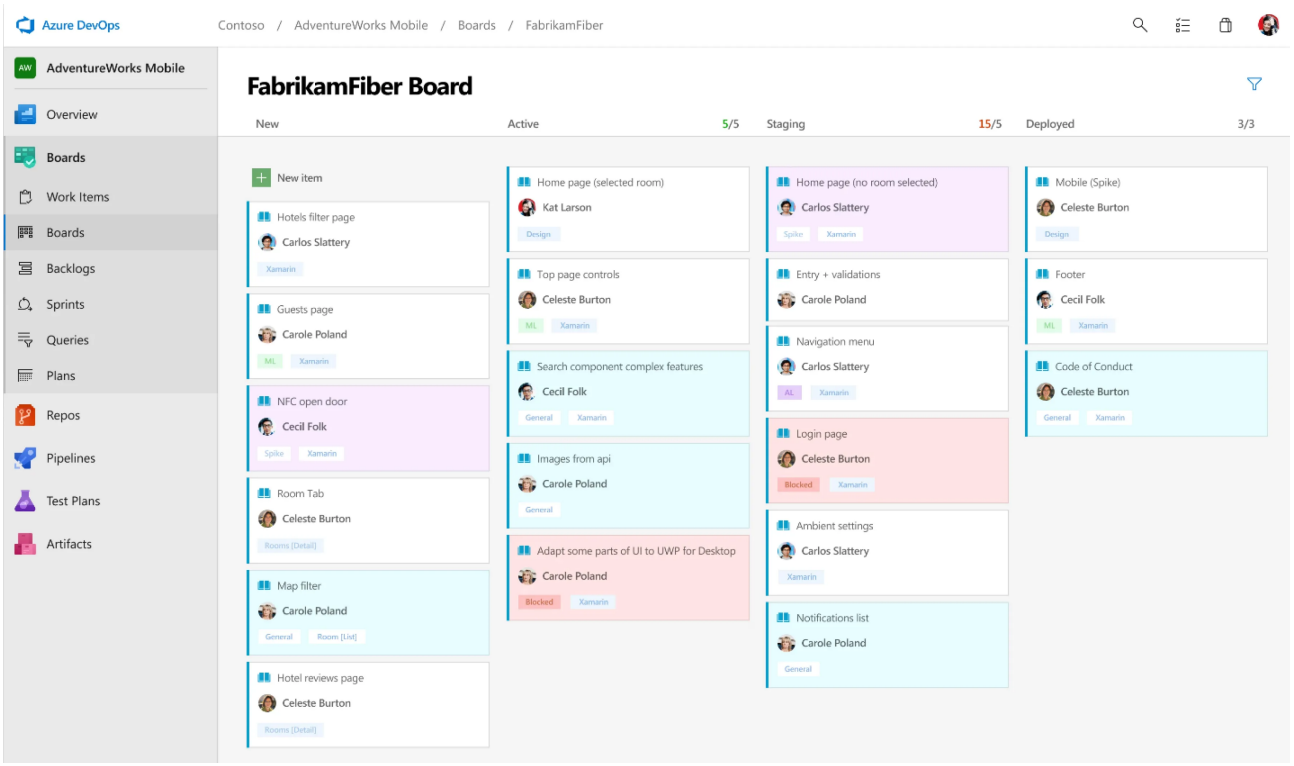


Рис. 1.4 Інтерфейс Azure DevOps

Microsoft Teams: Екосистема для комунікації та співпраці

Microsoft Teams представляє комплексне рішення, яке об'єднує корпоративну комунікацію, відеоконференції та файлосховище [10]. Його ключова перевага полягає у глибокій інтеграції з екосистемою Microsoft 365, що полегшує спільну роботу над документами, але не роботу з Azure DevOps, де зазвичай відбувається трекінг завдань, та основна важлива комунікація, що повинна бути збережена. Основний фокус полягає в каналах та чатах з миттєвими повідомленнями, та зустрічах. Головним недоліком є те, що власні інструменти не підтримують складних робочих процесів, детальної кастомізації зв'язків між завданнями. Таким чином знову виникає проблема для вирішення якої необхідна інтеграція з зовнішніми трекерами, де можлива втрата контексту. Інтерфейс сервісу виглядає наступним чином (див. рис. 1.5).

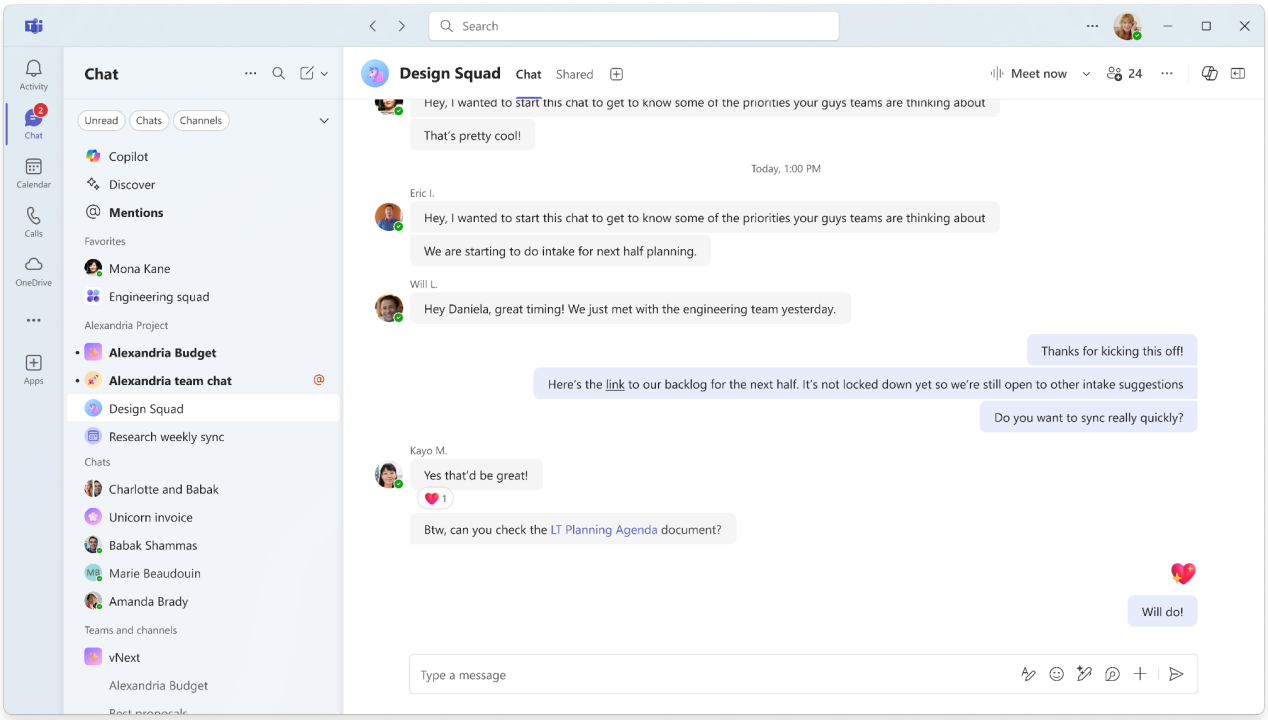


Рис. 1.5 Інтерфейс Teams

Порівняльний аналіз функціональних можливостей

Для наочної демонстрації функціональних прогалин у контексті інтеграції ключових доменів представлена порівняльна таблиця. У ній чітко видно, що на ринку домінують рішення, які є сильними лише в одній або двох сферах.

Порівняльний аналіз аналогів (Jira, Trello, Azure DevOps, Slack, Microsoft Teams) за ключовими функціональними областями подано в таблиці 1.1.

Таблиця 1.1

Порівняльна таблиця аналогів

Характеристика	**Jira**	**Trello**	**Azure DevOps**	**Slack**	**Microsoft Teams**
Основний фокус	Управління проєктами (ALM), Bug Tracking	Візуалізація робочого процесу (Kanban)	Управління життєвим циклом розробки (ALM), CI/CD	Корпоративна комунікація	Комунікація та Екосистема

Продовж. табл. 1.1

Управління завданнями	Високе: складні Workflows та кастомні поля	Середнє: прості картки та списки	Високе: ієрархічні Work Items, глибока кастомізація	Низьке: потребує зовнішніх інтеграцій	Середнє: через спрощений Planner/To Do
Чат у реальному часі	Відсутній: спілкування лише через коментарі	Відсутній: обмежено лінійними коментарями	Відсутній: лише коментарі, потребує інтеграції з Teams/Slack	Високе: спеціалізований сервіс для обміну повідомленнями	Високе: нативні канали та миттєві повідомлення
Збереження контексту	Низьке: інформація розпорошена, немає прямого зв'язку з обговореннями	Низьке: коментарі перетворюються на неструктурований потік	Середнє: є зв'язок завдань з кодом, але комунікація винесена в інші додатки	Критично низьке: рішення в чатах часто втрачаються та фрагментуються	Середнє: дані фрагментовані між чатами та зовнішніми трекерами
Складність налаштування	Висока: потребує спеціальних знань та часу на конфігурацію	Низька: інтуїтивно зрозумілий інтерфейс для швидкого старту	Висока: вимагає часу на конфігурацію процесів та прав доступу	Низька: швидке розгортання та додавання учасників	Середня: вимагає інтеграції з екосистемою Microsoft 365
Управління подіями	Відсутнє	Відсутнє	Відсутнє (лише планування спринтів)	Відсутнє	Середнє: створення постів та інтеграція з календарем

Продовж. табл. 1.1

Підтримка Agile-процесів	Повна: нативна підтримка Scrum та Kanban	Базова: фокус на візуальних Kanban-дошках	Повна: нативна підтримка Scrum, Agile, CMMI	Відсутня: лише через ботів та плагіни	Обмежена: через інтегровані додатки
Журнал змін та історія	Високий: детальний аудит усіх дій у системі	Базовий: спрощена історія активності в картках	Високий: детальний аудит Work Items та репозиторії в	Відсутній: історія обмежується лише текстом повідомлень	Фрагментований: розділений між різними додатками екосистеми
Багатоканальні сповіщення	Складна: часто призводить до надмірної кількості листів	Базова: переважно внутрішньосистемні сповіщення	Середня: переважно email та вебхуки, складні налаштування	Висока: фокус на миттєвих повідомленнях у реальному часі	Висока: інтегрована в загальну систему сповіщень ОС
Взаємозв'язки сутностей	Середні: вимагають складного налаштування Workflow та плагінів	Низькі: відсутня нативна підтримка складних залежностей	Високі: гнучка система зв'язків між завданнями, комітами, гілками	Відсутні: повідомлення не мають системного зв'язку із завданнями	Обмежені: не підтримують глибокої кастомізації зв'язків

Проведений детальний аналіз існуючих на ринку систем управління проєктами та комунікаціями чітко показує їхню сфокусованість на окремих, ізольованих функціональних сферах. Жоден з представлених аналогів не має нативної, доступної «з коробки» взаємодії між завданнями та комунікацією на необхідному рівні.

Таким чином, результати цього аналізу лягли в основу формулювання функціональних вимог і визначення архітектури майбутньої системи, метою якої є вирішення проблеми фрагментації контексту, ідентифікованої у постановці задачі.

1.3 Вибір архітектури

Архітектура системи є одним з найвизначніших етапів життєвого циклу розробки, що формує фундамент для майбутньої продуктивності, гнучкості додатку. На поточному ринку сучасних розподілених систем, що обробляють величезні обсяги даних та інтенсивність комунікацій зростає експоненційно, архітектура повинна не лише підтримувати основні бізнес-процеси, а й мати потенціал для горизонтального масштабування без деградації продуктивності. Аналіз різних архітектур має охопити широкий спектр патернів, від класичних найпростіших монолітів до сучасних комплексних структур, з метою пошуку оптимального балансу між складністю розробки та операційною стабільністю. Це є критично важливим для інтеграції різнорідних доменів, наприклад, управлінням завданнями та асинхронна комунікація в реальному часі.

Найпростішою з архітектур є монолітна (див. рис. 1.6). Це традиційна модель для побудови систем, що забезпечує інтеграцію функціональних модулів у виконуваний контекст із загальною базою даних [16]. Хоч такий підхід й значно спрощує процеси початкового розгортання та розробки, він створює серйозні ризики у випадку високого навантаження на систему. При виході одного з модулів, наприклад, модуля обробки нотифікацій, це може призвести до виходу з ладу усієї платформи. Окрім того, така архітектура не дозволяє гнучкого масштабування системи. При необхідності збільшити кількість ресурсів для одного модуля доведеться надавати ресурси усьому моноліту, що не є ефективною економічною моделлю [1]. Також у майбутньому

складна взаємозалежність компонентів буде уповільнювати процес впровадження нових функцій через обов'язкове перетестування всієї системи.

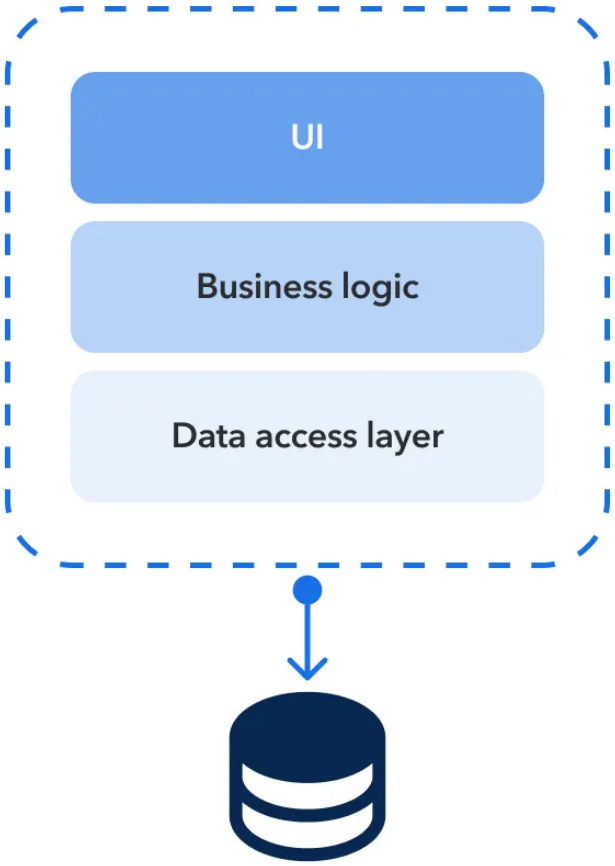


Рис. 1.6 Приклад монолітної архітектури

З іншого боку є мікросервісна архітектура, вона потребує повної декомпозиції на автономні сервіси (див. рис. 1.7). Це забезпечує найвищий рівень відмовостійкості та технологічної незалежності. В такій архітектурі кожен сервіс має власну базу даних та окремий стек технологій. Проте подібний підхід потребує великої кількості ресурсів для розробки та підтримки через велику складність у підтримці цілісності даних та управлінні мережевою інфраструктурою. Витрати ресурсів на організацію міжсервісної взаємодії, моніторинг та оркестрацію можуть перевищити всі переваги від гнучкості. Крім того, мережеві затримки при великій кількості внутрішніх викликів можуть негативно вплинути на користувацький досвід у динамічних інтерфейсах [6].

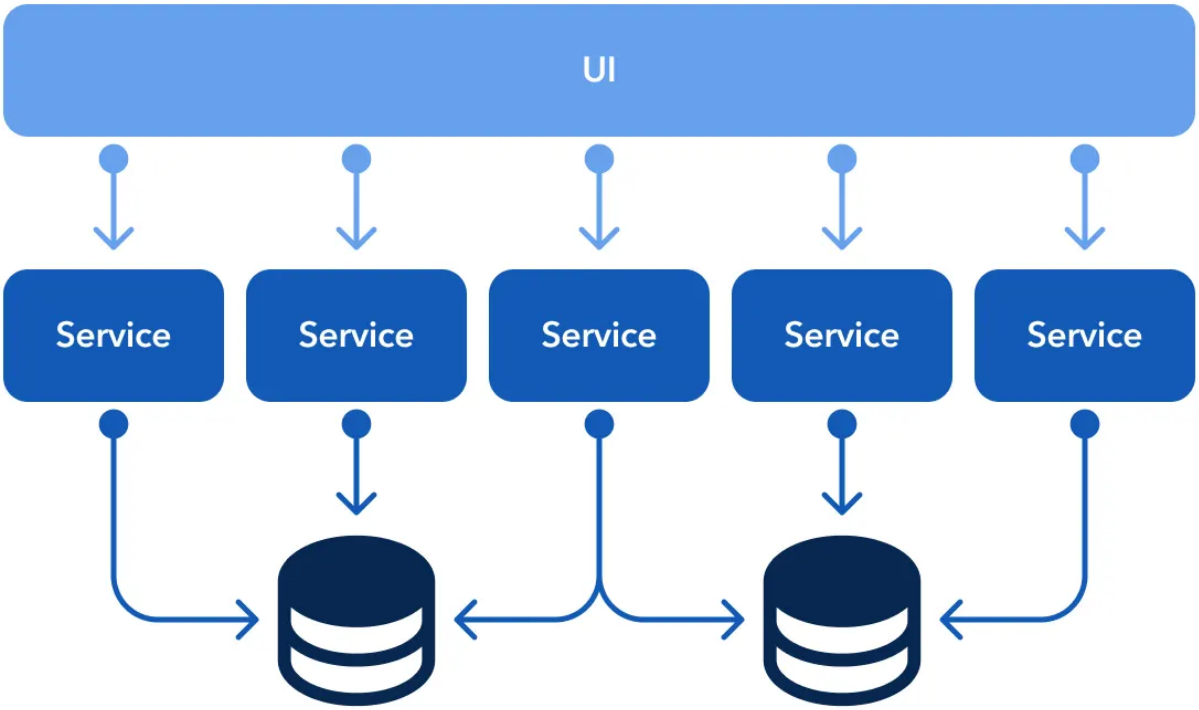


Рис. 1.7 Мікросервісна архітектура

Тому для розробки системи було обрано гібридну сервісно-орієнтовану модель, яка дозволяє використовувати спеціалізовані середовища виконання для різних функціональних завдань. Таке рішення ґрунтується на необхідності поділу системи на два ключових домени: стабільне середовище для бізнес-логіки та реактивне середовище для взаємодії з системою в реальному часі [15]. Таке розділення відповідальності гарантує, що транзакційні операції з даними не будуть заблоковані інтенсивним мережевим трафіком від модулів спілкування.

Основним сервісом є домен бізнес-логіки, він побудований з використанням суворої типізації та розвиненими механізмами валідації. Він є єдиним джерелом істини для усіх сутностей системи: завдань, проєктів, подій, користувачів. Використання надійних систем управління даними дозволить

забезпечити цілісність зв’язків та автоматизацію типових операцій через стандартизовані інтерфейси взаємодії. Це дозволить сконцентруватися на бізнес-логіці, не витрачаючи зайвий час на низькорівневу реалізацію отримання даних.

Наступним є сервіс комунікацій реального часу, в якому для забезпечення миттєвої реакції на дії користувача використовується асинхронне середовище, оптимізоване для обробки великої кількості одночасних з’єднань. Цей сервіс відповідає винятково за чати, статуси присутності та передачу сигналів про нові події. Завдяки неблокуючій моделі вводу-виводу, система здатна підтримувати високу динаміку обговорень без навантаження на основний сервер бази даних. Це дозволить ізолювати частину системи, змінюється найчастіше, зробивши загальну архітектуру більш стійкою до пікових сплесків активності.

Наступним є інтеграційний шар сповіщень, що дозволить взаємодію між двома доменами для передачі повідомлень, тобто виступить сполучною ланкою. Він забезпечить трансляцію подій із комунікаційного домену, наприклад, згадок або публікації повідомлень, в уніфіковану чергу обробки основного домену. Це дозволить реалізувати багатоканальну доставку інформації через різні інтерфейси та зовнішні сервіси інформування. Такий підхід усуне розрив між спілкуванням та виконанням завдань, створивши цілісний контекст роботи для кожного учасника команди.

Стратегічною перевагою обраного підходу є те, що подібна модель забезпечує унікальну можливість незалежного масштабування кожного сегмента системи залежно від характеру навантаження. Це дозволить оптимізувати використання серверних ресурсів та гнучко адаптувати архітектуру до змінних потреб проєкту. Система зможе продемонструвати високу відмовостійкість та надійність завдяки чіткому розподілу відповідальності та використанню асинхронних паттернів. Це створить міцну

основу для додавання нової функціональності, зберігаючи при цьому чистоту коду та зручність його підтримки.

1.4 Обґрунтування вибору інструментальних засобів та вимоги до програмного забезпечення

Вибір технологічного стеку для розробки інтегрованої системи базується на необхідності поєднання високої продуктивності, надійності та здатності обробляти великі масиви даних у реальному часі. Далі наведено розширене обґрунтування вибору кожного компонента системи:

Для фронтенд частини було обрано React, бо це одна з найпопулярніших бібліотек для розробки користувацького інтерфейсу у світі, тому має велику підтримку зі сторони спільноти. Також одним з головних критеріїв є швидкодія та простота, так як у React використовується компонентна архітектура це забезпечує високу швидкість рендерингу складних інтерфейсів завдяки механізму Virtual DOM [8]. Це критично важливо в системі де елементи повинні часто оновлюватись без перезавантаження сторінки. TypeScript додає статичну типізацію, що дозволяє виявляти помилки на етапі написання коду, що значно спрощує підтримку великої кодової бази при командній розробці [13]. Redux Toolkit обрано як стандартну бібліотеку для керування складними станами додатку, за допомогою цієї бібліотеки можливо зручно оновлювати дані та кешувати їх, що дозволяє знизити навантаження на сервер. Для забезпечення професійного візуального вигляду та швидкості розробки було використано Material UI, тому що в цій бібліотеці зібрані майже усі компоненти, що можуть бути необхідні під час розробки, та що відповідають гайдлайнам дизайну від Google. Це дозволяє сконцентруватись на бізнес-логіці, а не на базовій стилізації.

Для модуля основної бізнес-логіки було обрано Symfony через його зрілість, високий рівень безпеки та модульну архітектуру, що дозволяє підключати лише необхідні компоненти. Для автоматичної генерації RESTful

API обрано API Platform, що добре інтегрується із Symfony та відповідає сучасним стандартам. Для зручного доступу до БД використовується Doctrine ORM за допомогою паттерну Data Mapper [11]. Це дозволяє працювати з сутностями як з чистими об'єктами (POPO), що підвищує чистоту архітектури та полегшує складність розробки. Для виконання важких операцій, таких як розсилка email-сповіщень або обробка завантажених файлів використовується Symfony Messenger. Він додає шину повідомлень та асинхронну обробку завдань за допомогою черг, що гарантує швидкий час відгуку системи для кінцевого користувача [12].

Як основну СУБД було обрано MySQL через її репутацію як однієї з найсучасніших реляційних баз даних з відкритим кодом. Ця СУБД повністю відповідає вимогам ACID, які є важливі для структурних даних завдань та загальної роботи системи. Також MySQL підтримує роботу з чергами та повнотекстового пошуку, що в майбутньому може прибрати необхідність впровадження додаткових інструментів, наприклад, ElasticSearch.

Для модуля чату було обрано NestJS, як прогресивний Node.js фреймворк. Він використовує асинхронну неблокуючу модель подій, що ідеально підходить для високонавантажених WebSocket-з'єднань [21]. NestJS використовує TypeScript, ту саму мову що й на фронтенді, та має багато схожостей з Symfony, що спрощує розробку. Для зручної роботи з WebSockets використовується бібліотеку Socket.IO, що виступає як абстрактний шар для автоматичного перепідключення, підтримки «кімнат» для групових чатів та роботу через HTTP long-polling у разі обмежень мережі [22].

Для розгортання та інфраструктури використовується Docker, що дозволяє контейнеризувати кожен сервіс системи, гарантуючи ідентичність середовища розробки, тестування та продуктової експлуатації. Ця технологія дозволяє усунути проблему «працює на моєму комп'ютері» та дозволяє просте горизонтальне масштабування окремих частин системи [23]. Nginx обрано як

високоефективний вебсервер та реверс-проксі. Він відповідає з термінацію SSL-трафіку, load balancing між контейнерами та ефективне повернення статичних файлів фронтенду. Ідеальним вхідним вузлом для реалізуємої системи його робить те що він здатен обробляти тисячі одночасних з’єднань із низьким споживанням пам’яті [24]. Та phpMyAdmin для зручного доступу до баз даних всередині проєкту.

Для забезпечення високої продуктивності, відмовостійкості та горизонтального масштабування гібридної архітектури необхідно використовувати потужне апаратне забезпечення. Зокрема, сервери застосунків, які обслуговують основний бекенд на Symfony та модуль чату на NestJS, повинні бути оснащені швидкими NVMe-накопичувачами, процесорами з високою тактовою частотою (мінімум 4, а краще від 8 ядер) та щонайменше 16 ГБ оперативної пам'яті для ефективної обробки паралельних запитів, роботи PHP-FPM і підтримки численних WebSocket-з'єднань.

Сервер бази даних MySQL потребує значно більше ресурсів: для швидкого кешування та обробки складних запитів необхідні процесори мінімум на 8 ядер, від 32 до 64 ГБ оперативної пам'яті та надшвидкі NVMe-диски або RAID-масиви з високим показником IOPS для швидкої роботи з трансакційними даними.

Щоб об'єднати ці компоненти в контейнеризованому середовищі Docker, необхідна надійна мережева інфраструктура з пропускною здатністю від 1 до 10 Гбіт/с, що гарантуватиме мінімальні затримки під час обміну даними між мікросервісами. З боку кінцевого користувача вимоги залишаються базовими — для повноцінної роботи системи достатньо будь-якого сучасного веббраузера з підтримкою стандартів HTML5, CSS3 та протоколу WebSocket.

Висновки до першого розділу

У першому розділі було проведено детальний аналіз проблем командної взаємодії в ІТ-проєктах, який виявив критичну необхідність подолання

фрагментації робочого контексту. Дослідження існуючих аналогів, таких як Jira, Trello, Azure DevOps, Slack та Microsoft Teams, підтвердило, що сучасні рішення часто фокусуються лише на одному аспекті — або на управлінні завданнями, або на комунікації — що змушує користувачів постійно перемикатися між додатками та призводить до втрати важливої інформації. Отже, було обґрунтовано доцільність розробки інтегрованої вебсистеми, яка об'єднує ці домени в єдиному цифровому просторі для підвищення прозорості та ефективності роботи команди.

Для забезпечення високої відмовостійкості та продуктивності системи було обрано гібридну сервісно-орієнтовану архітектуру. Такий підхід дозволяє розділити стабільне середовище бізнес-логіки та реактивне середовище для взаємодії в реальному часі, запобігаючи блокуванню трансакційних операцій інтенсивним мережевим трафіком від чатів. Обрана модель дозволяє незалежно масштабувати окремі сегменти системи залежно від навантаження, що створює гнучкий та надійний фундамент для подальшої розробки.

Технологічний стек системи був сформований з урахуванням вимог до швидкодії та типізації даних: бібліотека React із Redux Toolkit для фронтенду, фреймворк Symfony для основної бізнес-логіки та NestJS для модуля чатів у реальному часі. Використання Docker для контейнеризації та Nginx як реверс-проксі гарантує ідентичність середовищ розробки та експлуатації, а також спрощує горизонтальне масштабування. Визначені апаратні вимоги до серверної інфраструктури забезпечують стабільну обробку великої кількості одночасних WebSocket-з'єднань та швидку роботу бази даних MySQL.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБ-СИСТЕМИ УПРАВЛІННЯ ЗАВДАННЯМИ ТА КОМАНДНОЇ ВЗАЄМОДІЇ В ІТ-ПРОЄКТАХ

2.1 Визначення варіантів використання та об’єктно-орієнтованої структури системи

Основна мета проєкту полягає у спрощенні управління проєктами та поліпшенні комунікації в них. З огляду на це, базовим функціоналом системи є створення задач, подій, обговорень для ідей, налаштування сповіщень користувачів та забезпечення можливості спілкування через чати. Робота безпосередньо із завданнями охоплює їх створення, зручне редагування та видалення. Важливою вимогою до системи є забезпечення ефективного пошуку за допомогою інструментів фільтрації, сортування та пагінації. Для глибокої класифікації та ефективного управління завданнями передбачена комплексна система полів, яка дозволяє структурувати всю необхідну інформацію.

Кожна задача ідентифікується за допомогою унікального номера (ідентифікатора). Він має бути простим, переважно інкрементальним, що дозволяє користувачам легко знаходити та передавати завдання без необхідності запам'ятовувати чи вказувати повну назву - для розуміння контексту достатньо назвати кілька цифр. Головним інформаційним полем виступає назва, яка стисло сумаризує суть проблеми, що підлягає вирішенню. Для надання ширшого контексту використовується поле опису, яке може оновлюватися з плином часу для підтримання актуальності інформації. У цьому полі реалізована можливість вибору стилю відображення тексту: від звичайного формату до розширеного з використанням базового синтаксису Markdown.

Життєвий цикл задачі відстежується за допомогою статусу, який відображає поточний етап роботи - від потреби в додатковому аналізі до готовності до розробки або етапу тестування. На пріоритетність та терміновість виконання впливає відповідне поле пріоритету: чим він вищий, тим більше уваги команда має приділити цьому завданню. Для зручного розподілу робіт за

обсягом та специфікою використовується тип завдання, що дозволяє диференціювати їх на звичайні задачі, баги, епіки тощо. Додатковим критерієм класифікації за довільними ознаками слугують теги. Окрім цього, для кожного завдання фіксується термін виконання, який вказує на орієнтовну дату завершення запланованих робіт.

Особлива увага приділяється розподілу відповідальності, що реалізується через поле виконавців. Це дозволяє делегувати задачу конкретному учаснику, після чого вона з'являється в його персональному списку завдань. Також передбачена можливість призначати задачі на цілі ролі, закріплені за ідеєю. Це значно підвищує оперативність процесів: будь-який вільний фахівець із відповідною роллю може швидко взяти термінове завдання в роботу.

Система також передбачає тісний взаємозв'язок між різними сутностями. Обов'язковим атрибутом під час створення задачі є прив'язка до конкретної ідеї, яка заповнюється автоматично (при створенні з екрана самої ідеї) або вказується вручну (з розділу персональних задач). Для розширення контексту реалізована можливість пов'язувати завдання з певними подіями (івентами), що стосуються відповідної ідеї. Крім того, задля збереження повноти інформації та історії рішень, до задач можна прикріплювати чати для обговорень або посилання на конкретні повідомлення.

Після створення завдання повинна бути можливість додавання вкладень, що будуть асоціюватись з задачею. Також у цих вкладеннях повинна бути можливість перегляду усіх вкладень, що доступні для задачі, тобто можливий перегляд й файлів, що прикріплені у коментарях.

Важливим для завдань також є можливість «м’якого» видалення. У випадку коли користувач випадково видалив завдання у котрому було пророблено багато роботи, та воно має цінність, адміністратор сайту повинен мати можливість відновити задачу зі збереженням усіх зв’язків та контенту задачі.

Не менш потрібною є функція перегляду історії змін задачі, тобто якщо якийсь користувач змінить опис, або видалить файл запис про це повинен бути доступним для перегляду усім.

Для важливих завдань повинна бути можливість «підписки» на зміни. Коли задача буде змінена користувачем усі юзери, що підписані на задачу отримають повідомлення у ті канали, що вони для цього вибрали, наприклад, пошту, в браузері, чи просто на сайті. Та при натиску на ці повідомлення повинна відбуватися переадресація на сайт та на цільову задачу.

Зручність налаштування простору дуже залежить від гнучкості налаштування параметрів завдань на стороні користувача, таких як статус, пріоритет, теги. У них повинна бути присутня «вага» для відповідного розміщення їх у елементі вибору потрібного значення. Для зручності для цих налаштувань повинні бути встановлені значення за замовчуванням, але за необхідності користувач повинен мати можливість перейти на варіант з використанням унікальних налаштувань, тобто бути у змозі змінювати, додавати, видаляти та змінювати вагу дефолтних налаштувань. Також необхідний функціонал за якого можна буде повернутись до дефолтних налаштувань не втрачаючи їх взагалі, за допомогою меню переведення одних у інші.

Наступною важливою функціональністю системи є управління подіями (івентами). Їхня головна мета полягає в оперативному інформуванні залучених учасників про важливі зміни або заходи в межах певної ідеї. Під час створення події передбачено спеціальний перемикач, який дозволяє обрати: надіслати сповіщення всім учасникам чи просто мовчки зафіксувати подію в системі.

Кожна подія формується на основі низки інформаційних полів. Базовим орієнтиром є назва, яка стисло підсумовує суть заходу. Більш розгорнута інформація зберігається в описі, де зосереджено основний контент. Для гнучкості оформлення тексту передбачена можливість вибору стилю

відображення: від звичайного формату до розширеного форматування за допомогою синтаксису Markdown. Візуальну складову доповнюють фотографії. Окрім можливості вбудовувати зображення безпосередньо в текст опису через Markdown, реалізована функція окремого прикріплення світлин до події для їх зручного перегляду.

Для ефективної класифікації та точного таргетування використовуються категорії. Якщо користувач має хоча б одну категорію з цього переліку у списку, що йому цікаві, та увімкнені сповіщення, коли ця подія буде створена він повинен отримати про це нотифікацію. У контексті роботи над конкретною ідеєю до події можна додавати пов'язані завдання, які стосуються саме цієї ідеї (однак, якщо івент створюється для сайту загалом, прив'язка задач недоступна).

Наступною важливою складовою є коментарі. Вони повинні бути універсальною сутністю та працювати у великій кількості випадків, наприклад, у завданнях, подіях, ідеях тощо. Головна задача коментарів це бути вмістом інформації, можливо великого об’єму, що не загубиться та буде доступним завжди, та підтримує складні обговорення. Вони повинні підтримувати ієрархію, що являє собою гілки спілкування, різні реакції, зміну та видалення. При відображенні коментаря важливо показувати повний час коли він був створений, та при наявності змін час оновлення, з відображенням прапора, що оригінальний коментар був змінений.

Також важливо зауважити, що ці коментарі повинні оновлюватись в режимі реального часу, тобто якщо два користувачі відкриють одну й ту саму сторінку де будуть присутні коментарі та один з них додасть коментар, його змінить, додасть реакцію тощо, інший миттєво це побачить [30].

Чати є дуже важливою складовою комунікації між користувачами. Вони можуть бути між двома, та більше юзерами, загальними, не прив’язаними до ідеї, або таким, що мають пряму залежність до ідеї. На загальній сторінці спілкування повинна бути кнопка для початку розмови, після котрої можна

буде назвати бесіду та обрати учасників, що потрібно додати до чату. Ці користувачі автоматично отримають повідомлення про створення, або додавання їх до групи та зможуть перейти на сторінку чату.

Система повинна мати окрему сторінку, де зібрані всі доступні чати, з можливістю їх фільтрації за конкретними ідеями. У загальному списку для кожного чату необхідно відображати його назву, текст і час надсилання останнього повідомлення, аватарки поточних учасників, а також назву пов'язаної ідеї, якщо чат до неї відноситься.

Інтерфейс відкритого чату має бути виконаний у типовому стилі та забезпечувати необхідний набір інструментів для комунікації. Користувачі повинні мати змогу надсилати звичайні повідомлення, відповідати на конкретні репліки, а також редагувати чи видаляти їх. Для розширення можливостей спілкування передбачено прикріплення до повідомлень різноманітних файлів та медіа з функцією їх зручного перегляду. Окремо реалізовано можливість копіювання прямого посилання на будь-яке конкретне повідомлення в межах бесіди.

Управління складом чату включає можливості додавання нових людей та перегляду поточного списку учасників, при цьому функція видалення користувачів доступна виключно творцю цього чату. Для забезпечення комфортного інформаційного фону передбачено налаштування ввімкнення або вимкнення сповіщень для кожної окремої бесіди. Інтерактивність та зручність процесу спілкування підтримується завдяки індикаторам перегляду надісланих повідомлень, а також відображенню статусу написання тексту іншими співрозмовниками в реальному часі.

Чат є способом швидкого обміну інформації, але водночас може використовуватися для зберігання важливої інформації завдяки можливості збереження посилань на конкретні повідомлення та додавання зв’язків між

задачами та подіями та чатами. Це буде забезпечувати наскрізний контекст протягом усієї роботи.

Окремо потрібно зазначити про повідомлення, вони повинні бути 4 типів: оновлення задач, згадки в коментарях, повідомлення та івенти. Меню з цими повідомленнями повинно бути доступне з будь-якого місця на сайті, з можливістю фільтрації за типом. Також для зручності при натисканні на повідомлення повинна відбуватися навігація на об’єкт повідомлення.

Тобто головними реляціями в системі будуть ідея як основна сутність, навколо якої побудовані усі процеси, та завдання котрі можуть бути пов’язані лише з однією ідеєю, як і події та чати. А завдання, події, чати пов’язані між собою зв’язком багато до багатьох.

2.2 Розробка бази даних системи

Сутність завдання виступає центральним хабом для всієї проєктної роботи. Її зв'язки визначають не лише саму суть роботи, яку необхідно виконати, але й відповідальних осіб, місце завдання на загальному таймлайні та його інтеграцію з іншими інформаційними блоками.

Для зручності передачі інформації між користувачами використовується простий інкрементальний id, що замінює громіздкі унікальні коди або повні назви. Кожне завдання обов'язково має бути прив'язане до певної батьківської ідеї через поле idea. У системі також реалізовано спеціальний двошаровий підхід для чіткого розділення стандартних налаштувань і кастомних (специфічних для конкретної ідеї). Через цю архітектурну особливість передбачено дублювання параметрів: за статус відповідають taskStatus та ideaTaskStatus, за пріоритет - taskPriority та ideaTaskPriority, за тип завдання - taskType та ideaTaskType, а за теги - відповідно tasksTags та ideaTasksTags.

Розподіл відповідальності здійснюється завдяки можливості призначити конкретного виконавця у полі assignee або закріпити завдання за певними

ролями через assignedRoles. Окремо фіксується користувач, який є автором задачі, що зберігається у полі creator. Суть проблеми, яку має вирішити завдання, стисло сумаризується в його title, тоді як увесь необхідний контекст детально розкривається в полі description. Формат тексту опису визначається полем descriptionFormat, який за замовчуванням є звичайним, проте для розширеного форматування підтримується використання Markdown.

Для управління робочим часом у задачі передбачено встановлення дедлайну через due_date, а після завершення роботи система фіксує точну дату виконання у полі completedAt. Завдання можна додати до конкретного спринту (sprint), який визначає тимчасову мету команди; якщо такої прив'язки немає, задача автоматично залишається в беклозі. Для оцінки складності та загального обсягу робіт задачі присвоюється певна кількість storyPoints.

Щоб зберігати повноту контексту, до задачі можна неформально прив'язувати ідентифікатори відповідних чатів через chatIds та прикріплювати різноманітні файли у полі files. Для предметних обговорень безпосередньо всередині завдання функціонує окрема гілка comments. Зацікавлені користувачі мають змогу підписатися на задачу через поле followers, щоб своєчасно отримувати сповіщення про будь-які оновлення. Також для побудови наскрізного контексту до завдання можуть бути додані пов'язані події через поле events.

Крім того, система підтримує розгалужену структуру залежностей. Поле sourceRelations зберігає список задач, які безпосередньо заблоковані поточною, тоді як targetRelations містить усі інші задачі з іншими типами реляцій. Увесь життєвий цикл сутності ретельно фіксується: система автоматично зберігає час створення (createdAt) та кожного оновлення (updatedAt). Для захисту інформації використовується механізм м'якого видалення, який просто записує час видалення задачі у поле deletedAt, залишаючи можливість її повноцінного відновлення за потреби.

Приблизна схема зв’язків сутності Task виглядає наступним чином (див. рис. 2.1).

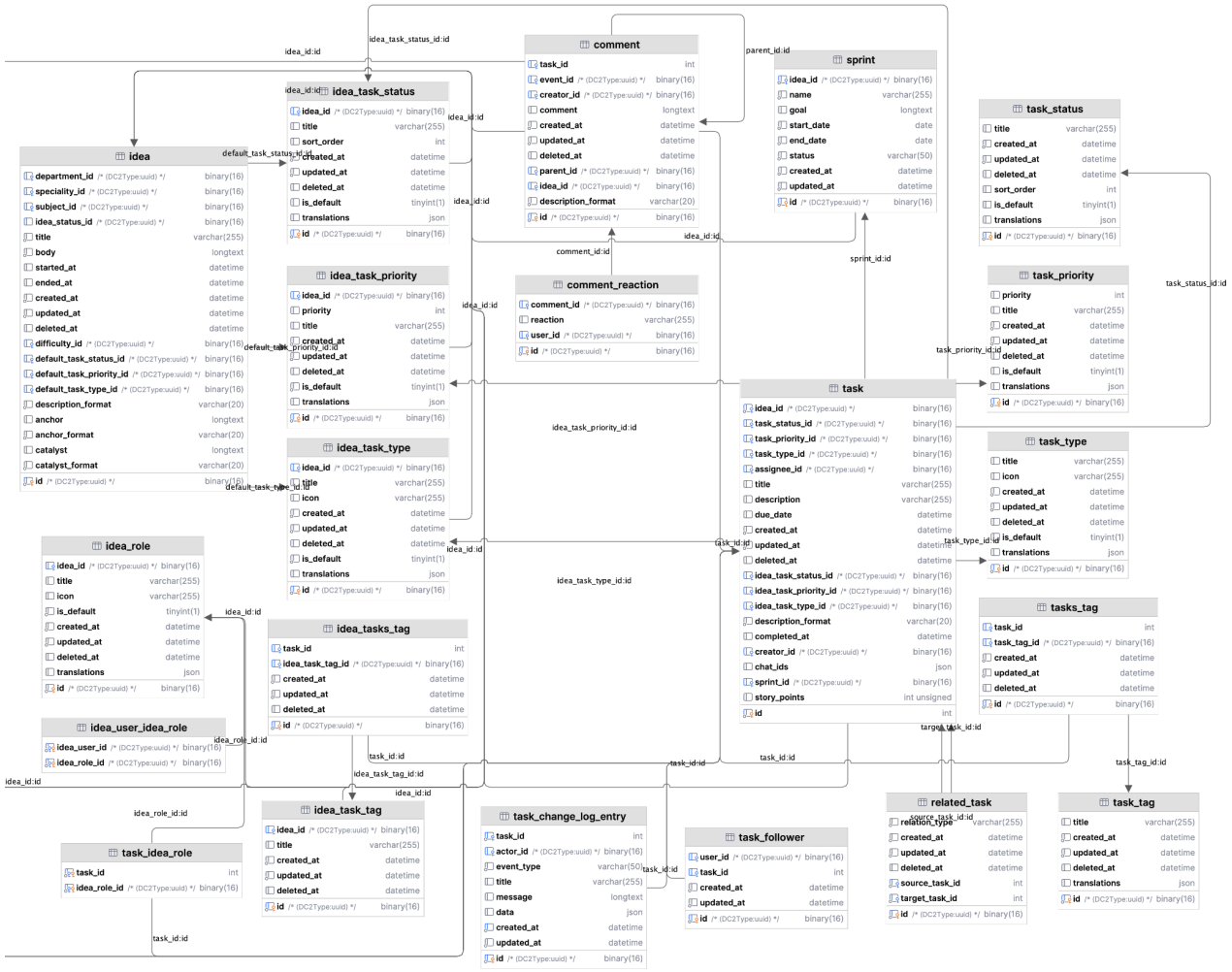


Рис. 2.1 Основні реляції сутності Task у базі даних

Розглянемо простішу, але не менш важливу схему бази даних сервісу чатів, що відповідає за надсилання повідомлень у режимі реального часу (див. рис. 2.2).

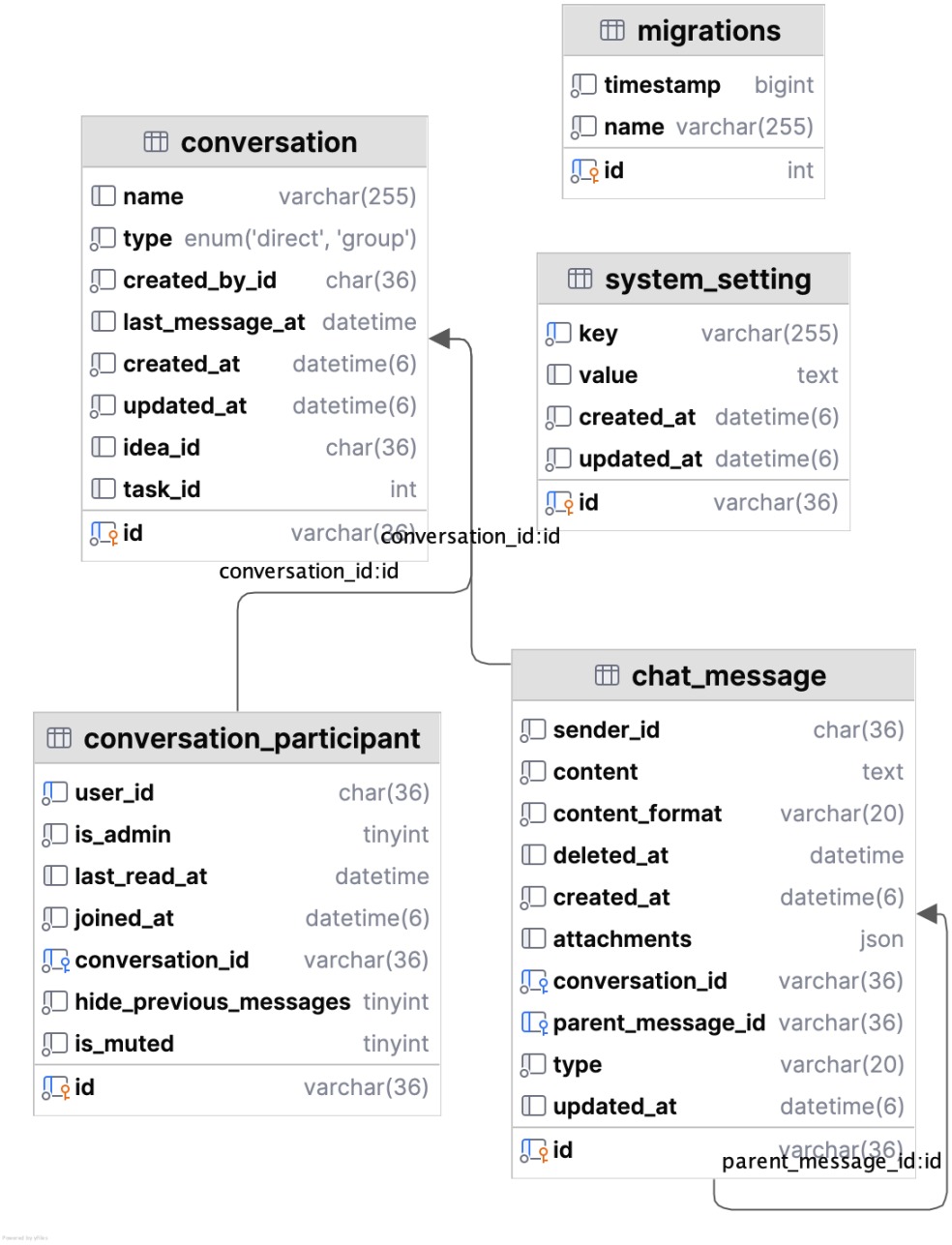


Рис. 2.2 Схема бази даних чат сервісу

Conversation — це сутність-контейнер для усіх чатів, це приватна група, з можливим посиланням на задачу, або ідею. Тому що чат може мати чітку тему, та зручної навігації та пошуку потрібних чатів ці дані зберігаються в даній сутності.

Єдині хто можуть переглядати повідомлення та їх надсилати це користувачі у сутності ConversationParticipant. В ній збережена інформація

щодо того коли приєднався користувач, та чи є у нього право переглядати повідомлення до його приєднання. Також тут вказано чи потрібно нотифікувати користувача при надходженні нових повідомлень в чат.

Сутність ChatMessage відповідає саме за повідомлення, що може належати лише одному чату. Також тут можливі вкладення файлів та медіафайлів, для їх швидкого обміну. Реалізована підтримка відповідей на конкретні повідомлення для утримання контексту розмови.

2.3 Проєктування та реалізація алгоритмів роботи системи

Загальна схема сервісів системи виглядає наступним чином (див. рис. 2.3).

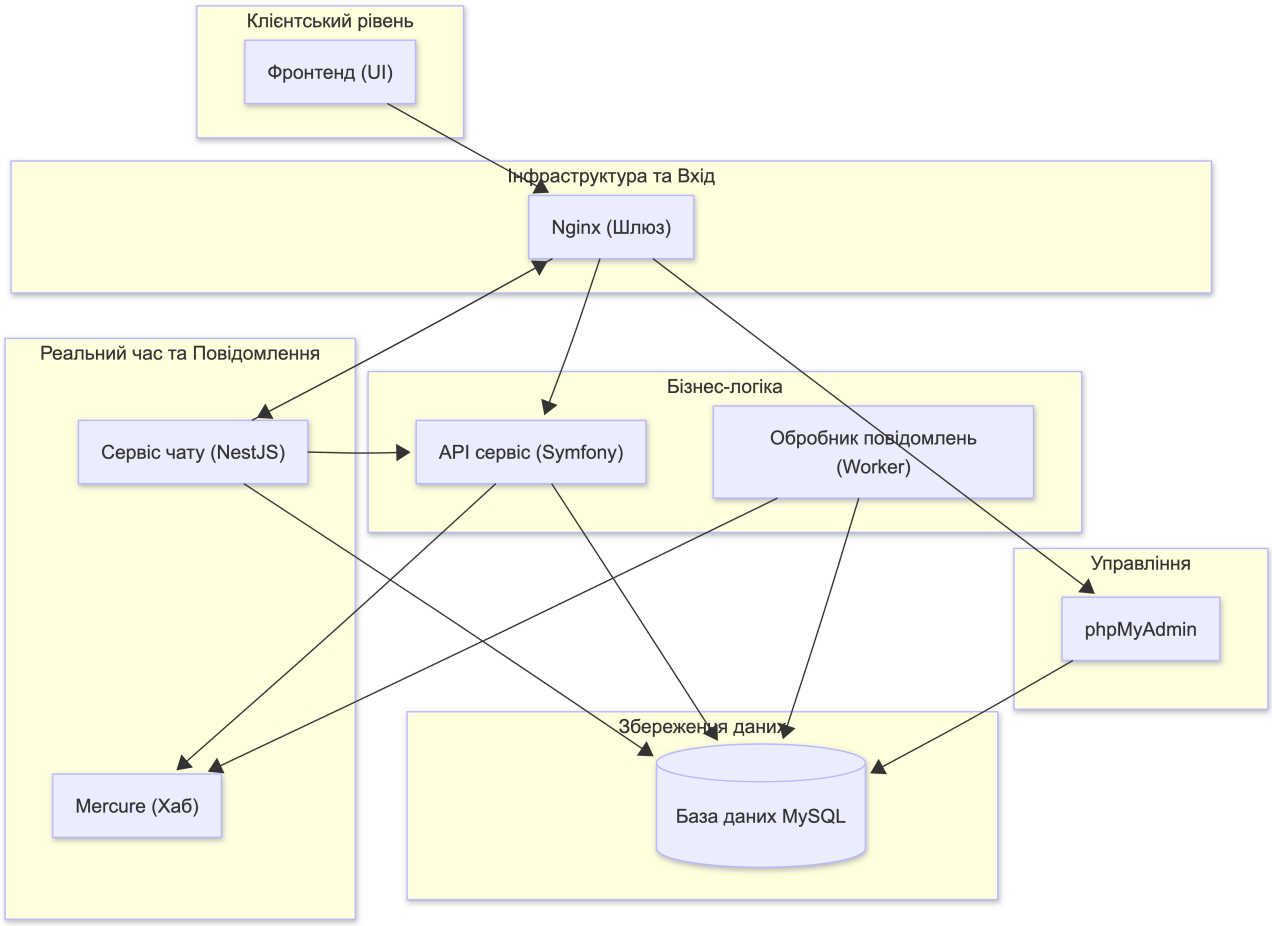


Рис. 2.3 Компонентна діаграма сервісів проєкту

Роль компоненту Фронтенд (UI) у клієнтському шарі полягає у відображенні інтерфейсу користувача, як single-page application (SPA), що означає відсутність необхідності перезавантаження сторінки для оновлення даних на ній. Також цей компонент підтримує постійний зв’язок з Mercure Hub для отримання останніх повідомлень, та іноді підключається до Сервісу чату (NestJS) для повноцінної роботи чатів.

Далі ми можемо побачити Nginx (Шлюз) — єдина точка входу для взаємодії з серверною частиною системи. Вона відповідає за управління SSL/TLS сертифікатами, слугує reverse-proxy, та перенаправляє трафік в залежності від шляху запиту, наприклад, шлях, що починається на /api буде направлений до API сервіс (Symfony), а у випадку /chat-api на Сервіс чату.

Наступними йдуть сервіси з шару реального часу та повідомлень — це Сервіс чату та Mercure (Хаб). Чат сервіс — це високопродуктивний NestJS сервіс, що відповідає за з’єднання у режимі реального часу та підтримує свою власну логіку для зберігання повідомлень та інформації про чати. Для отримання додаткової інформації, наприклад, про користувачів, або для перевірки захищеності звертається до API сервісу за допомогою back-channel [27]. Та Mercure — це сервіс, що використовується апі сервісом для надсилання Server-Send Events (SSE) до користувачів, це дозволяє не перевантажувати використання Websockets для надсилання простих нотифікацій про різні події, що виникають на сайті [28].

Перейдемо до сервісів з шару бізнес логіки, тут знаходиться головний API сервіс, що є «серцем» системи. Він управляє усіма зв’язками за допомогою Doctrine ORM та надає REST API для взаємодії з даними. Та за допомогою вже описаного Mercure має можливість надсилати оновлення користувачам. Також в цьому шарі присутній Обробник подій (worker) — PHP-процес, керований супервізором, який обробляє «асинхронну» частину бізнес-логіки, щоб забезпечити оперативність роботи основного API.

Окремо можна помітити наявність шару для менеджменту, що складається з одного сервісу phpmyadmin. Він необхідний для забезпечення графічного інтерфейсу для адміністрування баз даних та налагодження під час розробки.

Загальну структуру взаємодії з даними можна розглянути, на прикладі Task (див. рис. 2.4). Вона полягає у взаємодії фронтенду з API, тому кожен запит, що стосується конкретного завдання, перевіряється сервером на наявність прав доступу до запитуваного ресурсу. Також, при зміні статусу, пріоритету, або типу завдання специфічна логіка (процесори) відповідають за правильний вибір налаштувань для застосування до завдання, забезпечуючи, щоб усі зміни були валідні. Коли задача зберігається — в базі даних «просинаються» Listener, що відстежують різницю в даних між старими та новими значеннями. Використовується TaskChangeLogListener — система створює запис в TaskChangeLog таблиці для збереження виявлених змін та їх дату. Також ці дані про зміну надсилаються у TaskChangeListener, що формує зміни в повідомлення та надсилає його в MessageBus. Потім вони обробляються у CreateNotificationsHandler. Цей хендлер перевіряє якими саме каналами повинна надсилатись нотифікація клієнту: електронна пошта, оновлення на сайті, чи ще й в браузері. Далі вже сформовані фінальні нотифікації надсилаються до Mercure Hub що відповідає за саму доставку повідомлень. Далі візьмемо ситуацію, що другий користувач був підписаний на зміни завдання, тоді його браузер через пряме SSE (server side events) з’єднання отримує сигнал для того, щоб оновити список нових повідомлень, та після цього відображає нове повідомлення самому користувачу.

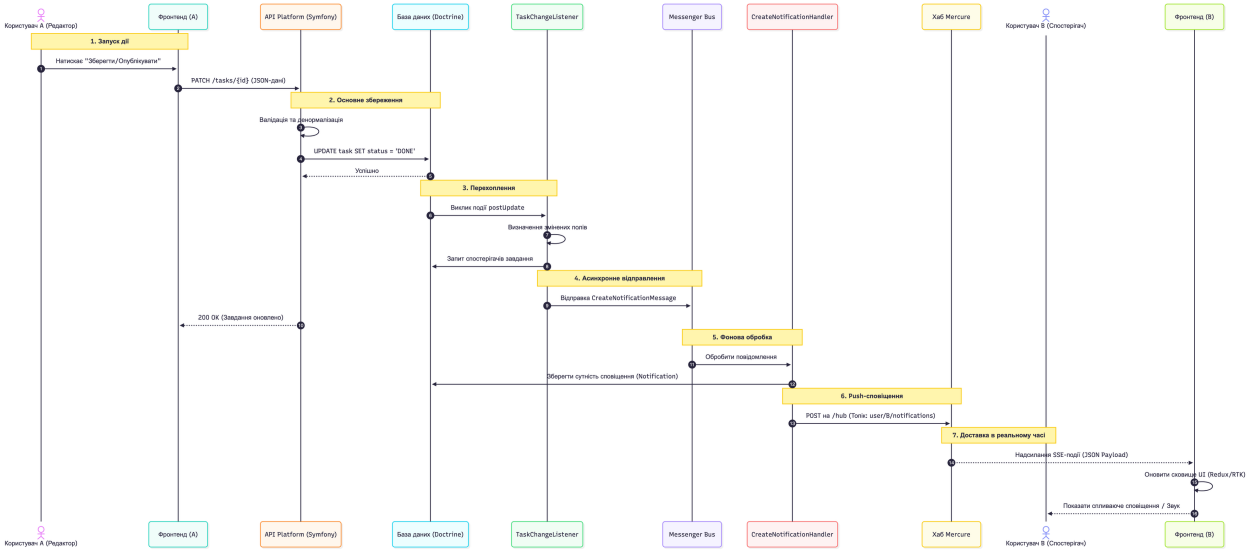


Рис. 2.4 Діаграма послідовності процесу зміни завдання та отримання іншими користувачем повідомлення про це

Наступним розглянемо процес нотифікації користувачів при надсиланні повідомлення у чат (див. рис. 2.5). Такий варіант буде гарною демонстрації взаємодії сервісів між собою.

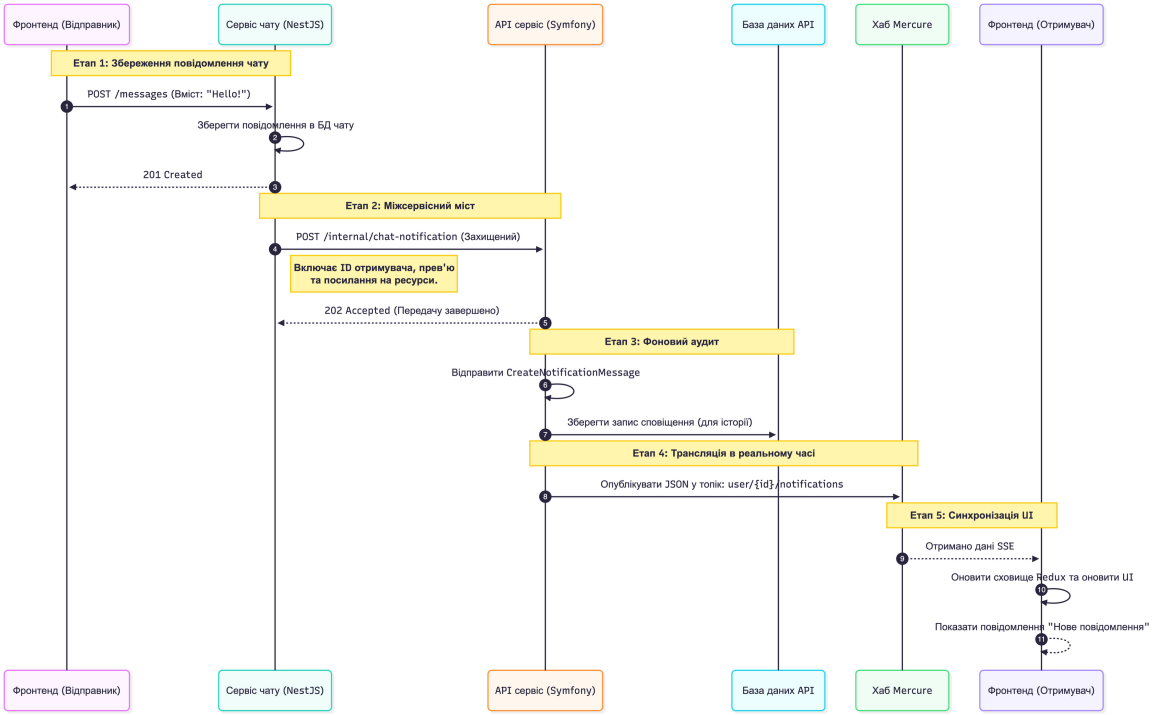


Рис. 2.5 Діаграма послідовності надсилання повідомлення в чат та нотифікування користувачів про оновлення в чаті

Першим кроком процес починається коли користувач А натискає кнопку надсилання повідомлення в чат. Це каже фронтенду звернутися до NestJS сервіса для обробки повідомлення.

Другим кроком чат-сервіс локально зберігає повідомлення, визначивши кого з користувачів потрібно проінформувати про отримання нових повідомлень він звертається до Symfony API через захищений back-channel. Це звернення необхідне тому що саме у Symfony зберігається повний список нотифікацій, що надсилаються користувачам та налаштування того як вони вподобають ці повідомлення отримувати.

Третім кроком Symfony отримує цей запит, та замість того щоб змушувати користувача чекати, той надсилає CreateNotificationMessage у внутрішній Message Bus. Та чат-сервіс миттєво отримуючи «202 Accepted» може переходити до інших задач.

Четвертий крок передбачає обробку повідомлення background worker. Він створює сутність Notification та зберігає її, для того що навіть у випадку коли отримувач офлайн, при заході на сайт він побачить усі останні повідомлення.

На п’ятому кроці background worker викликає MercureNotificationPublisher, щоб він надіслав сформований JSON в Mercure Hub. Це хаб виступає як «гаряча лінія» тримаючи тисячі браузерних підключень активними.

На шостому кроці браузер користувача Б отримує нотифікацію через довготривале підключення (SSE) та фронтенд, без необхідності перезавантаження сторінки, запитує останні повідомлення та показує їх користувачу.

Наступним можна роздивитись функціонал пов’язаний з спринтами (див. рис. 2.6).

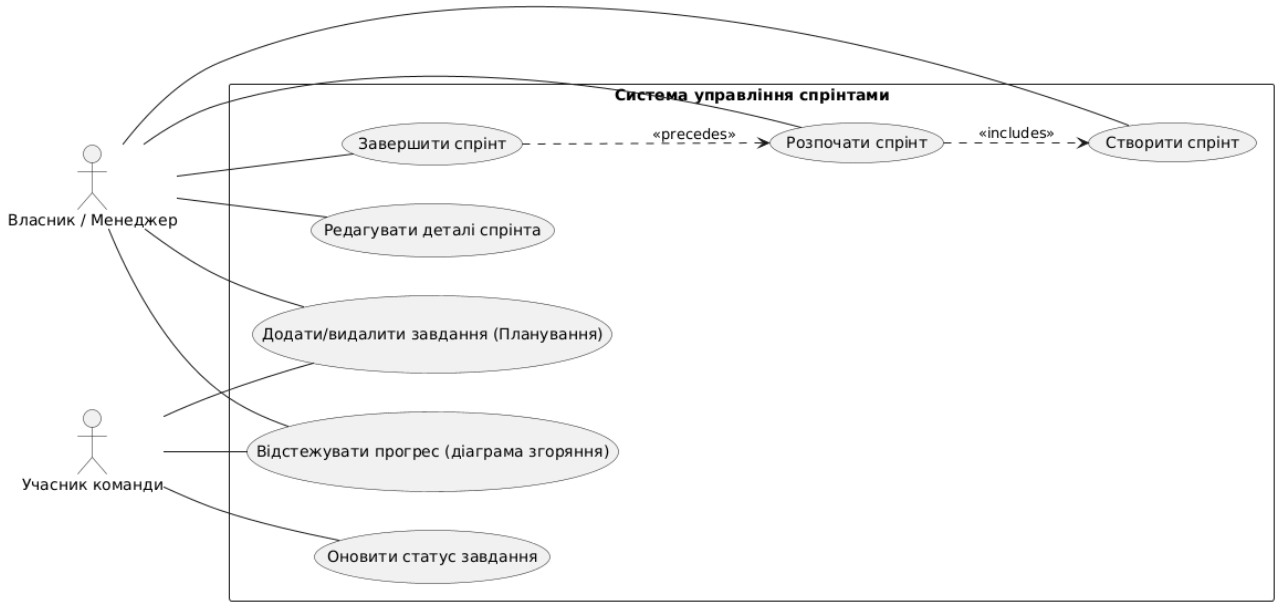


Рис. 2.6 Use cases діаграма для управління спринтами

Кожен овал означає спосіб використання, а саме інтеракцію користувача та системи. Перелік представлених use cases:

1. Створити спринт — початковий крок де менеджер створює «контейнер» для обмеженого по часу циклу розробки (зазвичай 2 тижні). Користувач вводить назву спринту, його мету та дату початку, та кінця. Після створення система позначає його статусом «planned» (заплановано);
2. Змінити деталі спринту — зміна деталей існуючого спринту у статусі «planned», або «active». Користувач натискає кнопку «Змінити» для зміни цілі спринту, або його термінів. PATCH запит надсилається для зміни даних залишаючи зв’язок для доданих задач;
3. Додати/Видалити задачі — спільний процес відбору, та визначення кількості сторі поінтів, завдань із загального списку завдань (беклогу), які мають бути виконані протягом спринту. Менеджер натискає на задачу та після кліку на поле «Спринт» у випадаючому списку обирає потрібний спринт та вказує числом кількість сторі поінтів. Система оновлює поле

sprint_id на ідентифікатор обраного спринту та поле totalStoryPoints новим значенням;

4. Почати спринт — формальне переведення спринту зі статусу «заплановано» до «активний». Менеджер натискає на кнопку «Почати спринт» у вигляді деталей спринту. Система змінює статус на «active» та створює SprintBurndownSnapshot з початковою точкою T= 0, та ідеальною кривою згоряння на графіку.
5. Перевірити прогрес — відображення темпів роботи команди та обсяг задач, що залишились. Усі користувачі мають доступ до перегляду цього графіку для того, щоб бачити чи встигає команда з обсягом роботи. Система обраховує суму виконаних сторі поінтів на осі Y та час на осі X, та показує їх у порівнянні з лінійною регресією поінтів до часу.
6. Оновлення статусу задачі — щоденний цикл оновлень статусів задач, для підтримання актуальності інформації про проєкт. Член команди змінює статус задачі на сторінці деталей. У випадку зміни статусу на «Завершено» система позначає сторі поінти задачі як «завершені» та враховує це в наступних обрахунках графіку згоряння.
7. Завершити спринт — фінальне закриття спринту. Менеджер натискає кнопку «Завершити спринт» на сторінці деталей про спринт. Система змінює його статус на «completed», та переносить усі незавершені задачі зі спринту до беклогу.

Важливо зазначити, що для генерації даних для графіків згоряння застосовується cron-job, що виконується щодня (зазвичай опівночі) для своєчасного оновлення даних.

2.4 Реалізація вебсистеми управління завданнями та командної взаємодії в IT-проєктах

Розглянемо фрагмент коду, що може потребувати пояснення для правильного розуміння:

```
1. private function notifyMentions(Comment $comment, array $mentionIds, string $creatorName, ?string $creatorId, array &$notifiedIds): void
2. {
3.     $ctx = $this->resolveResourceContext($comment);
4.
5.     // a. Direct User Mentions
6.     foreach ($this->userRepository->findBy(['id' => $mentionIds]) as $user) {
7.         $uid = (string)$user->getId();
8.         if ($uid !== $creatorId) {
9.             $this->dispatch($uid, NotificationType::MENTION, "{$creatorName} mentioned you", $comment, $ctx);
10.            $notifiedIds[] = $uid;
11.        }
12.    }
```

Цей фрагмент взятий з сутності CommentMentionProcessor, що відповідає за обробку згадувань користувачів, або ролей у коментарі [25]. На початку методу notifyMentions використовується інший метод для визначення типу сутності: завдання, подія, чи ідея. Надалі ця інформація буде використана для правильного визначення каналу, через який вона буде надсилатися. Фрагмент на рядках 6-11 відповідає за проходження по усім ідентифікаторам, що приходять у аргументі $mentionIds, та спробі отримання такого користувача, та якщо він існує та не є користувачем, що й створив коментар, на 9 рядку викликає метод для надсилання повідомлення [26].

```
13.    // b. Role Mentions (Expand to Users)
14.    $roles = $this->roleRepository->findBy(['id' => $mentionIds]);
15.    foreach ($roles as $role) {
16.        foreach ($role->getIdeaUsers() as $iu) {
17.            if ($user = $iu->getUser()) {
18.                $uid = (string)$user->getId();
19.                if ($uid !== $creatorId && !in_array($uid, $notifiedIds, true)) {
```

```
20.                     $this->dispatch($uid,                 NotificationType::MENTION,
   "{$creatorName}    mentioned    @{$role->getTitle()}",    $comment,    $ctx,    $role-
   >getTitle());
21.                     $notifiedIds[] = $uid;
22.                 }
23.             }
24.         }
25.     }
26. }
```

Далі в рядках 15-26 описаний більш комплексний варіант, при якому спочатку отримуються ролі, що були згадані, за ролями та ідеєю дістаються користувачі. Якщо вони не були вже проінформовані про згадку, та не є авторами створеного коментаря, то для них надсилається повідомлення.

Цей фрагмент коду може бути складним для розуміння через достатню кількість використаних зв'язків між сутностями, а саме: Comment -> Role -> IdeaUser -> User. Також окремої уваги заслуговує масив $notifiedIds, що відповідає за збереження ідентифікаторів користувачів, що вже були проінформовані про згадку в коментарі, для уникнення надсилання двох однакових повідомлень.

Наступним фрагментом коду буде функціонал навігації до конкретного повідомлення в чаті лише за його ідентифікатором.

```
1. private async fetchTargetMessageWindow(
2.  baseQuery: SelectQueryBuilder<ChatMessage>,
3.  conversationId: string,
4.  targetId: string,
5.  itemsPerPage: number,
6. ): Promise<ChatMessage[]> {
7.  const targetMessage = await this.messageRepository.findOne({
8.    where: { id: targetId, conversationId },
9.  });
10. if (!targetMessage) {
11.   return                          baseQuery.clone().orderBy('message.createdAt',
   'DESC').take(itemsPerPage).getMany();
12. }
```

На рядку 7-9 ми отримуємо цільове повідомлення, за його ідентифікатором та ідентифікатором розмови, щоб упевнитись, що воно взагалі

існує. У випадку якщо такого повідомлення не знайдено, як вказано на рядках 10-12, ми завантажуємо та повертаємо останні доступні повідомлення.

```
13.  const half = Math.floor(itemsPerPage / 2);
14.  const olderMessages = await baseQuery
15.    .clone()
16.    .andWhere('message.createdAt      <=      :targetDate',      {      targetDate:
     targetMessage.createdAt })
17.    .orderBy('message.createdAt', 'DESC')
18.    .take(half + 1)
19.    .getMany();
20.
21.  const newerMessages = await baseQuery
22.    .clone()
23.    .andWhere('message.createdAt      >      :targetDate',      {      targetDate:
     targetMessage.createdAt })
24.    .orderBy('message.createdAt', 'ASC')
25.    .take(half)
26.    .getMany();
27.
28.  return [...newerMessages.reverse(), ...olderMessages];
     }
```

Тепер для зручності користувача та ефективності системи нам потрібно отримати усі повідомлення навколо цільового в «радіусі» itemsPerPage/2, для цього ми спочатку завантажили половину необхідної кількості сутностей, що старші за ціль, плюс одне повідомлення, щоб увімкнути саму ціль до масиву, та іншу половину, що молодше цілі. Потім, для того, щоб повернути усі отримані повідомлення у правильному порядку ми реверсуємо масив повідомлень, що молодші за ціль, та приєднуємо іншу половину.

Складність даного фрагменту полягає в тому, що SQL-запити зазвичай повертають дані в одному порядку сортування, але ця функція вимагає «розгалуження» від центральної позначки часу.

Далі розглянемо функціонал, що є достатньо важким для розуміння, а саме сутність TaskChangeLogBuilder. Цей двигун використовується для підтримки журналу змін задачі, головна мета якого оцінити те як змінилась задача і перетворити та окремо зберегти ці зміни у зручному, зрозумілому для

людини форматі. Головна складність полягає в необхідності обробки великої кількості видів вхідної інформації та її подальшої нормалізації.

У першу чергу потрібно оглянути метод в котрому використана високий ступінь гнучкості для нормалізації:

```
private function normalizeValue(mixed $value): mixed
{
   return match (true) {
       $value instanceof User => $value->getFullName(),
       $value instanceof \DateTimeInterface => $value->format(\DateTimeInterface::ATOM),
       $value instanceof \BackedEnum => $value->value,
       is_object($value) => $this->normalizeObject($value),
       default => $value,
   };
}
```

Коли поле задачі змінюється змінна Value може мати різний тип, але все одно добре обробляється.

Далі на рядках 3-11 використовується одна з просунутих можливостей PHP — рефлексія, для динамічного визначення правильного відображення об’єкту замість прописування усіх варіантів вручну, воно шукає на типові назви полів:

```
1.  private function normalizeObject(object $value): mixed
2.  {
3.      $reflection = new \ReflectionObject($value);
4.      $title = null;
5.
6.      foreach     (['getFullName',     'getTitle',     'getName',     'getOriginalName',
    '__toString'] as $method) {
7.          if ($reflection->hasMethod($method)) {
8.              $title = (string) $value->$method();
9.              break;
10.         }
11.     }
12.
13.     $data = ['title' => $title];
14.
15.     if ($reflection->hasMethod('getTranslations')) {
16.         $translations = $value->getTranslations();
17.         if ($translations !== null && $translations !== []) {
18.             $data['translations'] = $translations;
```

```
        }
    }

    return ($data['title'] !== null || isset($data['translations'])) ? $data : $value;
}
```

Висновки до другого розділу

У другому розділі було детально спроєктовано структуру та функціональні можливості системи, де центральною сутністю виступає «Ідея», навколо якої інтегруються завдання, події та чати. Визначені варіанти використання охоплюють повний життєвий цикл управління проєктами: від створення гнучких завдань із підтримкою markdown та пріоритезації до розгалуженої системи комунікацій через контекстні коментарі та чати. Особливу увагу приділено збереженню наскрізного контексту роботи, що реалізується через можливість додавання зв’язків повідомлень до конкретних об'єктів та підтримку історії змін для кожного завдання.

Розробка архітектури бази даних та компонентної схеми підтвердила доцільність обраного гібридного підходу, де основний API на Symfony керує цілісністю бізнес-даних у MySQL, а сервіс на NestJS забезпечує високопродуктивну роботу чатів. Проєктування зв'язків між сутностями дозволило реалізувати складні реляції «багато до багатьох» між завданнями, подіями та обговореннями, що є ключовим для усунення інформаційних розривів. Використання Mercure Hub як окремого шару для Server-Send Events (SSE) дозволяє ефективно доставляти нотифікації в реальному часі без надмірного навантаження на основні сервіси системи.

Заключним етапом проєктування став опис алгоритмів взаємодії сервісів та логіки обробки даних, зокрема процесів синхронізації станів через Message Bus та фонового виконання важких операцій. Реалізація таких компонентів, як CommentMentionProcessor для обробки згадок та TaskChangeLogBuilder для

автоматичного ведення журналу змін, забезпечує високу прозорість дій команди та автоматизацію рутинних процесів. Спроєктована система управління спринтами з використанням діаграм згоряння (Burndown Charts) створює надійний інструментарій для моніторингу прогресу, що в сукупності з іншими модулями формує цілісне середовище для ефективної командної взаємодії.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК ВИКОРИСТАННЯ ДОДАТКУ

3.1 Порядок та опис роботи з програмним додатком (Опис інтерфейсу)

Інтерфейс користувача розроблено на основі сучасних принципів вебдизайну, тобто основний акцент було зроблено на: простоту навігації, мінімалізм, чітку ієрархію та адаптивність під різні типи пристроїв. Для зручності розробки та підтримки єдиної дизайн-системи було використано бібліотеку Material UI (MUI), що повністю відповідає стандартам Google Material Design.

У режимі робочого столу інтерфейс складається з двох основних зон (див. рис. 3.1 та 3.2).

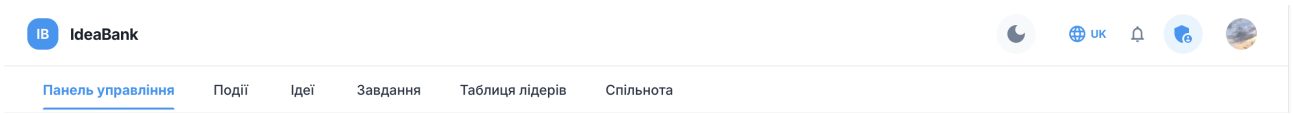


Рис. 3.1 Стандартний вигляд верхньої панелі керування

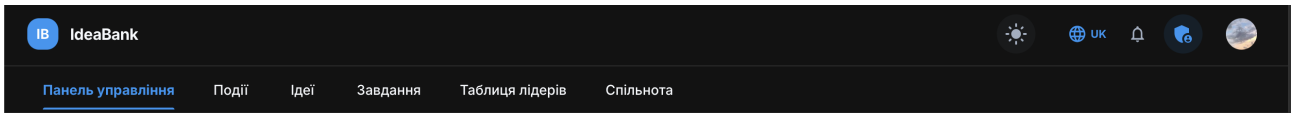


Рис. 3.2 Стандартний вигляд верхньої панелі керування за темної теми

Верхня панель керування містить такі елементи: назву додатку; перемикач теми сайту зі світлої на темну, та навпаки; перелік мов для відображення сайту; меню сповіщень; при наявності в користувача прав адміністратора також відображається кнопка для переходу в панель адміністратора; іконка користувача з мінімальним набором дій. Та під цим рядком елементів знаходяться посилання на основні сторінки системи.

Розберемося з елементами верхньої панелі керування детальніше. Перелік мов на поточний момент включає у себе лише англійську та українську мови з відповідними іконками, та може бути розширений пізніше (див. рис. 3.3).

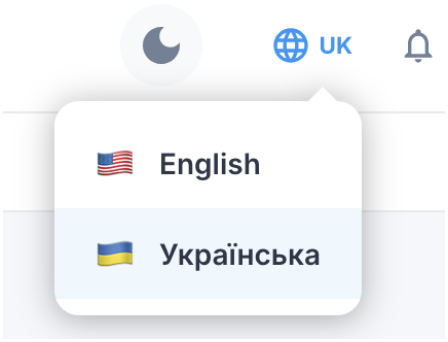


Рис. 3.3 Вигляд меню для вибору мови сайту

Розглянемо детальніше меню сповіщень (див. рис. 3.4). Кнопка для відкриття цього меню має вигляд дзвіночка з червоним значком з кількістю непрочитаних повідомлень. При натисканні на кнопку ми можемо бачити один загальний та 4 специфічних фільтри. За допомогою них ми можемо фільтрувати усі повідомлення за типами, наприклад, чи повідомлення стосується оновлень у задачах, згадки користувача, нових повідомлень у чатах, або нових подій. Присутня кнопка для автоматичного прочитання усіх повідомлень, при натисканні на яку усі повідомлення будуть позначені як прочитані та червоний бейджик пропаде.

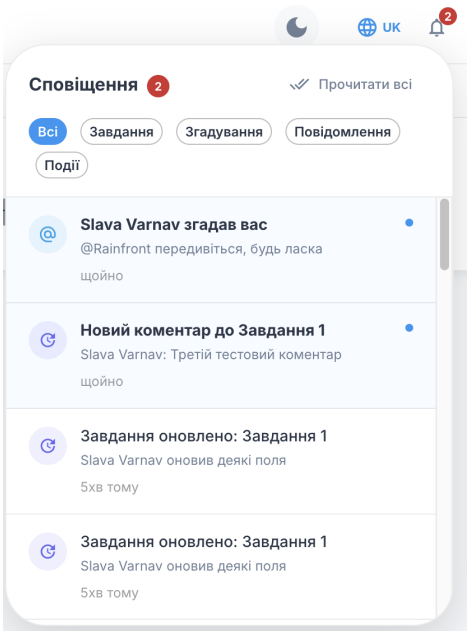


Рис. 3.4 Загальний вигляд меню з повідомленнями

При натисканні на повідомлення воно позначається прочитаним та перенаправляє на відповідну сторінку, та, наприклад, для згадок коментарях, після завантаження нової сторінки вона буде прогорнута до необхідного елементу та підсвітить його (див. рис. 3.5).

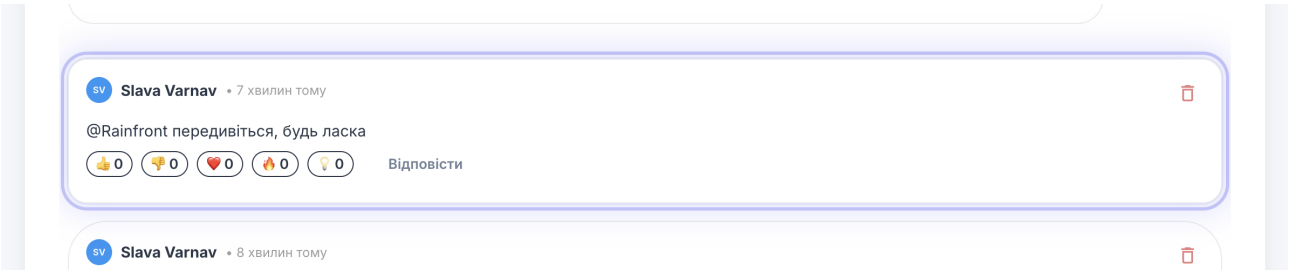


Рис. 3.5 Приклад підсвітки коментаря при перенаправлені на сторінку

Якщо користувач натисне на нотифікацію (див. рис. 3.6) про нове повідомлення він також буде перенаправлений до чату.

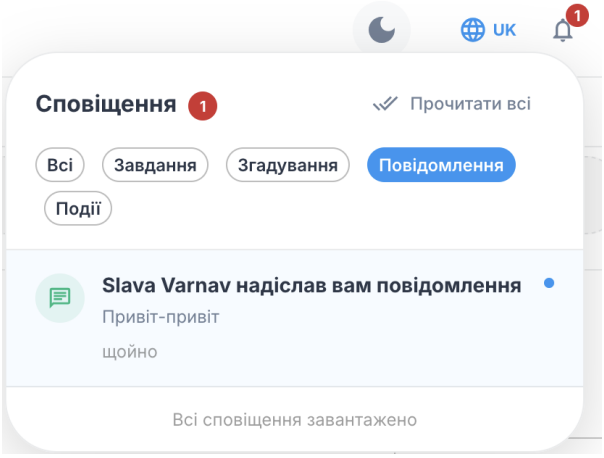


Рис. 3.6 Приклад нотифікації про нове повідомлення в чаті

Та після завантаження сторінки відповідне повідомлення також буде підсвічено (див. рис. 3.7).

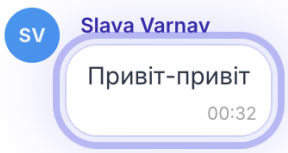


Рис. 3.7 Підсвітка повідомлення після натиснення на нотифікацію в меню

Наступною йде кнопка панелі адміністратора. Вона доступна користувачу лише у тому разі, якщо він має роль Адміністратора в системі (див. рис. 3.8).

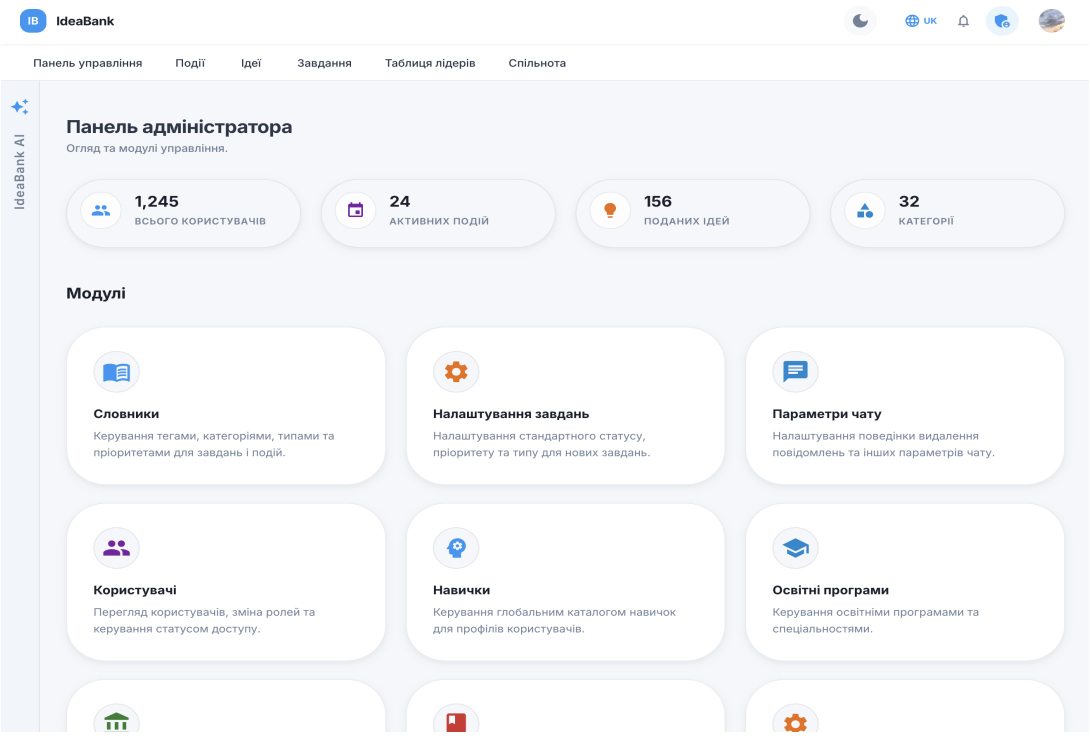


Рис. 3.8 Панель адміністратора

Далі можна помітити аватар користувача, при натисканні на який відкриється меню з декількома варіантами вибору (див. рис. 3.9).

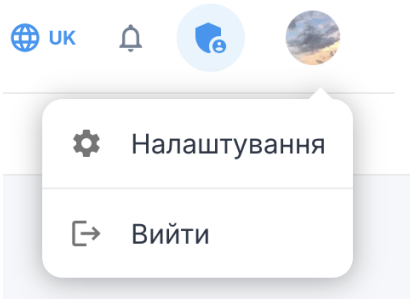


Рис. 3.9 Меню користувача

На сторінці ідеї розглянемо доступні опції для навігації (див. рис. 3.10). Тут нас цікавлять навігаційні кнопки «Чати», «Завдання» та «Спринти», ці посилання є основними, що використовуються для управління командною розробкою.

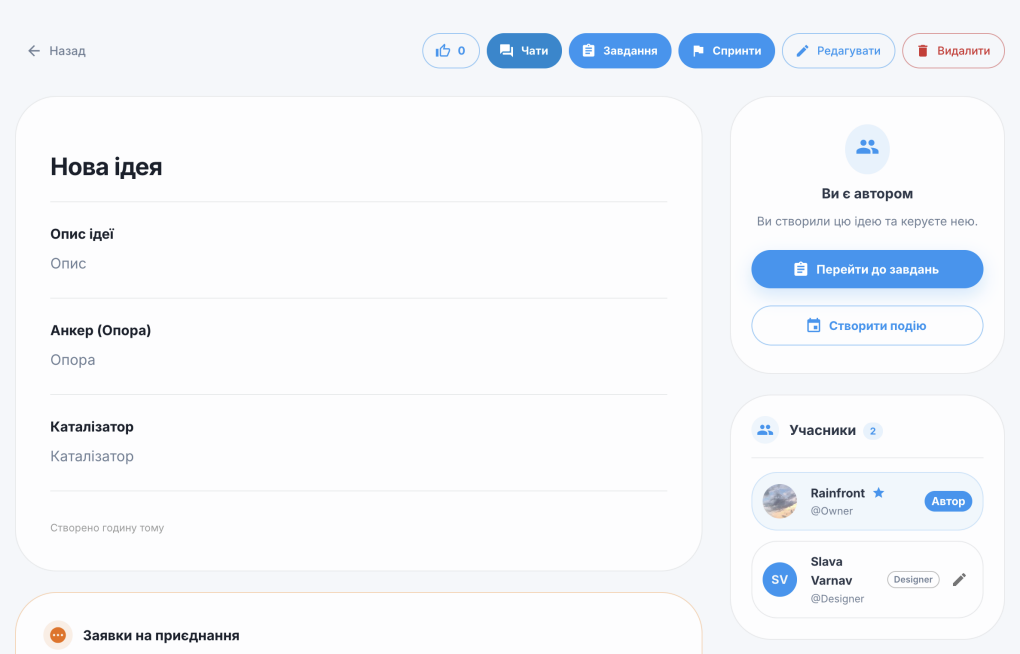


Рис. 3.10 Сторінка ідеї та доступні опції для навігації

Перейдемо до сторінки завдань ідеї (див. рис. 3.11).

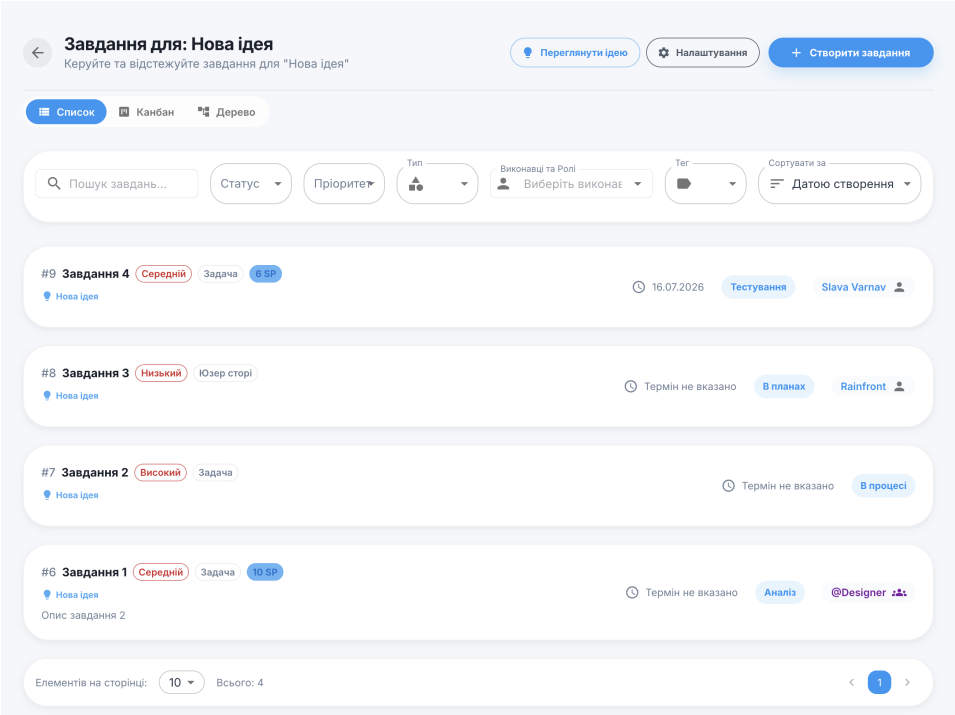


Рис. 3.11 Загальний вигляд сторінки з завданнями

Розберемо елементи на цій сторінці в деталях. Праворуч від назви ідеї можемо помітити кнопку, щоб повернутися до ідеї не залежно від того з якого місця ми потрапили на сторінку задач до цього. Далі праворуч знаходиться

кнопка для задання індивідуальних налаштувань для задач ідеї. Одразу після створення ідеї вона використовує стандартні значення (див. рис. 3.12), що задані адміністратором на відповідній сторінці в адмін панелі (див. рис. 3.13 – 3.16).

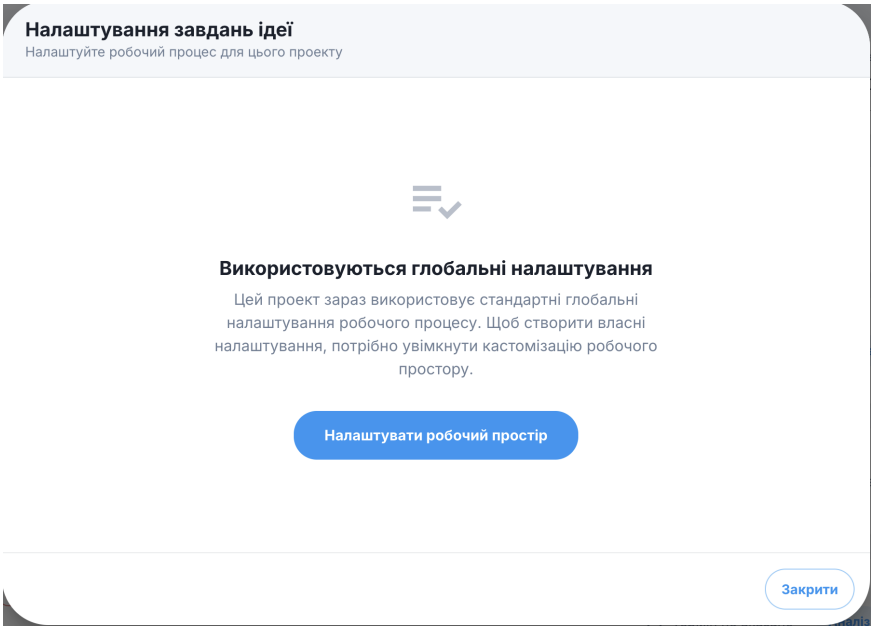


Рис. 3.12 Повідомлення про використання налаштувань за замовчуванням

При натисненні на кнопку «Налаштувати робочий простір» користувачу стане доступне меню налаштувань основних сутностей індивідуально для ідеї.

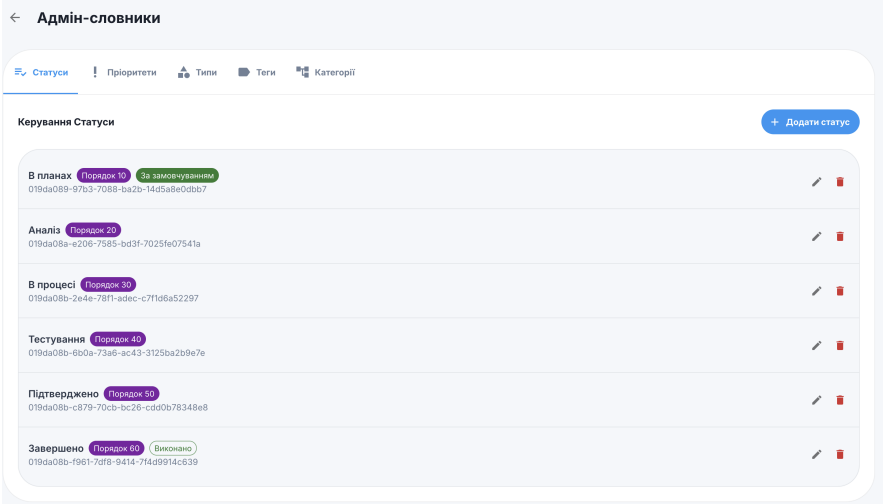


Рис. 3.13 Стандартні налаштування статусів в адмін панелі

Для усіх подібних сутностей є можливість задати «вагу», що буде використана для визначення порядку значень у меню вибору.

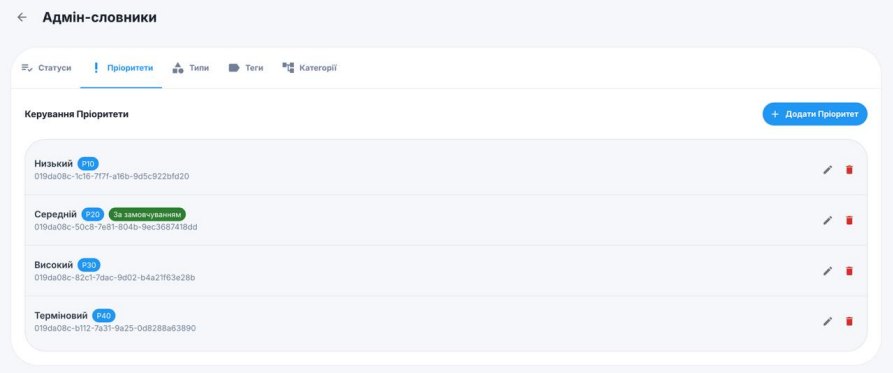


Рис. 3.14 Стандартні налаштування пріорітетів в адмін панелі

Для зручності адміністрування при відображенні подібних значень ми можемо побачити відразу й їх ідентифікатор. З його допомогою можливо швидше знайти необхідні записи у базі даних.

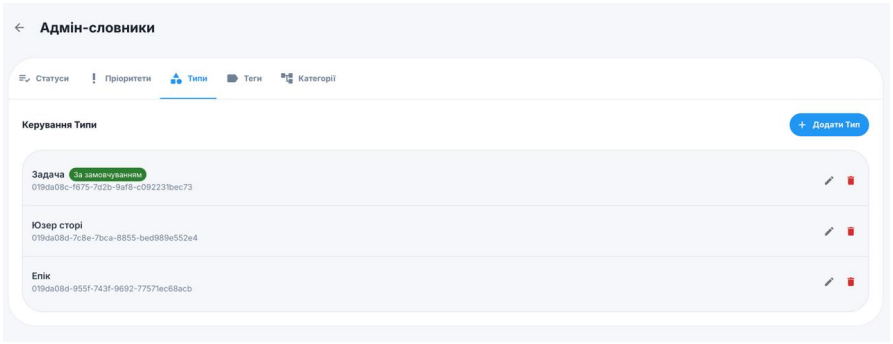


Рис. 3.15 Стандартні налаштування типів в адмін панелі

Стандартні значення для завдань можна додавати не лише через індивідуальні налаштування для типів, а й на окремій сторінці в налаштуваннях (див. рис. 3.16)

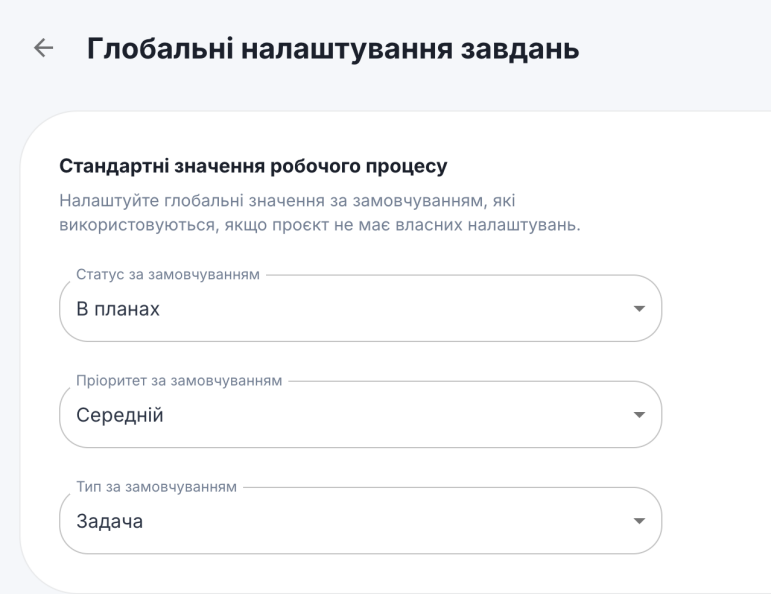


Рис. 3.16 Налаштування стандартних значень для завдань в адмін панелі

Для кожного з показаних налаштувань можливо змінити порядок, та становити за замовчуванням. Також для зручності користувачів, що користуються різними мовами, можливо ввести переклад цих значень на інші мови (див. рис. 3.17).

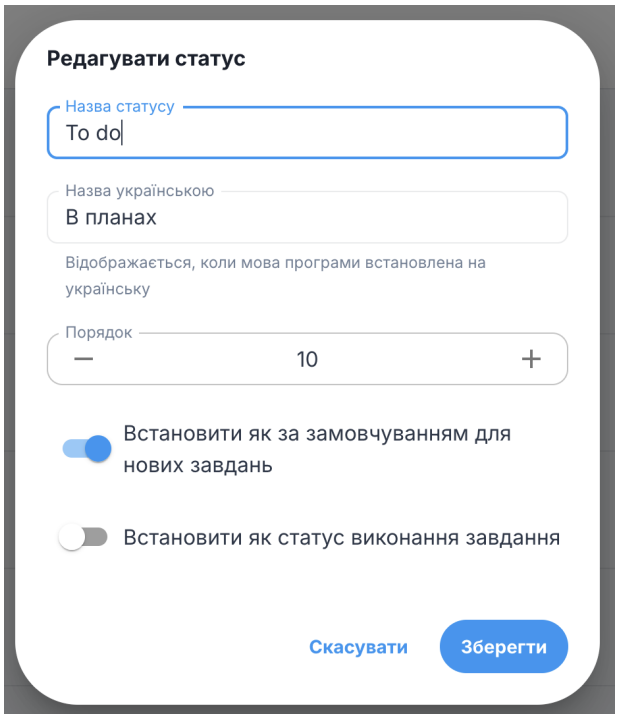


Рис. 3.17 Форма редагування статусу задачі

Повернемося до індивідуальних налаштувань для певної ідеї. Після натиснення на кнопку «Налаштувати робочий простір». Після цього з’являється можливість налаштувати усі значення індивідуально під проєкт (див. рис. 3.18).

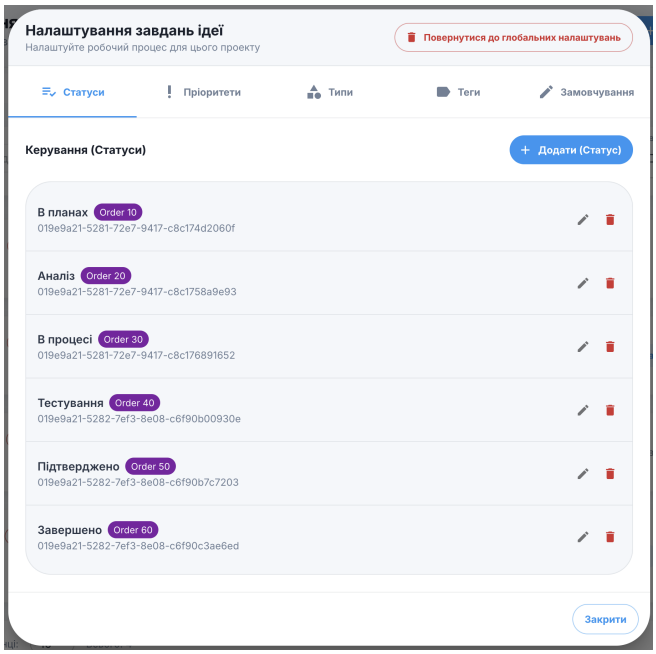


Рис. 3.18 Модальне вікно для налаштування індивідуальних значень для ідеї

При спробі повернення до глобальних налаштувань натискаючи відповідну кнопку. Користувач матиме змогу трансформувати нові значення, що він створив до стандартних. Також, для зручності користування, є можливість автоматичного порівняння назв значень, для швидшої взаємодії (див. рис. 3.19).

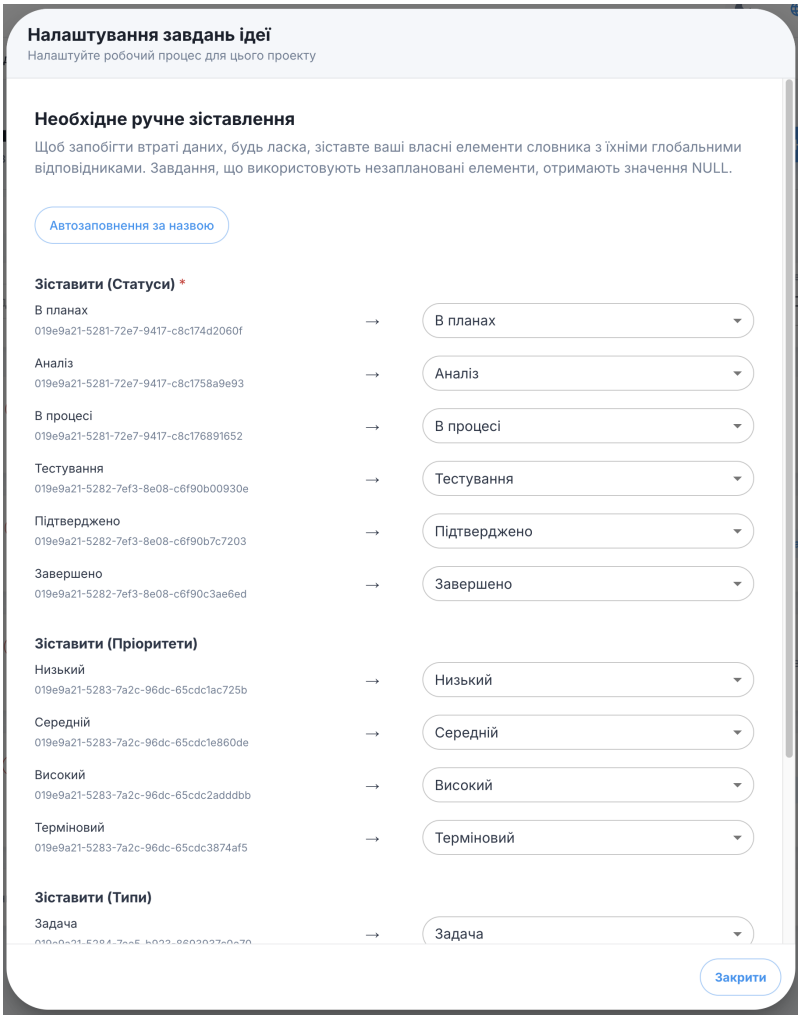


Рис. 3.19 Модальне вікно для зіставлення індивідуальних та стандартних значень

Повернемось до відображення завдань для ідеї. Там ми можемо побачити різні варіанти відображення списку задач, а саме: Список, Канбан та Дерево.

При використанні відображення Списком усі задачі знаходяться один під одним для швидкого й зручного перегляду інформації про них. Наступним є варіант відображення Канбан, що передбачає відображення задач у колонках, що відповідають їх статусам (див. рис. 3.20). Для зручності та швидкості задачі можна переносити між колонками для зміни їх статусу за допомогою Drag’n’Drop. Також можливе налаштування колонок, що повинні відображатися при цьому варіанті відображення.

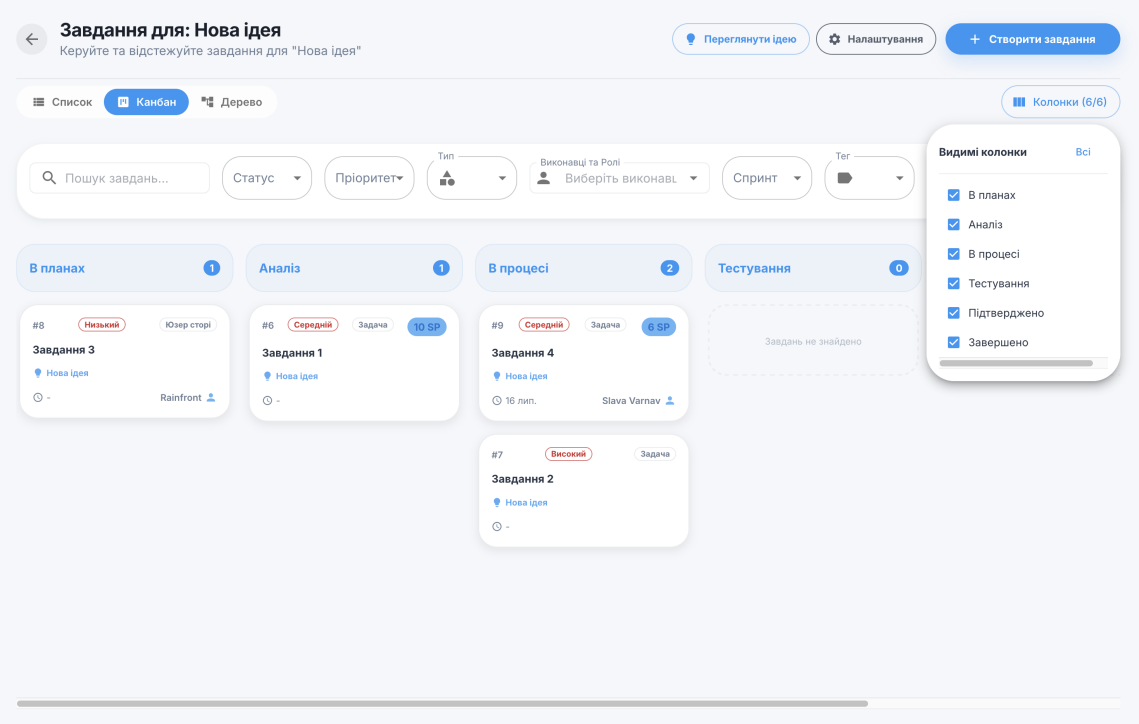


Рис. 3.20 Варіант відображення Канбан

Наступним йде варіант відображення Дерево, що потрібен для зручного розглядання залежностей між задачами, наприклад, яка задач є батьківською для інших (див. рис. 3.21).

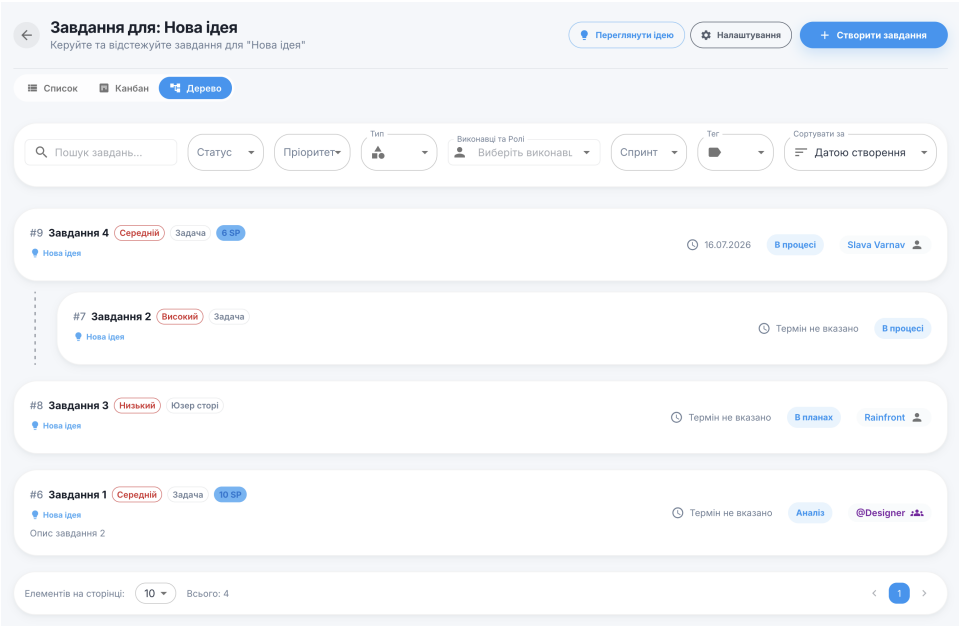


Рис. 3.21 Варіант відображення Дерево

Необхідно зазначити, що для цих відображень є можливість тонкого налаштування фільтрів для пошуку необхідних задач. Присутні фільтри за: статусом, пріоритетом, типом, виконавцем, тегом. Та можливе сортування за датою створення, або терміном виконання. Для усіх цих полів передбачено випадаючий список доступних значень (див. рис. 3.22).

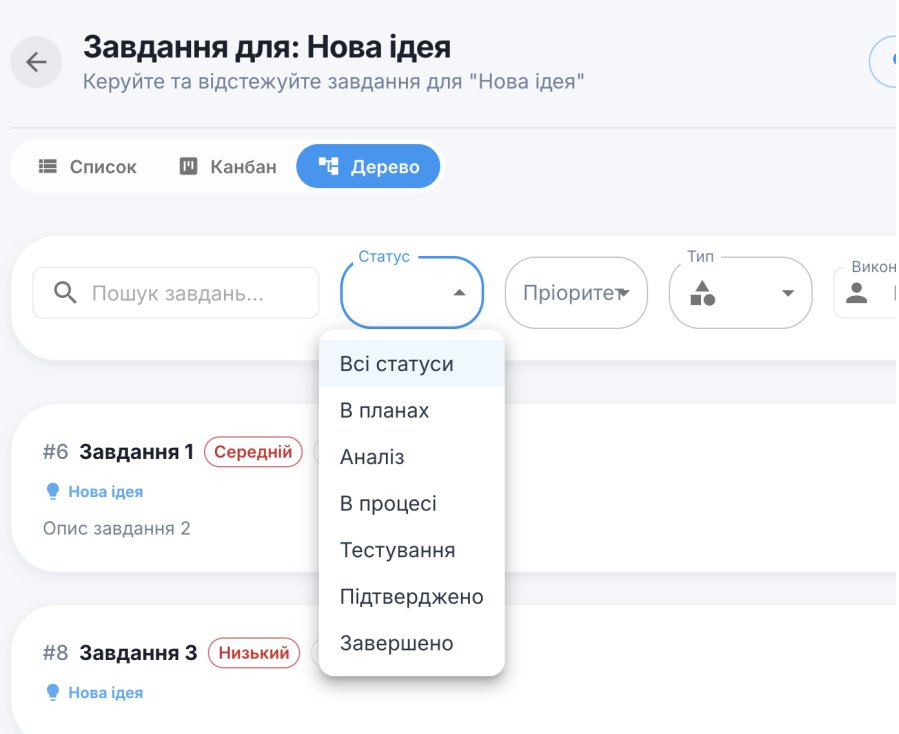


Рис. 3.22 Випадаючий список доступних значень для фільтрів

У випадку великої кількості задач у ідеї для зручної навігації між ними передбачено гнучку пагінацію (див. рис. 3.23).

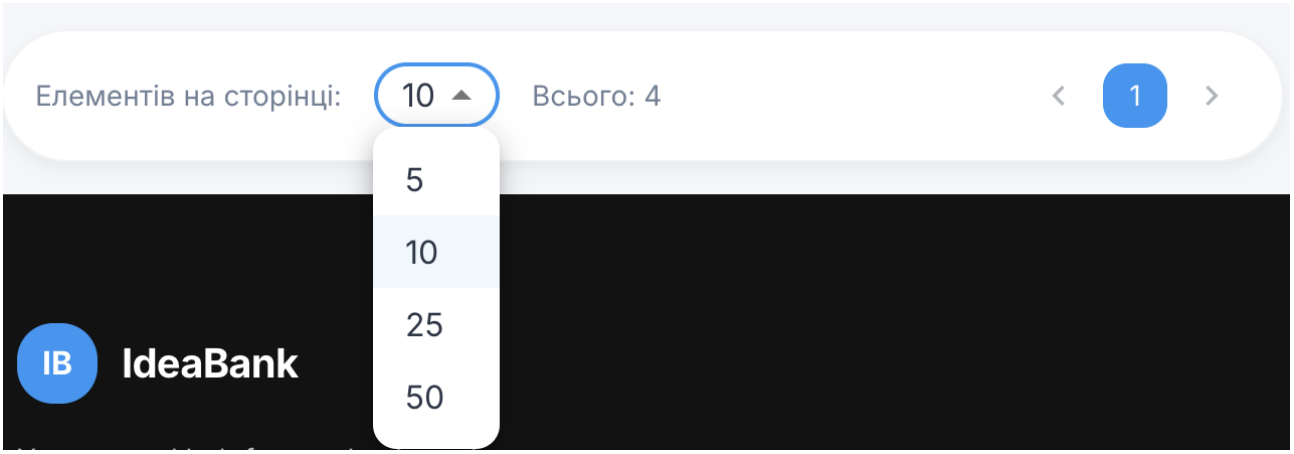


Рис. 3.23 Пагінація для задач

Для зручного та швидкого переглядання інформації щодо задачі при натисканні на неї відкривається швидкий перегляд, що має повний функціонал. Та при необхідності, натиснувши на кнопку «Повні деталі» є можливість перейти на окрему сторінку задачі (див. рис. 3.24).

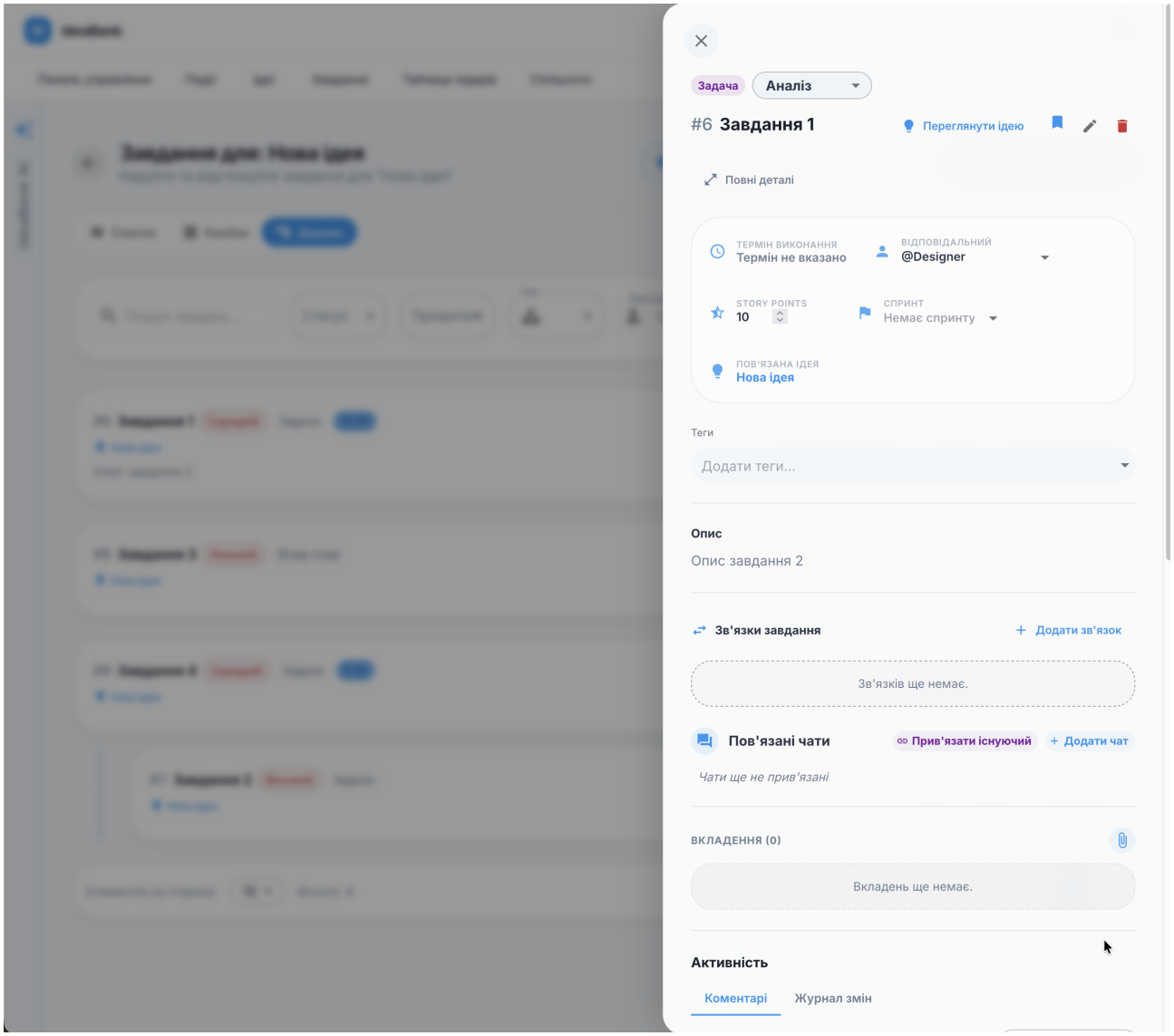


Рис. 3.24 Швидкий перегляд задачі

Тут ми вже можемо переглянути деталі задачі, та змінити інформацію про неї у пару кліків. Поля що змінюються найчастіше, наприклад: статус, відповідальний тощо можливо змінити прямо з сторінки деталей. Для зміни інших полів потрібно використати модальне вікно (див. рис. 3.25).

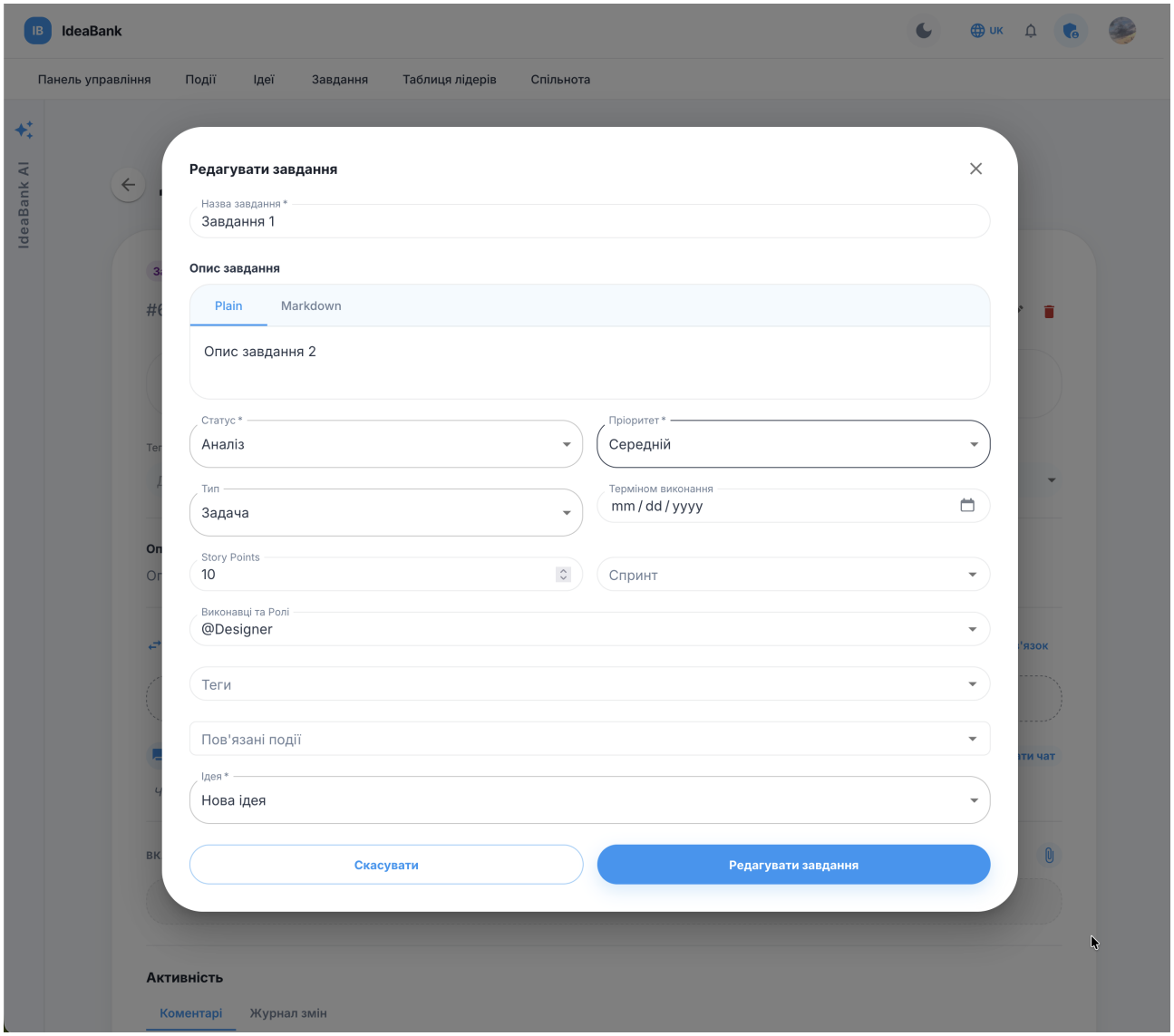


Рис. 3.25 Модальне вікно для створення або зміни задачі

На сторінці деталей ми можемо помітити можливість додавання тегів, встановлення зв’язків між різними задачами, прив’язки чатів, та подій, додавання вкладень.

Наступною важливою складової задач є комунікація. Вона може відбуватися у прив’язаних чатах, та для збереження даних на довгий час краще за все використовувати коментарі, що оновлюються в режимі реального часу, з ієрархією відповідей та реакціями, та можливістю використання розмітки markdown та сортування. Також може виникати необхідність у зверненні та прямому інформуванні її про залученість до певної людини, або ролі, що

працює у ідеї, для цього є механізм згадок, де можна обрати потрібну людину (див. рис. 3.26 та 3.27).

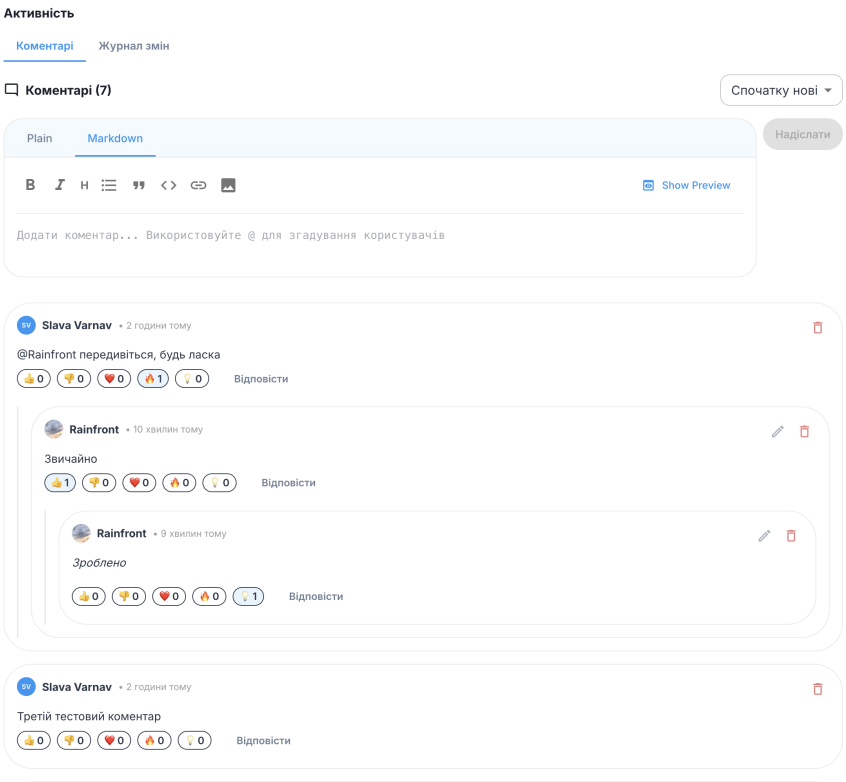


Рис. 3.26 Коментарі, їх ієрархія, реакція та розмітка

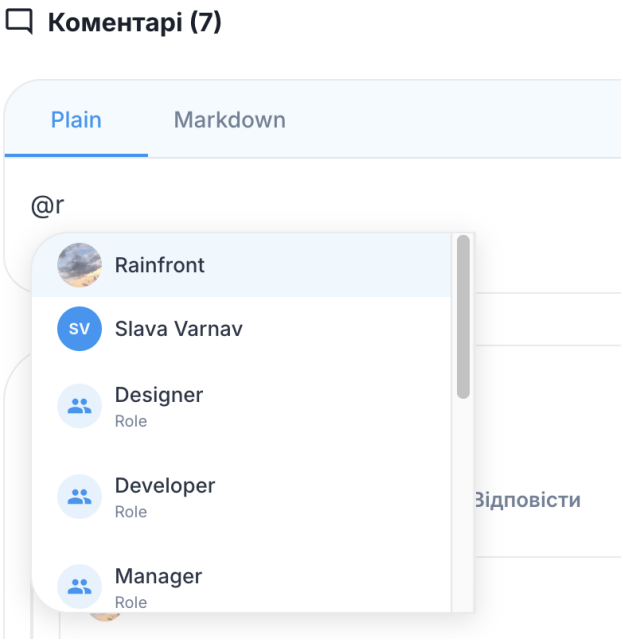


Рис. 3.27 Список доступних згадок

Важливим складником виконання задачі є інформованість про її зміни, тому у кожного користувача є можливість підписатися на зміни, та отримувати про це повідомлення. Та для того щоб передивитися хронологію змін можна відкрити вкладку «Журнал подій» з повним списком усіх змін, що стосуються задачі (див. рис. 3.28).

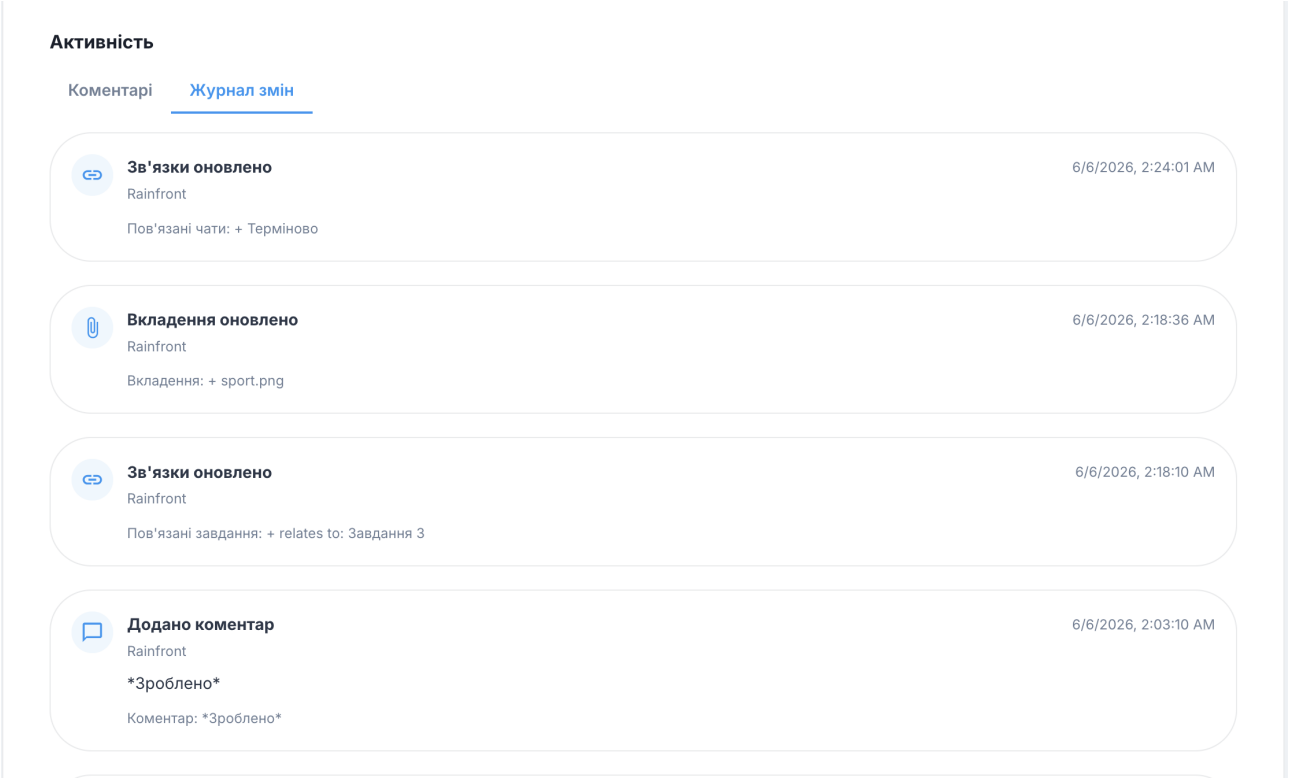


Рис. 3.28 Журнал змін, що стосується задачі у хронологічному порядку

Розглянемо наступну важливу частину функціональності, що стосується управління завданнями у ідеї — планування ітерацій. Цей модуль розроблений для планування короткострокових етапів розробки ідей. При створенні ідеї, після переходу на сторінку спринтів для користувача доступний беклог — список усіх незавершених та незапланованих завдань (див. рис. 3.29). Менеджер проєкту може готуванти задачі для наступних спринтів звідси.

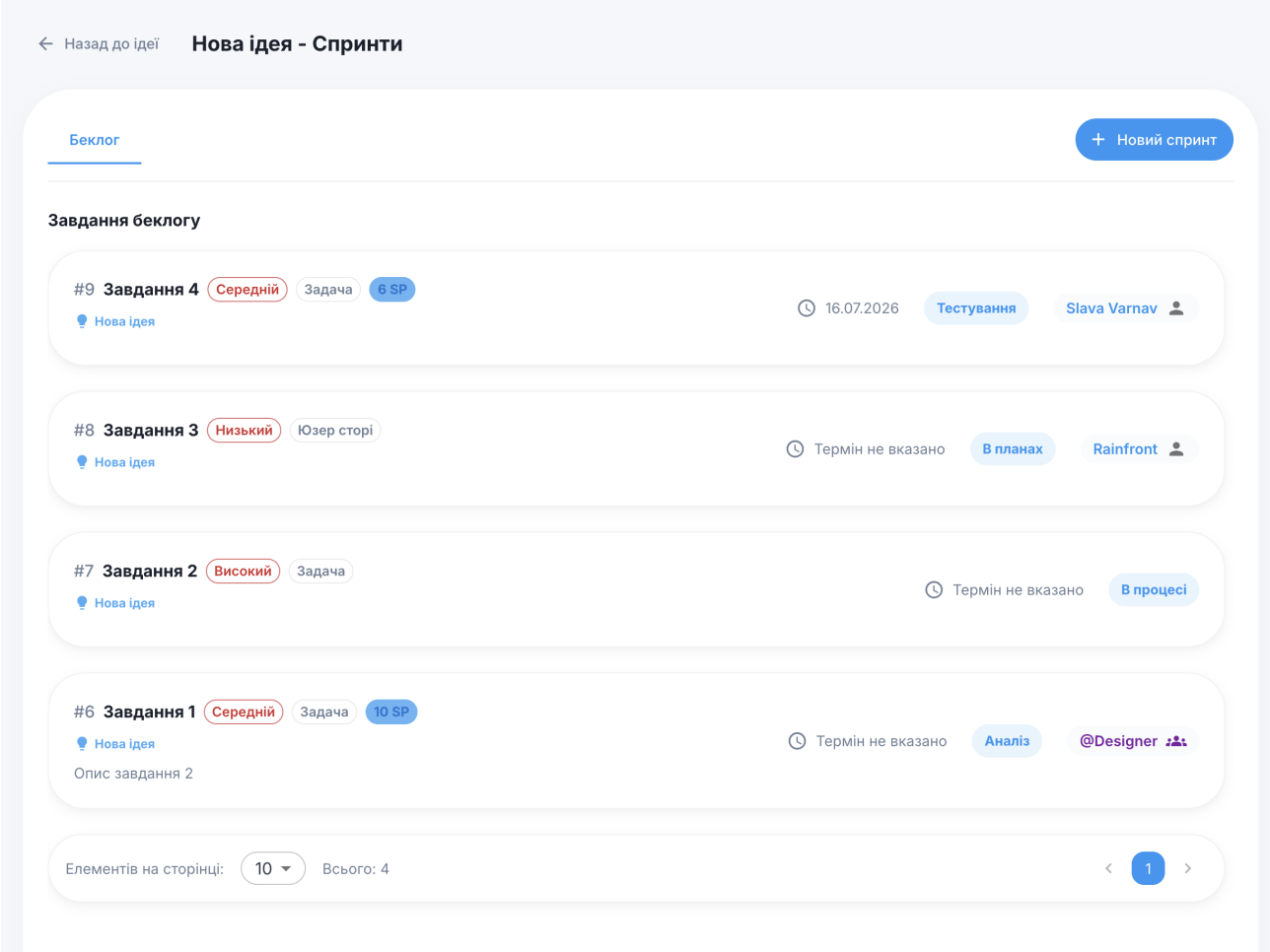


Рис. 3.29 Беклог ідеї

Після створення спринту він переходить у стан «запланований», тобто він має назву та ціль, але робота над ним ще не розпочата. Подібні спринти слугують для групування завдань на наступні ітерації. Активні спринти (зазвичай один) відображаються спочатку, та містять завдання, що команда виконує прямо зараз. Також доступні архівні спринти, вони використовуються для перегляду статистики виконаних завдань.

Розглянемо можливості керування спринтом (див. рис. 3.30).

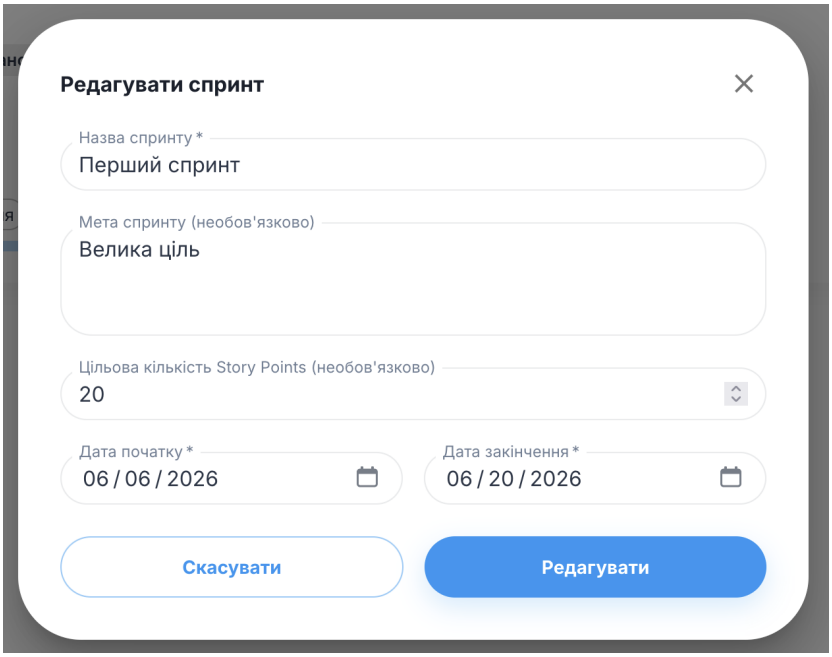


Рис. 3.30 Форма створення та/або редагування спринту

У шапці відображається його назва, дати початку та кінця та прогрес-бар (співвідношення виконаних сторі поінтів до запланованої кількості), та менеджеру доступна кнопка «Завершити спринт», що при натисканні аналізує поточний статус завдань (див. рис. 3.31).

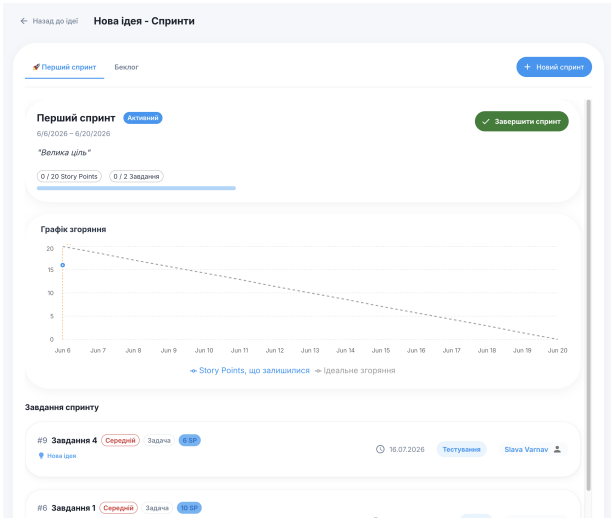


Рис. 3.31 Активний спринт з наявним графіком згоряння

Якщо є невиконані завдання, система пропонує обрати для них дію: перенести в загальний беклог, чи автоматично перекинути у наступний запланований спринт (див. рис. 3.32).

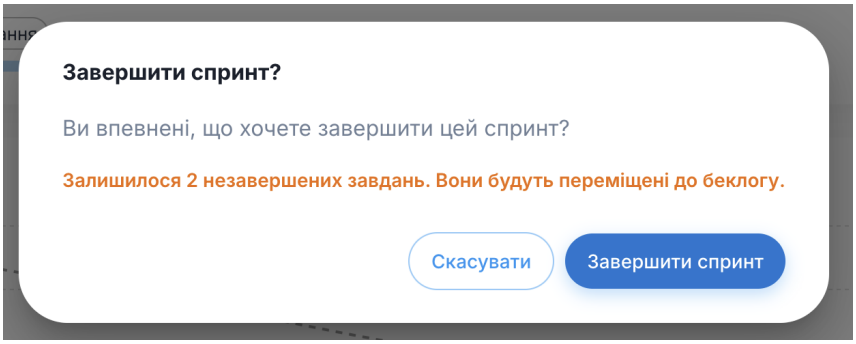


Рис. 3.32 Попередження при спробі завершити спринт при наявності

Це економить час менеджера на ручне перепризначення завдань. Кнопка «Почати спринт» доступна лише для запланованих спринтів, вона активує ітерацію, та починає процес моніторингу (див. рис. 3.33 та 3.34).

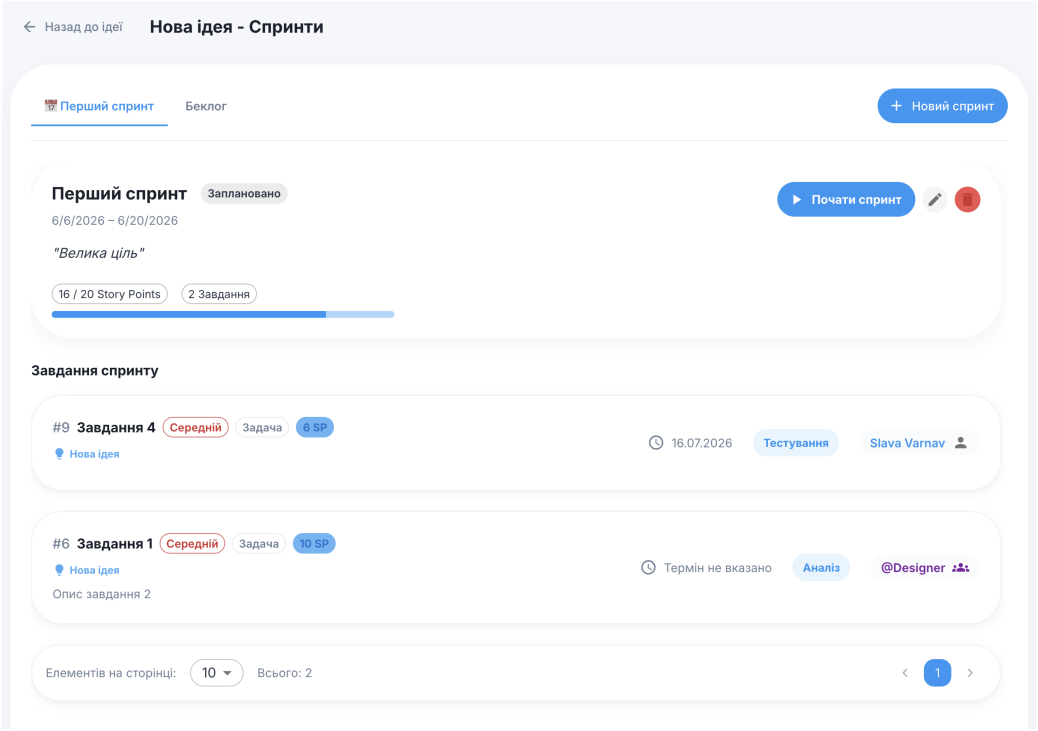


Рис. 3.33 Запланований спринт з задачами, що мають сторі поінти

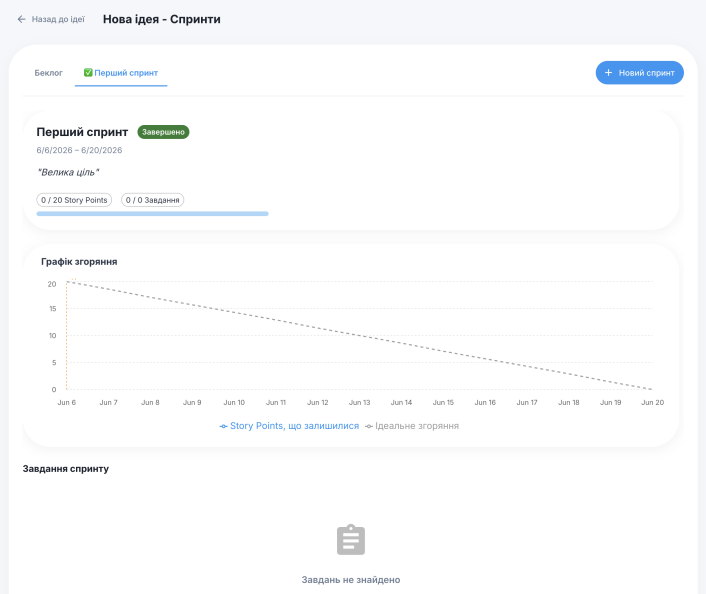


Рис. 3.34 Архівований спринт

Розглянемо принципи роботи графіку згоряння завдань. Його головна задача це візуалізація швидкості роботи команди, що допомагає прогнозувати дату завершення ітерації. На графіку відображаються 2 лінії:

- Планована лінія, що є прямою пунктирною лінією від початковою суми сторі поінтів у перший день спринту до 0 в останній день;
- Фактична лінія, що є суцільною лінією, яка оновлюється щодня на основі накопичених даних про виконані завдання. Вона падає вниз сходинками, коли завдання закриваються.

Графік автоматично адаптується під ширину контейнера та підтримує інтерактивні підказки — при наведенні курсору на точку графіка з'являється детальна інформація про поточну дату та кількість сторі поінтів, що залишилися. Оновлення даних для графіка відбувається у фоновому режимі раз на добу за допомогою консольної команди Symfony `app:sprint:snapshot`, яка запускається планувальником завдань Ofelia.

Далі розглянемо функціонал організації подій. На головній сторінці подій сайту присутні самі картки подій та пошук та швидки фільтри (див. рис. 3.35). Картки містять банер події (зображення з ефектом легкого масштабування при

наведенні курсору), назву події, дату та час початку (з автоматичним форматуванням локалі), іконку категорії та аватар організатора події. У нижній частині картки розміщено кнопку «Детальніше», яка перенаправляє на індивідуальну сторінку події. Текстовий пошук виконується з дебаунсом (затримка 500 мс) — запит відправляється на сервер лише після того, як користувач завершив введення тексту, що знижує навантаження на API. Категорії обираються через автозаповнення з підтримкою мульти-вибору та видалення тегів-категорій хрестиком.

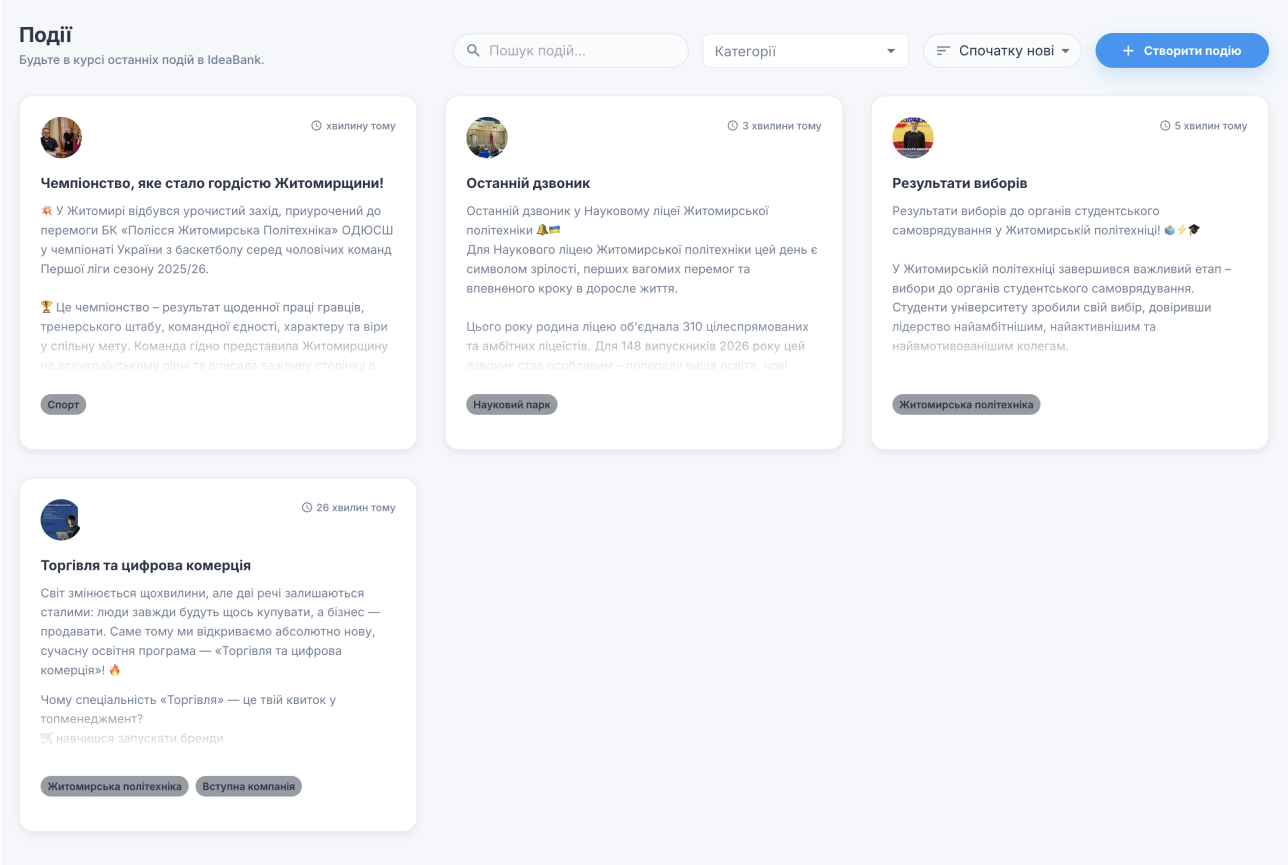


Рис. 3.35 Сторінка подій на сайті

Також приділяється увага адаптивності. На екранах мобільних пристроїв фільтри приховуються, щоб звільнити місце для карток подій. Замість фільтрів знизу посередині екрана з'являється плаваюча кнопка дії з текстом «Фільтри». Натискання на кнопку відкриває нижній висувний контейнер , де розміщено повний набір фільтрів (пошук, вибір категорій, сортування за назвою або

датою) та велика кнопка «Показати результати» (див. рис. 3.36). Це забезпечує зручність взаємодії однією рукою на смартфонах.

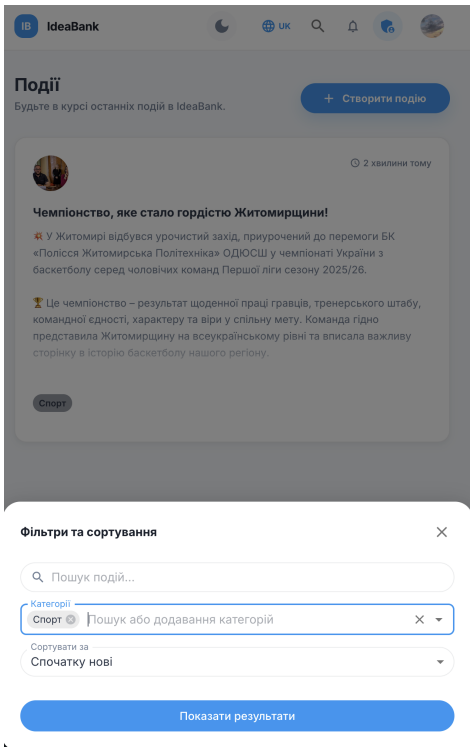


Рис. 3.36 Адаптивність для малих екранів

Перейдемо до детальної сторінки події. У верхній частині сторінки розміщено великий банер події. Нижче йде блок інформації: дата та час проведення, опис події. Якщо поточний користувач є організатором події або адміністратором платформи, йому відображаються кнопки «Редагувати» та «Видалити» (див. рис. 3.37).

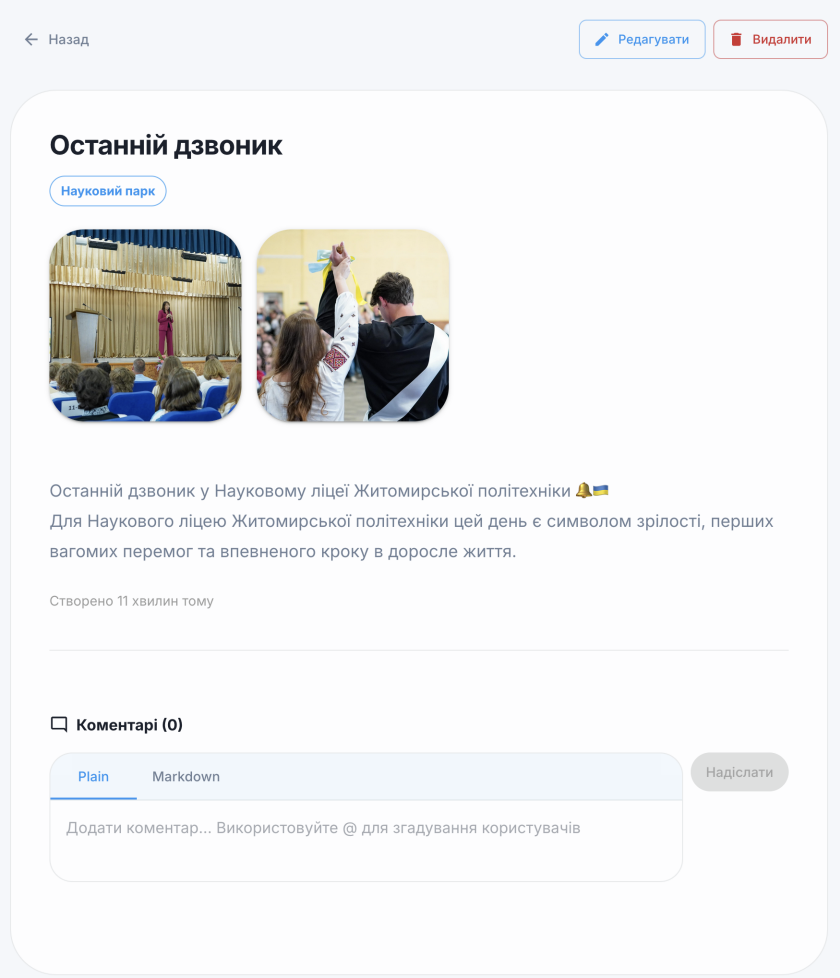


Рис. 3.37 Вигляд сторінки деталей події

Перейдемо до модуля комунікації у реальному часу, а саме — чатів. Їх інтерфейс розділено на три функціональні зони: бокову панель діалогів, панель повідомлень та шапку чату.

Бокова панель діалогів відображає стрічку діалогів користувача (див. рис. 3.38). Кожен рядок діалогу містить аватар співрозмовника (або іконку групи для групових чатів), назву, текст останнього повідомлення, дату останньої активності та синій круглий бейдж із кількістю непрочитаних повідомлень. Зверху розміщено випадаючий список вибору ідеї. Користувач може відфільтрувати чати, показуючи лише розмови, що стосуються проєкту «А», або переглянути всі чати разом.

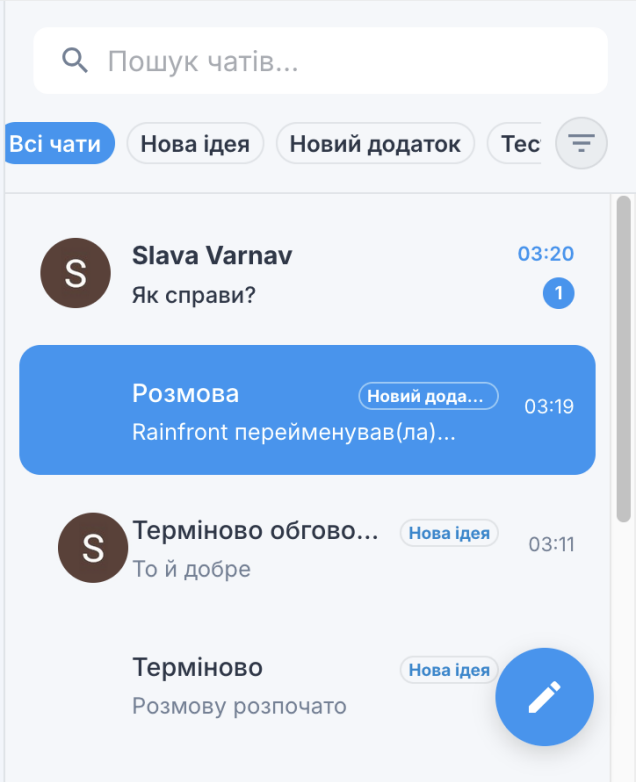


Рис. 3.38 Бокова панель діалогів

Також є поле пошуку для фільтрації списку чатів за ім'ям співрозмовника чи назвою групи. Кнопка «Новий чат» відкриває модальне вікно, де користувач може обрати тип чату (приватний діалог або групова кімната), ввести назву групи, обрати пов'язану ідею та вибрати учасників зі списку друзів або колег (див. рис. 3.39).

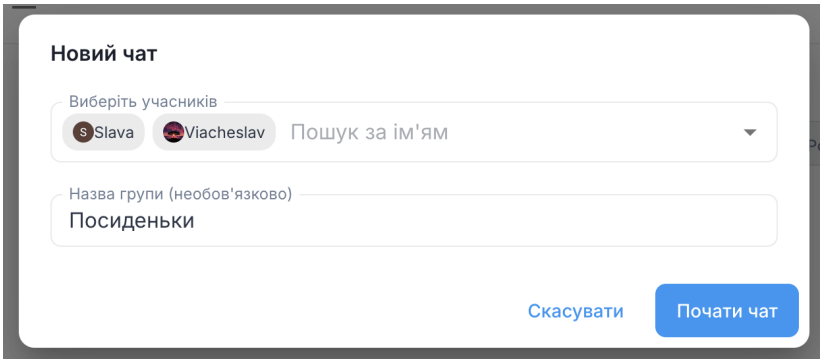


Рис. 3.39 Модальне вікно створення нового чату

Панель повідомлень відображає історію повідомлень. Повідомлення поточного користувача вирівнюються за правим краєм і мають світло-зелений

фон, повідомлення інших учасників — за лівим краєм із білим фоном (див. рис. 3.40).

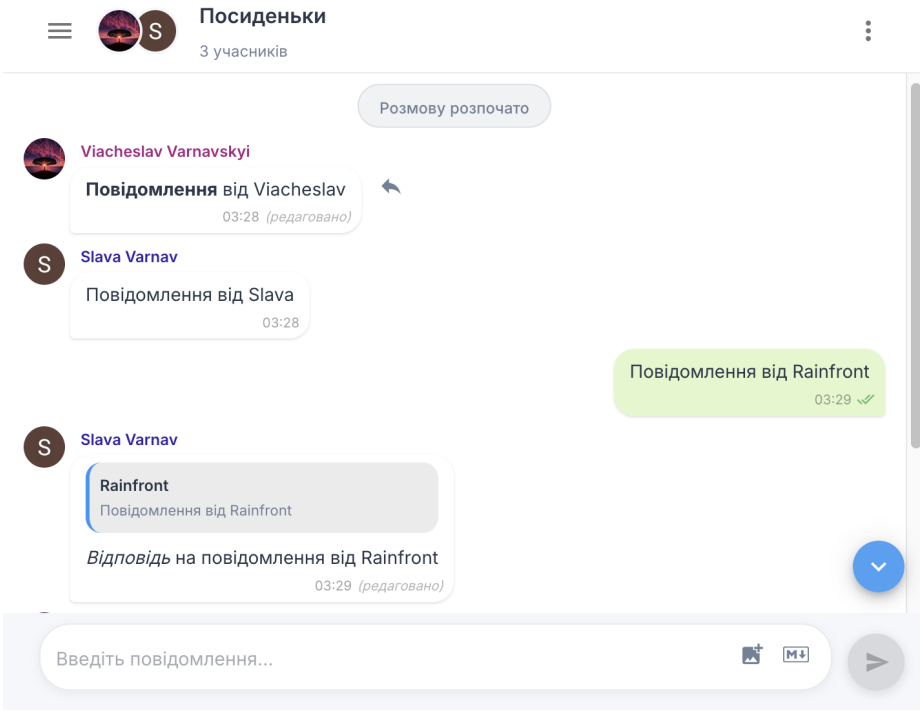


Рис. 3.40 Панель повідомлень

Повідомлення групуються за датами; нові повідомлення плавно додаються знизу. Індикатор друку реалізований через події WebSockets. Коли користувач вводить текст у поле введення, на сервер відправляється подія про те що користувач почав друкувати текст. Усі інші учасники чату бачать у шапці та панелі діалогів напис «[Ім'я] друкує...». Після зупинки введення на 2 секунди відправляється подія `typing_stop`, і напис зникає (див. рис. 3.41).

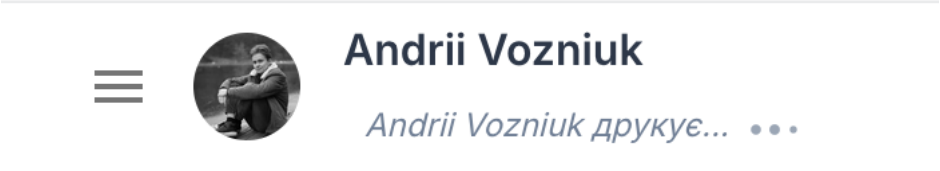


Рис. 3.41 Індикатор друкування користувача

Також при отриманні нового повідомлення стрічка автоматично прокручується до самого низу. Проте, якщо користувач прокрутив історію вгору для читання старих повідомлень, автоскрол блокується, щоб не збивати

фокус читання, а внизу праворуч з'являється кнопка «Вниз». Якщо повідомлення було написано використовуючи режим markdown, тоді тексти повідомлення перед виведенням на екран обробляється плагіном `react-markdown`, що дозволяє форматувати текст (використовувати списки, посилання, жирний шрифт, цитати). Для запобігання XSS-атакам весь згенерований HTML-код очищується бібліотекою `DOMPurify`.

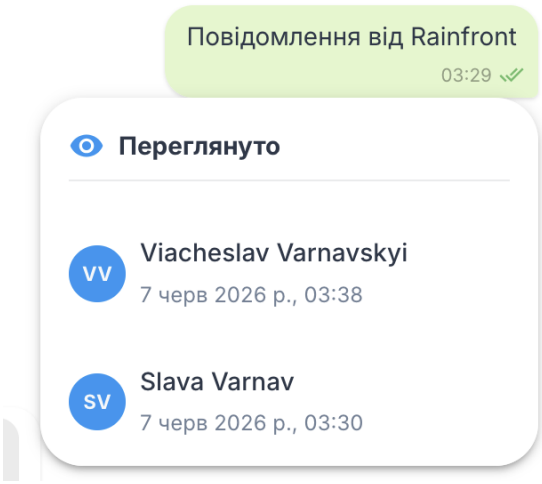


Рис. 3.42 Інформація про повідомлення

Шапка чату відображає деталі розмови та кнопки дій: запросити учасника, що відкриває модальне вікно запрошення учасників, де організатор чату може знайти будь-якого зареєстрованого користувача системи за його ім’ям чи прізвищем та додати до поточної розмови; переглянути список учасників, відкриваючи модальне вікно зі списком усіх підключених до чату користувачів, їхніми ролями та кнопками видалення учасників для адміністратора групи (див. рис. 3.43).

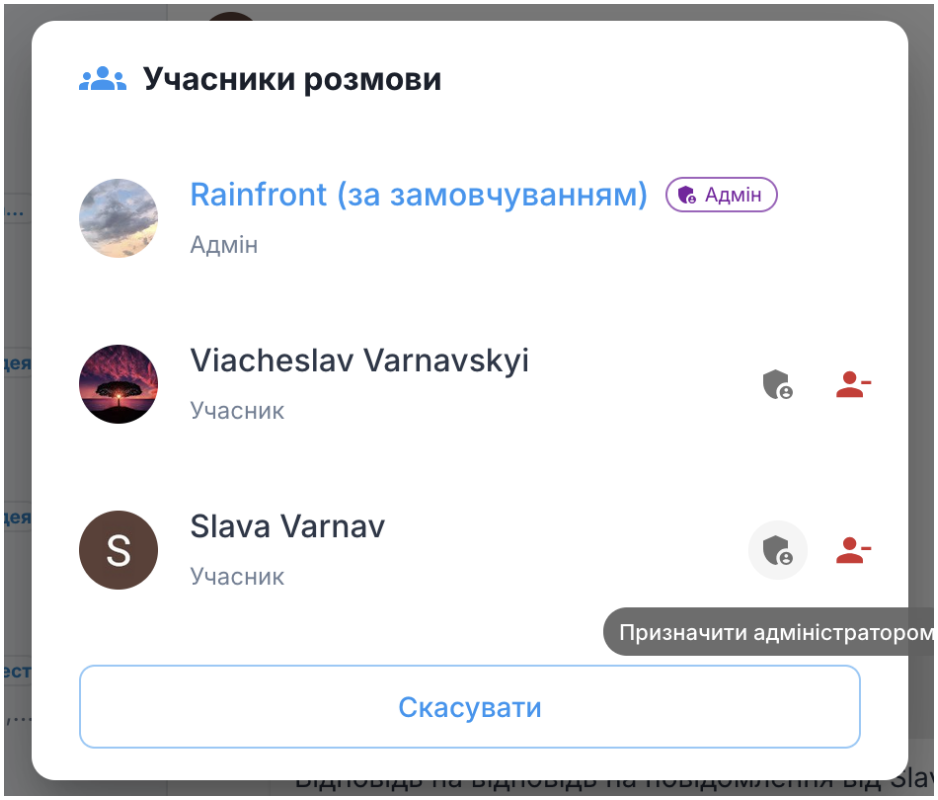


Рис. 3.43 Список учасників чату

Та користувач має можливість відразу переглянути усі медіа та файли у чаті, прив’язати чат до завдання, тощо (див. рис. 3.44).

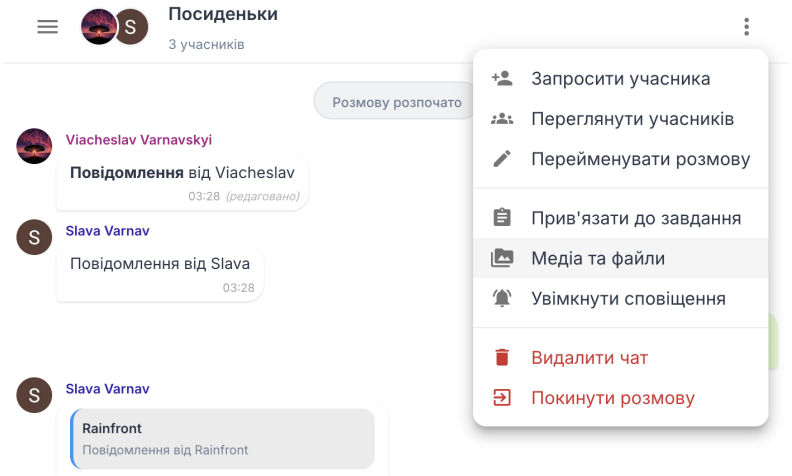


Рис. 3.44 Перелік доступних функцій для чату

3.2 Тестування роботи програмного комплексу

Для підтвердження коректності взаємодії UX-елементів розроблено наскрізний тестовий приклад спільної роботи команди над проєктом.

Користувач В’ячеслав реєструється в системі за допомогою Google SSO, заповнює свій профіль, вказує навички та інтереси (категорії) (див. рис. 3.45).

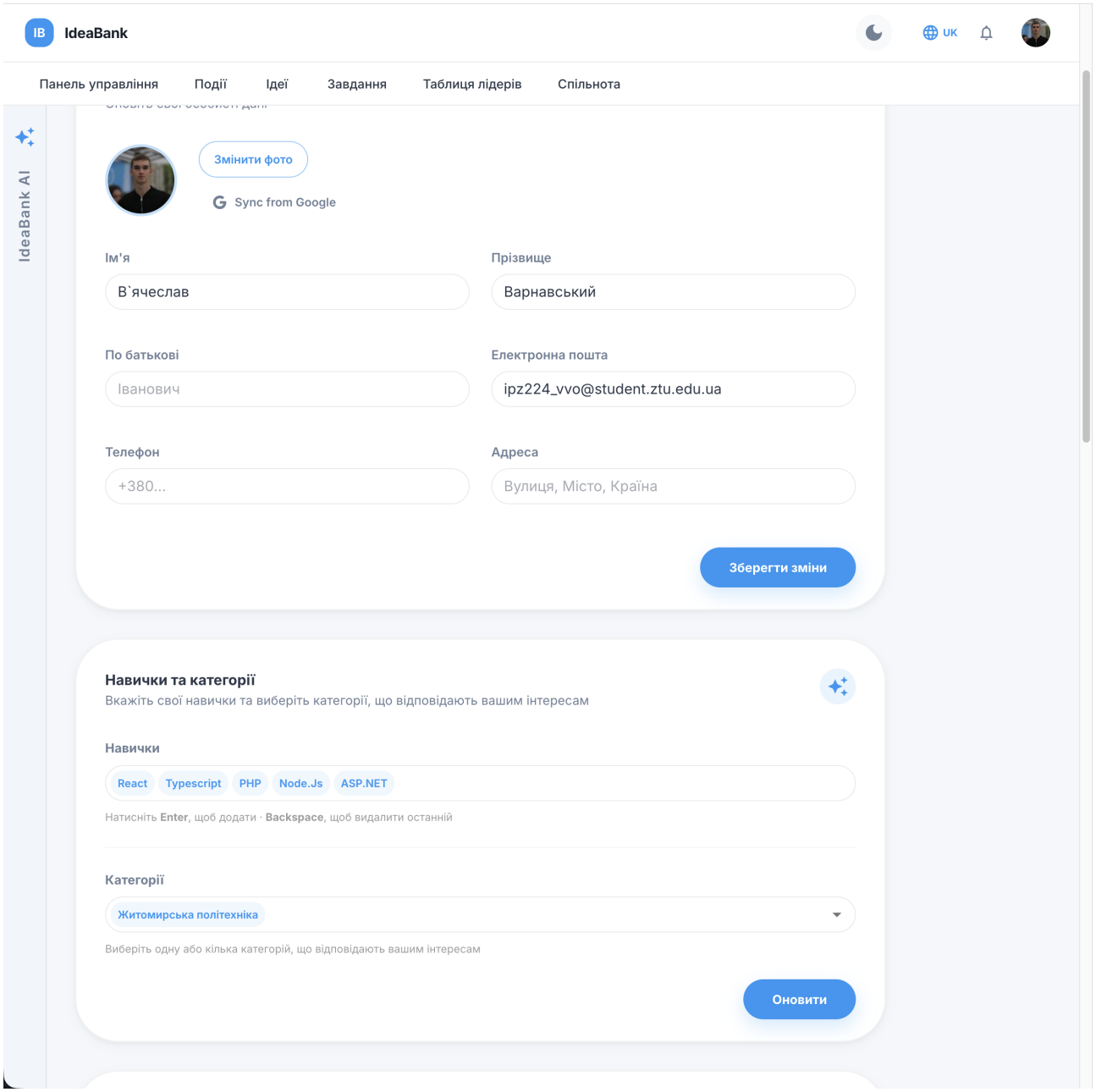


Рис. 3.45 Налаштування зареєстрованого користувача

В’ячеслав створює нову ідею «Розробка платформи автоматичного розгортання мікросервісів». В описі вказує мету та залучає кафедру комп'ютерних систем та мереж. Інший користувач, Марія (розробниця), знаходить цю ідею в загальному каталозі, натискає кнопку «Подати запит на приєднання» та описує свою мотивацію. В’ячеслав переглядає «Заявки на

приєднання» в деталях ідеї та натискає «Прийняти». Марія стає учасником команди (див. рис. 3.46).

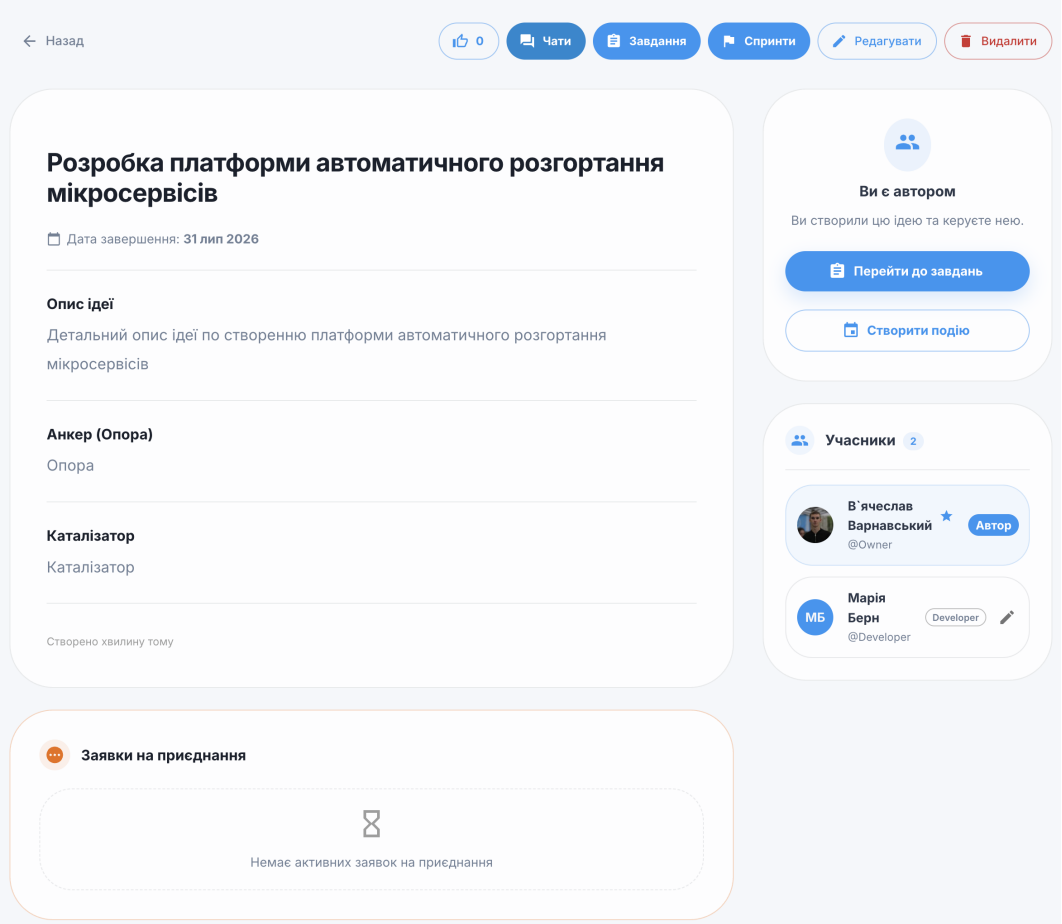


Рис. 3.46 Створена ідея та 2 учасника у ній

В’ячеслав відкриває розділ «Спринти» цієї ідеї та натискає «Новий спринт». У модальному вікні вводить назву `Спринт 1: Проєктування архітектури та CI/CD` та вказує термін — 1 тиждень. Спринт створюється у статусі `planned` (заплановано). В’ячеслав переходить у вкладку `Беклог` та створює три завдання (див. рис. 3.47):

1. Розробка схеми бази даних (складність: 5 Story Points, тип: Задача, пріоритет: Високий).

2. Налаштування Docker-контейнерів та CI/CD пайплайну (складність: 3 Story Points, виконавець: Марія).

3. Створення макетів інтерфейсу (складність: 2 Story Points).

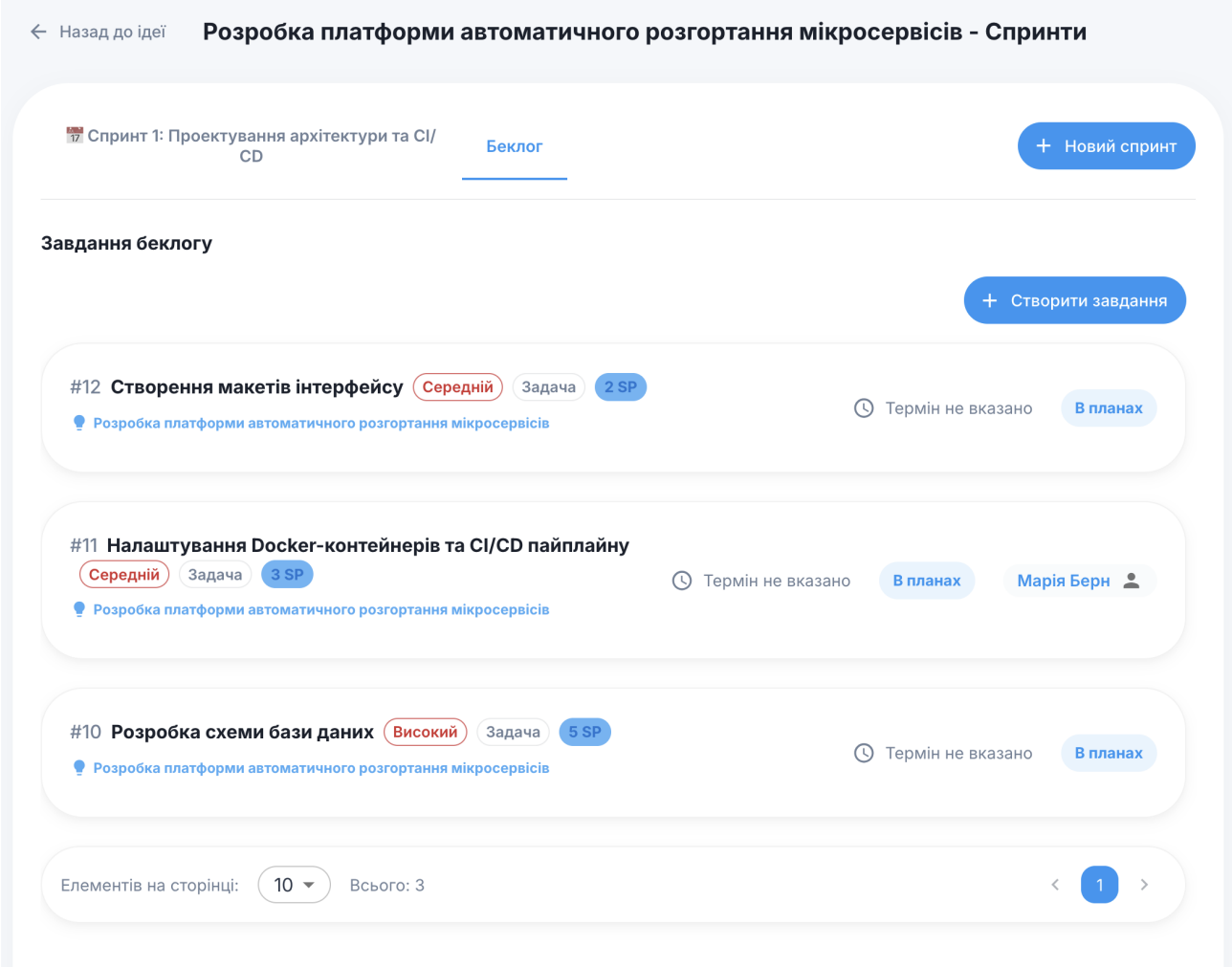


Рис. 3.47 Створені задачі у беклозі

В’ячеслав відкриває створений спринт, додає до нього перші два завдання з беклогу та натискає кнопку «Запустити спринт». Статус спринту змінюється на активний (🚀) (див. рис. 3.48).

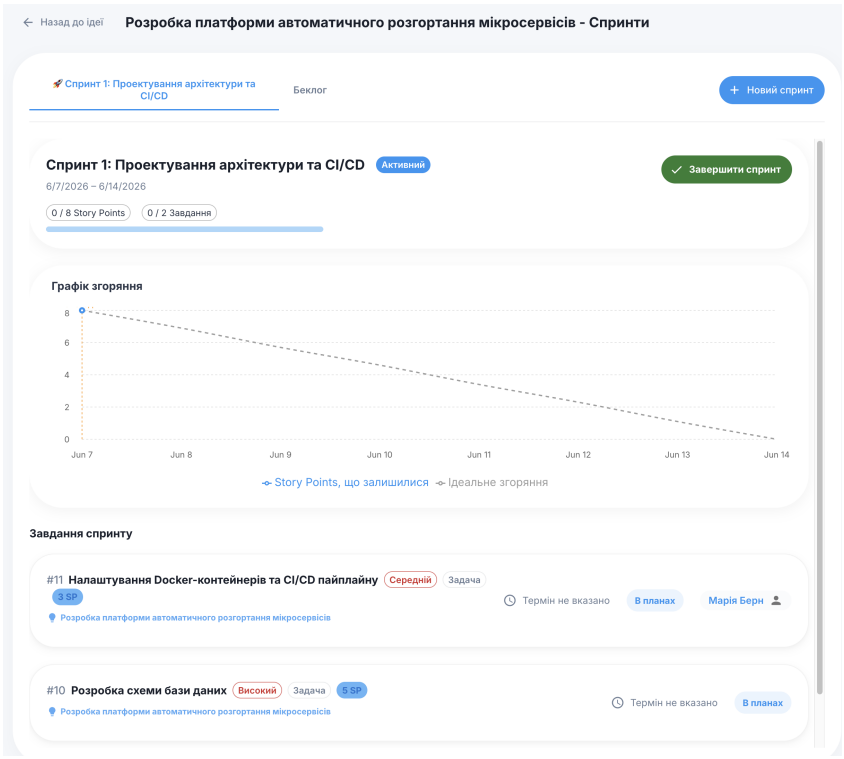


Рис. 3.48 Активний спринт

Марія відкриває Kanban-дошку, фільтрує її лише до першого спринта. Вона бачить призначене їй завдання у колонці «В планах» та перетягує картку завдання в колонку «В процесі» (див. рис. 3.49).

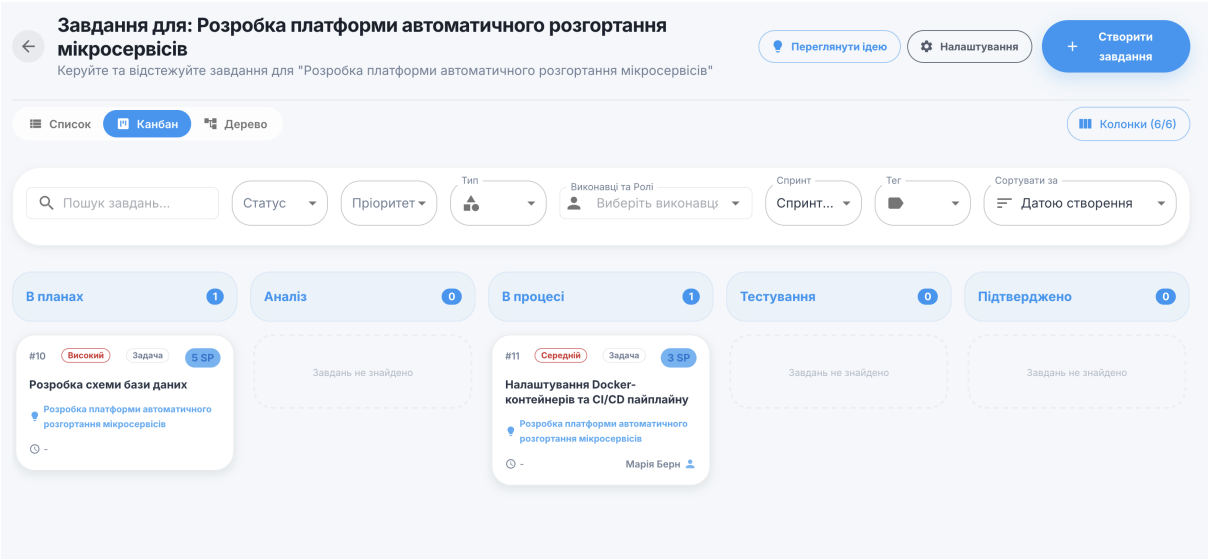


Рис. 3.49 Відфільтровані задачі та змінений статус задачі, що закріплена за Марією

Виконавши завдання, Марія відкриває детальну панель завдання, прикріплює звіт у форматі PDF, пише коментар «@В’ячеслав завдання виконано, файли прикріпила» та переміщує картку в колонку «Виконано» (див. рис. 3.50, 3.51).

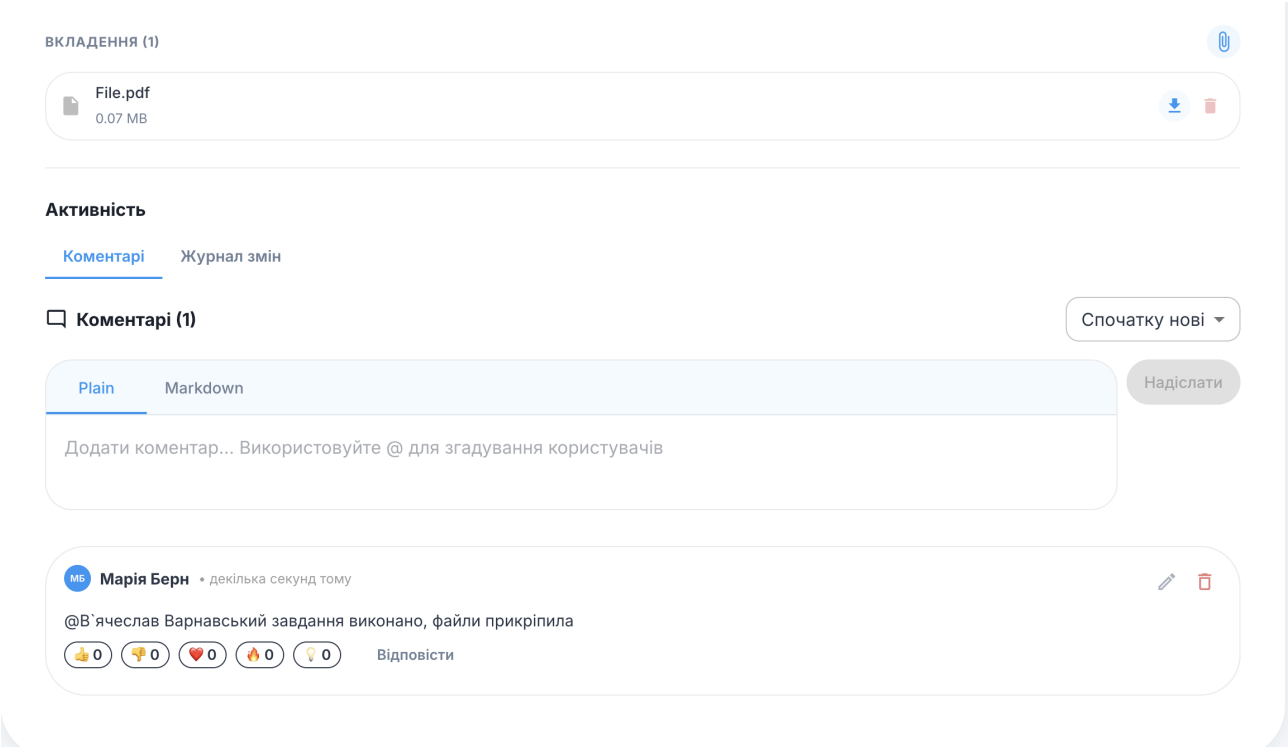


Рис. 3.50 Прикріплений файл та коментар зі згадкою В’ячеслава

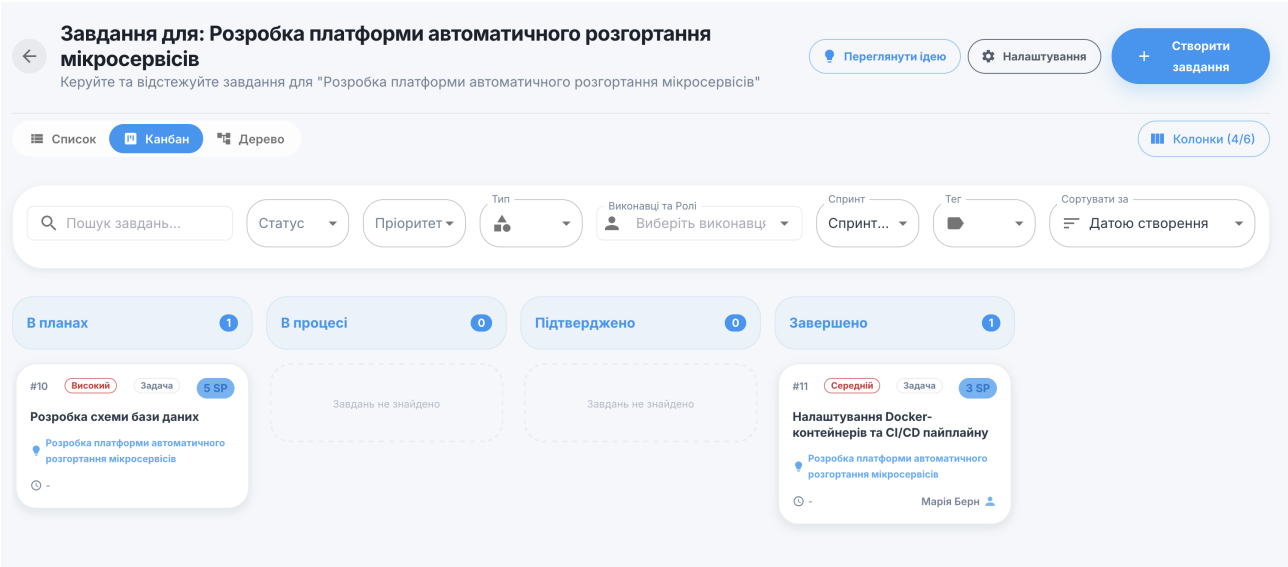


Рис. 3.51 Завдання переміщено у колонку «Завершено»

В’ячеслав отримує сповіщення про згадування його імені у коментарі (див. рис. 3.52).

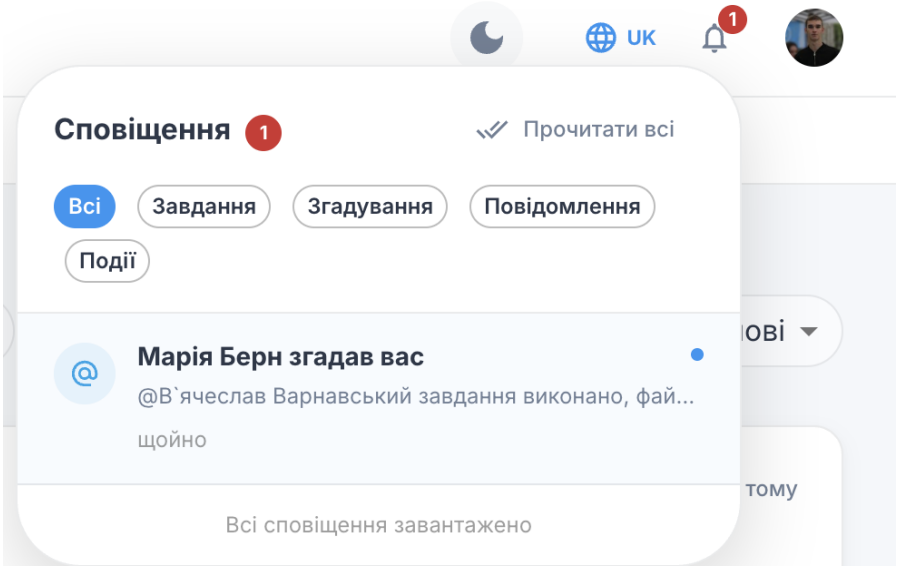


Рис. 3.52 Отримане сповіщення щодо згадки в задачі

Він клікає на сповіщення та відразу потрапляє на сторінку деталі завдання (див. рис. 3.53).

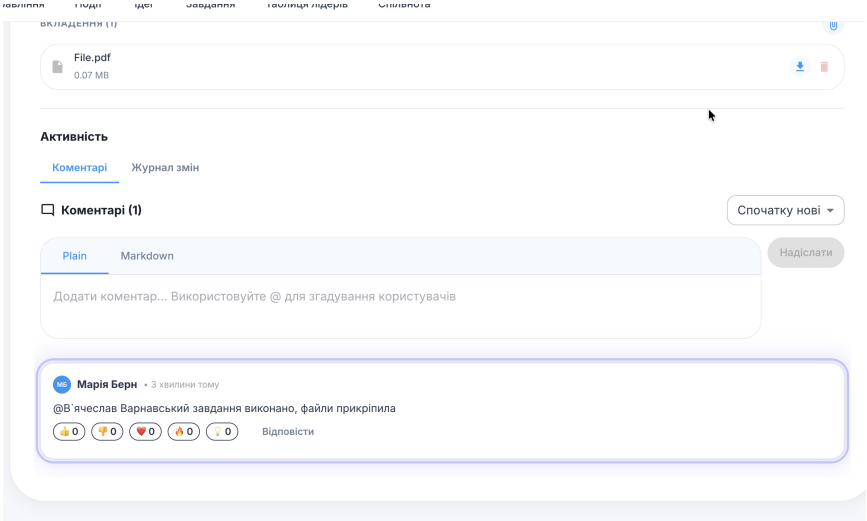


Рис. 3.53 Прикріплений файл та коментар зі згадкою для В’ячеслава

Для швидкого обговорення деталей В’ячеслав створює новий груповий чат для команди проєкту під назвою «Автодеплой мікросервісів» та додає туди Марію. Вони обговорюють структуру даних у реальному часі. Система

відображає повідомлення та статус «Марія друкує...» під час написання нею відповіді (див. рис. 3.54).

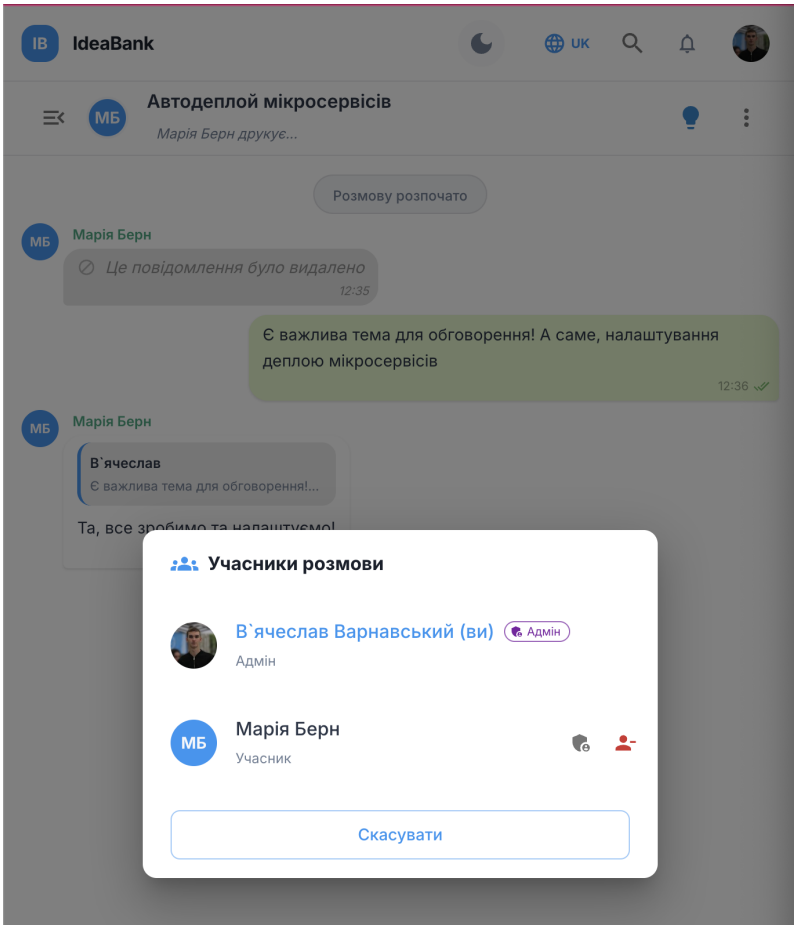


Рис. 3.54 Обговорення у створеному чаті, повідомлення про друк користувача та список учасників

Щоб проінформувати учасників ідеї про те що терміни виконання роботи продовжено, В’ячеслав переходить на сторінку ідеї та створює про це подію (див. рис. 3.55).

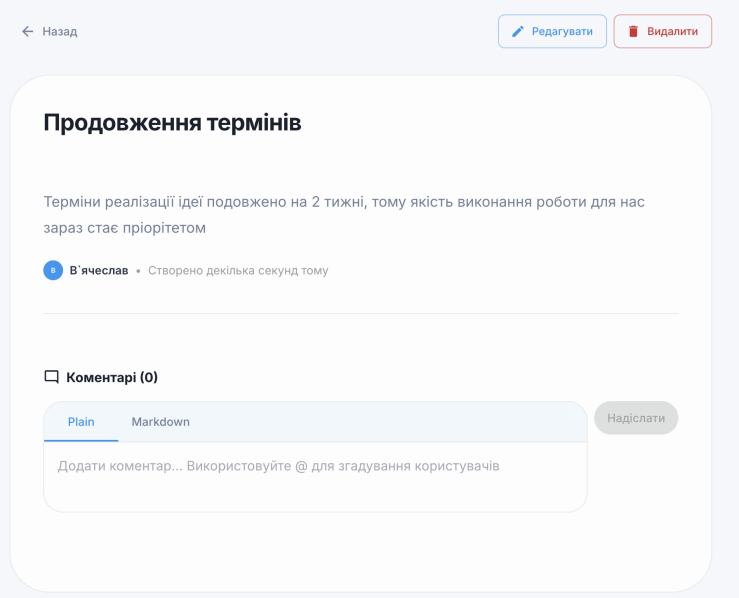


Рис. 3.55 Створена подія

Таким чином Марія отримає про це повідомлення, та перейде за ним (див. рис. 3.56).

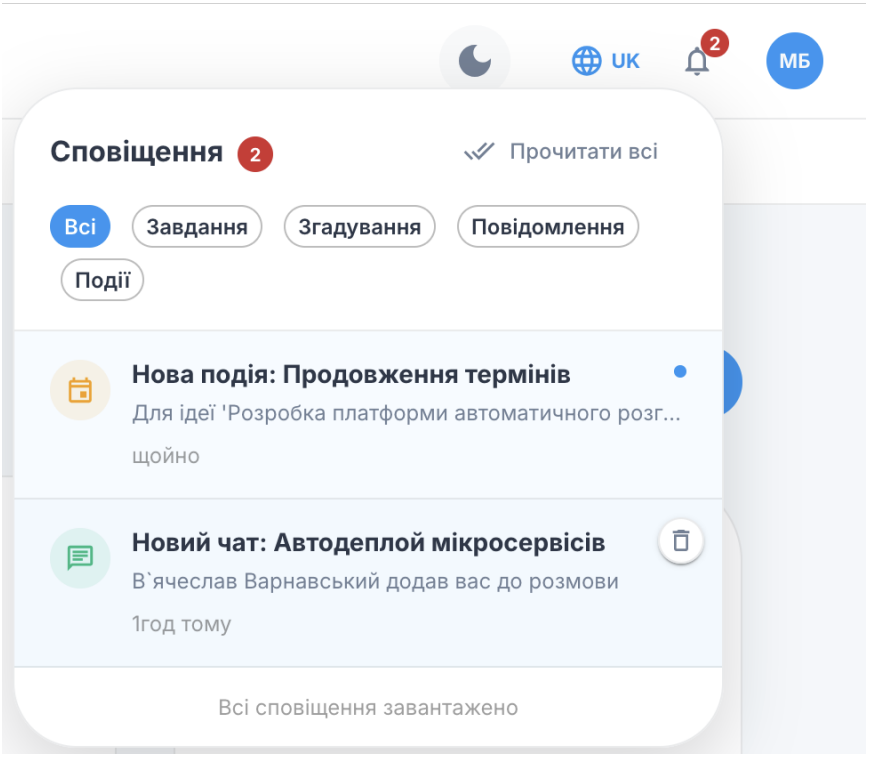


Рис. 3.56 Повідомлення до Марії про створення нової події для ідеї

В’ячеслав переглядає графік згоряння спринту. Він бачить, що фактична лінія згоряння сторі поінтів йде згідно з графіком. Під час спринту В’ячеслав натискає кнопку «Завершити спринт» (див. рис. 3.57).

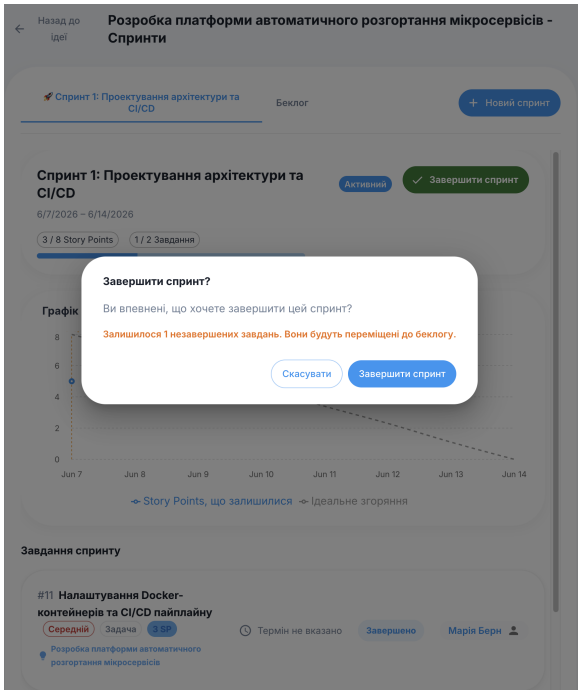


Рис. 3.57 Завершення спринту

Оскільки третє завдання не було перенесене в спринт і залишилося невиконаним, діалогове вікно пропонує перенести його в наступний спринт або повернути в беклог. В’ячеслав обирає беклог та підтверджує завершення (див. рис. 3.58).

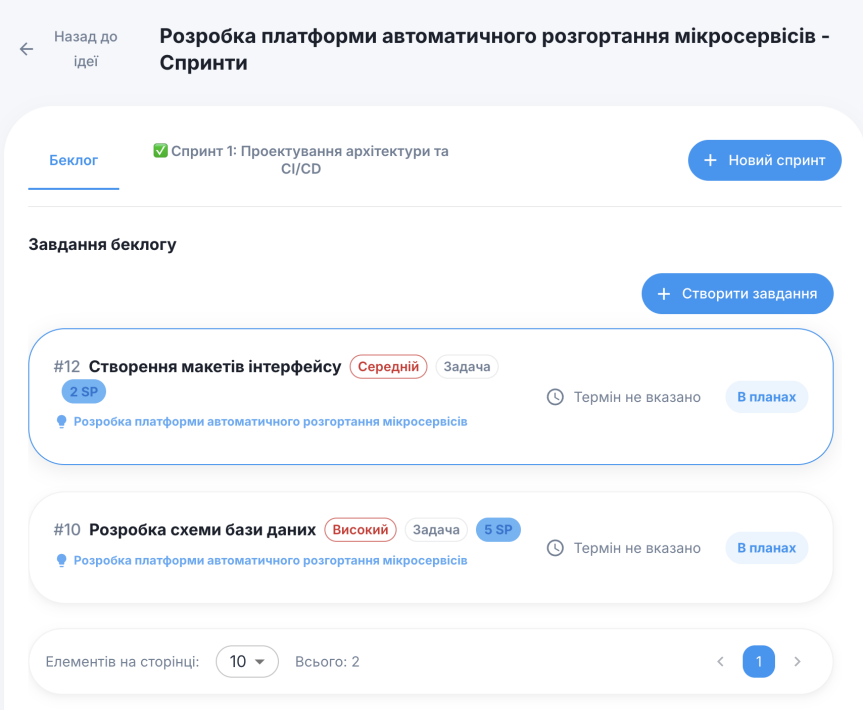


Рис. 3.58 Оновлений беклог

Спринт переходить в архів (вкладка зі значком ✅) (див. рис. 3.59).

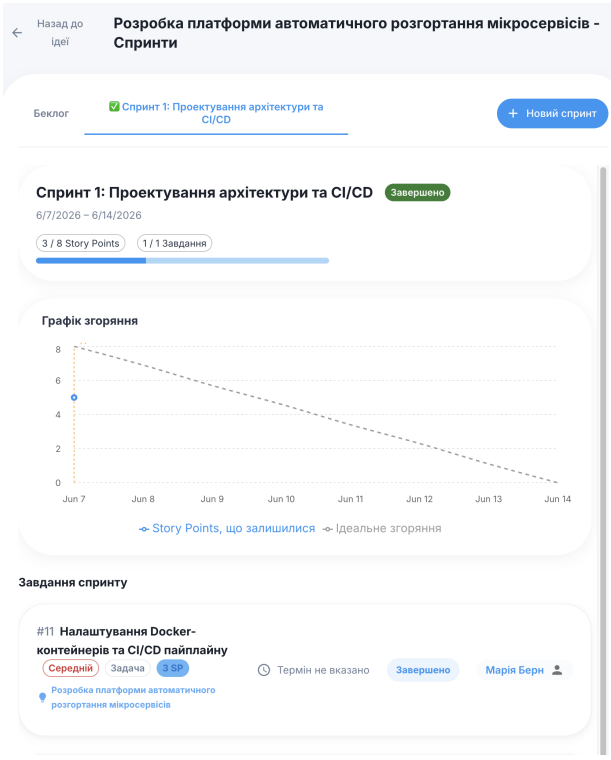


Рис. 3.59 Завершений спринт

3.3 Порядок налаштування та встановлення системи

Для успішного запуску системи на локальному комп'ютері розробника або сервері мають бути встановлені такі програмні засоби:

1. **Git** — система контролю версій (для отримання коду).
2. **Docker Engine** (версії 20.10 або новішої) та **Docker Compose** (версії 2.0 або новішої).
3. Текстовий редактор або IDE (наприклад, PhpStorm, VS Code) для редагування конфігураційних файлів.
4. Доступ до мережі Інтернет для завантаження базових образів контейнерів та бібліотек-залежностей.

Початковим кроком є завантаження вихідного коду проєкту з віддаленого репозиторію на локальну машину. Для цього у терміналі виконується команда:

git clone https://github.com/andrewvoznuyk/IB.git

Система має модульну архітектуру та складається з декількох сервісів. Налаштування параметрів безпеки, доступу до баз даних та зовнішніх API здійснюється за допомогою файлів конфігурації оточення .env. Для цього Необхідно створити два основні .env файли на основі прикладів .env.example, що знаходяться у кореневій директорії та директорії «api» [29].

Для автоматизації процесу ініціалізації середовища використовується скрипт **recreate.sh**. Для його запуску виконайте команду:

bash ./recreate.sh

Після запуску контейнерів бази даних ще не містять необхідної структури таблиць. Для автоматичного налаштування використовується скрипт **migrate.sh**. Для його запуску виконайте команду:

bash ./migrate.sh

Для забезпечення безпечної авторизації користувачів за технологією JSON Web Token (JWT) API-сервіс використовує асиметричне шифрування (публічний та приватний OpenSSL ключі). Генерація пари ключів виконується всередині контейнера api за допомогою консольної команди Symfony:

docker compose exec api php bin/console lexik:jwt:generate-keypair

Після успішного завершення всіх етапів встановлення система стає доступною за адресами наведеними в таблиці 1.2.

Таблиця 1.2

Локальні адреси сервісів та їх опис

Сервіс / Інтерфейс	**Локальна URL-адреса**	**Опис**
Основний вебсайт (Frontend/API)	https://localhost	Клієнтський інтерфейс та точка входу
phpMyAdmin	http://localhost:8081	Веб-інтерфейс керування базою даних MySQL
Chat Service (NestJS)	http://localhost:3100	Сервер обміну миттєвими повідомленнями
Mercure Hub	http://localhost:3000	Трансляція real-time подій

Після реєстрації на сайті користувач може надати собі права адміністратора. Для цього потрібно пройти аутентифікацію у phpMyAdmin з логіном та паролем, що вказані у .env файлі. Далі перейти до бази даних «project» та знайти у ній таблицю «user» (див. рис. 3.60).

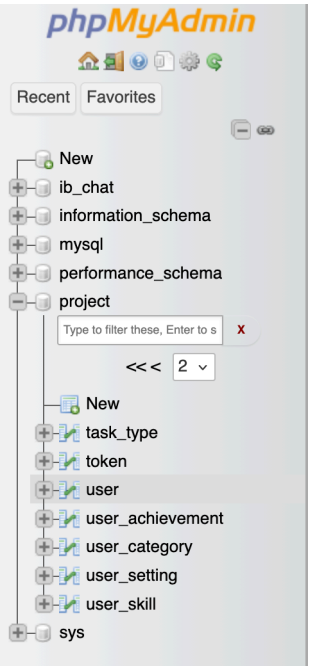


Рис. 3.60 Перелік баз даних, та таблиць всередині «project»

Далі потрібно знайти запис про потрібного користувача, та список його ролей. Після додавання значення «ROLE_ADMIN» до списку ролей користувач отримує права адміністратора. Для застосування змін необхідно вийти та зайти у акаунт знову, щоб отримати з серверу оновлений токен, що підтверджує наявність додаткової ролі.

Після встановлення значення пріоритетів, статусів та типів задач відсутні (див. рис. 3.61).

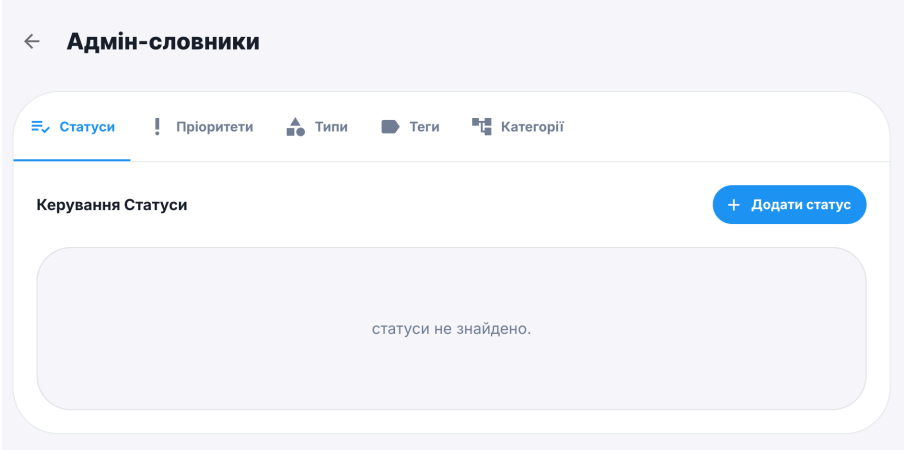


Рис. 3.61 Пусте меню зі статусами задач

Після заповнення необхідних даних також можна вказати глобальне налаштування для чатів, а саме чи відображати показник про видалене повідомлення (див. рис. 3.62).

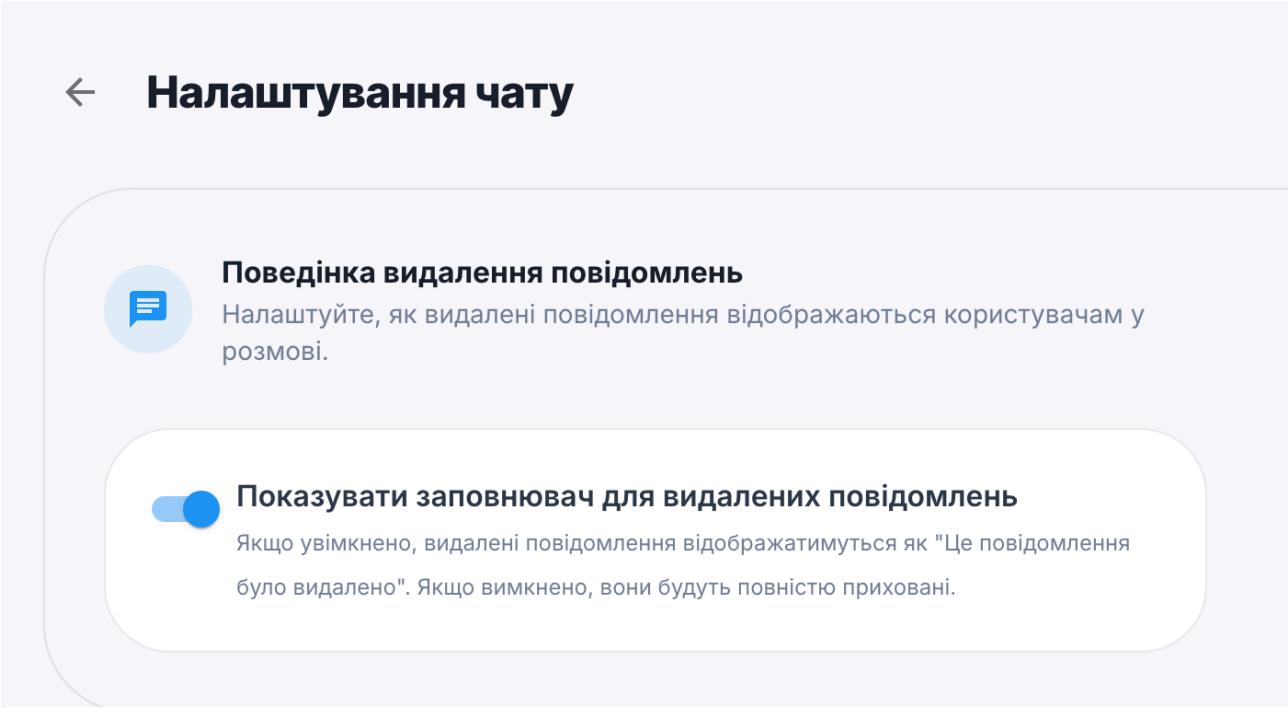


Рис. 3.62 Налаштування відображення видалених повідомлень

Ця інструкція повністю покриває життєвий цикл первинного розгортання та гарантує відтворюваність роботи системи на будь-якому сумісному комп'ютері.

Висновки до третього розділу

У третьому розділі було детально розглянуто практичну реалізацію розробленої вебсистеми управління завданнями та командної взаємодії, включаючи опис користувацького інтерфейсу, наскрізне тестування функціоналу та порядок розгортання програмного комплексу.

Продемонстровано, що користувацький інтерфейс системи, побудований на основі принципів Material Design, забезпечує інтуїтивно зрозумілу та зручну взаємодію з користувачем. Завдяки реалізації різноманітних варіантів

відображення завдань, таких як Канбан-дошка з підтримкою drag-and-drop, ієрархічне дерево та класичний список, система гнучко адаптується під потреби різних ІТ-команд. Інтеграція спеціалізованих модулів, таких як система багатоканальних сповіщень, багаторівневі контекстні коментарі зі згадками та чати в режимі реального часу, дозволяє вирішити головну проблему сучасних ІТ-проєктів — втрату робочого контексту при перемиканні між додатками. Впроваджений механізм управління спринтами та графіки згоряння (Burndown charts) додатково підвищують прозорість та ефективність планування ітерацій.

Для перевірки коректності роботи реалізованих алгоритмів було проведено комплексне наскрізне тестування на основі реального сценарію командної взаємодії. Був продемонстрований тестовий сценарій, що охоплював повний життєвий цикл роботи над проєктом — від реєстрації користувачів, створення ідеї, формування беклогу та спринтів до паралельної комунікації та оновлення статусів завдань — підтвердив високу надійність системи. Усі компоненти, включаючи трансляцію подій через Server-Send Events (SSE) та миттєвий обмін повідомленнями через WebSockets, працюють синхронно та без збоїв.

Крім того, у розділі наведено вичерпну інструкцію з налаштування та розгортання системи. Завдяки використанню технологій контейнеризації Docker та Docker Compose, процес ініціалізації середовища, міграції бази даних та генерації ключів для JWT-авторизації максимально автоматизовано. Це гарантує відтворюваність та ідентичність роботи програмного комплексу в середовищах розробки та продуктової експлуатації, нівелюючи проблему сумісності програмного та апаратного забезпечення.

Таким чином, результати третього розділу підтверджують, що розроблена система є повністю функціональним, зручним та відмовостійким інструментом, що досягає поставленої мети — створення єдиного цифрового простору для ефективного управління ІТ-проєктами малих та середніх розмірів.

ВИСНОВКИ

У кваліфікаційній роботі успішно вирішено завдання подолання фрагментації контексту та розпорошення інформації під час управління IT-проєктами шляхом створення інтегрованої вебсистеми.

Основним результатом є розробка єдиного цифрового простору, який органічно поєднує процеси відстеження завдань із засобами комунікації, що дозволяє командам оперативно приймати рішення без втрати важливих даних під час перемикання між додатками.

Під час роботи реалізовано гібридну сервісно-орієнтовану архітектуру, що розділяє стабільне середовище бізнес-логіки на базі Symfony та MySQL і реактивне середовище для миттєвої взаємодії на базі NestJS, гарантуючи високу продуктивність системи. Створений модуль управління підтримує гнучкі методології через планування беклогу, ведення спринтів та автоматичну генерацію графіків згоряння, пропонуючи зручну візуалізацію у вигляді списку, Канбан-дошки або дерева залежностей. Комунікація забезпечується через чати в режимі реального часу та багаторівневі контекстні коментарі з підтримкою розмітки Markdown і згадувань учасників, тоді як автоматизований журнал змін надійно фіксує всі дії для забезпечення прозорості робочого процесу.

Практична цінність результатів полягає у створенні готового відмовостійкого програмного продукту, який адаптовано під специфічні потреби невеликих і середніх IT-команд та який ефективно усуває розрив між плануванням і спілкуванням. Завдяки використанню технологій контейнеризації Docker та наявності автоматизованих скриптів розгортання, комплекс легко та надійно інтегрується в будь-яке інфраструктурне середовище без проблем із сумісністю.

Для подальшого вдосконалення системи пропонується впровадити алгоритми штучного інтелекту на базі розробленого сервісу аналізу для транскрибації тексту, автоматизації пріоритетів і генерації підсумків

обговорень у чатах. Крім того, розширення аналітичного модуля новими метриками та створення нативного мобільного застосунку додатково підвищать загальну ефективність управління ресурсами та зручність користування платформою.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Building microservices: designing fine-grained systems. O'Reilly Media, Incorporated, 2020. 250 p.
2. Designing Data-Intensive Web Applications / M. Brambilla et al. Elsevier Science & Technology Books, 2003.
3. Implementing azure devops solutions: learn about azure devops services to successfully apply devops strategies. de Gruyter GmbH, Walter, 2020.
4. Iqbal M. 18th state of agile report. *https://www.scrum.org/resources/blog/18th-state-agile-report*. URL: https://www.scrum.org/resources/blog/18th-state-agile-report.
5. Jira | project management for the AI era | atlassian. *Collaboration software for software, IT and business teams | Atlassian*. URL: https://www.atlassian.com/software/jira.
6. Kleppmann M. Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems. O'Reilly Media, Incorporated, 2017. 616 p.
7. Knapp B. W. Understanding the PMBOK guide - the PMBOK guide in plain english. Sturgeon Publishing, 2005. 54 p.
8. Learning react: modern patterns for developing react apps. O'Reilly Media, 2020. 310 p.
9. Mark G., Gudith D., Klocke U. The cost of interrupted work: more speed and stress. *Proceedings of the SIGCHI conference on human factors in computing systems*. New York, NY, USA, 2008. P. 107–110. URL: https://doi.org/10.1145/1357054.1357072.
10. Microsoft teams: the ai-powered platform for work. *Microsoft Teams*. URL: https://www.microsoft.com/en-us/microsoft-teams/group-chat-software.

11. Fowler M. Patterns of enterprise application architecture. Addison-Wesley Professional, 2002. 560 p.
12. Potencier F. Symfony 5: the fast track. Symfony SAS, 2019. 324 p.
13. Programming typescript: making your javascript applications scale. O'Reilly Media, 2019. 322 p.
14. Quiroz-Vázquez C., Goodwin M. What is application lifecycle management (ALM)? | IBM. *IBM*. URL: https://www.ibm.com/think/topics/application-lifecycle-management.
15. Richards M., Ford N. Fundamentals of software architecture: an engineering approach. O'Reilly Media, 2020. 432 p.
16. Richardson C. Microservices patterns: with examples in java. Manning Publications Co. LLC, 2018.
17. Schwaber K., Sutherland J. The scrum guide the definitive guide to scrum: the rules of the game. *Home | Scrum Guides*. URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf.
18. The work life of developers: activities, switches and perceived productivity / A. N. Meyer et al. *IEEE transactions on software engineering*. 2017. Vol. 43, no. 12. P. 1178–1193. URL: https://doi.org/10.1109/TSE.2017.2656886.
19. Trello: capture, organize, and tackle your to-dos from anywhere. *Capture, organize, and tackle your to-dos from anywhere*. URL: https://trello.com/home.
20. Using objectives and key results (okrs) and slack: a case study of coordination in large-scale distributed agile / V. Stray et al. *Hawaii international conference on system sciences*. 2022. URL: https://doi.org/10.24251/hicss.2022.883.
21. Casciaro M. Node. js design patterns. Packt Publishing, Limited, 2014.
22. WebSocket: lightweight client-server communications. O'Reilly Media, Incorporated, 2015. 144 p.

23. Matthias K., Kane S. P. Docker: up & running: shipping reliable containers in production. O'Reilly Media, 2018. 352 p.
24. NGINX cookbook: advanced recipes for high performance load balancing. O'Reilly Media, Incorporated, 2020. 145 p.
25. Martin R. Clean code: a handbook of agile software craftsmanship. Pearson Education, Limited.
26. Vernon V. Implementing Domain-Driven Design. Pearson Education, Limited.
27. Siriwardena P. Advanced API Security: Securing APIs with OAuth 2. 0, OpenID Connect, JWS, and JWE. Apress L. P., 2014.
28. Grigorik I. High performance browser networking: what every web developer should know about networking and web performance. O'Reilly Media, Incorporated, 2013. 400 p.
29. Wiggins A. The Twelve-Factor App. *The Twelve-Factor App*. URL: https://12factor.net/config.
30. Sasu T. Principles and patterns. *Publish/Subscribe systems*. Chichester, UK, 2012. P. 79–104. URL: https://doi.org/10.1002/9781118354261.ch4.

ДОДАТКИ

Додаток А

Лістинг коду TaskStatusProcessor.php:

```
<?php

namespace App\State;

use ApiPlatform\Metadata\Operation;
use ApiPlatform\Metadata\Post;
use ApiPlatform\State\ProcessorInterface;
use App\Entity\Task;
use App\Service\TaskChangeLogBuilder;
use App\Service\TaskNotificationService;
use Symfony\Component\DependencyInjection\Attribute\Autowire;

class TaskStatusProcessor implements ProcessorInterface
{
    public function __construct(
        #[Autowire(service:
'api_platform.doctrine.orm.state.persist_processor')]
        private ProcessorInterface $persistProcessor,
        private \Doctrine\ORM\EntityManagerInterface $entityManager,
        private \Symfony\Bundle\SecurityBundle\Security $security,
        private TaskNotificationService $notificationService,
        private TaskChangeLogBuilder $taskChangeLogBuilder,
    ) {
    }

    public function process(mixed $data, Operation $operation, array
$uriVariables = [], array $context = []): mixed
    {
        if ($data instanceof Task && $operation instanceof Post) {
            $idea = $data->getIdea();

            // 1. Try Project-specific defaults
            if ($idea) {
                if (null === $data->getIdeaTaskStatus() && null !== $idea-
>getDefaultTaskStatus()) {
                    $data->setIdeaTaskStatus($idea->getDefaultTaskStatus());
                }
                if (null === $data->getIdeaTaskPriority() && null !== $idea-
>getDefaultTaskPriority()) {
                    $data->setIdeaTaskPriority($idea->getDefaultTaskPriority());
                }
                if (null === $data->getIdeaTaskType() && null !== $idea-
>getDefaultTaskType()) {
                    $data->setIdeaTaskType($idea->getDefaultTaskType());
                }
            }
```

```
            // 2. Try Global defaults as fallback
            if (null === $data->getIdeaTaskStatus()) {
                $globalDefault              =              $this->entityManager-
>getRepository(\App\Entity\IdeaTaskStatus::class)
                    ->findOneBy(['idea' => null, 'isDefault' => true]);
                if ($globalDefault) {
                    $data->setIdeaTaskStatus($globalDefault);
                }
            }
            if (null === $data->getIdeaTaskPriority()) {
                $globalDefault              =              $this->entityManager-
>getRepository(\App\Entity\IdeaTaskPriority::class)
                    ->findOneBy(['idea' => null, 'isDefault' => true]);
                if ($globalDefault) {
                    $data->setIdeaTaskPriority($globalDefault);
                }
            }
            if (null === $data->getIdeaTaskType()) {
                $globalDefault              =              $this->entityManager-
>getRepository(\App\Entity\IdeaTaskType::class)
                    ->findOneBy(['idea' => null, 'isDefault' => true]);
                if ($globalDefault) {
                    $data->setIdeaTaskType($globalDefault);
                }
            }

            // 3. Set creator
            $creator = $this->security->getUser();
            if ($creator instanceof \App\Entity\User) {
                $data->setCreator($creator);
            }
        }

        $result     =     $this->persistProcessor->process($data,     $operation,
$uriVariables, $context);

        if ($data instanceof Task && ($operation instanceof Post || $operation
instanceof \ApiPlatform\Metadata\Patch)) {
            $this->notificationService->notifyAssignments($data);
        }

        if ($data instanceof Task && $operation instanceof Post) {
            $actor = $this->security->getUser();
            $entry = $this->taskChangeLogBuilder->buildInitialAssignmentEntry(
                task: $data,
                actor: $actor instanceof \App\Entity\User ? $actor : null,
            );

            if ($entry) {
                $this->entityManager->persist($entry);
                $this->entityManager->flush();
```

```
            }
        }

        return $result;
    }
}
```

Лістинг коду SprintBurndownSnapshotService.php:

```
<?php

namespace App\Service;

use App\Entity\Sprint;
use App\Entity\SprintBurndownSnapshot;
use App\Entity\Task;
use Doctrine\ORM\EntityManagerInterface;
use DateTime;

class SprintBurndownSnapshotService
{
    private EntityManagerInterface $entityManager;

    public function __construct(EntityManagerInterface $entityManager)
    {
        $this->entityManager = $entityManager;
    }

    public function snapshotActiveSprints(): void
    {
        $today = new DateTime('today');

        $activeSprints   =   $this->entityManager->getRepository(Sprint::class)-
>findBy([
            'status' => 'active'
        ]);

        foreach ($activeSprints as $sprint) {
            if ($sprint->getEndDate() < $today) {
                // Sprint has ended, or should we still capture if it wasn't
completed?
                // Depending on rules... Let's just process it if it's active.
            }

            $remainingPoints = 0;
            $completedPoints = 0;
            $totalTasks = 0;
            $completedTasks = 0;

            foreach ($sprint->getTasks() as $task) {
```

```
                $totalTasks++;
                $points = $task->getStoryPoints() ?? 0;

                // Simple check: if there is a completedAt date, it's done.
                if ($task->getCompletedAt() !== null) {
                    $completedTasks++;
                    $completedPoints += $points;
                } else {
                    $remainingPoints += $points;
                }
            }

            // Upsert mechanism:
            // Check if snapshot for today exists
            $snapshot                  =                  $this->entityManager-
>getRepository(SprintBurndownSnapshot::class)
                ->findOneBy([
                    'sprint' => $sprint,
                    'snapshotDate' => $today
                ]);

            if (!$snapshot) {
                $snapshot = new SprintBurndownSnapshot();
                $snapshot->setSprint($sprint);
                $snapshot->setSnapshotDate($today);
                $this->entityManager->persist($snapshot);
            }

            $snapshot->setRemainingPoints($remainingPoints);
            $snapshot->setCompletedPoints($completedPoints);
            $snapshot->setTotalTasks($totalTasks);
            $snapshot->setCompletedTasks($completedTasks);
        }

        $this->entityManager->flush();
    }
}
```

Лістинг коду TaskChangeLogListener.php:

```
<?php

namespace App\EventListener;

use App\Entity\Event;
use App\Entity\RelatedTask;
use App\Entity\Task;
use App\Entity\TaskChangeLogEntry;
use App\Entity\User;
use App\Service\TaskChangeLogBuilder;
```

```
use Doctrine\Bundle\DoctrineBundle\Attribute\AsDoctrineListener;
use Doctrine\ORM\EntityManagerInterface;
use Doctrine\ORM\Event\OnFlushEventArgs;
use Doctrine\ORM\PersistentCollection;
use Doctrine\ORM\UnitOfWork;
use Symfony\Bundle\SecurityBundle\Security;

#[AsDoctrineListener(event: 'onFlush')]
class TaskChangeLogListener
{
    private const WATCHED_FIELDS = [
        'title',
        'description',
        'due_date',
        'taskStatus',
        'taskPriority',
        'taskType',
        'assignee',
        'ideaTaskStatus',
        'ideaTaskPriority',
        'ideaTaskType',
    ];

    public function __construct(
        private TaskChangeLogBuilder $taskChangeLogBuilder,
        private Security $security,
    ) {
    }

    public function onFlush(OnFlushEventArgs $args): void
    {
        $entityManager = $args->getObjectManager();
        $unitOfWork = $entityManager->getUnitOfWork();
        $metadata = $entityManager->getClassMetadata(TaskChangeLogEntry::class);
        $actor = $this->security->getUser();
        $actor = $actor instanceof User ? $actor : null;

        // 1. Task field updates
        foreach ($unitOfWork->getScheduledEntityUpdates() as $entity) {
            if ($entity instanceof Task) {
                $this->handleTaskUpdate($entity,      $actor,      $unitOfWork,
$entityManager, $metadata);
            } elseif ($entity instanceof RelatedTask) {
                $this->handleRelatedTaskUpdate($entity,   $actor,   $unitOfWork,
$entityManager, $metadata);
            } elseif ($entity instanceof File) {
                $this->handleFileUpdate($entity,      $actor,      $unitOfWork,
$entityManager, $metadata);
            }
        }
```

```
        // 2. RelatedTask creations
        foreach ($unitOfWork->getScheduledEntityInsertions() as $entity) {
            if ($entity instanceof RelatedTask) {
                $this->handleRelatedTaskInsertion($entity,               $actor,
$entityManager, $metadata);
            }
        }

        // 3. RelatedTask deletions
        foreach ($unitOfWork->getScheduledEntityDeletions() as $entity) {
            if ($entity instanceof RelatedTask) {
                $this->handleRelatedTaskDeletion($entity,               $actor,
$entityManager, $metadata);
            }
        }

        // 4. Collection updates (ManyToMany: assignedRoles, events)
        foreach ($unitOfWork->getScheduledCollectionUpdates() as $collection) {
            if (!$collection instanceof PersistentCollection) {
                continue;
            }
            $this->handleCollectionUpdate($collection,  $actor,  $entityManager,
$unitOfWork, $metadata);
        }
    }

    private  function  handleTaskUpdate(Task  $task,  ?User  $actor,  UnitOfWork
$unitOfWork, EntityManagerInterface $entityManager, $metadata): void
    {
        $changeSet = $unitOfWork->getEntityChangeSet($task);

        if (array_key_exists('chatIds', $changeSet)) {
            $oldChatIds = $changeSet['chatIds'][0] ?? [];
            $newChatIds = $changeSet['chatIds'][1] ?? [];
            if ($oldChatIds === null) {
                $oldChatIds = [];
            }
            if ($newChatIds === null) {
                $newChatIds = [];
            }
            $added = array_values(array_diff($newChatIds, $oldChatIds));
            $removed = array_values(array_diff($oldChatIds, $newChatIds));

            $entry  =  $this->taskChangeLogBuilder->buildChatRelationEntry($task,
$actor, $added, $removed);
            if ($entry) {
                $entityManager->persist($entry);
                $unitOfWork->computeChangeSet($metadata, $entry);
            }
        }
```

```
        $watchedChanges = array_intersect_key($changeSet,
array_flip(self::WATCHED_FIELDS));
        $entry = $this->taskChangeLogBuilder->buildFieldChangeEntry($task,
$actor, $watchedChanges);

        if ($entry) {
            $entityManager->persist($entry);
            $unitOfWork->computeChangeSet($metadata, $entry);
        }
    }

    private function handleRelatedTaskInsertion(RelatedTask $relatedTask, ?User
$actor, EntityManagerInterface $entityManager, $metadata): void
    {
        $this->persistRelatedTaskLog($relatedTask, $actor, $entityManager,
$metadata, 'added');
    }

    private function handleRelatedTaskDeletion(RelatedTask $relatedTask, ?User
$actor, EntityManagerInterface $entityManager, $metadata): void
    {
        $this->persistRelatedTaskLog($relatedTask, $actor, $entityManager,
$metadata, 'removed');
    }

    private function handleRelatedTaskUpdate(RelatedTask $relatedTask, ?User
$actor, UnitOfWork $unitOfWork, EntityManagerInterface $entityManager,
$metadata): void
    {
        $changeSet = $unitOfWork->getEntityChangeSet($relatedTask);
        if (isset($changeSet['relationType'])) {
            $this->persistRelatedTaskLog($relatedTask, $actor, $entityManager,
$metadata, 'updated', $changeSet['relationType'][0]);
        }
    }

    private function persistRelatedTaskLog(RelatedTask $relatedTask, ?User
$actor, EntityManagerInterface $entityManager, $metadata, string $action,
$oldType = null): void
    {
        $source = $relatedTask->getSourceTask();
        $target = $relatedTask->getTargetTask();
        $type = $relatedTask->getRelationType();

        if ($source && $target) {
            $unitOfWork = $entityManager->getUnitOfWork();

            // Log for source
            $entry = $this->taskChangeLogBuilder->buildTaskRelationEntry(
                $source,
                $actor,
```

```
                $action !== 'removed' ? [['task' => $target, 'type' => $type]] :
[],
                $action !== 'added' ? [['task' => $target, 'type' => $oldType ??
$type]] : [],
            );
            if ($entry) {
                $entityManager->persist($entry);
                $unitOfWork->computeChangeSet($metadata, $entry);
            }

            // Log for target
            $reciprocalType = $type ? $type->getReciprocal() : null;
            $oldReciprocalType = $oldType ? $oldType->getReciprocal() : null;
            $entry = $this->taskChangeLogBuilder->buildTaskRelationEntry(
                $target,
                $actor,
                $action !== 'removed' ? [['task' => $source, 'type' =>
$reciprocalType]] : [],
                $action !== 'added' ? [['task' => $source, 'type' =>
$oldReciprocalType ?? $reciprocalType]] : [],
            );
            if ($entry) {
                $entityManager->persist($entry);
                $unitOfWork->computeChangeSet($metadata, $entry);
            }
        }
    }

    private function handleFileUpdate(File $file, ?User $actor, UnitOfWork
$unitOfWork, EntityManagerInterface $entityManager, $metadata): void
    {
        $changeSet = $unitOfWork->getEntityChangeSet($file);
        if (isset($changeSet['deletedAt']) && $changeSet['deletedAt'][0] ===
null && $changeSet['deletedAt'][1] !== null) {
            foreach ($file->getTasks() as $task) {
                $entry = $this->taskChangeLogBuilder->buildFileChangeEntry(
                    task: $task,
                    actor: $actor,
                    addedFiles: [],
                    removedFiles: [$file],
                );
                if ($entry) {
                    $entityManager->persist($entry);
                    $unitOfWork->computeChangeSet($metadata, $entry);
                }
            }
        }
    }
```

```
    private function handleCollectionUpdate(PersistentCollection $collection,
?User $actor, EntityManagerInterface $entityManager, UnitOfWork $unitOfWork,
$metadata): void
    {
        $mapping = $collection->getMapping();
        $owner = $collection->getOwner();
        $fieldName = $mapping['fieldName'] ?? null;

        if ($owner instanceof Task) {
            $entry = null;
            if ($fieldName === 'assignedRoles') {
                $entry = $this->taskChangeLogBuilder->buildRoleAssignmentEntry(
                    task: $owner,
                    actor: $actor,
                    addedRoles: $collection->getInsertDiff(),
                    removedRoles: $collection->getDeleteDiff(),
                );
            } elseif ($fieldName === 'events') {
                $entry = $this->taskChangeLogBuilder->buildEventRelationEntry(
                    task: $owner,
                    actor: $actor,
                    addedEvents: $collection->getInsertDiff(),
                    removedEvents: $collection->getDeleteDiff(),
                );
            } elseif ($fieldName === 'files') {
                $entry = $this->taskChangeLogBuilder->buildFileChangeEntry(
                    task: $owner,
                    actor: $actor,
                    addedFiles: $collection->getInsertDiff(),
                    removedFiles: $collection->getDeleteDiff(),
                );
            }

            if ($entry) {
                $entityManager->persist($entry);
                $unitOfWork->computeChangeSet($metadata, $entry);
            }
        } elseif ($owner instanceof Event && $fieldName === 'tasks') {
            $addedTasks = $collection->getInsertDiff();
            $removedTasks = $collection->getDeleteDiff();

            foreach ($addedTasks as $task) {
                if ($task instanceof Task) {
                    $entry            =            $this->taskChangeLogBuilder-
>buildEventRelationEntry($task, $actor, [$owner], []);
                    if ($entry) {
                        $entityManager->persist($entry);
                        $unitOfWork->computeChangeSet($metadata, $entry);
                    }
                }
            }
```

```
            foreach ($removedTasks as $task) {
                if ($task instanceof Task) {
                    $entry            =            $this->taskChangeLogBuilder-
>buildEventRelationEntry($task, $actor, [], [$owner]);
                    if ($entry) {
                        $entityManager->persist($entry);
                        $unitOfWork->computeChangeSet($metadata, $entry);
                    }
                }
            }
        }
    }
}
```

Лістинг коду chat.gateway.ts:

```
import {
  ConnectedSocket,
  MessageBody,
  OnGatewayConnection,
  OnGatewayDisconnect,
  OnGatewayInit,
  SubscribeMessage,
  WebSocketGateway,
  WebSocketServer,
} from '@nestjs/websockets';
import { forwardRef, Inject, Logger } from '@nestjs/common';
import { Server, Socket } from 'socket.io';
import { SymfonyAuthService } from '../auth/symfony-auth.service';
import { ChatService } from './chat.service';
import { AuthenticatedUser } from '../auth/auth.types';

interface TypingPayload {
  conversationId: string;
  isTyping: boolean;
  userId?: string;
  userName?: string;
}

@WebSocketGateway({
  namespace: '/chat',
  cors: {
    origin: true,
    credentials: true,
  },
})
export class ChatGateway implements OnGatewayConnection, OnGatewayDisconnect,
OnGatewayInit {
  private readonly logger = new Logger(ChatGateway.name);
```

```
  constructor(
    private readonly symfonyAuthService: SymfonyAuthService,
    @Inject(forwardRef(() => ChatService))
    private readonly chatService: ChatService,
  ) {}

  @WebSocketServer()
  server: Server;

  private getUserDisplayName(user: AuthenticatedUser | null): string {
    if (!user) return 'Unknown';
    return `${user.firstName ?? ''} ${user.lastName ?? ''}`.trim() || user.email
|| 'Unknown';
  }

  private getSocketUser(client: Socket): AuthenticatedUser | null {
    return ((client.data as Record<string, any>)?.user as AuthenticatedUser) ??
null;
  }

  afterInit(server: Server) {
    server.use((socket, next) => {
      (async () => {
        try {
          const token =
            (socket.handshake.auth?.token as string | undefined) ??
            socket.handshake.headers.authorization?.replace(/^Bearer\s+/i, '');

          if (!token) {
            return next(new Error('Unauthorized'));
          }

          const user = await this.symfonyAuthService.introspectToken(token);
          socket.data = { ...(socket.data as Record<string, any>), user };
          next();
        } catch (e) {
          const errorMessage = e instanceof Error ? e.message : String(e);
          this.logger.warn(`Gateway auth failed: ${errorMessage}`);
          next(new Error('Unauthorized'));
        }
      })().catch((err) => {
        this.logger.error('Unexpected error in middleware', err);
        next(new Error('Internal Server Error'));
      });
    });
  }

  handleConnection(client: Socket) {
    const user = this.getSocketUser(client);
```

```
    this.logger.log(`Socket connected: ${client.id} user=${user?.id ?? 'none'}`);
  }

  handleDisconnect(client: Socket) {
    this.logger.log(`Socket disconnected: ${client.id}`);
  }

  @SubscribeMessage('chat.joinConversation')
  async handleJoinConversation(
    @ConnectedSocket() client: Socket,
    @MessageBody('conversationId') conversationId: string,
  ) {
    const user = this.getSocketUser(client);
    if (!user) {
      client.emit('chat.error', { message: 'Unauthorized' });
      return;
    }

    const isParticipant = await this.chatService.isConversationParticipant(user.id, conversationId);
    if (!isParticipant) {
      client.emit('chat.error', { message: 'Forbidden' });
      return;
    }

    await client.join(`conversation:${conversationId}`);
    this.logger.log(`User ${user.id} joined room conversation:${conversationId}`);

    client.emit('chat.conversation.joined', {
      conversationId,
      userName: this.getUserDisplayName(user),
    });
  }

  @SubscribeMessage('chat.joinUser')
  async handleJoinUser(@ConnectedSocket() client: Socket, @MessageBody('userId') userId: string) {
    const user = this.getSocketUser(client);
    if (!user || user.id !== userId) {
      client.emit('chat.error', { message: 'Forbidden' });
      return;
    }
    await client.join(`user:${userId}`);
    client.emit('chat.user.joined', { userId });
  }

  @SubscribeMessage('chat.leaveConversation')
  async handleLeaveConversation(
    @ConnectedSocket() client: Socket,
```

```
    @MessageBody('conversationId') conversationId: string,
  ) {
    const user = this.getSocketUser(client);
    await client.leave(`conversation:${conversationId}`);

    client.emit('chat.conversation.left', {
      conversationId,
      userName: this.getUserDisplayName(user),
    });
  }

  @SubscribeMessage('chat.typing')
  async handleTyping(@ConnectedSocket() client: Socket, @MessageBody() payload:
TypingPayload) {
    const user = this.getSocketUser(client);
    if (!user) {
      return;
    }

    const                  isParticipant                  =                  await
this.chatService.isConversationParticipant(user.id, payload.conversationId);
    if (!isParticipant) {
      return;
    }

    const typingPayload: TypingPayload = {
      ...payload,
      userId: user.id,
      userName: this.getUserDisplayName(user),
    };

    // Broadcast via service to reach everyone's user rooms
    await this.chatService.broadcastTyping(user.id, typingPayload);
  }

  emitTypingToUser(userId: string, payload: TypingPayload) {
    this.server.to(`user:${userId}`).emit('chat.typing.changed', payload);
  }

  emitConversationMessage(conversationId: string, payload: unknown) {
    this.logger.log(`Emitting              chat.message.created              for
conversation:${conversationId}`);

this.server.to(`conversation:${conversationId}`).emit('chat.message.created',
payload);
  }

  emitUserConversationUpdate(userId: string, payload: unknown) {
    this.logger.log(`Emitting chat.conversation.updated for user:${userId}`);
    this.server.to(`user:${userId}`).emit('chat.conversation.updated', payload);
  }
```

```
  emitConversationUpdatedToRoom(conversationId: string) {
    this.logger.log(`Emitting chat.conversation.updated for room
conversation:${conversationId}`);

this.server.to(`conversation:${conversationId}`).emit('chat.conversation.updated
', { conversationId });
  }

  emitReadReceipt(conversationId: string, payload: { userId: string; lastReadAt:
string }) {
    this.logger.log(`Emitting chat.read.receipt for
conversation:${conversationId}`);
    this.server.to(`conversation:${conversationId}`).emit('chat.read.receipt',
payload);
  }

  emitMessageUpdated(conversationId: string, payload: unknown) {
    this.logger.log(`Emitting chat.message.updated for
conversation:${conversationId}`);

this.server.to(`conversation:${conversationId}`).emit('chat.message.updated',
payload);
  }

  emitMessageDeleted(conversationId: string, payload: { id: string;
conversationId: string }) {
    this.logger.log(`Emitting chat.message.deleted for
conversation:${conversationId}`);

this.server.to(`conversation:${conversationId}`).emit('chat.message.deleted',
payload);
  }
}
```

Лістинг коду chat-hydration.service.ts:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Conversation } from '../entities/conversation.entity';
import { ChatMessage } from '../entities/chat-message.entity';
import { SystemSetting } from '../entities/system-setting.entity';
import { SymfonyAuthService } from '../../auth/symfony-auth.service';
import { HydratedConversation, HydratedParticipant } from '../chat.types';

@Injectable()
export class ChatHydrationService {
  constructor(
    @InjectRepository(ChatMessage)
```

```
    private readonly messageRepository: Repository<ChatMessage>,
    @InjectRepository(SystemSetting)
    private readonly systemSettingRepository: Repository<SystemSetting>,
    private readonly symfonyAuthService: SymfonyAuthService,
  ) {}

  /**
   * Calculates unread message counts for a user in the given conversations.
   */
  async hydrateUnreadCounts(userId: string, conversations: Conversation[]):
Promise<void> {
    if (!conversations.length) return;

    for (const conv of conversations) {
      const participant = conv.participants?.find((p) => p.userId === userId);
      if (!participant) {
        (conv as unknown as HydratedConversation).unreadCount = 0;
        continue;
      }

      const query = this.messageRepository
        .createQueryBuilder('message')
        .where('message.conversationId = :conversationId', { conversationId:
conv.id })
        .andWhere('message.senderId != :userId', { userId });

      if (participant.lastReadAt) {
        query.andWhere('message.createdAt > :lastReadAt', { lastReadAt:
participant.lastReadAt });
      }

      if (participant.hidePreviousMessages && participant.joinedAt) {
        query.andWhere('message.createdAt >= :joinedAt', { joinedAt:
participant.joinedAt });
      }

      (conv as unknown as HydratedConversation).unreadCount = await
query.getCount();
    }
  }

  /**
   * Fetches user names from Symfony and attaches them as a virtual `user`
property
   * on each ConversationParticipant so the frontend can display names.
   */
  async hydrateParticipants(conversations: Conversation[]): Promise<void> {
    // Collect unique user IDs across all conversations
    const userIdSet = new Set<string>();
    for (const conv of conversations) {
      for (const p of conv.participants ?? []) {
```

```
        if (p.userId) userIdSet.add(p.userId);
      }
    }

    if (!userIdSet.size) return;

    const                  userDetails                  =                  await
this.symfonyAuthService.fetchUsersDetails([...userIdSet]);

    // Build an ID -> details map for fast lookup
    const userMap = new Map(userDetails.map((u) => [u.id, u]));

    // Attach user data to each participant
    for (const conv of conversations) {
      for (const p of conv.participants ?? []) {
        const details = userMap.get(p.userId);
        if (details) {
          (p as unknown as HydratedParticipant).user = {
            id: details.id,
            firstName: details.firstName ?? null,
            lastName: details.lastName ?? null,
            email: details.email,
            avatar: details.avatar ?? null,
          };
        }
      }
    }
  }

  /**
   * Hydrates the last message for each conversation.
   */
  async hydrateLastMessages(conversations: Conversation[]): Promise<void> {
    if (!conversations.length) return;

    const showDeletedSetting = await this.systemSettingRepository.findOne({
      where: { key: 'chat_show_deleted_messages' },
    });
    const showDeleted = showDeletedSetting?.value === 'true';

    for (const conv of conversations) {
      if (!conv.lastMessageAt) {
        conv.lastMessage = null;
        continue;
      }

      const queryBuilder = this.messageRepository
        .createQueryBuilder('message')
        .where('message.conversationId  =  :conversationId',  {  conversationId:
conv.id });
```

```
      if (!showDeleted) {
        queryBuilder.andWhere('message.deletedAt IS NULL');
      }

      const lastMessage = await queryBuilder.orderBy('message.createdAt',
'DESC').getOne();

      if (lastMessage) {
        conv.lastMessage = {
          content: lastMessage.content,
          senderId: lastMessage.senderId,
          createdAt: lastMessage.createdAt,
          deletedAt: lastMessage.deletedAt,
        };
      } else {
        conv.lastMessage = null;
      }
    }
  }
}
```

Лістинг коду chatRealtimeClient.ts:

```
import { io, Socket } from 'socket.io-client';

let socket: Socket | null = null;

const resolveSocketUrl = (): string => {
  const configured = import.meta.env.VITE_CHAT_SOCKET_URL as string | undefined;
  if (configured) {
    return configured;
  }
  return window.location.origin;
};

export const getChatSocket = (): Socket => {
  if (!socket) {
    socket = io(`${resolveSocketUrl()}/chat`, {
      autoConnect: false,
      transports: ['websocket'],
      auth: (cb) => cb({
        token: localStorage.getItem('token'),
      }),
      withCredentials: true,
    });
  }

  if (!socket.connected) {
    socket.connect();
  }
```

```
  return socket;
};

export const disconnectChatSocket = () => {
  if (socket?.connected) {
    socket.disconnect();
  }
};
```

Лістинг коду SprintBurndownChart.tsx:

```
import React, { memo, useMemo } from 'react';
import { Box, Typography, Skeleton } from '@mui/material';
import { useTranslation } from 'react-i18next';
import {
  LineChart,
  Line,
  XAxis,
  YAxis,
  CartesianGrid,
  Tooltip,
  Legend,
  ResponsiveContainer,
  ReferenceLine,
} from 'recharts';
import { useGetSprintBurndownQuery } from 'behavior/sprint/sprintApi';

interface SprintBurndownChartProps {
  sprintId: string;
}

const SprintBurndownChart = memo(function SprintBurndownChart({ sprintId }:
SprintBurndownChartProps) {
  const { t } = useTranslation('sprints');
  const { data, isLoading, error } = useGetSprintBurndownQuery(sprintId);

  const { chartData, todayStr, isTodayInRange } = useMemo(() => {
    if (!data) return { chartData: [], todayStr: '', isTodayInRange: false };

    const chartDataMap = new Map<string, { date: string; ideal?: number;
actual?: number }>();

    data.ideal.forEach((point) => {
      chartDataMap.set(point.date, { date: point.date, ideal:
point.remainingPoints });
    });

    data.snapshots.forEach((snap) => {
      const existing = chartDataMap.get(snap.date) ?? { date: snap.date };
```

```
      chartDataMap.set(snap.date, { ...existing, actual: snap.remainingPoints
});
    });

    const sorted = Array.from(chartDataMap.values()).sort(
      (a, b) => new Date(a.date).getTime() - new Date(b.date).getTime(),
    );

    const today = new Date().toISOString().substring(0, 10);

    return {
      chartData: sorted,
      todayStr: today,
      isTodayInRange: sorted.some((d) => d.date === today),
    };
  }, [data]);

  if (isLoading) {
    return <Skeleton variant="rectangular" height={300} sx={{ borderRadius: 4 }}
/>;
  }

  if (error || !data) {
    return (
      <Box sx={{ height: 300, display: 'flex', alignItems: 'center',
justifyContent: 'center', bgcolor: 'background.default', borderRadius: 4 }}>
        <Typography color="text.secondary">{t('chart.error', 'Could not load
burndown chart.')}</Typography>
      </Box>
    );
  }

  return (
    <Box sx={{ width: '100%', height: 350, p: 2, bgcolor: 'background.paper',
borderRadius: 4, boxShadow: '0 4px 20px rgba(0,0,0,0.05)' }}>
      <Typography variant="subtitle1" fontWeight={700} mb={2} px={2}>
        {t('chart.title', 'Burndown Chart')}
      </Typography>
      <ResponsiveContainer width="100%" height="85%">
        <LineChart data={chartData} margin={{ top: 5, right: 30, left: 0,
bottom: 5 }}>
          <CartesianGrid strokeDasharray="3 3" vertical={false} stroke="#E0E0E0"
/>
          <XAxis
            dataKey="date"
            tickFormatter={(tick: string) => new
Date(tick).toLocaleDateString(undefined, { month: 'short', day: 'numeric' })}
            tick={{ fontSize: 12 }}
            axisLine={false}
            tickLine={false}
            dy={10}
```

```
          />
          <YAxis
            tick={{ fontSize: 12 }}
            axisLine={false}
            tickLine={false}
            dx={-10}
          />
          <Tooltip
            labelFormatter={(label:            any)            =>            new
Date(label).toLocaleDateString()}
            contentStyle={{ borderRadius: 8, border: 'none', boxShadow: '0 4px
12px rgba(0,0,0,0.1)' }}
          />
          <Legend wrapperStyle={{ paddingTop: 20 }} />
          {isTodayInRange && (
            <ReferenceLine
              x={todayStr}
              stroke="#ff9800"
              strokeDasharray="3 3"
              label={{ position: 'top', value: t('chart.today', 'Today'), fill:
'#ff9800', fontSize: 12 }}
            />
          )}
          <Line
            type="monotone"
            dataKey="ideal"
            name={t('chart.ideal', 'Ideal Burndown')}
            stroke="#9e9e9e"
            strokeWidth={2}
            strokeDasharray="5 5"
            dot={false}
          />
          <Line
            type="monotone"
            dataKey="actual"
            name={t('chart.actual', 'Remaining Story Points')}
            stroke="#2196f3"
            strokeWidth={3}
            activeDot={{ r: 6 }}
          />
        </LineChart>
      </ResponsiveContainer>
    </Box>
  );
});

export default SprintBurndownChart;
```

Лістинг коду EventPublishedListener.php:

```php
<?php

namespace App\EventListener;

use App\Entity\Event;
use App\Service\EventNotificationService;
use Doctrine\Bundle\DoctrineBundle\Attribute\AsDoctrineListener;
use Doctrine\ORM\Event\PostPersistEventArgs;
use Doctrine\ORM\Event\PostUpdateEventArgs;

#[AsDoctrineListener(event: 'postPersist')]
#[AsDoctrineListener(event: 'postUpdate')]
class EventPublishedListener
{
    public function __construct(
        private EventNotificationService $eventNotificationService
    ) {
    }

    public function postPersist(PostPersistEventArgs $args): void
    {
        $entity = $args->getObject();

        if (!$entity instanceof Event) {
            return;
        }

        if ($entity->isPublished()) {
            $this->eventNotificationService->notifyPublished($entity);
        }
    }

    public function postUpdate(PostUpdateEventArgs $args): void
    {
        $entity = $args->getObject();

        if (!$entity instanceof Event) {
            return;
        }

        $changeSet = $args->getObjectManager()->getUnitOfWork()-
>getEntityChangeSet($entity);

        if (isset($changeSet['isPublished'])) {
            [$oldValue, $newValue] = $changeSet['isPublished'];
            if (!$oldValue && $newValue) {
                $this->eventNotificationService->notifyPublished($entity);
            }
        }
    }
}
```

Лістинг коду TaskKanban.tsx:

```
import { useState, useEffect } from 'react';
import { Box, Typography, Paper, alpha, useTheme, Chip, CircularProgress } from
'@mui/material';
import { useTranslation } from 'react-i18next';
import { DragDropContext, Droppable, DropResult } from '@hello-pangea/dnd';
import { Task } from 'behavior/task/taskApi';
import KanbanCard from 'components/tasks/list/KanbanCard';
import { TaskStatus, IdeaTaskStatus } from 'behavior/task/types';
import { getLocalizedTitle } from 'utils/localization';

interface TaskKanbanProps {
  tasks: Task[];
  statuses: Array<TaskStatus | IdeaTaskStatus>;
  visibleStatusIds: string[];
  onTaskClick: (task: Task) => void;
  onTaskMove: (taskId: number, newStatusId: string) => void;
  isLoading?: boolean;
}

const TaskKanban = ({
  tasks,
  statuses,
  visibleStatusIds,
  onTaskClick,
  onTaskMove,
  isLoading
}: TaskKanbanProps) => {
  const theme = useTheme();
  const { t, i18n } = useTranslation('tasks');
  // Local state for session-based ordering and move tracking
  const     [columnOrders,     setColumnOrders]     =     useState<Record<string,
number[]>>({});
  const [pendingMoves, setPendingMoves] = useState<Record<number, string>>({});

  // Sync columnOrders with tasks prop while respecting local optimistic moves
  useEffect(() => {
    // Atomic calculation
    let pendingChanged = false;
    const remainingPending = { ...pendingMoves };

    tasks.forEach(task => {
      const        serverSid        =        (task.ideaTaskStatus?.id        ||
task.taskStatus?.id)?.toString();
      const targetSid = remainingPending[task.id];

      // If server status now matches our optimistic target, the move is
fulfilled
```

```
      if (targetSid && serverSid === targetSid) {
        delete remainingPending[task.id];
        pendingChanged = true;
      }
    });

    const nextOrders = { ...columnOrders };
    let ordersChanged = false;

    // Distribute all current tasks into their "correct" columns
    const allStatusIds = statuses.map(s => s.id.toString());
    allStatusIds.forEach(sid => {
      const localOrder = nextOrders[sid] || [];

      // Find all tasks that should be in this column
      const assignedIds = tasks.filter(task => {
        const targetSid = remainingPending[task.id];
        if (targetSid) return targetSid === sid;
        const serverSid = (task.ideaTaskStatus?.id ||
task.taskStatus?.id)?.toString();
        return serverSid === sid;
      }).map(t => t.id);

      // Maintain local order for existing items, append any truly new items
      const maintainedItems = localOrder.filter(id => assignedIds.includes(id));
      const newItems = assignedIds.filter(id => !maintainedItems.includes(id));
      const finalOrder = [...maintainedItems, ...newItems];

      if (JSON.stringify(finalOrder) !== JSON.stringify(localOrder)) {
        nextOrders[sid] = finalOrder;
        ordersChanged = true;
      }
    });

    if (pendingChanged) {
      setPendingMoves(remainingPending);
    }
    if (ordersChanged) {
      setColumnOrders(nextOrders);
    }
  }, [tasks, statuses, columnOrders, pendingMoves]);


  if (isLoading) {
    return (
      <Box display="flex" justifyContent="center" alignItems="center" py={8}
width="100%">
        <CircularProgress size={24} sx={{ mr: 2 }} />
        <Typography color="text.secondary">{t('common.loading', { ns:
'translation', defaultValue: 'Loading...' })}</Typography>
      </Box>
```

```
    );
  }

  const filteredStatuses = statuses.filter(s =>
visibleStatusIds.includes(s.id.toString()));

  const handleDragEnd = (result: DropResult) => {
    const { destination, source, draggableId } = result;
    if (!destination) return;
    if (destination.droppableId === source.droppableId && destination.index ===
source.index) return;

    const taskId = parseInt(draggableId, 10);
    const sourceSid = source.droppableId;
    const destSid = destination.droppableId;

    // 1. Mark move as pending to prevent useEffect from snapping it back
    setPendingMoves(prev => ({ ...prev, [taskId]: destSid }));

    // 2. Perform local reorder for instant, stable feedback
    setColumnOrders(prev => {
      const next = { ...prev };

      // Remove from source column
      next[sourceSid] = (next[sourceSid] || []).filter(id => id !== taskId);

      // Insert into destination column at requested index
      const destList = [...(next[destSid] || [])];
      destList.splice(destination.index, 0, taskId);
      next[destSid] = destList;

      return next;
    });

    onTaskMove(taskId, destSid);
  };

  return (
    <DragDropContext onDragEnd={handleDragEnd}>
      <Box
        sx={{
          display: 'flex',
          overflowX: 'auto',
          pb: 2,
          minHeight: '600px',
          gap: 2,
          '&::-webkit-scrollbar': {
            height: 8,
          },
          '&::-webkit-scrollbar-track': {
            background: alpha(theme.palette.background.paper, 0.5),
```

```
            borderRadius: 4,
          },
          '&::-webkit-scrollbar-thumb': {
            background: alpha(theme.palette.primary.main, 0.2),
            borderRadius: 4,
            '&:hover': {
              background: alpha(theme.palette.primary.main, 0.3),
            }
          }
        }}
      >
        {filteredStatuses.map(status => {
          const sid = status.id.toString();
          const orderedIds = columnOrders[sid] || [];

          // Get the actual task objects in the locally maintained order
          const statusTasks = orderedIds
            .map(tid => tasks.find(t => t.id === tid))
            .filter(Boolean) as Task[];

          return (
            <Box
              key={status.id}
              sx={{
                minWidth: 280,
                maxWidth: 280,
                display: 'flex',
                flexDirection: 'column'
              }}
            >
              <Paper
                elevation={0}
                sx={{
                  p: 2,
                  mb: 2,
                  borderRadius: 2,
                  bgcolor: alpha(theme.palette.primary.main, 0.05),
                  border: '1px solid',
                  borderColor: alpha(theme.palette.primary.main, 0.1),
                  display: 'flex',
                  justifyContent: 'space-between',
                  alignItems: 'center'
                }}
              >
                <Typography         variant="subtitle1"         fontWeight={700}
color="primary.main">
                  {getLocalizedTitle(status, i18n.language)}
                </Typography>
                <Chip
                  label={statusTasks.length}
                  size="small"
```

```
                    color="primary"
                    sx={{ fontWeight: 700, height: 20 }}
                  />
                </Paper>

                <Droppable droppableId={status.id.toString()}>
                  {(provided, snapshot) => (
                    <Box
                      ref={provided.innerRef}
                      {...provided.droppableProps}
                      sx={{
                        flex: 1,
                        px: 0.5,
                        minHeight: 100,
                        bgcolor:              snapshot.isDraggingOver             ?
alpha(theme.palette.primary.main, 0.05) : 'transparent',
                        transition: 'background-color 0.2s ease',
                        borderRadius: 2
                      }}
                    >
                      {statusTasks.length === 0 ? (
                        <Box
                          sx={{
                            p: 4,
                            textAlign: 'center',
                            border: '2px dashed',
                            borderColor: 'divider',
                            borderRadius: 2,
                            opacity: 0.5
                          }}
                        >
                          <Typography variant="caption" color="text.secondary">
                            {t('no_tasks')}
                          </Typography>
                        </Box>
                      ) : (
                        statusTasks.map((task, index) => (
                          <KanbanCard
                            key={task.id}
                            task={task}
                            index={index}
                            onClick={() => onTaskClick(task)}
                          />
                        ))
                      )}
                      {provided.placeholder}
                    </Box>
                  )}
                </Droppable>
              </Box>
            );
```

```
        })}
      </Box>
    </DragDropContext>
  );
};

export default TaskKanban;
```

Лістинг коду EventNotificationService.php:

```
<?php

namespace App\Service;

use App\Entity\Event;
use App\Entity\User;
use App\Enum\NotificationType;
use App\Message\CreateNotificationMessage;
use Symfony\Component\Messenger\MessageBusInterface;
use Symfony\Bundle\SecurityBundle\Security;

class EventNotificationService
{
    public function __construct(
        private MessageBusInterface $messageBus,
        private Security $security
    ) {
    }

    public function notifyPublished(Event $event): void
    {
        $idea = $event->getIdea();
        if (!$idea) {
            return;
        }

        $actor = $this->security->getUser();
        $actorId = $actor instanceof User ? (string) $actor->getId() : null;

        $eventId = (string) $event->getId();
        $eventTitle = $event->getTitle() ?? "Event #{$eventId}";
        $ideaTitle = $idea->getTitle() ?? "Idea";

        foreach ($idea->getIdeaUsers() as $ideaUser) {
            $user = $ideaUser->getUser();
            if (!$user) {
                continue;
            }

            $userId = (string) $user->getId();
```

```
            // Don't notify the person who published it
            if ($userId === $actorId) {
                continue;
            }

            $this->messageBus->dispatch(new CreateNotificationMessage(
                recipientUserId: $userId,
                type: NotificationType::EVENT_PUBLISHED,
                title: "New Event: {$eventTitle}",
                message:   "A   new   event   has   been   published   for   idea
'{$ideaTitle}': {$eventTitle}",
                resourceType: 'event',
                resourceId: $eventId,
                data: [
                    'eventId' => $eventId,
                    'eventTitle' => $eventTitle,
                    'ideaId' => (string) $idea->getId(),
                    'ideaTitle' => $ideaTitle
                ]
            ));
        }
    }
}
```