Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

Бакалавр
(назва освітнього ступеня)

на тему: **Розробка програмного забезпечення вебзастосунку для організації роботи команди на платформі .NET**

Виконав: студент IV курсу, групи СП-42
121 Інженерія програмного забезпечення
спеціальності
(шифр і назва спеціальності)

_____ Радюк В.К.
(підпис) (прізвище та ініціали)

Керівник _____ Цуприк Г.Б.
(підпис) (прізвище та ініціали)

Нормоконтроль _____ Стоянов Ю.М.
(підпис) (прізвище та ініціали)

Завідувач кафедри _____ Петрик М.Р.
(підпис) (прізвище та ініціали)

Рецензент _____ _____
(підпис) (прізвище та ініціали)

Тернопіль 2026

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Петрик М.Р.
(підпис) (прізвище та ініціали)
« 6 » квітня 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня бакалавр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

Студенту Радюку Владиславу Костянтиновичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програмного забезпечення вебзастосунку для організації роботи команди на платформі .NET

Керівник роботи Цуприк Галина Богданівна к.т.н., доц.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від « 6 » квітня 2026 року № 4/9-170
2. Термін подання студентом завершеної роботи 22.06.2026
3. Вихідні дані до роботи наукові літературні джерела

4. Зміст роботи (перелік питань, які потрібно розробити)
Вступ. 1 Аналіз предметної області та вимоги до програмного забезпечення.
2. Проєктування програмного забезпечення Zent
3. Реалізація та тестування програмного забезпечення
4. Безпека життєдіяльності, основи охорони праці
Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
Ілюстрації, діаграми, знімки екрану з демонстрацією реалізованого застосунку, слайди презентації до захисту КРБ: титульний слайд; актуальність роботи; мета та завдання роботи; основні завдання кваліфікаційної роботи; технології та методологія розробки; аналіз існуючих рішень; вимоги до програмного забезпечення; ієрархічна модель програми; варіанти використання: гість та учасник; варіанти використання: адміністратор та власник; архітектура програмного забезпечення; діаграма класів програмного забезпечення; переміщення задачі між колонками; ER-діаграма бази даних; інтерфейс застосунку; тестування програмного забезпечення; висновки

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Безпека життєдіяльності, основи охорони праці	к.т.н., доцент Мариненко С.Ю.		
Нормоконтроль	к.т.н., доц. каф. ПІ. Стоянов Ю.М.		

7. Дата видачі завдання 6 квітня 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	*Розробка технічного завдання*	*6.04 – 12.04*	*виконано*
2.	*Робота над першим розділом «Аналіз предметної області та вимоги до програмного забезпечення»*	*13.04 – 26.04*	*виконано*
3.	*Робота над другим розділом «Проєктування програмного забезпечення Zent»*	*27.04 – 03.05*	*виконано*
4.	*Робота над третім розділом «Реалізація та тестування програмного забезпечення»*	*04.05 – 17.05*	*виконано*
5.	*Робота над четвертим розділом «Безпека життєдіяльності, основи охорони праці»*	*18.05 – 24.05*	*виконано*
6.	*Оформлення пояснювальної записки і графічного матеріалу*	*25.05 – 7.06*	*виконано*
7.	*Перевірка на академічний плагіат, перевірка керівником та консультантами*	*8.06 – 14.06*	*виконано*
8.	*Попередній захист кваліфікаційної роботи бакалавра*	*15.06 – 21.06*	*виконано*
9.	*Захист кваліфікаційної роботи бакалавра*		
.			

Студент _____ Радюк В.К.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Цуприк Г.Б.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Розробка програмного забезпечення вебзастосунку для організації роботи команди на платформі .NET // Кваліфікаційна робота освітнього рівня «Бакалавр» // Радюк Владислав Костянтинович // Тернопільський національний технічний університет імені Івана Пулюя, факультет комп'ютерно-інформаційних систем і програмної інженерії, кафедра програмної інженерії, група СП-42 // Тернопіль, 2026 // с. – 87, рис. – 36, табл. – 4, додат. – 1, бібліогр. – 38.

Ключові слова: .NET, ASP.NET Core, React, CQRS, Entity Framework Core, PostgreSQL, канбан-дошка, управління проєктами, drag-and-drop, JWT-автентифікація.

Метою кваліфікаційної роботи є розробка веборієнтованої системи Zent для командної роботи над проєктами на основі канбан-методології. У проєкті реалізовано п'ять рівнів вкладеності – команда, проєкт, дошка, колонка, задача. Вони утворюють основну структуру даних. Кожен рівень має власний набір операцій.

Аналіз існуючих інструментів виявив спільну слабкість: або надмірна складність налаштування, або відсутність гнучкої рольової моделі – саме ці прогалини стали відправною точкою при формуванні вимог. Бекенд побудовано на ASP.NET Core Minimal API з власним CQRS-диспетчером замість MediatR, дані зберігаються у PostgreSQL через Entity Framework Core. Фронтенд на React 19 використовує TanStack Query для синхронізації стану та dnd-kit для drag-and-drop; окремо реалізовано оптимістичний інтерфейс зі знімком стану та відкатом при серверній помилці. Автентифікацію побудовано на JWT-токенах з рольовим розмежуванням доступу на рівні команди; коректність логіки підтверджено unit- та інтеграційними тестами.

Предмет дослідження: підходи до побудови повностекових вебзастосунків на платформі .NET із реалізацією канбан-методології, власного CQRS-диспетчера та оптимістичного інтерфейсу користувача.

ABSTRACT

Development of a Web Application for Team Workflow Management on the .NET Platform // Radiuk Vladyslav Kostiantynovych // Ternopil Ivan Puluj National Technical University, Faculty of Computer and Information Systems and Software Engineering, Department of Software Engineering, Group SP-42 // Ternopil, 2026 // p. – 87, fig. – 36, tab. – 4, app. – 1, bibl. – 38.

Keywords: .NET, ASP.NET Core, React, CQRS, Entity Framework Core, PostgreSQL, Kanban board, project management, drag-and-drop, JWT authentication.

The purpose of the qualification thesis lies in the development of the web-oriented system Zent for team-based project work grounded in the Kanban methodology. The project introduces five hierarchical levels–team, project, board, column, and task–which together form the core data structure. Each level includes a dedicated set of operations.

Analysis of existing tools revealed a shared limitation: either excessive configuration complexity or the absence of a flexible role model. These gaps shaped the system requirements. The backend relies on ASP.NET Core Minimal API and incorporates a custom CQRS dispatcher instead of MediatR; data persistence uses PostgreSQL via Entity Framework Core. The frontend, implemented with React 19, applies TanStack Query for state synchronization and dnd-kit for drag-and-drop interactions. An optimistic user interface operates with state snapshots and rollback mechanisms in case of server-side errors. Authentication uses JWT tokens with role-based access control at the team level, while unit and integration tests confirm the correctness of the implemented logic.

The object of research covers multi-user interaction in project management systems. The subject focuses on architectural approaches and tools for building full-stack web applications with role-based access control.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

RBAC – Role-Based Access Control (рольова модель керування доступом)

CQRS – Command Query Responsibility Segregation (розмежування відповідальності команд і запитів)

API – Application Programming Interface (інтерфейс програмування застосунків)

REST – Representational State Transfer (передача репрезентативного стану)

JWT – JSON Web Token (токен веб-автентифікації)

UML – Unified Modeling Language (уніфікована мова моделювання)

WBS – Work Breakdown Structure (ієрархічна структура робіт) DI – Dependency Injection (впровадження залежностей)

OCP – Open/Closed Principle (принцип відкритості/закритості)

DRY – Don't Repeat Yourself (принцип уникнення повторень)

SaaS – Software as a Service (програмне забезпечення як послуга)

FSD – Feature-Sliced Design (архітектура на основі функціональних зрізів)

DnD – Drag and Drop (перетягування елементів інтерфейсу)

EF – Entity Framework (об'єктно-реляційний маппер для .NET)

SQL – Structured Query Language (мова структурованих запитів)

UUID – Universally Unique Identifier (універсальний унікальний ідентифікатор)

HTTP – HyperText Transfer Protocol (протокол передачі гіпертексту)

DAC – Discretionary Access Control (дискреційне керування доступом)

UI – User Interface (інтерфейс користувача

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

ЗМІСТ

ВСТУП

Попит на інструменти для спільної роботи над проєктами перетворились із допоміжного ПЗ на критичну інфраструктуру команди. Розподілені учасники щодня стикаються з ситуацією, коли несвоєчасна передача інформації між членами групи безпосередньо зміщує строки.

Серед підходів до організації командного потоку робіт канбан-методологія (запозичена з виробничої практики Toyota) зберігає широке застосування в розробці ПЗ. За даними досліджень, вебплатформа з канбан-інтерфейсом скорочує час узгодження між учасниками й знижує частку пропущених дедлайнів.

Ринок наразі розколотий навпіл. Jira тисне корпоративним функціоналом, Trello й Asana – надто спрощені для команд із диференційованими ролями. Гнучке розмежування прав у межах однієї організаційної одиниці більшість інструментів середнього сегменту не підтримує взагалі. Окремо варто виділити інтерфейсну затримку під час drag-and-drop: 200–300 мс очікування після перетягування картки руйнують суб'єктивне відчуття плавності роботи. Оптимістичне оновлення UI розв'язує проблему на рівні клієнта – інтерфейс фіксує зміну негайно, а при серверній помилці відкочується до збереженого знімка стану.

Повностековий застосунок із декількома шарами вимагає суворого розподілу відповідальності між компонентами. CQRS відокремлює читання від запису: обробники команд і запитів тестуються ізольовано, зв'язність між модулями падає. Дослідники фіксують, що поєднання CQRS із багаторівневою архітектурою скорочує міжкомпонентні залежності й утримує вартість змін на прийнятному рівні навіть у міру зростання кодової бази. Власний диспетчер замість MediatR дає повний контроль над маршрутизацією команд і усуває зайву транзитивну залежність.

Актуальність теми зумовлена браком легковагових інструментів, які б поєднували гнучку рольову модель, канбан-інтерфейс і шарову архітектуру без надмірних сторонніх бібліотек.

Мета роботи – спроєктувати й реалізувати систему Zent для командного управління проєктами на основі канбан-методології, платформи .NET 9 і фреймворку React 19.

Для досягнення мети в роботі вирішено такі завдання:

– проаналізовано наявні рішення у сфері управління проєктами й виявлено їх архітектурні обмеження;

– побудовано ієрархічну модель предметної області – команда, проєкт, дошка, колонка, задача;

– спроєктовано бекенд із власним CQRS-диспетчером;

– розроблено клієнтську частину з drag-and-drop і оптимістичним UI;

– налаштовано JWT-автентифікацію з рольовим розмежуванням;

– написано unit- та інтеграційні тести.

Об'єктом дослідження виступають процеси багатокористувацької взаємодії в програмних системах управління проєктами. Предмет – підходи до побудови повностекових вебзастосунків із рольовим розмежуванням доступу.

Практична цінність роботи полягає у готовому програмному продукті для реального командного середовища. Zent веде користувача від реєстрації команди до переміщення картки між колонками дошки. Логіку перевірено двічі: на рівні ізольованих обробників команд і через реальний HTTP-хост засобами WebApplicationFactory. У ході виконання роботи спроєктовано і реалізовано повностековий застосунок Zent. Бекенд базується на ASP.NET Core Minimal API з власним CQRS-диспетчером і PostgreSQL; фронтенд – на React 19, де TanStack Query синхронізує серверний стан, а dnd-kit керує перетягуванням карток. Алгоритм динамічного впорядкування елементів дошки підтримує переміщення задач між колонками зі збереженням порядкових індексів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Управління проєктами в розподілених командах стає дедалі складнішим завданням, що вимагає спеціалізованих інструментів із чіткою структурою та гнучким розмежуванням доступу. У першому розділі ставиться мета проаналізувати предметну область командного управління проєктами, розглянути існуючі програмні рішення та сформулювати вимоги до розроблюваної системи.

1.1 Аналіз предметної області та огляд існуючих рішень

Два десятиліття тому більшість команд будувала процес розробки навколо фіксованих вимог і довгих циклів планування. Waterfall-підхід передбачав, що аналітик збирає вимоги раз і назавжди, а команда реалізує їх без відхилень. На практиці така модель ламалась щоразу, коли замовник змінював пріоритети або ринок рухався швидше за проєктний план. Agile-методології виникли як відповідь на цю нежиттєздатність. Гнучкі методології запропонували іншу логіку: планувати коротко, перевіряти часто, змінювати курс без катастрофи. Галузевий звіт State of Agile 2023 зафіксував, що близько 86% опитаних компаній або вже перейшли на ітеративну розробку, або мають такі плани – і серед них швидкість постачання продукту в середньому на 40% вища, ніж у командах на waterfall [4].

Виробнича система Toyota, де картки сигналізували про потребу в нових деталях, дала канбану назву й базову ідею – візуалізувати кожну одиницю роботи й відстежувати її рух крізь стадії виконання. Перенесена в розробку ПЗ, методологія набула додаткового виміру: WIP-ліміти обмежують кількість паралельно відкритих задач і дозволяють виявляти вузькі місця до того, як вони блокують команду. Зв'язок між канбан-дошками й покращенням координації учасників підтверджено в дослідженнях командної продуктивності [5]. Що

стосується ринку – у 2024 році відповідне ПЗ оцінювалось у 1,1 млрд дол., а до 2033 аналітики прогнозують подвоєння цього показника [6].

Окремо варто розглянути проблему розмежування доступу. Команда з трьох людей і команда з п'ятдесяти мають несхожі потреби в управлінні правами – і ця різниця не кількісна, а якісна. Менша команда покладається на довіру й неформальні домовленості; більша потребує чіткої ієрархії ролей, де власник проєкту бачить одне, рядовий учасник – інше. Практика розробки вебзастосунків підтверджує: відсутність гнучкої рольової моделі змушує команди або переплачувати за корпоративні тарифи, або шукати менш захищені обхідні рішення [7].

Нарешті, якість самого інтерфейсу канбан-дошки суттєво впливає на те, чи стане інструмент частиною щоденної практики команди. Перетягування картки між колонками – найчастіша операція на дошці, і навіть 200–300 мс затримки після неї руйнують відчуття плавності.

Оптимістичне оновлення інтерфейсу вирішує цю задачу: зміна фіксується візуально миттєво, а при серверній помилці система відкочується до збереженого знімка стану [8].

Огляд існуючих аналогів П'ять інструментів охоплюють більшу частину ринку командного управління проєктами і водночас добре ілюструють спектр компромісів між простотою й функціональністю.

Jira з'явилась у 2002 році і з того часу залишається еталоном для команд розробки. Підтримка kanban- і scrum-дошок, спринтів, епіків, story points, інтеграцій із Git і CI/CD – усе це робить Jira практично незамінною для складних технічних процесів.

Права доступу налаштовуються на рівні проєкту, компонента й окремого тикету – гнучкість максимальна. Але саме повнота функціоналу стає головним бар'єром: нові учасники команди витрачають дні на те, щоб зрозуміти логіку інструменту, а налаштування під конкретний процес нерідко потребує окремого адміністратора.

Для команд із 3–5 осіб без виділеного DevOps-спеціаліста Jira – надмірне рішення. Інтерфейс з основними функціями вебзастосунку продемонстровано на рисунку 1.1.

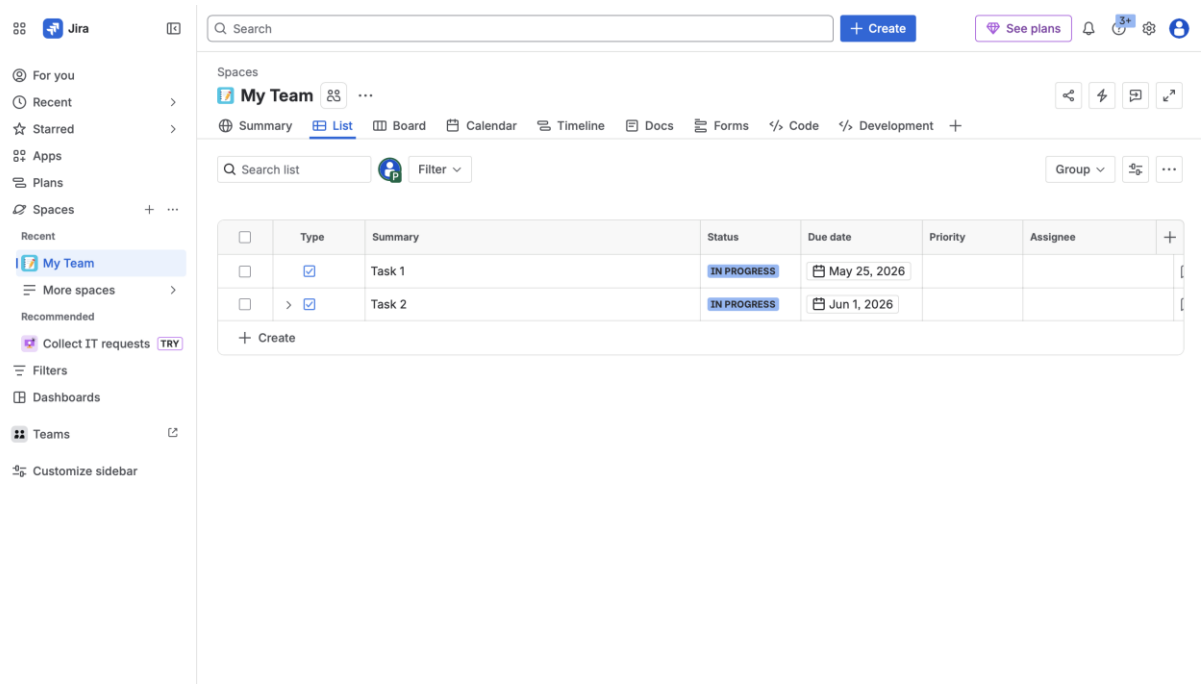

Рисунок 1.1 – Інтерфейс вебзастосунку Jira

Trello – протилежність за складністю. Дошка, картки, чеклісти, кольорові мітки – і нічого понад. Почати роботу можна за п'ять хвилин без жодного навчання.

Проблема виникає при зростанні команди або проєкту: нативна ієрархія задач у Trello відсутня, а роль учасника зводиться до двох варіантів – «може редагувати» або «лише переглядає». Будь-яка серйозна автоматизація потребує платних розширень.

Для особистого планування або мікрокоманди Trello є оптимальним вибором, але вже для десяти осіб із різними зонами відповідальності його функціоналу не вистачає. Інтерфейс вебзастосунку продемонстровано на рисунку 1.2.

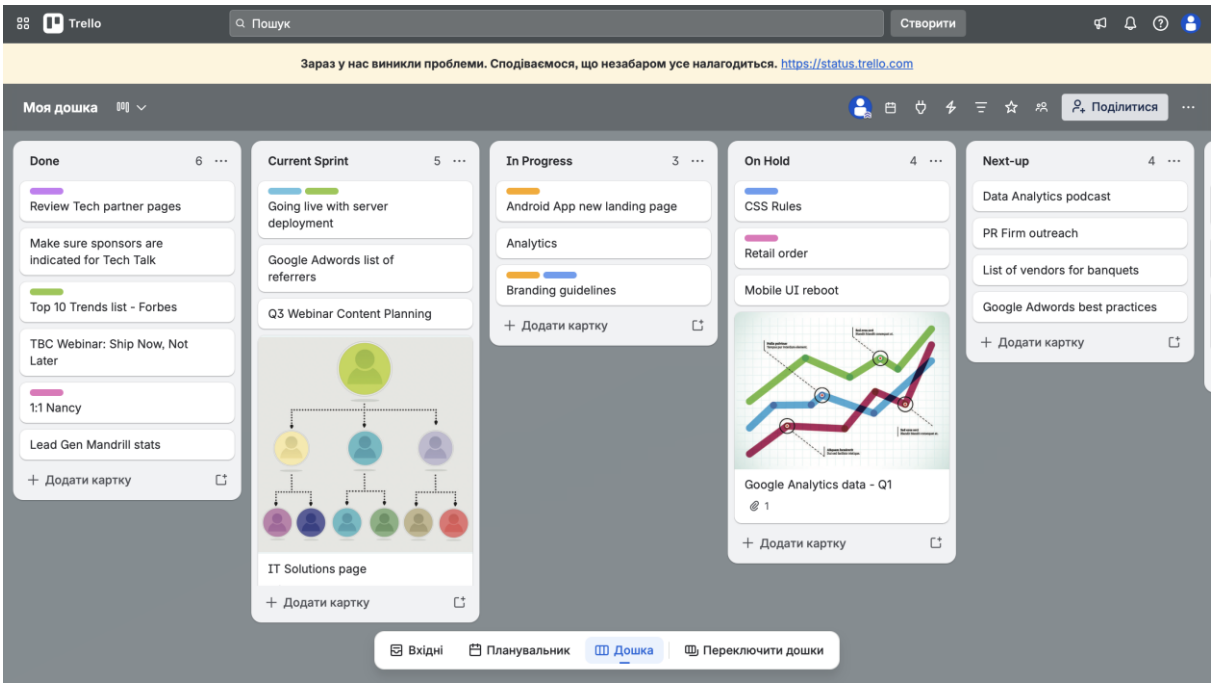

Рисунок 1.2 – Інтерфейс вебзастосунку Trello

Asana займає середину між двома полюсами. Кілька варіантів відображення задач – список, дошка, часова шкала, календар – разом із шаблонами й правилами автоматизації роблять її гнучким інструментом для бізнес-команд.

Залежності між задачами, вбудований пошук, звіти – все є. Проте Asana будувалась для менеджерів і маркетологів, а не для розробників. Концепція гілок, пул-реквестів чи деплойментів у неї відсутня, а функціонал рольового управління доступний лише на платних тарифах від 10 дол. на користувача на місяць. Переглянути інтерфейс вебдодатку можна на рисунку 1.3.

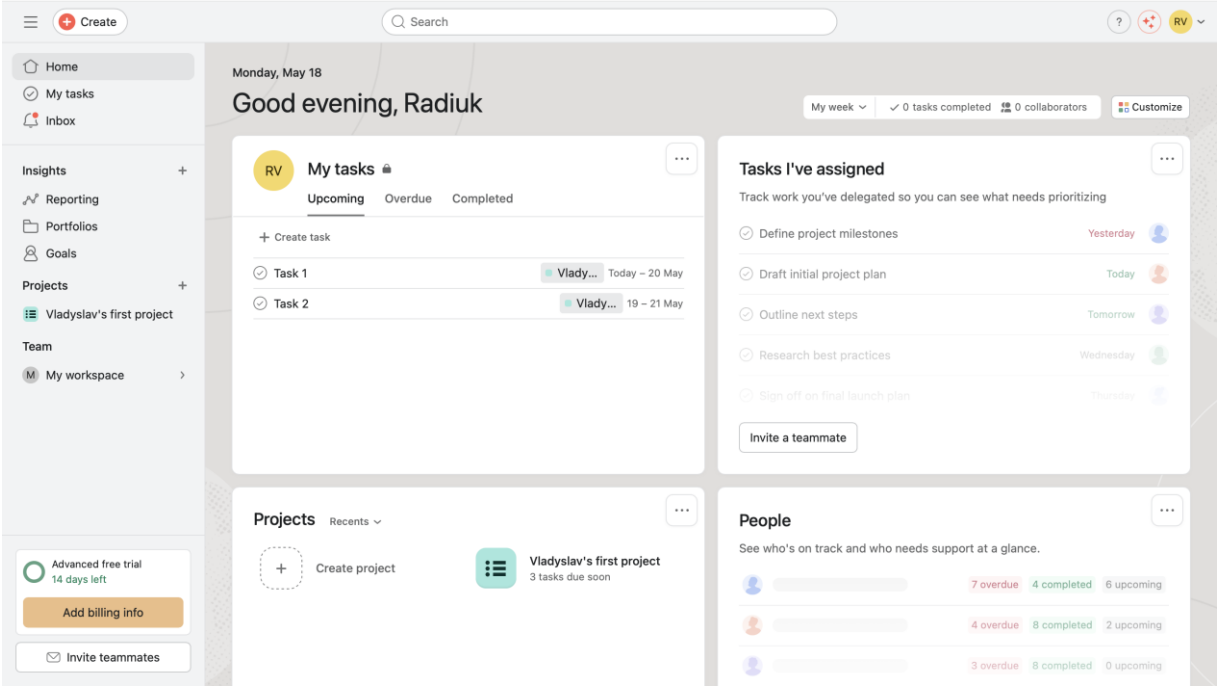

Рисунок 1.3 – Інтерфейс вебзастосунку Asana

ClickUp з'явився у 2017 році з амбіцією замінити всі робочі інструменти одразу. Кастомні поля, автоматизації, mind maps, вбудований відеозапис, документи – перелік величезний.

Для команди, що готова витратити час на налаштування, ClickUp справді стає потужним центром управління роботою. Але широта охоплення обертається хаосом для новачків: інтерфейс перевантажений, початкова конфігурація займає години.

Відома проблема – нестабільна продуктивність на навантажених workspace, що фіксується у відгуках користувачів роками. Інтерфейс вебзастосунку продемонстровано на рисунку 1.4.

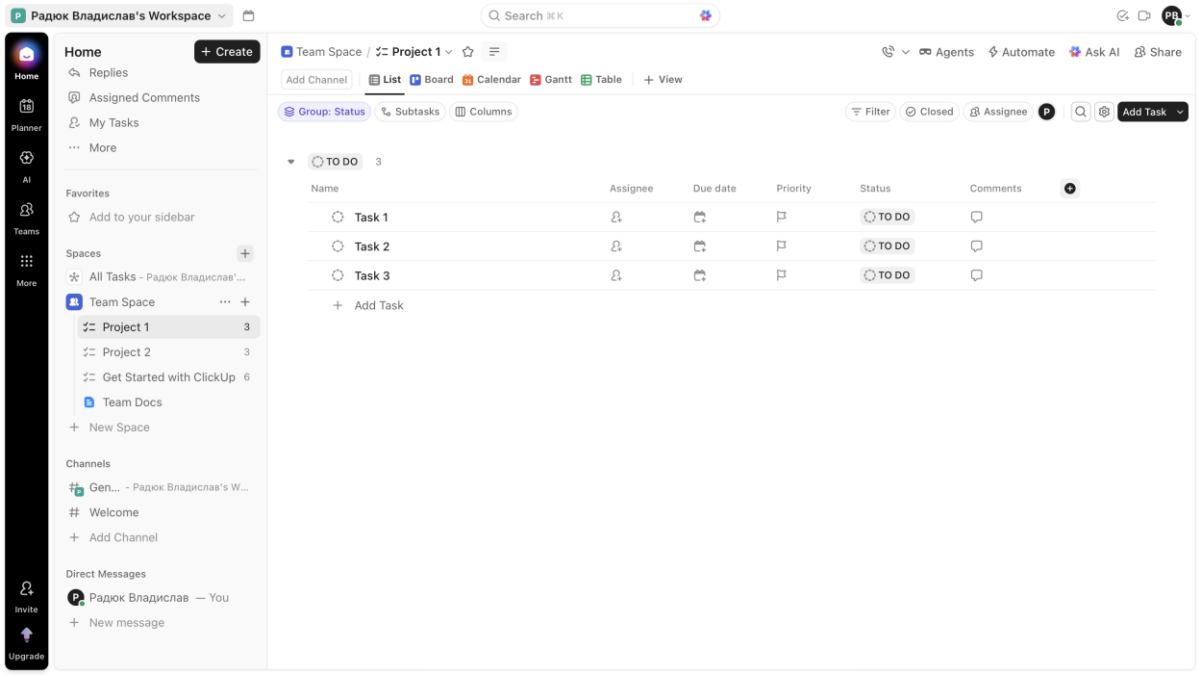

Рисунок 1.4 – Інтерфейс вебзастосунку ClickUp

Linear – наймолодший із п'яти, запущений у 2019 році. Ставка на швидкість і лаконічність принесла свої плоди: інтерфейс реагує миттєво, клавіатурні скорочення прискорюють навігацію, нативні інтеграції з GitHub і GitLab зроблені добре.

Ієрархія проста й прозора, складається з команд, проєктів, циклів та задач. Саме через цю простоту Linear підходить не всім: кастомізації мало, безкоштовний план обмежений п'ятьма учасниками, а для організацій із нестандартними процесами гнучкості може не вистачити. Переглянути інтерфейс вебдодатку можна на рисунку 1.5.

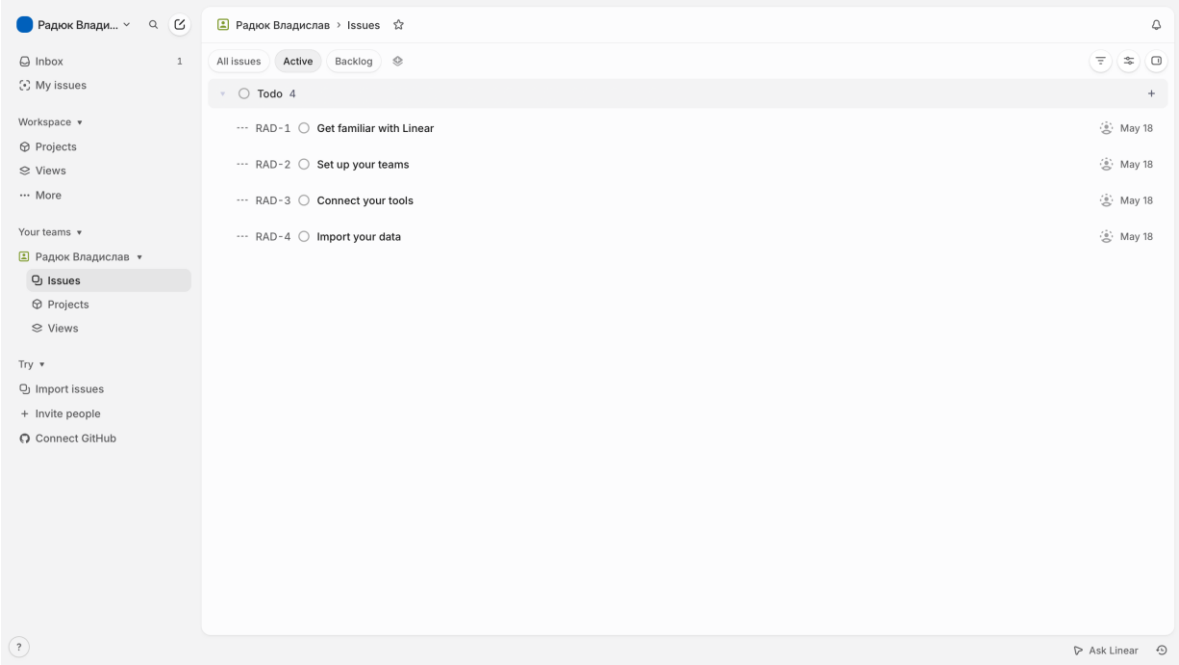

Рисунок 1.5 – Інтерфейс вебзастосунку Linear

Результати аналізу існуючих ринкових рішень підсумовано у таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз існуючих аналогів

Критерій	**Jira**	**Trello**	**Asana**	**ClickUp**	**Linear**
1	**2**	**3**	**4**	**5**	**6**
Канбан-дошка	+	+	+	+	+
Ієрархія сутностей	Проєкт → Епік → Story → Sub-task	Дошка → Картка	Проєкт → Секція → Задача	Простір → Список → Задача → Підзадача	Задача → Підзадача→ Команда → Проєкт → Задача
Рольова модель	Гнучка, багаторівнева	Мінімальна (2 ролі)	На рівні команди (платно)	Кастомна (платно)	На рівні команди
Drag-and-drop	+	+	+	+	+
Оптимістичний UI	Частково	Відсутній	Частково	Частково	+

Продовження таблиці 1.4

1	2	3	4	5	6
Складність входження	Висока	Низька	Середня	Висока	Низька
Цільова аудиторія	Dev-команди	Малі команди	Бізнес-команди	Будь-які команди	Продуктові команди
Безкоштовний план	До 10 користувачів	До 10 дошок	До 15 користувачів	Необмежено (обмежені функції)	До 5 користувачів

Таблиця фіксує закономірність, яка повторюється незалежно від конкретного інструменту: складність і гнучкість корелюють позитивно, а простота і функціональність – негативно.

Команда, що обирає Trello, жертвує ієрархією й ролями. Команда, що обирає Jira або ClickUp, платить тижнями на налаштування й онбординг.

Жоден із п'яти інструментів не поєднує одночасно низький поріг входження, повноцінну п'ятирівневу ієрархію сутностей і гнучку рольову модель на рівні команди без переплати за корпоративний тариф. Саме на заповнення цієї ніші спрямована система Zent.

На відміну від розглянутих аналогів, Zent реалізує ієрархію Team – Project – Board – Column – Task, де кожен рівень має власний набір операцій і права доступу. Ролі Owner, Admin і Member прив'язані до конкретної команди, а не до облікового запису глобально – один і той самий користувач керує однією командою й підпорядковується в іншій. Перетягування картки між колонками фіксується в інтерфейсі миттєво; при серверній помилці система повертається до знімка стану, збереженого перед операцією.

1.2 Ієрархічна модель предметної області

Організація нетривіальної роботи потребує структури, що відображає реальні відносини між об'єктами предметної області. У традиційному управлінні проєктами подібну роль виконує Work Breakdown Structure (WBS) – метод ієрархічної декомпозиції великої ініціативи на менші, конкретні та вимірювані частини [9].

WBS починається з проєкту як цілого: спочатку він ділиться на фази або напрями, кожна з яких розпадається на задачі з виконавцем, строком і критерієм завершення. Agile-методології переосмислили концепцію декомпозиції, але не відкинули її. Планування відбувається поступово: на старті команда знає великі блоки, а деталі наступного кроку стають відомі лише по мірі наближення до нього [9].

Дослідники підтверджують: стратегія декомпозиції безпосередньо впливає на зв'язність артефактів та частоту невдач при реалізації [10]. Деталізація «наперед» поступається гнучкій ітерації.

Для програмного застосунку, що підтримує командну роботу, ієрархічна модель предметної області виконує дві функції.Перша – відображення реальної організаційної структури: хто з ким працює, над чим і в яких межах. Друга – визначення меж видимості й доступу: учасник бачить рівно те, до чого його залучено.

Рівні ієрархії системи Zent У системі Zent визначено п'ять рівнів предметної області, що утворюють суворо вкладену структуру: команда – проєкт – дошка – колонка – задача. Рівні наведено на рисунку 1.6.

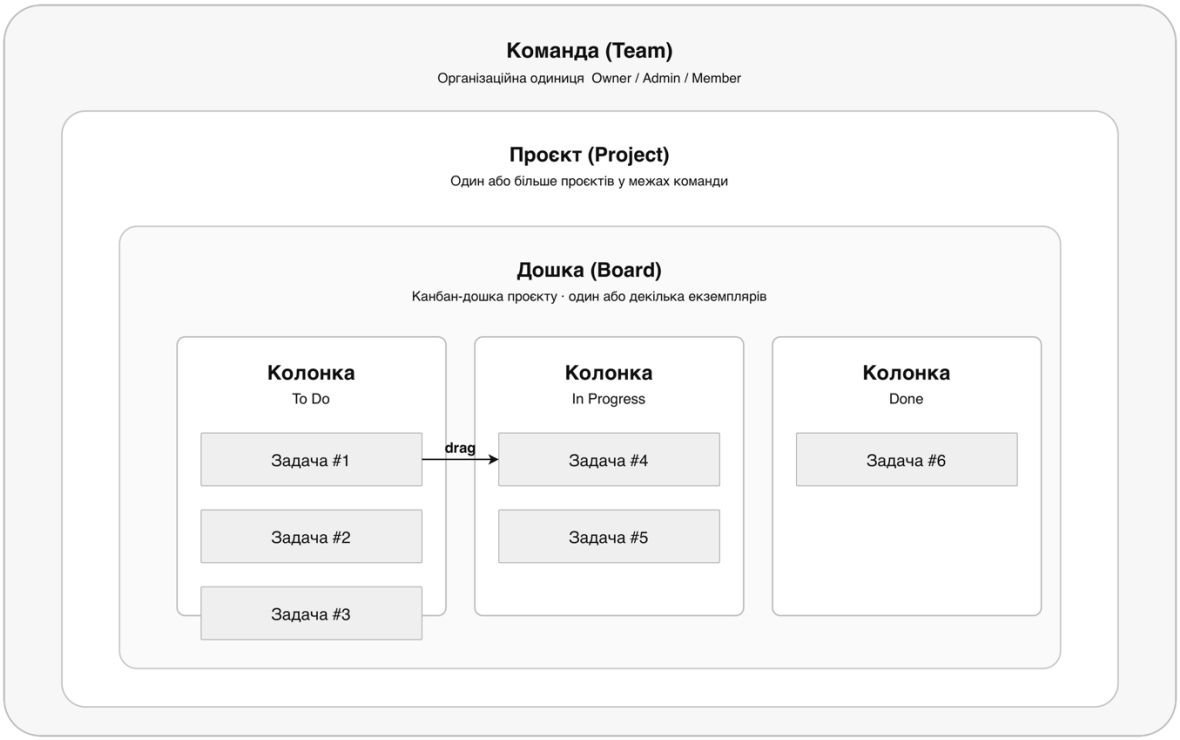

Рисунок 1.6 – Рівні ієрархії системи Zent

Команда (Team) – верхній рівень моделі. Користувачів об'єднує спільна мета, а сама команда слугує контейнером для проєктів, дошок та учасників. Керування членством і розподіл ролей відбуваються саме тут. Owner, Admin,

Member – три ролі, кожна прив'язана до конкретної команди, а не до облікового запису глобально. Той самий користувач може бути власником в одній команді й рядовим учасником в іншій. Контекстне розмежування відображає реальну організаційну практику, де людина паралельно бере участь у кількох командних структурах із різним рівнем відповідальності [11].

Проєкт (Project) – другий рівень. Всередині команди проєкт представляє конкретний напрям роботи, а саме продукт, клієнтський контракт або внутрішню ініціативу. Членство в проєкті є окремим поняттям від членства в команді: учасника можна долучити до одних проєктів і не надавати доступу до інших. Зв'язок між користувачем і конкретним проєктом зберігає окрема сутність ProjectMember, незалежна від командної ролі.

Дошка (Board) – третій рівень. Дошка є головним робочим простором проєкту й реалізує канбан-методологію в класичному вигляді. Один проєкт може

містити кілька дошок. Наприклад, окремі для бекенду й фронтенду або для різних спринтів. Дошка не має власного стану завершеності, залишаючись постійно актуальним відображенням поточного стану задач.

Колонка (Column) – четвертий рівень. Колонки формують структуру дошки й представляють стадії виконання задач. Типовий набір: «To Do», «In Progress», «Done», але система не фіксує назви: кожна команда визначає власні стадії відповідно до процесу. Числовий атрибут Order визначає позицію колонки на дошці; при переміщенні система перераховує порядок усіх колонок від 1 до N.

Задача (Task) є нижнім, найдеталізованішим рівнем. Атомарна одиниця роботи. Назва, опис, пріоритет (Low, Medium, High, Critical) і числовий порядок у межах колонки є основними атрибутами задачі. Атрибут Order визначає позицію задачі всередині поточної колонки; при переміщенні між колонками порядкові індекси обох зачеплених колонок перераховуються.

Відносини між рівнями. Між сутностями діє суворо ієрархічний принцип «один-до-багатьох» на кожному переході: одна команда містить довільну кількість проєктів, проєкт – дошок, дошка – колонок, колонка – задач. Зворотній зв'язок відсутній. Задача не може належати двом колонкам одночасно, а колонка не існує поза дошкою.

Сувора вкладеність спрощує логіку перевірки доступу: щоб визначити право користувача на редагування задачі, застосунок рухається по ієрархії вгору: від колонки через дошку до проєкту й команди. Перевіряє членство та роль на відповідних рівнях.

У Zent права доступу організовано за принципами RBAC (Role-Based Access Control): кожному користувачеві призначається роль, а роль визначає перелік дозволених операцій [11].

Глобальні права, характерні для монолітних систем, тут поступаються командно-рівневому розмежуванню – підхід, що відповідає практиці сучасних багатокористувацьких SaaS-платформ.

Таблиця 1.2 описує дозволені операції для кожної ролі.

Таблиця 1.2 – Матриця доступу за ролями

Операція	**Member**	**Admin**	**Owner**
1	**2**	**3**	**4**
Переглядати дошки та задачі	+	+	+
Створювати та переміщувати задачі	+	+	+
Керувати колонками дошки	+	+	+
Створювати та редагувати дошки	–	+	+
Створювати проєкти	–	+	+
Додавати учасників до команди	–	+	+
Видаляти команду або проєкт	–	–	+
Змінювати ролі учасників	–	–	+

Роль є атрибутом відносини «користувач – команда», а не атрибутом самого користувача. Сутність TeamMember зберігає посилання на User, посилання на Team і значення TeamRole. Один обліковий запис отримує різні ролі в різних командах – без дублювання запису, лише через різні рядки TeamMember.

Атрибути задачі та пріоритизація. Задача – найбільш деталізована сутність моделі – містить кілька атрибутів, що впливають на організацію роботи команди. Перелік TaskPriority фіксує чотири градації: Low, Medium, High, Critical. Agile-команди покладаються на пріоритизацію як на ключову практику фокусування – без неї всі задачі набувають формально однакового статусу, що ускладнює прийняття рішень [9]. Атрибут Order забезпечує детерміновану послідовність задач у межах колонки. Числовий порядок зберігається в базі даних і перераховується при кожному переміщенні – послідовність задач на дошці однакова для всіх учасників команди незалежно від моменту завантаження сторінки.

Зв'язок моделі з програмною реалізацією. Описана доменна модель безпосередньо відображається на структуру бази даних і код бекенду. Кожен рівень ієрархії отримав окрему сутність EF Core: TeamEntity, ProjectEntity, BoardEntity, ColumnEntity, TaskEntity. Зовнішні ключі реалізують відносини

«один-до-багатьох»; каскадне видалення налаштовано так, що видалення дошки автоматично поширюється на всі її колонки й задачі.

Тісна відповідність між доменною моделлю і схемою бази даних – архітектурне рішення, прийняте свідомо. Дослідники у сфері проєктування програмних систем вказують: коли семантичний розрив між моделлю предметної області та моделлю даних мінімальний, складність коду знижується, а ризик помилок при еволюції схеми зменшується [10].

1.3 Актанти системи та діаграма варіантів використання

UML визначає актанта (actor) як суб'єкта, що діє поза межами системи, – людину, пристрій або зовнішню програму. На відміну від внутрішніх компонентів, актант не входить до складу системи, а лише взаємодіє з нею ззовні.

Варіант використання (use case) описує одну таку взаємодію. Від ініціювання до отримання вимірюваного результату [12].

Серед артефактів аналізу вимог діаграма варіантів використання виконує особливу функцію: вона окреслює межі системи й показує, хто з нею взаємодіє і навіщо. Головною перевагою є не технічна точність, а комунікативна. Дослідники підтверджують: аналітики, розробники й замовники однаково читають такі діаграми без жодної додаткової підготовки [13].

Актанти системи. Zent розрізняє чотири актанти залежно від ролі, яку він займає під час сеансу. Ролі вибудовані ієрархічно. Кожна наступна роль охоплює права попередньої й додає до них власні обмежені операції.

Гість (Guest) – неавтентифікований відвідувач. Доступ відкритий лише до двох точок: реєстрації нового облікового запису та входу з наявними даними. Пройшовши автентифікацію, гість набуває прав Учасника.

Учасник (Member) – автентифікований користувач, доданий щонайменше до однієї команди. Щодня він переглядає дошки, створює й переміщує задачі, керує структурою колонок. Базова роль – призначається автоматично при вступі до команди.

Адміністратор (Admin) – Учасник із розширеними правами в межах конкретної команди. Додатково керує складом команди, проєктами та дошками. Є спеціалізацією Учасника.

Власник (Owner) – найвищий рівень прав у межах команди. Отримує роль автоматично при створенні команди. Виконує всі операції Адміністратора плюс деструктивні дії: видалення учасників і команди. Є спеціалізацією Адміністратора. Порівняльну характеристику актантів продемонстровано у таблиці 1.3.

Таблиця 1.3 – Характеристики актантів системи Zent

Актант	**Статус**	**Код ролі**	**Умова отримання**
1	**2**	**3**	**4**
Гість	Неавтентифікований	–	За замовчуванням
Учасник	Автентифікований	TeamRole.Member	Додавання до команди
Адміністратор	Автентифікований	TeamRole.Admin	Призначення Власником
Власник	Автентифікований	TeamRole.Owner	Створення команди

Ідентифікатори призначаються за принципом: префікс відображає функціональну групу варіанту використання (див. табл.1.4).

- AUTH – автентифікація та реєстрація;
- VIEW – операції перегляду та навігації;
- TASK – операції з задачами;
- BOARD – операції з колонками та дошкою;
- TEAM – управління командою та учасниками;
- PROJ – управління проєктами;
- OWN – виключні операції Власника.

AUTH-01 – Реєстрація в системі. Актант: Гість. Гість вводить ім'я, email і пароль. Система перевіряє унікальність email, хешує пароль через BCrypt і створює обліковий запис. Результат: обліковий запис створено.

AUTH-02 – Вхід у систему. Актант: Гість. Гість вводить email і пароль. Система перевіряє відповідність хешу і повертає JWT-токен. Результат: користувач автентифікований, отримав токен доступу.

VIEW-01 – Перегляд списку команд. Актант: Учасник. Відображається перелік команд, до яких належить користувач, із назвою та кількістю учасників.

VIEW-02 – Перегляд проєктів команди. Актант: Учасник. Після вибору команди відображається список проєктів, до яких користувач доданий як ProjectMember.

VIEW-03 – Перегляд канбан-дошки. Актант: Учасник. Відображається обрана дошка з усіма колонками та картками задач у відповідному порядку (Order).

TASK-01 – Створення задачі. Актант: Учасник. Користувач вводить назву, опис і пріоритет. Задача додається до кінця обраної колонки. Результат: задача відображається на дошці.

TASK-02 – Редагування задачі. Актант: Учасник. Користувач змінює назву, опис або пріоритет задачі. Результат: оновлені атрибути збережено.

TASK-03 – Видалення задачі. Актант: Учасник. Задача видаляється, порядок решти задач у колонці перераховується.

TASK-04 – Переміщення задачі між колонками. Актант: Учасник. Картка перетягується в іншу колонку. Інтерфейс оновлюється оптимістично; при серверній помилці відкочується до збереженого знімка. Порядкові індекси обох колонок перераховуються.

TASK-05 – Переміщення задачі всередині колонки. Актант: Учасник. Позиція задачі змінюється в межах тієї самої колонки через drag-and-drop. Результат: значення Order задач у колонці перераховано.

BOARD-01 – Додавання колонки. Актант: Адміністратор. Вводиться назва нової колонки. Колонка додається до кінця дошки.

BOARD-02 – Перейменування колонки. Актант: Адміністратор. Назва наявної колонки змінюється.

BOARD-03 – Видалення колонки. Актант: Адміністратор. Колонка видаляється разом із усіма задачами, що в ній знаходились. Порядок решти колонок перераховується.

BOARD-04 – Переміщення колонки. Актант: Адміністратор. Порядок колонок на дошці змінюється через drag-and-drop. Всі колонки дошки отримують нові значення Order.

PROJ-01 – Створення проєкту. Актант: Адміністратор. Вводиться назва, проєкт пов'язується з командою. Результат: проєкт відображається у списку команди.

PROJ-02 – Створення дошки. Актант: Адміністратор. У межах проєкту створюється нова порожня дошка.

TEAM-01 – Пошук користувачів. Актант: Адміністратор. За ім'ям або email виконується пошук зареєстрованих користувачів для подальшого додавання до команди.

TEAM-02 – Додавання учасника до команди. Актант: Адміністратор. Знайдений користувач додається до команди з роллю Member. Результат: користувач отримав доступ до команди.

OWN-01 – Видалення учасника з команди. Актант: Власник. Учасник виключається з команди й втрачає доступ до всіх її ресурсів.

OWN-02 – Видалення команди. Актант: Власник. Команда видаляється каскадно разом із проєктами, дошками, колонками й задачами.

Таблиця 1.4 – Зведений каталог варіантів використання

ID	**Назва вимоги**	**Опис**	**Передумова**
1	**2**	**3**	**4**
AUTH-01	Реєстрація в системі	Гість	–
AUTH-02	Вхід у систему	Гість	Обліковий запис існує
VIEW-01	Перегляд списку команд	Учасник, Адміністратор, Власник	AUTH-02

Продовження таблиці 1.4

1	2	3	4
VIEW-02	Перегляд проєктів команди	Учасник, Адміністратор, Власник	VIEW-01
VIEW-03	Перегляд канбан-дошки	Учасник, Адміністратор, Власник	VIEW-02
TASK-01	Створення задачі	Учасник, Адміністратор, Власник	VIEW-03
TASK-02	Редагування задачі	Учасник, Адміністратор, Власник	VIEW-03
TASK-03	Видалення задачі	Учасник, Адміністратор, Власник	VIEW-03
TASK-04	Переміщення задачі між колонками	Учасник, Адміністратор, Власник	VIEW-03
TASK-05	Переміщення задачі всередині колонки	Учасник, Адміністратор, Власник	VIEW-03
BOARD-01	Додавання колонки	Адміністратор, Власник	VIEW-03
BOARD-02	Перейменування колонки	Адміністратор, Власник	VIEW-03
BOARD-03	Видалення колонки	Адміністратор, Власник	VIEW-03
BOARD-04	Переміщення колонки	Адміністратор, Власник	VIEW-03
PROJ-01	Створення проєкту	Адміністратор, Власник	VIEW-01
PROJ-02	Створення дошки	Адміністратор, Власник	VIEW-02
TEAM-01	Пошук користувачів	Адміністратор, Власник	VIEW-01
TEAM-02	Додавання учасника до команди	Адміністратор, Власник	TEAM-01
OWN-01	Видалення учасника з команди	Власник	VIEW-01
OWN-02	Видалення команди	Власник	VIEW-01

Діаграми варіантів використання. Нижче наведено чотири окремі діаграми – по одній для кожного актанта (див. рис. 1.7-1.10).

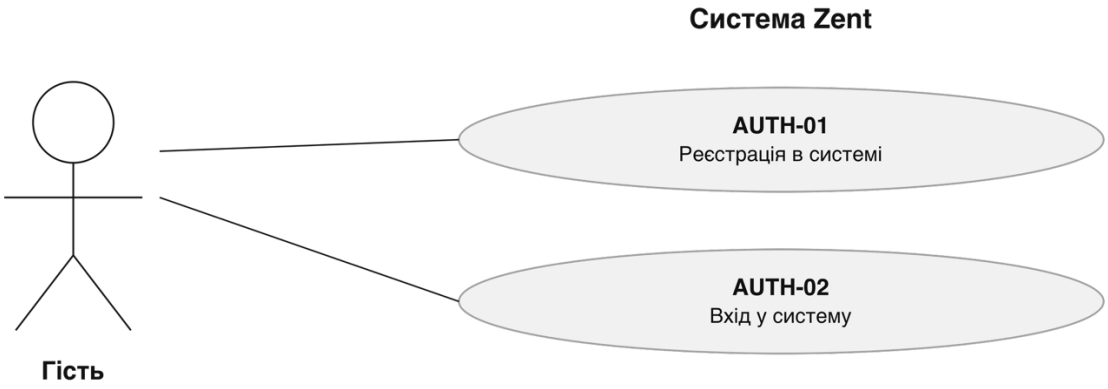

Рисунок 1.7 – Діаграма варіантів використання: Гість

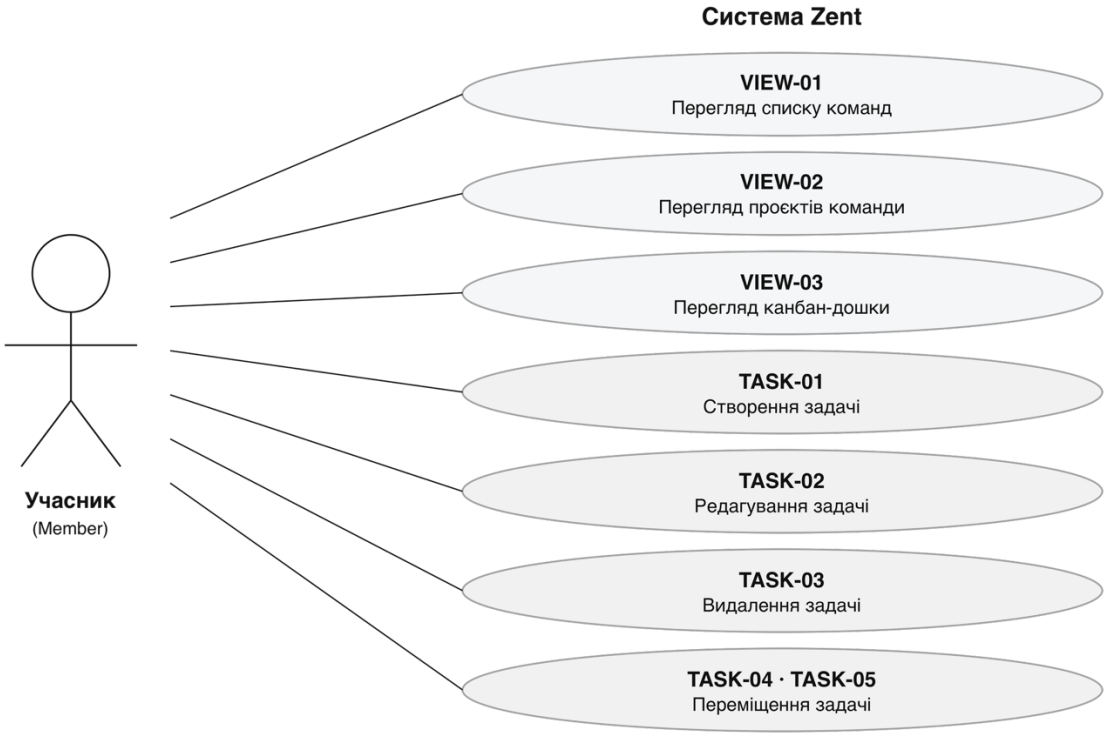

Рисунок 1.8 – Діаграма варіантів використання: Учасник

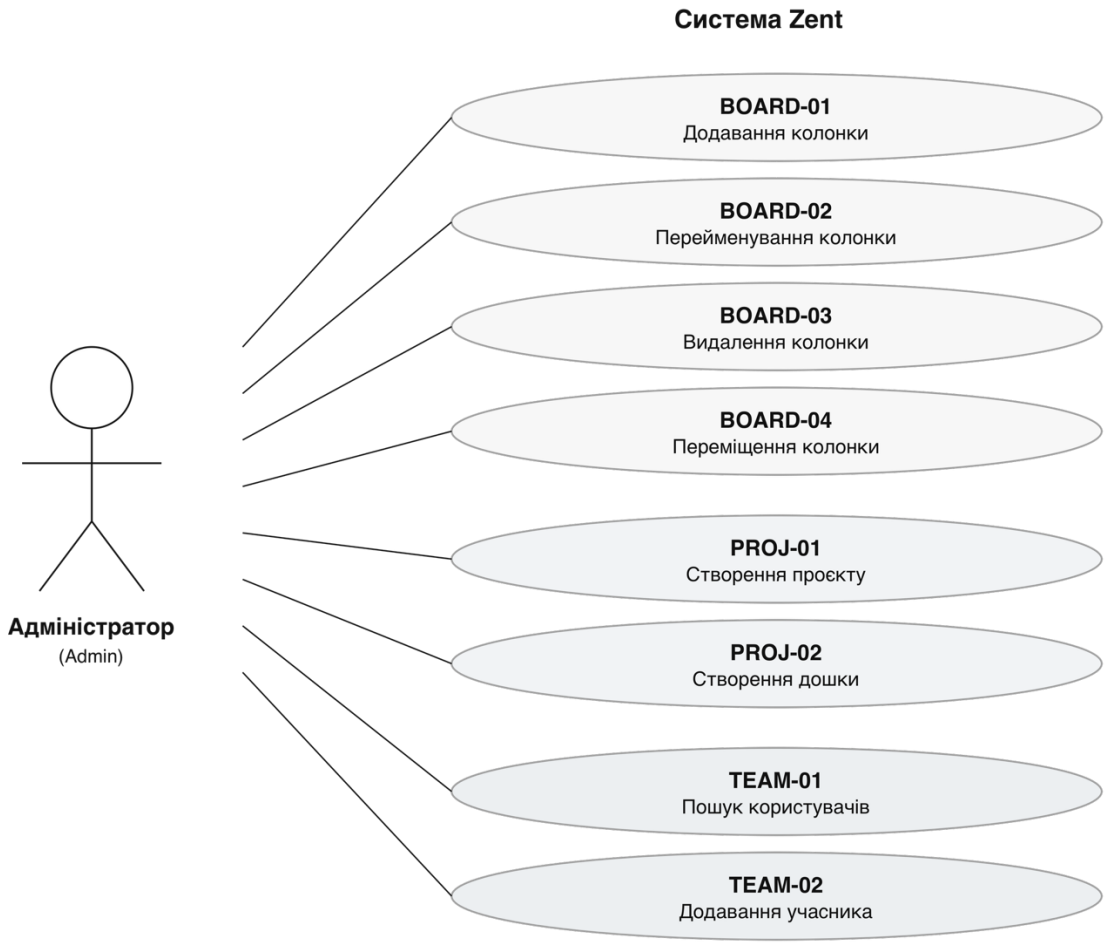

Рисунок 1.9 – Діаграма варіантів використання: Адміністратор

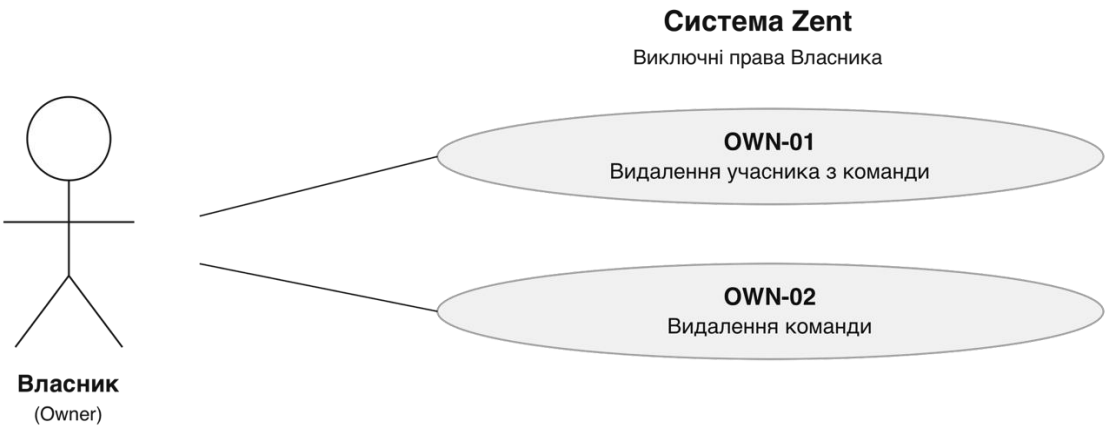

Рисунок 1.10 – Діаграма варіантів використання: Власник

Власник має лише два виключних варіанти використання – обидва деструктивні й незворотні. Решту операцій Власник успадковує від Адміністратора та Учасника через ієрархію узагальнення.ґ

Аналіз актантів системи Zent виявив чотири зовнішні суб'єкти взаємодії з чітко розмежованими наборами прав. Визначено двадцять варіантів використання, розподілених за сімома функціональними групами: AUTH, VIEW, TASK, BOARD, PROJ, TEAM, OWN. Ієрархія між актантами відповідає принципу найменших привілеїв [11]:

Гість отримує лише точку входу, Учасник – операції з задачами й перегляд, Адміністратор – управління структурою дошки та складом команди, Власник – виключне право на незворотні деструктивні дії. Отриманий каталог варіантів використання слугуватиме основою для формулювання функціональних вимог у наступному підрозділі.

1.4 Функціональні та нефункціональні вимоги до програмного забезпечення

Класифікація вимог до програмного забезпечення В інженерії вимог розрізняють дві принципово різні категорії. Функціональні вимоги (Functional Requirements, FR) фіксують, що саме виконує система – операції, поведінку й результати, доступні користувачу.

Нефункціональні вимоги (Non-Functional Requirements, NFR) стосуються іншого виміру: наскільки якісно система виконує свої функції – швидко, безпечно, без збоїв [14]. NFR нерідко відкладають «на потім». Закономірність, яку фіксують дослідники, однакова: вимоги до якості, проігноровані на етапі аналізу, проявляються в тестуванні або вже після релізу – і кожне таке відкриття коштує значно більше, ніж рання специфікація [15].

Дев'ять характеристик якості ПЗ: функціональна придатність, ефективність продуктивності, сумісність, взаємодія, надійність, безпека, зручність обслуговування, гнучкість і безпека середовища [16]. За цією класифікацією сформульовано NFR системи Zent.

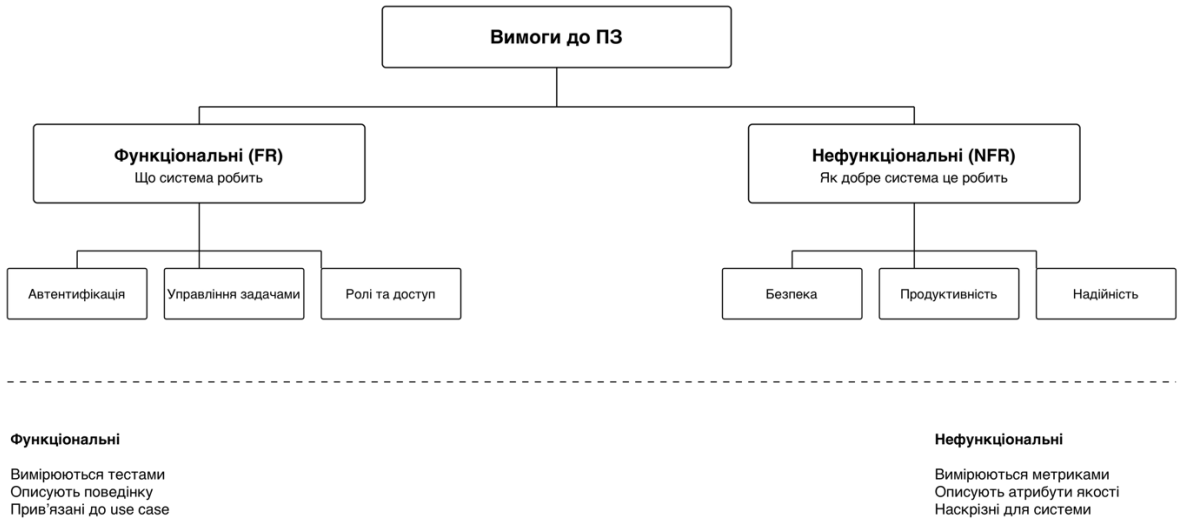

Рисунок 1.11 – Класифікація вимог до ПЗ

Функціональні вимоги Zent згруповано за тематичними блоками відповідно до груп варіантів використання з підрозділу 1.3.

Блок AUTH (Автентифікація):

– FR-AUTH-01. Реєструючись, користувач надсилає ім'я, адресу електронної пошти та пароль. Перед збереженням система хешує пароль алгоритмом BCrypt. Спроба зареєструватися з вже використаним email отримує повідомлення про помилку;

– FR-AUTH-02. Під час входу система звіряє введений пароль зі збереженим хешем. Коли перевірка проходить успішно, генерується JWT-токен з обмеженим терміном дії – клієнт отримує його у тілі відповіді;

– FR-AUTH-03. Захищений ендпоінт не обробляє запит, поки не підтвердить наявність і валідність JWT-токена у заголовку. Відсутній або прострочений токен – відповідь 401 Unauthorized.

Блок VIEW (Навігація та перегляд

– FR-VIEW-01. Увійшовши до системи, користувач бачить перелік усіх команд, до яких він належить, – із назвою та кількістю учасників кожної;

– FR-VIEW-02. Відкриваючи команду, користувач отримує лише ті проєкти, де він фігурує як ProjectMember. Проєкти без членства не потрапляють у відповідь;

– FR-VIEW-03. При відкритті дошки клієнт отримує повний її стан: колонки у порядку атрибута Order і задачі кожної колонки у тому самому порядку.

Блок TASK (Операції з задачами)

– FR-TASK-01. Назва є єдиним обов'язковим полем при створенні задачі – опис і пріоритет (Low, Medium, High, Critical) користувач може не вказувати. Нова задача займає останню позицію у колонці: її Order дорівнює поточній кількості задач плюс один;

– FR-TASK-02. Редагування задачі охоплює назву, опис і пріоритет. Атрибут Order при цьому не змінюється – лише явно вказані поля;

– FR-TASK-03. Після видалення задачі система перераховує Order решти задач у колонці, починаючи від 1;

– FR-TASK-04. Drag-and-drop переміщення задачі між колонками оновлює інтерфейс оптимістично – до підтвердження сервера. При помилці клієнт відкочується до знімка стану, збереженого перед операцією. На сервері перераховуються атрибути Order обох зачеплених колонок;

– FR-TASK-05. Переміщення задачі всередині колонки змінює Order усіх задач колонки з урахуванням нової позиції.

Блок BOARD (Структура дошки):

– FR-BOARD-01. Адміністратор або Власник додає колонку з назвою. Новий Order колонки дорівнює поточній кількості колонок дошки плюс один;

– FR-BOARD-02. Перейменування колонки зачіпає лише поле назви – Order залишається незмінним;

– FR-BOARD-03. Видалення колонки каскадно прибирає всі задачі, що в ній знаходились. Після видалення система перераховує Order решти колонок дошки;

– FR-BOARD-04. Drag-and-drop зміна порядку колонок оновлює атрибут Order для кожної колонки дошки.

Блок PROJ (Проєкти та дошки):

– FR-PROJ-01. Адміністратор або Власник створює проєкт із назвою в межах команди. Проєкт стає видимим для учасників, яких адміністратор до нього додасть;

– FR-PROJ-02. У межах проєкту Адміністратор або Власник створює порожню дошку – без колонок і задач.

Блок TEAM (Управління командою):

– FR-TEAM-01. Пошук зареєстрованих користувачів виконується за ім'ям або email. Система фільтрує результати за частковим збігом рядка;

– FR-TEAM-02. Знайденого користувача Адміністратор або Власник додає до команди з роллю Member. У таблиці TeamMember з'являється новий запис.

Блок OWN (Виключні права Власника):

– FR-OWN-01. Власник видаляє учасника з команди. Запис TeamMember прибирається, і користувач втрачає доступ до всіх ресурсів команди;

– FR-OWN-02. Власник видаляє команду. Каскадно прибираються всі проєкти, дошки, колонки, задачі й записи учасників.

Нефункціональні вимоги. NFR системи Zent сформульовано відповідно до характеристик ISO/IEC 25010:2023 [16]. Кожна вимога супроводжується вимірюваним критерієм – без нього перевірити виконання NFR під час тестування неможливо [14].

Продуктивність (Performance Efficiency):

– NFR-PERF-01. GET-запити (GetBoard, GetTeamProjects, GetTeams) сервер опрацьовує не довше 300 мс при навантаженні до 50 одночасних користувачів;

– NFR-PERF-02. Мутаційні запити (POST, PUT, DELETE), зокрема переміщення задач і колонок, сервер опрацьовує не довше 500 мс при тому самому навантаженні;

– NFR-PERF-03. Drag-and-drop операція відображається в інтерфейсі не пізніше ніж через 16 мс після завершення перетягування – це відповідає 60 fps і досягається оптимістичним оновленням стану до відповіді сервера.

Надійність (Reliability):

– NFR-REL-01. При серверній помилці під час drag-and-drop клієнт відкочує стан дошки до збереженого знімка. Відкат завершується протягом 100 мс після отримання помилки;

– NFR-REL-02. Некоректні вхідні дані не зупиняють бекенд аварійно – кожен невалідний запит отримує структуровану відповідь з HTTP-кодом і описом через GlobalExceptionHandler;

– NFR-REL-03. Міграції бази даних запускаються автоматично при старті застосунку. Невідповідність схеми зупиняє запуск із діагностичним повідомленням.

Безпека (Security):

– NFR-SEC-01. Паролі зберігаються виключно як BCrypt-хеші. Відновити пароль із хешу засобами системи неможливо;

– NFR-SEC-02. Протермінований або підроблений JWT-токен отримує відповідь 401 на всіх захищених ендпоінтах без винятку;

– NFR-SEC-03. Спроба учасника однієї команди отримати ресурси іншої повертає 403 Forbidden. Перевірка відбувається на рівні обробника кожного запиту;

– NFR-SEC-04. FluentValidation перехоплює некоректні або відсутні обов'язкові поля до передачі даних в обробник. Такі запити отримують 400 Bad Request на рівні API.

Зручність обслуговування (Maintainability):

– NFR-MAINT-01. Кожен новий варіант використання реалізується як окремий обробник – ICommandHandler або IQueryHandler – без змін у вже існуючих. Принцип відкритості/закритості (OCP) витримується на рівні архітектури;

– NFR-MAINT-02. Unit-тести охоплюють обробники автентифікації та критичну бізнес-логіку: переміщення задач і колонок. Для кожного обробника є тест на позитивний сценарій і тест на обробку виключення;

– NFR-MAINT-03. Інтеграційні тести перевіряють ендпоінти автентифікації через реальний HTTP-хост (WebApplicationFactory) з ізольованою тестовою базою даних у пам'яті.

Сумісність та розгортання (Compatibility):

– NFR-COMP-01. Браузери Chromium-родини і Firefox актуальних версій відображають клієнтську частину коректно на розширеннях від 1280px до 1920px;

– NFR-COMP-02. Кожен запит до REST API несе всю необхідну інформацію – сервер не зберігає стан сеансу між запитами, що відповідає stateless-принципу;

– NFR-COMP-03. Бекенд і PostgreSQL не прив'язані до конкретної ОС. Розгортання можливе на будь-якій платформі з підтримкою .NET 9.

Сформований перелік слугує вихідним матеріалом для проєктування архітектури системи в другому розділі.

Розділ охоплює три взаємопов'язані задачі: аналіз предметної області командного управління проєктами, огляд наявних програмних рішень і формування вимог до системи Zent. Перехід від Waterfall до Agile переосмислив не лише організаційні практики – він змінив і вимоги до інструментарію підтримки командної роботи. Канбан у цьому контексті вирізняється практичністю: методологія робить потік роботи видимим і дає змогу виявляти вузькі місця до того, як вони блокують команду. Ринок відреагував відповідно – 1,1 млрд доларів у 2024 році з очікуваним подвоєнням до 2033-го [6].

П'ять провідних інструментів – Jira, Trello, Asana, ClickUp і Linear – проаналізовано за спільним набором критеріїв. Картина виявилась однотипною: складні системи дають гнучкість, але потребують тижнів налаштування; прості – запускаються за п'ять хвилин, але не тягнуть повноцінну ієрархію й рольову модель без доплати. Поєднати низький поріг входження з п'ятирівневою

ієрархією сутностей і командно-рівневим RBAC у безкоштовному варіанті не вдається жодному. Саме тут і знаходиться ніша Zent. Ієрархічна модель предметної області охоплює п'ять рівнів: команда, проєкт, дошка, колонка, задача. Сувора вкладеність визначає логіку перевірки доступу, а рольова модель на принципах RBAC реалізована контекстно – на рівні окремої команди, що дозволяє одному користувачеві мати різні ролі без дублювання облікового запису.

Аналіз актантів визначив чотири суб'єкти взаємодії та двадцять варіантів використання, розподілених за сімома функціональними групами. Ієрархія між актантами відповідає принципу найменших привілеїв. На основі каталогу сформульовано функціональні та нефункціональні вимоги відповідно до ISO/IEC 25010:2023 із вимірюваними критеріями прийнятності. Отримані результати слугують вихідним матеріалом для проєктування архітектури в другому розділі.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ZENT

Проєктування програмної системи – це етап, на якому абстрактні вимоги набувають форми конкретних архітектурних рішень. Тут визначаються межі відповідальності: як компоненти взаємодіють, за якими принципами організовано код – усе визначається саме тут. Дослідники у сфері програмної інженерії фіксують закономірність: архітектурні недоліки, виявлені після початку реалізації, команда усуває в десятки разів дорожче, ніж ті самі недоліки на етапі проєктування [17].

Другий розділ охоплює повний цикл проєктувальних рішень системи – від вибору архітектурного стилю до моделювання структури класів. Кожне рішення супроводжується доцільним обґрунтуванням.

2.1 Архітектура програмного забезпечення та діаграми класів

Архітектура програмної системи – сукупність фундаментальних рішень щодо організації компонентів, їхніх відповідальностей і способів взаємодії [17].

Обраний підхід визначає, наскільки легко система еволюціонуватиме: чи вдасться додати новий варіант використання, замінити базу даних, масштабувати команду розробників без лавинних змін у коді. Погано спроєктована архітектура не дається взнаки одразу. Вона проявляється поступово – у вигляді дедалі складніших змін і зростаючого ризику регресій при кожному новому доповненні.

Для системи Zent обрано Clean Architecture – підхід, де внутрішні шари не залежать від зовнішніх [18]. Бізнес-логіка нічого не знає про базу даних, HTTP-протокол чи конкретний фреймворк – усі інфраструктурні деталі знаходяться на периферії і можуть бути замінені без змін у ядрі. Знижена зв'язність між компонентами дає розробнику змогу тестувати бізнес-логіку незалежно від інфраструктури.

Практично Clean Architecture реалізовано через поділ бекенду на шість окремих .NET-проєктів із суворою односпрямованістю залежностей:

Zent.Common ← Zent.Data ← Zent.Application ← Zent.Infrastructure ← Zent.Postgresql ← Zent.API Стрілка означає «залежить від». Жоден внутрішній шар не посилається на зовнішній.Zent.Application знає про інтерфейси сервісів, але не про конкретні реалізації. Вони надходять через dependency injection із Zent.Infrastructure. Zent.Common містить спільні перелічення, константи й типи виключень, доступні на всіх рівнях без циклічних залежностей. CQRS як патерн організації бізнес-логіки

Поряд із Clean Architecture у системі застосовано патерн CQRS (Command Query Responsibility Segregation) – розмежування операцій зміни стану від операцій читання [19]. Патерн вирішує конкретну проблему: клас, через який проходять і читання, і запис, неминуче обростає умовними розгалуженнями, порушуючи принцип єдиної відповідальності. Відокремивши команди від запитів, з'являється можливість оптимізувати кожен тип операції незалежно: читання через спрощені проєкції, запис через повну валідацію й транзакційний контекст.

У Zent CQRS реалізовано без сторонніх бібліотек. Система розрізняє два типи операцій: Команди – операції зміни стану. Кожна команда реалізує відповідний інтерфейс і обробляється окремим хендлером. Прикладами слугують створення задачі, переміщення колонки, видалення команди. Запити – операції читання без побічних ефектів. Кожен запит повертає дані, не модифікуючи стан системи. Прикладами є отримання дошки з колонками, перелік команд користувача, деталі задачі. Центральний диспетчер знаходить відповідний хендлер і делегує виконання йому. Кожен новий варіант використання реалізується як окремий хендлер – вже існуючі хендлери при цьому не змінюються, що витримує принцип відкритості/закритості на рівні архітектури.

Клієнтська частина організована за методологією Feature-Sliced Design (FSD) – архітектурним підходом для React-застосунків, де кодова база розбивається на шари з односпрямованими залежностями [20]. Традиційний

поділ за технічним типом файлів FSD замінює групуванням за функціональними доменами, що відповідають реальним можливостям застосунку.

Структура клієнтської частини охоплює чотири шари: app (ініціалізація застосунку), провайдери і маршрутизатор. pages (сторінки), кожна з яких компонує функціональні модулі. features – ізольовані функціональні модулі: автентифікація, дошки, колонки, задачі, команди. shared – HTTP-клієнт, зберігання токена, спільні хуки та компоненти розмітки.

Шар features є ключовим. Кожен модуль містить власні компоненти, хуки для роботи з серверним станом і типи. Залежності йдуть виключно вниз по шарах, що унеможливлює циклічні зв'язки й спрощує розуміння кодової бази при масштабуванні. Архітектуру шарів FSD проілюстровано на рисунку 2.1.

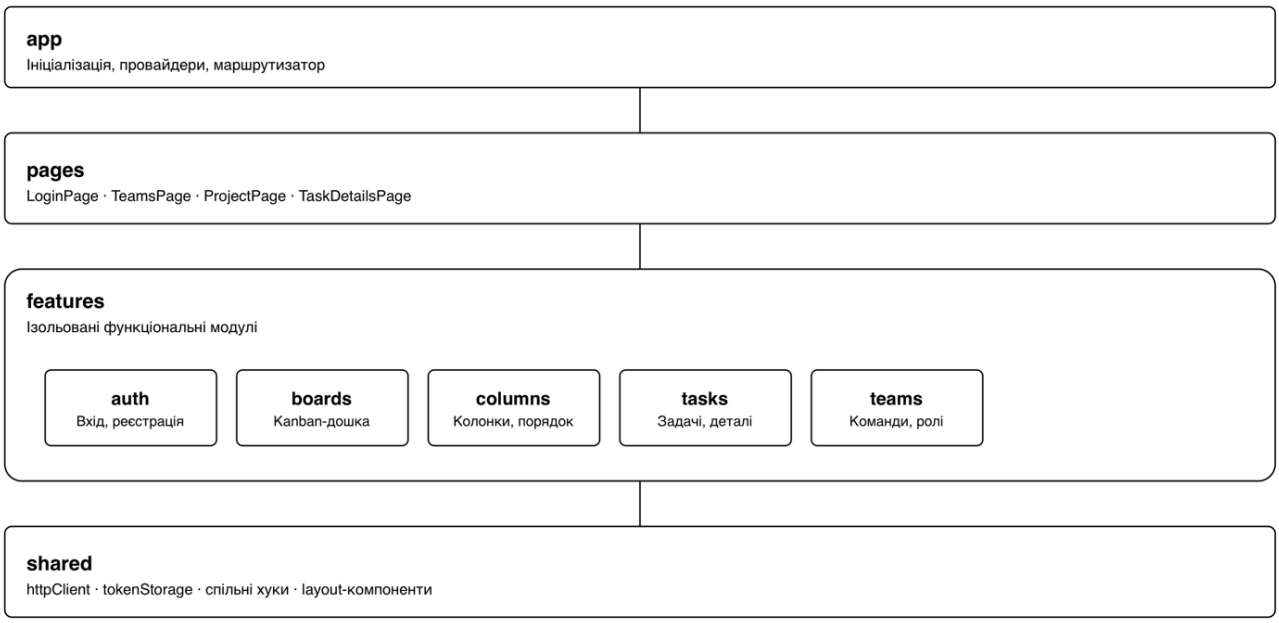

Рисунок 2.1 – Архітектура клієнтської частини застосунку

Взаємодія шарів системи Будь-який запит у системі Zent проходить однаковий маршрут крізь шари архітектури – незалежно від конкретного варіанту використання. Шар API приймає HTTP-запит, валідує вхідні дані й формує об'єкт команди або запиту. Шар Application отримує об'єкт через диспетчер, виконує бізнес-логіку й звертається до шару даних для читання або збереження. Infrastructure надає конкретні реалізації сервісів – генерацію токенів,

хешування паролів. База даних отримує фінальні SQL-команди від Entity Framework Core і повертає результат назад по ланцюжку. Кожен шар знає лише про той, що знаходиться безпосередньо під ним. Жоден шар не «перестрибує» через сусідній. Завдяки такій структурі заміна будь-якого компонента (бази даних, механізму автентифікації, HTTP-фреймворку) потребує змін лише в одному місці, не зачіпаючи решту системи [18].

Діаграма класів системи. Діаграма класів є одним із ключових артефактів об'єктно-орієнтованого проєктування. Вона відображає статичну структуру системи: які класи існують, які атрибути й методи вони містять і як між собою пов'язані [17].

На відміну від діаграм послідовності або станів, діаграма класів не описує поведінку в часі, вона фіксує архітектурний «знімок» доменного шару, що слугує основою для реалізації та документування системи.

У проєктуванні програмного забезпечення діаграма класів виконує роль технічного контракту між аналізом вимог і написанням коду. У контексті проєктування програмного забезпечення діаграми класів виконують роль технічного контракту між аналізом вимог і написанням коду.

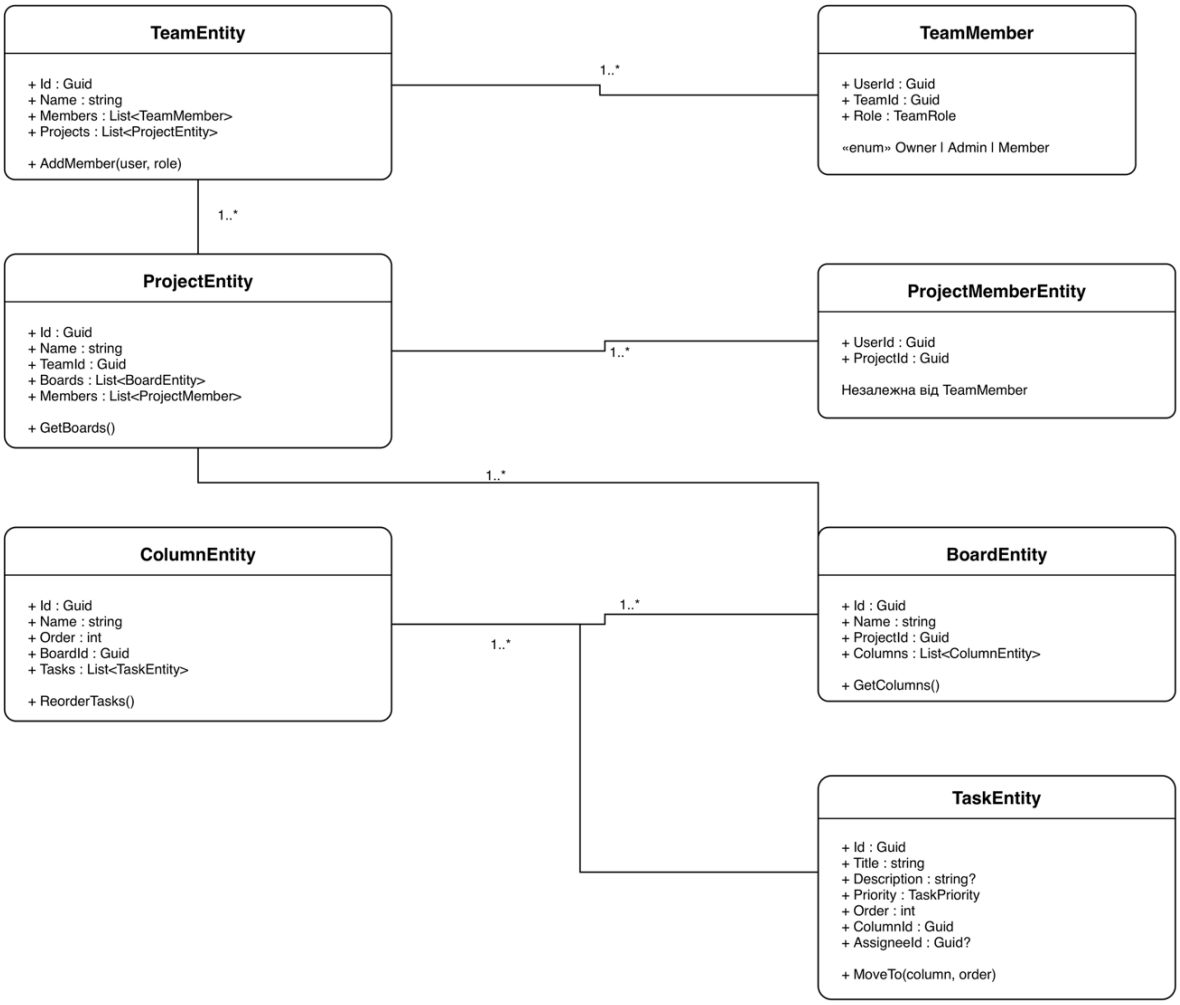

Рисунок 2.2 – Діаграма класів системи Zent

Діаграма класів (див. рис. 2.2) охоплює сім основних сутностей доменного шару та відображає відносини між ними. Центральне місце займає ієрархія вкладеності: TeamEntity агрегує проєкти, проєкти – дошки, дошки – колонки, колонки – задачі. Зв'язки між класами реалізовано через композицію з кардинальністю «один-до-багатьох» на кожному рівні. Допоміжні класи TeamMember і ProjectMemberEntity відображають дворівневу модель членства, а перелічення TaskPriority – систему пріоритетів задач.

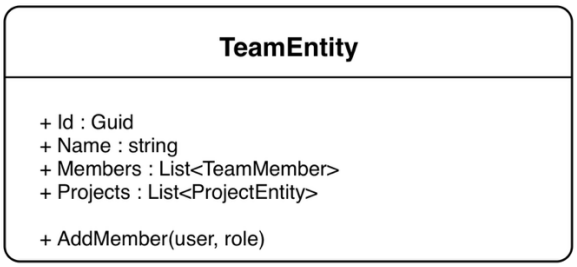

Рисунок 2.3 – Клас TeamEntity

TeamEntity (див. рис. 2.3) є кореневим агрегатом усієї ієрархії та точкою входу для більшості операцій з командою. Клас зберігає назву команди, колекцію учасників типу TeamMember і колекцію підпорядкованих проєктів. Метод AddMember інкапсулює логіку призначення ролі, унеможливлюючи пряму маніпуляцію колекцією ззовні. Через TeamEntity проходять усі перевірки членства й авторизації, що робить його центральним об'єктом рольової моделі системи.

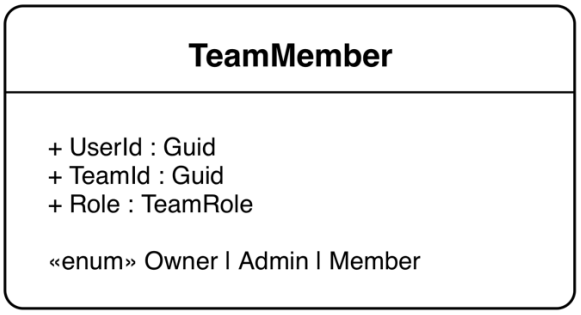

Рисунок 2.4 – Клас TeamMember

TeamMember (див. рис. 2.4) – клас-зв'язка між користувачем і командою, що зберігає контекстну роль учасника. Ключовим атрибутом є Role типу TeamRole з трьома значеннями: Owner, Admin, Member. Завдяки тому, що роль є атрибутом відносини, а не самого користувача, один обліковий запис може мати різні ролі в різних командах. Клас не є самостійним агрегатом і не існує поза контекстом команди.

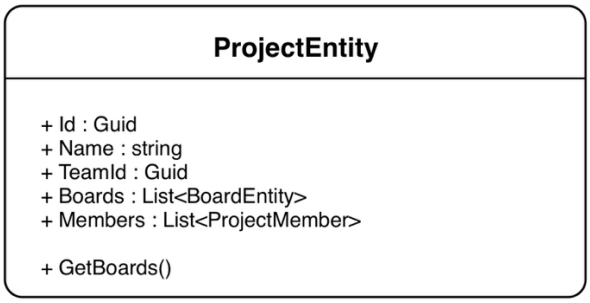

Рисунок 2.5 – Клас ProjectEntity

ProjectEntity (див. рис. 2.5) представляє конкретний напрям роботи всередині команди та слугує контейнером для дошок і учасників проєкту. Зовнішній ключ TeamId фіксує належність проєкту до команди. Клас агрегує дві незалежні колекції: Boards – дошки проєкту, і Members – учасників через ProjectMemberEntity. Метод GetBoards надає впорядкований доступ до дошок відповідно до прав поточного користувача.

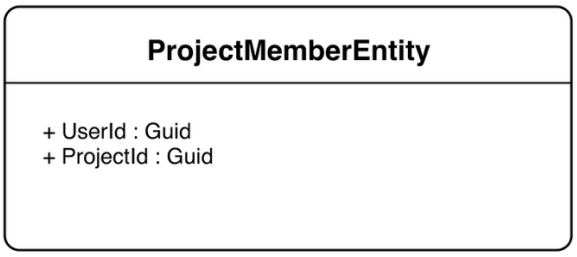

Рисунок 2.6 – Клас ProjectMemberEntity

ProjectMemberEntity (див. рис. 2.6) – окрема сутність, що зберігає зв'язок між користувачем і конкретним проєктом незалежно від командної ролі. Наявність двох атрибутів – UserId і ProjectId – дозволяє надавати або обмежувати доступ до окремих проєктів у межах однієї команди. Учасник команди може бути членом одних проєктів і не мати доступу до інших, що реалізує принцип найменших привілеїв на рівні проєкту. Відсутність власного атрибута ролі є свідомим рішенням: рольові права успадковуються з TeamMember.

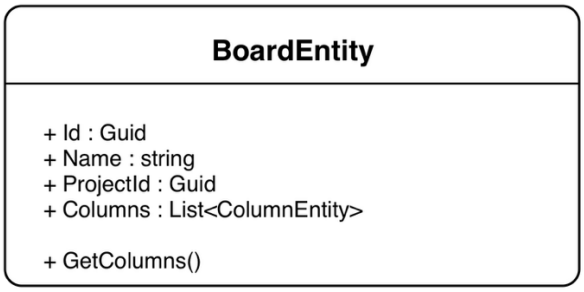

Рисунок 2.7 – Клас BoardEntity

BoardEntity (див. рис. 2.7) є головним робочим простором проєкту і реалізує канбан-дошку як об'єкт доменного шару. Клас прив'язаний до проєкту через ProjectId і агрегує колекцію колонок. Дошка не містить власного стану завершеності – вона є постійно актуальним відображенням поточного розподілу задач. Метод GetColumns повертає колонки в порядку, визначеному атрибутом Order кожної з них.

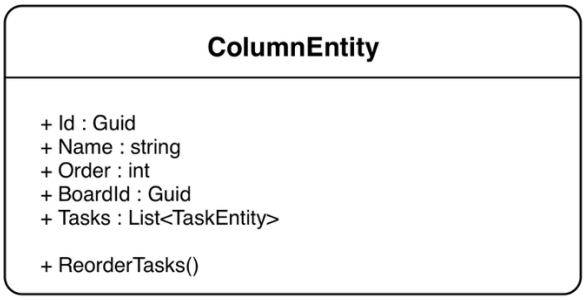

Рисунок 2.8 – Клас ColumnEntity

ColumnEntity (див. рис. 2.8) формує структуру дошки і представляє окрему стадію виконання задач. Атрибут Order визначає позицію колонки серед інших колонок дошки і перераховується щоразу, коли користувач переміщує колонку через drag-and-drop. Метод ReorderTasks централізує логіку перерахунку порядку задач після будь-якої операції переміщення всередині колонки. Каскадне видалення налаштовано так, що видалення колонки автоматично прибирає всі задачі, які в ній знаходились.

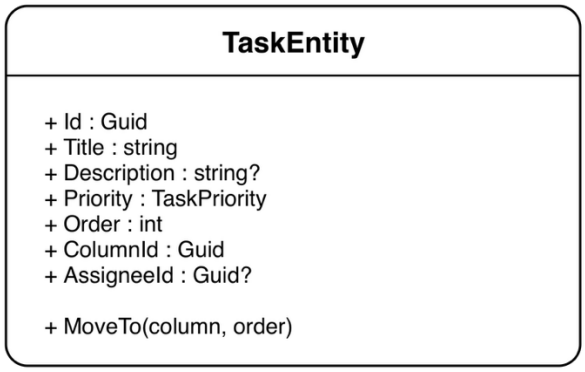

Рисунок 2.9 – Клас TaskEntity

TaskEntity (див. рис. 2.9) є найдеталізованішою сутністю доменного шару і атомарною одиницею роботи в системі. Клас містить назву, опис, пріоритет типу TaskPriority, порядковий атрибут Order і опціональний ідентифікатор виконавця AssigneeId. Атрибут Order перераховується як при переміщенні задачі всередині однієї колонки, так і при переміщенні між колонками – в останньому випадку перераховуються порядки обох зачеплених колонок. Метод MoveTo інкапсулює цю двоетапну логіку, приймаючи цільову колонку й нову позицію як параметри.

Патерни проєктування. Архітектура системи Zent спирається на кілька усталених патернів проєктування, кожен із яких вирішує конкретну задачу і застосовується на відповідному рівні системи [19]

Dependency Injection пронизує всю архітектуру бекенду. Інтерфейси сервісів оголошено у внутрішніх шарах, а конкретні реалізації зареєстровано у зовнішніх і передаються через конструктор. Жоден клас бізнес-логіки не створює залежності самостійно – усі вони отримуються ззовні. Завдяки цьому будь-який сервіс можна підмінити тестовою заглушкою без змін у коді, що перевіряється.

Chain of Responsibility реалізує обробку виключень у системі. Централізований глобальний обробник перехоплює типізовані виключення з усіх шарів і транслює їх у відповідні HTTP-статуси. Жоден ендпоінт не містить власної логіки обробки помилок, оскільки вся відповідальність делегована

одному компоненту. Такий підхід гарантує однорідні відповіді API незалежно від того, в якому хендлері виникла помилка.Builder Pattern застосовується у тестовому шарі для конструювання тестових об'єктів. Замість того щоб у кожному тесті вручну ініціалізувати складні сутності з десятками полів, спеціальний будівельник дозволяє декларативно задати лише ті атрибути, що важливі для конкретного тесту. Решта заповнюється розумними значеннями за замовчуванням, що зменшує дублювання коду й спрощує читання тестів.

Optimistic UI Update застосовано на клієнтській стороні для drag-and-drop операцій. Інтерфейс реагує на дію користувача миттєво, не чекаючи підтвердження сервера. Стан дошки оновлюється локально одразу після завершення перетягування. Якщо сервер повертає помилку, клієнт відкочує стан до збереженого знімка, зробленого перед операцією. Патерн усуває відчутну затримку між дією і реакцією інтерфейсу, критично важливу для операцій, що виконуються десятки разів на день [20].

2.2 Поведінкове моделювання засобами UML

Поведінкове моделювання системи засобами UML охоплює аспекти, які структурні діаграми не відображають: часову послідовність взаємодій, розгалуження алгоритмічних потоків і зміни станів сутностей [20].

Для системи Zent побудовано три діаграми – послідовності, діяльності та станів. Кожна з них прив'язана до конкретного процесу, а не до абстрактної функціональності. Вибір саме цих трьох типів не випадковий, оскільки діаграма послідовності фіксує, які компоненти обмінюються повідомленнями і в якому порядку.

Діаграма діяльності розкриває алгоритм дій всередині одного процесу. Діаграма станів показує, як змінюється сутність протягом усього свого існування. [21].

Діаграма послідовності: автентифікація користувача. Об'єктом моделювання було обрано саме автентифікацію , оскільки вона проходить крізь

усі архітектурні шари системи і добре ілюструє принципи їх взаємодії. Діаграма (див. рис. 2.10) містить шість учасників: користувача, клієнтський застосунок, API-рівень, шар бізнес-логіки, інфраструктурний шар і базу даних. Лінії часу позначено вертикальними пунктирними лініями, суцільні стрілки відповідають запитам, пунктирні – відповідям. Користувач заповнює форму та надсилає HTTP-запит на сервер. API-рівень валідує вхідні дані і передає запит до бізнес-логіки у вигляді команди. Обробник команди запитує дані користувача з бази, після чого інфраструктурний шар перевіряє пароль через хешування. Підтверджена автентифікація завершується генерацією JWT-токена, який повертається клієнту. Отримавши токен, застосунок зберігає його та перенаправляє користувача до захищеного розділу. Напрямки стрілок на діаграмі відображають ключову властивість Clean Architecture: залежності спрямовані лише від зовнішніх шарів до внутрішніх, зворотний напрямок неможливий [22].

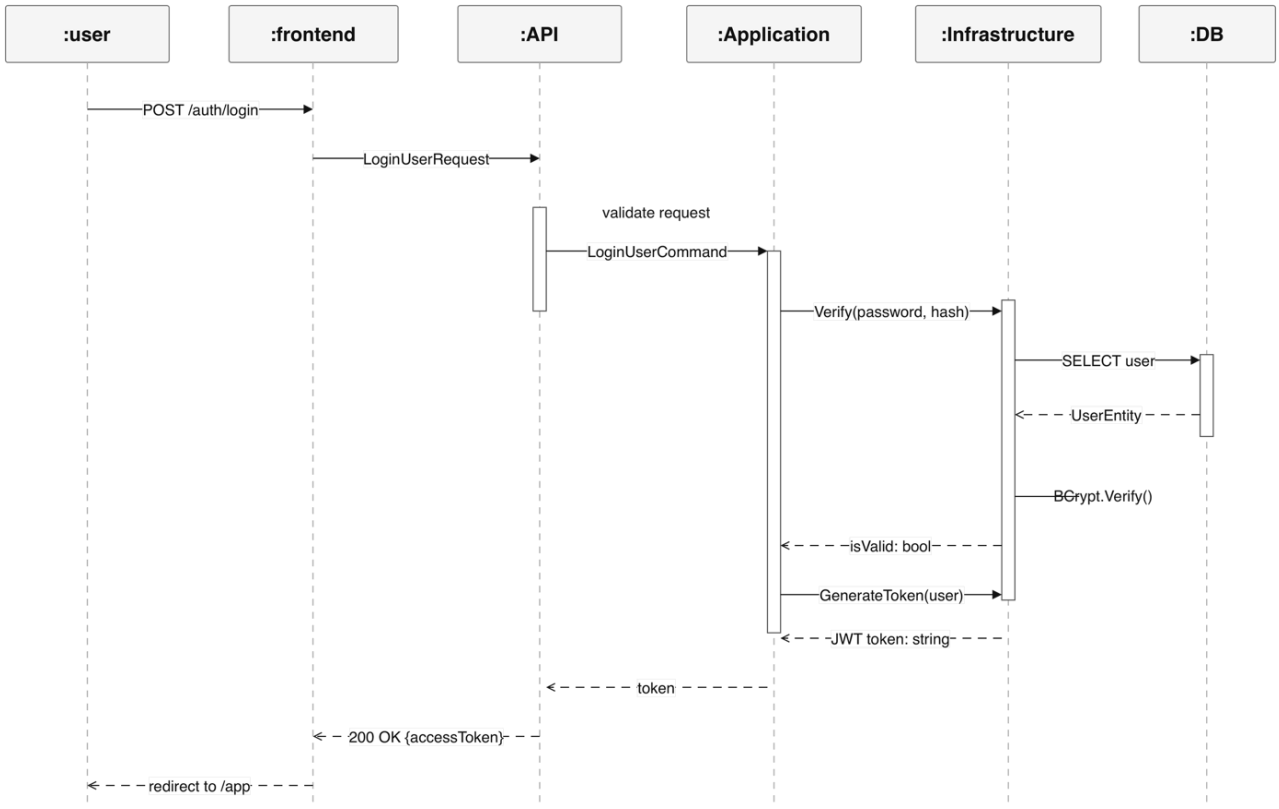

Рисунок 2.10 – Діаграма послідовності автентифікації користувача

Діаграма діяльності: переміщення задачі між колонками. Drag-and-drop переміщення задачі поєднує миттєве оновлення інтерфейсу, запис змін на сервері та відкат у разі помилки. З точки зору керування станом – один із найнетривіальніших процесів у системі. Нотація UML 2.5 передбачає початковий вузол (суцільне коло), фінальний вузол (концентричне коло), дії та вузли прийняття рішень [22].

Саме такою структурою побудовано діаграму на рисунку 2.11. Від початку перетягування клієнтська частина оновлює відображення задачі в реальному часі – без очікування серверної відповіді. Після завершення операції система визначає, чи змінилась колонка задачі. Залежно від результату виконується один із двох алгоритмів перепозиціонування, а порядкові номери задач у зачеплених колонках перераховуються. Сформований запит іде на сервер. Далі – розгалуження. Успішна відповідь запускає оновлення даних із сервера. Помилка повертає інтерфейс до знімку стану, збереженого перед початком перетягування.

Патерн Optimistic UI, реалізований у такий спосіб, широко застосовують у сучасних веб-застосунках для суб'єктивного пришвидшення взаємодії [23].

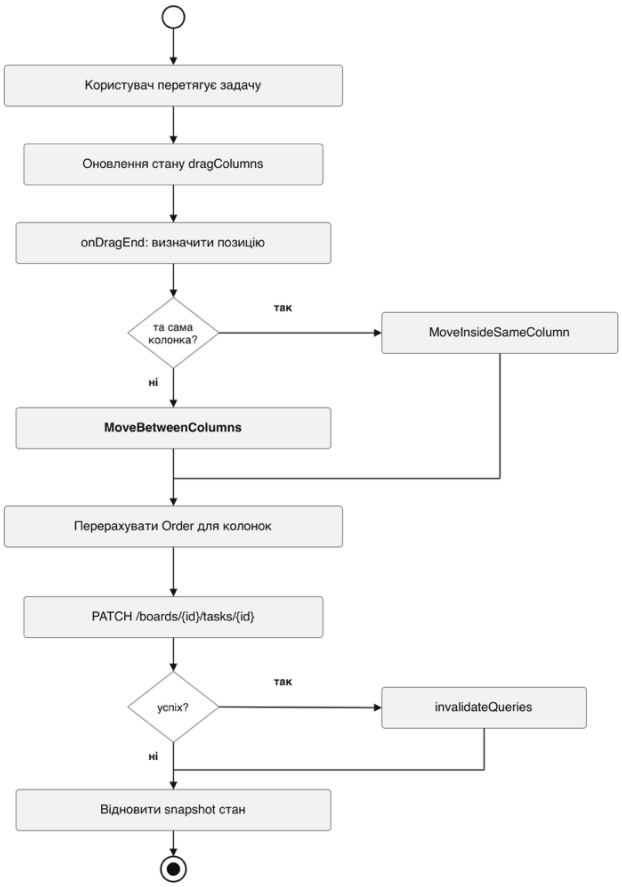

Рисунок 2.11 – Діаграма діяльності переміщення задачі між колонками

Діаграма станів: життєвий цикл дошки. Дошка (board) агрегує колонки та задачі й тому має виражену логіку переходів між станами. Саме тому вона обрана об'єктом для діаграми станів. Стани на рисунку 2.12 зображено прямокутниками із заокругленими кутами; у кожному виділено зону для дій входу (entry /) і поточних дій (do /). Переходи підписані подіями, що їх спричиняють [20].

Щойно створена, дошка фіксує лише назву. Додавання першої колонки переводить її у стан «Налаштовується» – стають доступними операції з колонками. Видалення всіх колонок повертає дошку назад. Перша задача переводить дошку до стану «Активна»: починається реальна робота з Kanban-потоком. Якщо всі задачі зрештою видалено, дошка повертається до «Налаштовується». Коли ж усі задачі опиняються у фінальній колонці – дошка переходить до «Завершена» і стає доступною лише для перегляду. Повторне відкриття будь-якої задачі повертає дошку в активний стан. Видалення дошки з будь-якого стану переводить її у термінальний стан «Видалена». Окремо варто

наголосити на архітектурній особливості: стан дошки не зберігається у базі даних окремим полем, а обчислюється динамічно зі стану дочірніх сутностей.

Діаграма моделює бізнес-логічний стан агрегату, а не технічний атрибут – підхід, характерний для предметно-орієнтованого проектування [21].

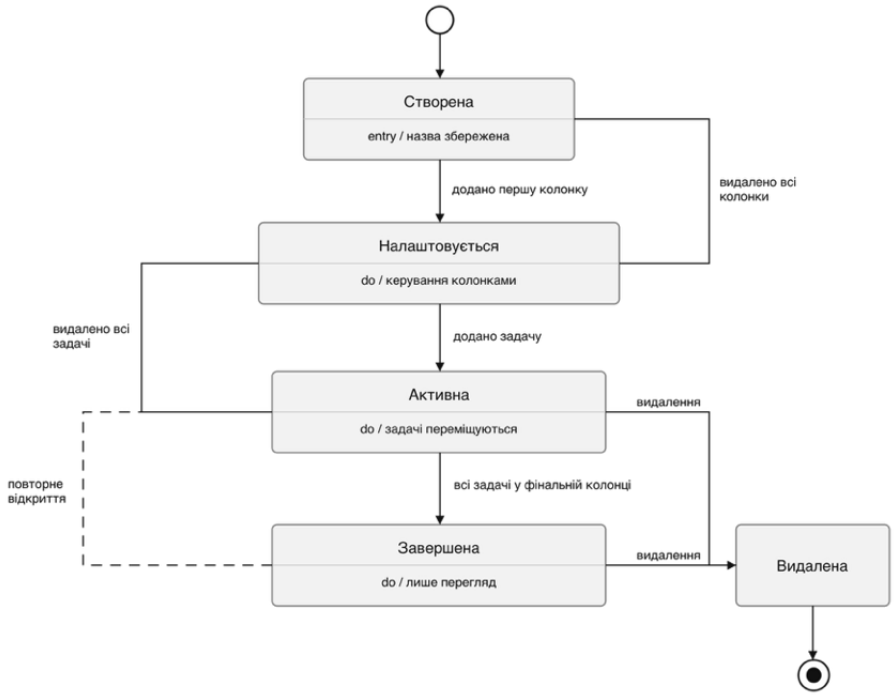

Рисунок 2.12 – Діаграма станів життєвого циклу дошки

Побудовані діаграми охоплюють поведінковий вимір системи Zent з різних кутів. Діаграма послідовності фіксує міжкомпонентний протокол автентифікації. Діаграма діяльності описує алгоритм операції з підтримкою оптимістичного оновлення. Діаграма станів документує повний life cycle ключової сутності.

Разом вони відповідають вимогам специфікації UML 2.5.1 до повноти поведінкового моделювання [20] і слугують основою для верифікації логіки на етапі проектування.

2.3 Проєктування системи розмежування доступу та бази даних

У Zent модель доступу не існує окремо від схеми бази даних. Обидва складники працюють разом: один описує права користувачів, другий фіксує місце зберігання правил доступу та спосіб їх перевірки під час запиту. Якщо розглядати рольову модель без структури таблиць, зникає практичний зв’язок між бізнес-логікою та даними.

У межах проєкту ролі користувачів не винесено в загальну властивість облікового запису. Вони належать до членства в конкретній команді. Один і той самий користувач може мати різні ролі в різних командах. База даних фіксує такі зв’язки окремо для кожного випадку, тому логіку не доводиться дублювати. Окремо працюють обмеження цілісності в СУБД. Вони стримують некоректні зміни навіть тоді, коли помилка виникає в прикладному коді. Вибір моделі контролю доступу Під час проєктування багатокористувацької системи потрібно визначити, хто отримує доступ до ресурсу і які дії для нього допустимі.

Теорія інформаційної безпеки пропонує для цього кілька моделей контролю доступу; найчастіше розглядають DAC, MAC і RBAC [24]. Для Zent найбільш придатною є RBAC-модель.

У командному SaaS-застосунку права зручніше призначати не окремим користувачам, а ролям. Адміністратор змінює набір прав один раз, після чого зміна поширюється на всіх учасників із відповідною роллю.

Такий підхід спрощує супровід системи, особливо після зростання команди. Класична RBAC-модель у стандарті NIST RBAC спирається на чотири елементи: користувачів, ролі, дозволи та сесії [24]. Користувач має роль. Роль об’єднує дозволи. Дозвіл описує конкретну дію над конкретним ресурсом. Логіка проста, але важлива: права не розкидані по окремих користувачах, а згруповані в ролях. Порівняно з DAC, RBAC зменшує навантаження на адміністрування. Якщо потрібно змінити доступ для групи учасників, достатньо відредагувати одну роль, а не десятки індивідуальних записів.

Контекстна RBAC у системі Zent. У Zent роль залежить від команди. Обліковий запис сам по собі не має постійного глобального статусу Owner, Admin або Member. Статус виникає лише в межах певної команди. Тому один користувач може бути Owner в одній команді й Member в іншій. Конфлікту між правами не виникає, бо система перевіряє не тільки ідентичність користувача, а й команду, у якій виконується дія.

У науковій літературі подібний підхід описують як context-aware RBAC: контекст авторизації охоплює суб'єкта та організаційне середовище операції [25]. У межах команди Zent використовує три ролі: Owner, Admin і Member. Owner з'являється під час створення команди або після передачі прав власника від іншого Owner. Власник має повноваження Admin, а також може виконувати дії з найбільшим ризиком: видаляти учасників і саму команду. Admin відповідає за керування учасниками, проєктами й дошками. Його призначає Owner. Member має базовий рівень доступу та працює переважно із задачами й колонками. Між ролями існує ієрархія: Owner → Admin → Member. Вищий рівень включає права нижчого рівня і додає власні. Ієрархія ролей узгоджується з принципом найменших привілеїв: кожна роль має лише ті повноваження, які потрібні для виконання відповідних функцій [24].

Дворівневе членство: команда і проєкт Zent розділяє членство в команді та членство в проєкті. Йдеться про два різні рівні доступу, хоча вони пов'язані між собою. Перший рівень описує TeamMember. Таблиця TeamMember фіксує належність користувача до команди й одночасно містить його роль. Наявність такого запису означає, що користувач входить до складу команди та має відповідний набір прав. Другий рівень описує ProjectMember. Тут фіксується зв'язок між користувачем і конкретним проєктом. Окремої ролі таблиця не містить, бо права беруться з TeamMember. Таке розділення потрібне для точнішого доступу всередині команди. Учасник може працювати з одним проєктом і не мати доступу до іншого. Межу між доступними й недоступними проєктами задає ProjectMember. Під час перевірки доступу застосунок рухається від ресурсу до проєкту, а потім до команди. Запис у ProjectMember показує, чи

має користувач доступ до проєкту. Роль у TeamMember уточнює, які саме дії він може виконувати. Разом два записи формують повну картину прав. Архітектура узгоджується з принципом defense in depth. Кожен рівень ієрархії працює як окремий бар'єр [25].

Нижче наведено діаграму (див. рис. 2.13), що ілюструє логіку перевірки доступу в системі Zent.

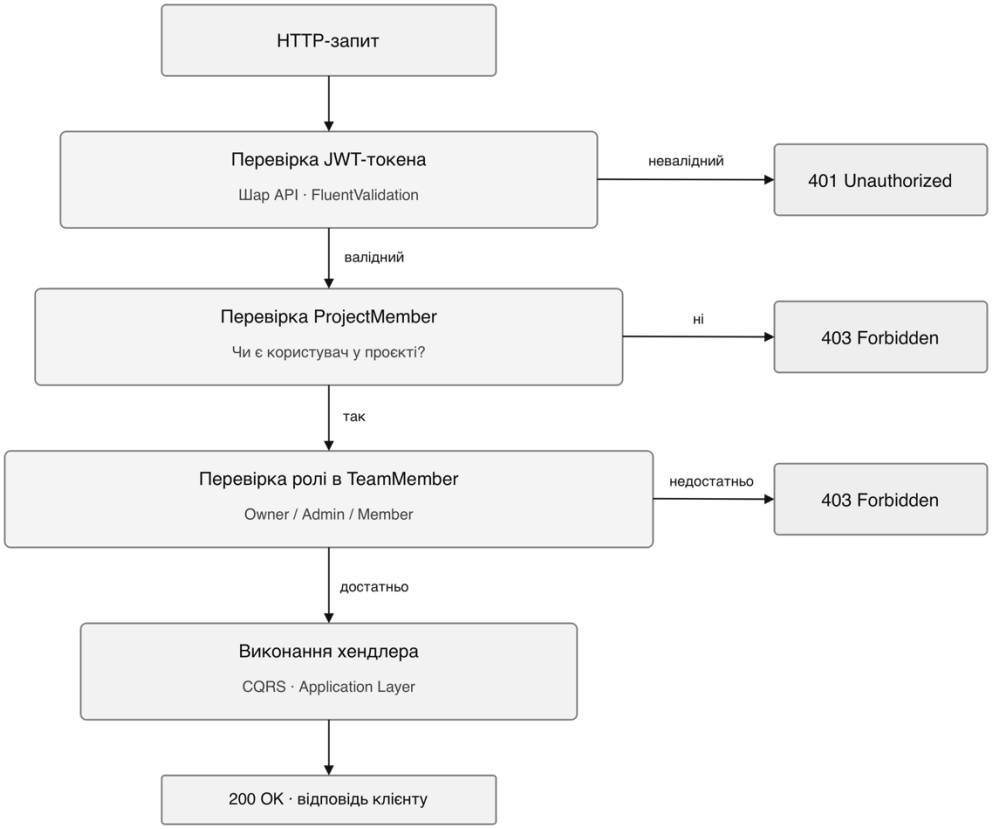

Рисунок 2.13 – Логіка перевірки доступу в системі Zent

Принципи організації схеми бази даних. Схема бази даних Zent повторює доменну модель. Для кожної сутності предметної області створено окрему таблицю. Між таблицями ієрархію задають зовнішні ключі. Для реляційної бази даних бажано, щоб структура таблиць якомога точніше відтворювала доменну модель. За такої відповідності код легше підтримувати, а зміни в схемі рідше призводять до помилок [26].

Для зберігання даних у Zent використано PostgreSQL. Обрана СУБД має відкритий код, підтримує розширені типи даних, транзакційну цілісність і складні JOIN-запити. Роботу між C#-кодом і базою даних виконує Entity Framework Core. Він перетворює LINQ-запити на SQL, керує міграціями й надає типобезпечний доступ до даних [27]. Первинні ключі в усіх таблицях мають тип UUID. Автоінкрементні INT-ідентифікатори частково показують кількість записів у таблиці, тоді як UUID такої інформації не відкриває. Є й інша перевага: застосунок може згенерувати новий ключ без попереднього запиту до бази даних [26].

Структура таблиць та зв’язки. Схема містить вісім основних таблиць. У таблиці users містяться облікові записи користувачів: ім’я, прізвище, унікальна електронна адреса та BCrypt-хеш пароля. Відкритий пароль до бази не потрапляє. Система зберігає лише хеш, тому відновити початкове значення неможливо. teams описує команди.

Кожна команда має назву. Зовнішні ключі в таблиці відсутні, оскільки команда є початковим елементом ієрархії. team_members пов’язує користувачів із командами. Окреме поле TeamRole зберігає роль учасника. Складений унікальний індекс на (user_id, team_id) не допускає ситуації, коли один користувач має дві ролі в одній команді. projects містить проєкти. Поле team_id пов’язує кожен проєкт із командою. Якщо команда видаляється, база даних каскадно прибирає всі її проєкти. project_members фіксує участь користувачів у проєктах. Роль у таблиці не зберігається, бо система бере права з team_members. Складений унікальний індекс на (user_id, project_id) захищає таблицю від дублювання записів. boards містить дошки.

Кожна дошка належить до певного проєкту через project_id. Каскадне видалення поширюється від проєкту до пов’язаних дошок. columns описує колонки на дошці. Таблиця містить зовнішній ключ board_id і числове поле order. Унікальний індекс на (board_id, order) не дозволяє створити дві колонки з однаковою позицією в межах однієї дошки. tasks зберігає задачі. До таблиці входять title, description, priority, order, column_id і необов’язковий assignee_id.

Якщо колонку видалено, база даних каскадно видаляє всі задачі, які були в ній розміщені.

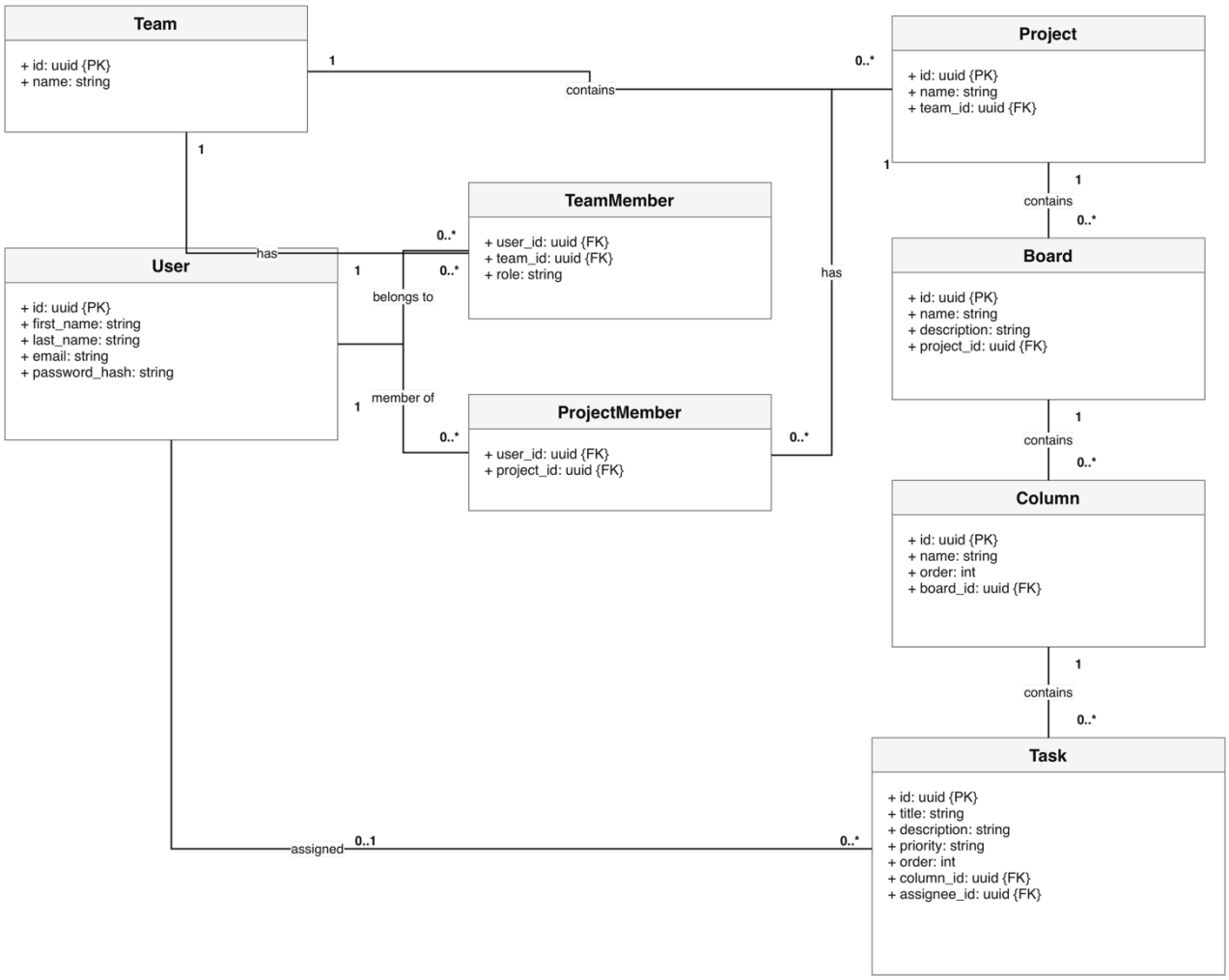

Рисунок 2.14 – ER-діаграма бази даних системи

Каскадне видалення та цілісність даних. Каскадне видалення працює на всіх рівнях ієрархії. Після видалення команди запускається ланцюг teams → projects → boards → columns → tasks. Якщо учасника прибрано з команди, база даних також видаляє його записи з project_members. У результаті в схемі не накопичуються осиротілі записи. Прикладний код не потребує окремої логіки очищення [27].

Обмеження цілісності FOREIGN KEY, UNIQUE і NOT NULL описані на рівні міграцій EF Core. PostgreSQL перевіряє їх незалежно від прикладного шару.

Некоректна транзакція не проходить навіть тоді, коли помилка міститься в коді хендлера. Дослідники наголошують на важливості багаторівневого захисту даних: обмеження на рівні СУБД залишаються останнім бар'єром перед пошкодженням даних [26].

У другому розділі було розглянуто проєктування системи Zent на архітектурному, поведінковому та структурному рівнях. Для серверної частини обрано Clean Architecture і CQRS, що розділяють відповідальності між шарами, зменшують зв'язність компонентів і спрощують розвиток системи. Клієнтську частину спроєктовано за методологією Feature-Sliced Design, де функціональні модулі ізольовані, а залежності мають чіткий напрям.

Поведінкові UML-діаграми деталізують ключові процеси: автентифікацію користувача, переміщення задачі між колонками та життєвий цикл дошки. Окремо спроєктовано модель розмежування доступу на основі контекстно-залежної RBAC і схему бази даних із використанням PostgreSQL, UUID-ідентифікаторів, зовнішніх ключів, унікальних обмежень і каскадного видалення.

У підсумку описані проєктні рішення формують технічну основу для реалізації, тестування та подальшого масштабування системи Zent.

3 РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Третій розділ охоплює практичну реалізацію системи Zent. Усі ключові архітектурні механізми та тестування функціональності на різних рівнях. Розподіл відповідальностей між шарами застосунку став одним із пріоритетів під час розробки, оскільки без нього підтримка та масштабування продукту в довгостроковій перспективі стають надмірно витратними.

Бізнес-логіка повністю відокремлена від деталей інфраструктури та рівня представлення – зміни в одному шарі не порушують контракти інших [28]. Платформою для реалізації обрано .NET 10 із власним CQRS-диспетчером, Minimal API ендпоінтами, Entity Framework Core для роботи з PostgreSQL, а також набором автоматизованих тестів обох типів.

3.1 Реалізація CQRS-диспетчера та шару бізнес-логіки

Центральним архітектурним рішенням у системі Zent стало застосування патерну CQRS (Command Query Responsibility Segregation). Патерн розмежовує операції читання (Query) і модифікації даних (Command) у незалежні потоки. Завдяки цьому кожен потік можна оптимізувати й тестувати окремо, а кожна операція інкапсульована у власному хендлері з єдиною відповідальністю [29]. Замість популярної бібліотеки MediatR було написано власний диспетчер – невеликий, повністю підконтрольний, без зайвих зовнішніх залежностей.

Абстракції команд, запитів та хендлерів. Фундамент CQRS-шару утворюють чотири інтерфейси, визначені у проекті Zent.Application. Маркери ICommand та ICommand<TResult> позначають команди – відповідно без результату і з ним. IQuery<TResult> позначає запит, що повертає дані. Хендлери реалізують один із трьох контрактів: ICommandHandler<TCommand>, ICommandHandler<TCommand, TResult> або IQueryHandler<TQuery, TResult> (див. лістинг 3.1):

Лістинг 3.1 – Програмний код

```
public interface ICommand;
public interface ICommand<out TResult>;
public interface IQuery<out TResult>;

public interface ICommandHandler<in TCommand>
    where TCommand : ICommand
{
    Task Handle(TCommand command, CancellationToken ct);
}

public interface IQueryHandler<in TQuery, TResult>
    where TQuery : IQuery<TResult>
{
    Task<TResult> Handle(TQuery query, CancellationToken ct);
}
```

DI-контейнер однозначно знаходить хендлер за типом команди чи запиту. Реєстрація відбувається через сканування збірки Zent.Application – новий хендлер не потребує жодного ручного запису в конфігурацію. Взаємодію між компонентами CQRS-шару ілюструє рисунок 3.1.

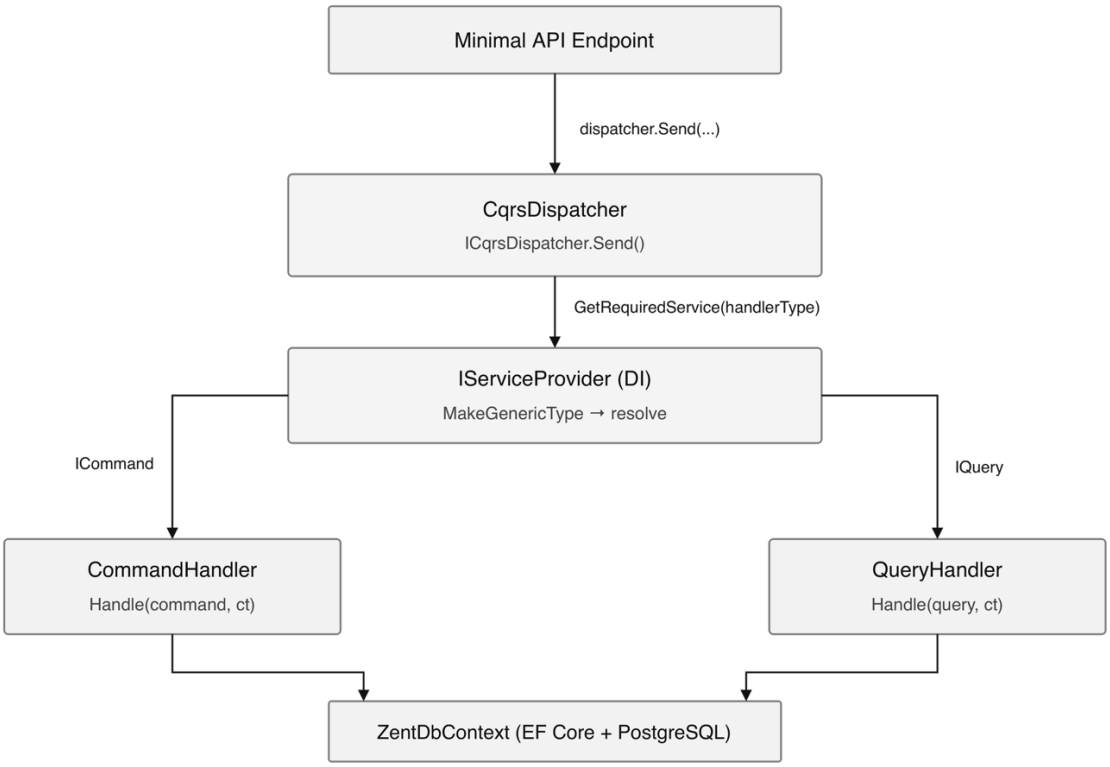

Рисунок 3.1 – Потік виконання операції через CQRS-диспетчер системи

Реалізація CqrsDispatcher. Клас CqrsDispatcher реалізує інтерфейс ICqrsDispatcher і містить три перевантаження методу Send: для команди без результату, для команди з результатом, для запиту. Алгоритм у кожному варіанті однаковий – за runtime-типом об'єкта система будує тип хендлера через MakeGenericType, отримує екземпляр із DI-контейнера і викликає Handle через рефлексію. Це демонструє лістинг 3.2.

Лістинг 3.2 – Програмний код

```
    public sealed class CqrsDispatcher(IServiceProvider services)
: ICqrsDispatcher
    {
        public async Task<TResult> Send<TResult>(
            IQuery<TResult> query, CancellationToken ct =
default)
        {
            var queryType   = query.GetType();
            var handlerType = typeof(IQueryHandler<,>)
                                .MakeGenericType(queryType,
typeof(TResult));
            try
            {
                var handler =
services.GetRequiredService(handlerType);
                var method  = handlerType.GetMethod("Handle");
                var task    =
(Task<TResult>)method!.Invoke(handler, [query, ct])!;
                return await task;
            }
            catch (InvalidOperationException)
            {
                throw new InvalidOperationException(
                    $"No handler registered for query
{queryType.Name}.");
            }
        }
    }
```

Застосування рефлексії є свідомим архітектурним вибором у цьому випадку. Альтернативою могла б бути кодогенерація або використання бібліотеки MediatR, яка вирішує ту саму задачу. Однак кодогенерація ускладнює читабельність проекту, а підключення MediatR – хоча і є стандартною практикою

– вносить додаткову зовнішню залежність, яка при використанні лише одного її аспекту є надлишковою. Власний диспетчер обсягом близько 60 рядків коду вирішує поставлену задачу повністю і залишається повністю під контролем розробника. Продуктивність при використанні рефлексії залишається прийнятною, оскільки GetMethod викликається один раз на тип запиту, а GetRequiredService делегується вбудованому DI-контейнеру .NET, що кешує фабрики сервісів при першій реєстрації [30]. Хендлери реєструються як Scoped-сервіси, що відповідає рекомендованому часу життя при роботі з DbContext у межах HTTP-запиту – кожен запит отримує власний екземпляр контексту бази даних, що усуває проблеми конкурентного доступу.

Хендлери організовані за принципом вертикальних зрізів: кожна бізнес-операція розміщена в окремій директорії, що містить команду або запит та відповідний хендлер. Наприклад, операція додавання дошки розміщена в Zent.Application/Features/Boards/AddBoard/ і включає AddBoardCommand.cs та AddBoardHandler.cs. Такий підхід контрастує з горизонтальним поділом, де всі команди лежать в одній папці, а всі хендлери – в іншій. При горизонтальному поділі навігація по коду стає громіздкою в міру зростання кількості фіч. Вертикальні зрізи натомість дозволяють знайти весь код, що стосується конкретної операції, в одному місці, не переходячи між директоріями.

Загалом шар Zent.Application містить хендлери для семи доменних областей: автентифікація (Auth), команди (Teams), проекти (Projects), дошки (Boards), колонки (Columns), задачі (Tasks) та користувачі (Users). Кожна область має від двох до восьми операцій, що у сукупності формують повний набір бізнес-можливостей системи.

Розглянемо типовий хендлер команди – AddBoardHandler. Його відповідальність включає: перевірку існування проекту; перевірку прав доступу (лише Owner або Admin команди можуть створювати дошки); перевірку унікальності назви дошки в межах проекту; створення сутності дошки разом із п'ятьма колонками за замовчуванням (Backlog, To Do, In Progress, Review, Done); збереження до бази даних і повернення ідентифікатора створеної дошки.

Важливо, що весь цей ланцюг виконується в межах одного виклику SaveChangesAsync, що гарантує атомарність операції – або дошка і всі п'ять колонок створені разом, або не створені взагалі (див. лістинг 3.3).

Лістинг 3.3 – Програмний код

```
    internal sealed class AddBoardHandler(ZentDbContext
dbContext)
        : ICommandHandler<AddBoardCommand, Guid>
    {
        public async Task<Guid> Handle(
            AddBoardCommand command, CancellationToken ct)
        {
            var project = await dbContext.Projects
                .AsNoTracking()
                .Where(x => x.Id == command.ProjectId)
                .Select(x => new { x.Id, x.TeamId })
                .FirstOrDefaultAsync(ct);

            if (project is null)
                throw new ProjectNotFoundException(
                    $"Project with id {command.ProjectId} was not
found.");

            var hasAccess = await dbContext.TeamMembers
                .AsNoTracking()
                .AnyAsync(x =>
                    x.TeamId == project.TeamId &&
                    x.UserId == command.UserId &&
                    (x.MemberRole == TeamRole.Owner ||
                     x.MemberRole == TeamRole.Admin), ct);

            if (!hasAccess)
                throw new TeamAccessDeniedException(
                    "You do not have permission to create boards
in this team.");
            return entity.Id;
        }
    }
```

Варто звернути увагу на використання проєкції .Select(x => new { x.Id, x.TeamId }) при читанні проекту. Замість завантаження всієї сутності через EF Core запитується лише два поля, необхідні для перевірки доступу. Це відповідає принципу мінімальних запитів до бази даних і зменшує обсяг переданих даних між СУБД і застосунком [31].

Такий підхід застосовується в усіх хендлерах системи Zent там, де з отриманих даних використовується лише підмножина полів.

Логіка переміщення колонок і задач. Найбільш складними з точки зору бізнес-логіки є хендлери MoveColumnHandler та MoveTaskHandler, що реалізують перебудову порядку елементів при drag-and-drop операціях у Kanban-дошці. Ці операції є критичними для коректності відображення: якщо порядок елементів після переміщення збережеться неправильно, користувач побачить дошку, що не відповідає його діям, і змушений буде перезавантажити сторінку.

Обидва хендлери дотримуються однієї стратегії: завантажити всі елементи поточного контексту, відсортовані за полем Order; видалити переміщуваний елемент із поточної позиції; вставити його на нову; перенумерувати всі елементи послідовно від 1. Такий підхід із повною перенумерацією є простішим і надійнішим порівняно зі спробою зсунути лише сусідні елементи: він гарантує, що після будь-якої кількості переміщень значення Order залишаються щільними цілими числами без прогалин і дублікатів.

Для MoveTaskHandler окремо обробляються два сценарії. Якщо переміщення відбувається всередині однієї колонки – перебудовується порядок лише в ній. Якщо задача переміщується між колонками – завантажуються задачі обох колонок і порядок перебудовується в кожній незалежно. Метод Math.Clamp застосовується до цільової позиції, щоб запобігти виходу за межі при граничних значеннях – наприклад, якщо клієнт надішле TargetOrder = 0 або значення, більше за кількість елементів, що демонструється на лістингу 3.4.

Лістинг 3.4 – Програмний код

```
if (sourceColumnId == targetColumnId)
{
    await MoveInsideSameColumnAsync(
        taskId, sourceColumnId, command.TargetOrder, ct);
}
else
{
    await MoveBetweenColumnsAsync(
        taskId, sourceColumnId, targetColumnId,
command.TargetOrder, ct);
```

```
}
await dbContext.SaveChangesAsync(ct);
```

Обидва внутрішні методи не викликають SaveChangesAsync самостійно – збереження виконується один раз після завершення будь-якого з них. Це гарантує атомарність операції переміщення: зміни або зберігаються повністю, або не зберігаються зовсім у разі виникнення помилки.

Розширення запитів через TeamMemberQueryExtensions. Авторизаційна модель системи Zent побудована на двох незалежних рівнях перевірки доступу. Перший рівень – команда (Team): операції зі створення проектів та дошок доступні лише членам із роллю Owner або Admin. Другий рівень – проект (Project): операції з читання і запису задач, переміщення колонок, перегляду дошки доступні будь-якому члену проекту незалежно від ролі в команді. Така дворівнева модель дозволяє гнучко розмежовувати адміністративні дії та повсякденну роботу з задачами.

Для повторного використання логіки перевірки першого рівня реалізовано статичний клас TeamMemberQueryExtensions із використанням нового синтаксису C# 14 – розширювальних методів через блок extension(...). Методи WithProjectManageAccess() та WithBoardManageAccess() фільтрують членів команди за роллю, залишаючи лише Owner та Admin. Оскільки методи повертають IQueryable<TeamMemberEntity>, фільтрація транслюється безпосередньо у SQL-запит і не відбувається на рівні .NET-процесу, що забезпечує оптимальну продуктивність (див. лістинг 3.5).

Лістинг 3.5 – Програмний код

```
public static class TeamMemberQueryExtensions
{
    extension(IQueryable<TeamMemberEntity> query)
    {
        public IQueryable<TeamMemberEntity>
WithProjectManageAccess()
            => query.Where(x =>
                x.MemberRole == TeamRole.Owner ||
                x.MemberRole == TeamRole.Admin);
    }
```

```
    }

    var hasAccess = await dbContext.TeamMembers
        .Where(x => x.TeamId == command.TeamId && x.UserId ==
command.UserId)
        .WithProjectManageAccess()
        .AnyAsync(ct);
```

Застосування таких розширень сприяє дотриманню принципу DRY: логіка перевірки ролей не дублюється в кожному хендлері, а виражена єдиним читабельним викликом методу з промовистою назвою. Це також підвищує супроводжуваність коду при зміні правил авторизації, наприклад при додаванні нової ролі Moderator із правом керувати проектами, достатньо оновити один метод у TeamMemberQueryExtensions, і зміна автоматично поширюється на всі хендлери, що його використовують.

Таким чином, шар бізнес-логіки системи Zent побудований на чітких абстракціях CQRS, власному легковаговому диспетчері та організації коду за вертикальними зрізами. Кожен хендлер є самодостатньою одиницею, що інкапсулює повний сценарій однієї бізнес-операції: від перевірки прав доступу до збереження результату. Подібна архітектура відповідає принципу єдиної відповідальності на рівні класів і принципу відкритості/закритості на рівні модулів – додавання нової операції не змінює жодного існуючого коду, а лише додає новий хендлер із відповідними командою або запитом. Саме така структура забезпечує як зручність підтримки, так і зручність ізольованого тестування кожної операції.

3.2 Реалізація інтерактивної канбан-дошки з drag-and-drop та оптимістичним UI

Канбан-дошка є центральним елементом інтерфейсу системи Zent. Вона відображає колонки з задачами, дозволяє переміщувати їх перетягуванням і відображає зміни миттєво, не очікуючи відповіді сервера. Реалізація цього

функціоналу охоплює три пов'язані задачі: організацію drag-and-drop взаємодії, синхронізацію стану з сервером та відкат змін у разі помилки.

Бібліотека dnd-kit та архітектура drag-and-drop. Для реалізації drag-and-drop обрано бібліотеку @dnd-kit у складі пакетів @dnd-kit/core, @dnd-kit/sortable та @dnd-kit/modifiers. Бібліотека побудована на основі нативного Pointer Events API і не залежить від застарілого HTML5 Drag and Drop API, що дає кращу підтримку сенсорних пристроїв і точніший контроль над поведінкою перетягування.

Компонент Board є кореневим для всієї дошки. Він рендерить DndContext із налаштованими обробниками подій та SortableContext для впорядкування колонок. Кожна колонка обгорнута у SortableBoardColumn, який надає їй ідентифікатор для сортування та слухачі подій. Аналогічно, кожна задача всередині колонки обгорнута у SortableTaskCard. Обробка перетягування розподілена між трьома подіями DndContext: onDragStart, onDragOver та onDragEnd. При старті фіксується знімок поточного стану дошки. Під час перетягування (onDragOver) стан оновлюється локально для живого відображення позиції елемента. При завершенні (onDragEnd) визначається тип операції і надсилається запит до сервера.

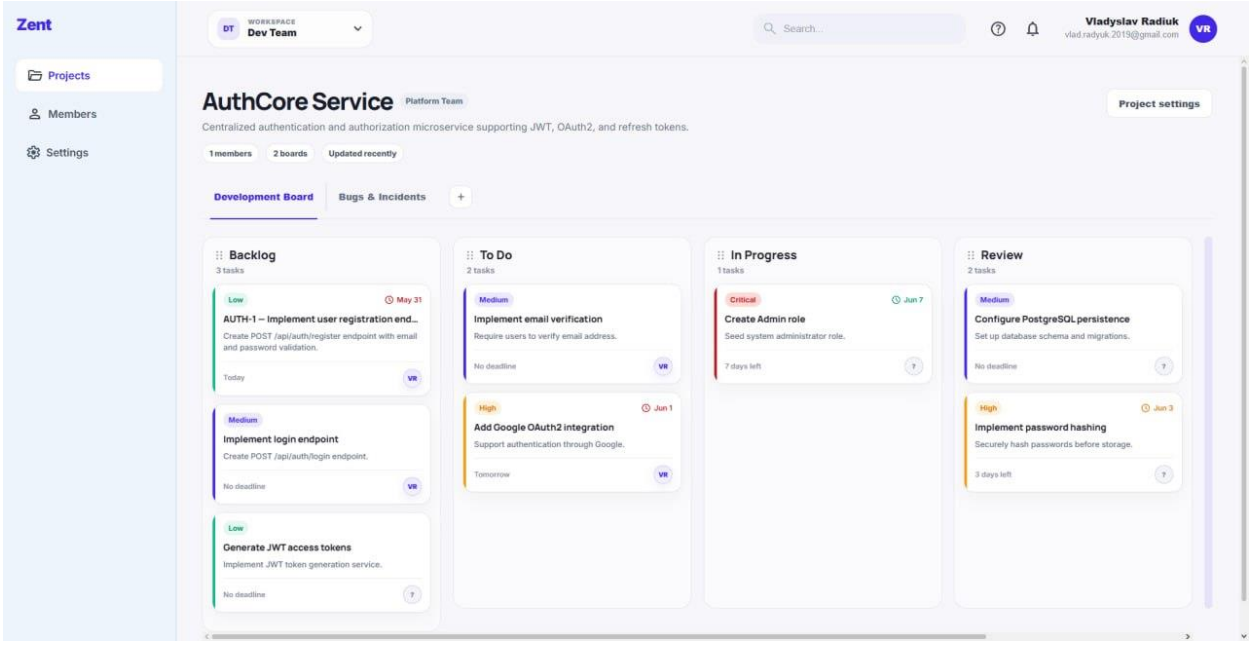

Рисунок 3.2 – Загальний вигляд канбан-дошки з колонками та задачами

Розмежування операцій переміщення. Система розрізняє два типи переміщення: переміщення колонки та переміщення задачі. Кожен тип обробляється окремим хуком і відповідним API-викликом.

Хук useMoveColumn викликає boardsApi.moveColumn() із новим порядковим номером цільової позиції. На сервері MoveColumnHandler завантажує всі колонки дошки, видаляє переміщувану з поточної позиції, вставляє її на нову і послідовно перенумеровує весь список. Таким чином, поле Order завжди зберігає щільну послідовність без прогалин.

Переміщення задачі обробляє хук useMoveTask, який викликає tasksApi.moveTask(). Сервер через MoveTaskHandler розрізняє два підсценарії. Якщо задача переміщується в межах однієї колонки, перенумеровується лише вона. Якщо задача переходить між колонками, перенумеровуються обидві зачеплені колонки, а поле ColumnId задачі оновлюється.

Оптимістичне оновлення UI є патерном, при якому інтерфейс відображає результат дії негайно, не чекаючи підтвердження від сервера. Це усуває відчутну затримку між завершенням перетягування і візуальною реакцією дошки, яка при мережевих запитах може становити від 100 до 500 мілісекунд і більше [34]. Реалізація спирається на механізм optimistic updates бібліотеки TanStack Query. Перед відправкою запиту до сервера хук скасовує активні запити до кешу дошки (cancelQueries), зберігає поточний знімок стану (getQueryData) і негайно записує оновлений стан до кешу (setQueryData). Інтерфейс перемальовується одразу, відображаючи нове розташування елемента. Уривок програмної реалізації продемонстровано на лістингу 3.6.

Лістинг 3.6 – Програмний код

```
    onMutate: async ({ columnId, targetOrder }) => {
      await queryClient.cancelQueries({ queryKey: ['board',
boardId] });
      const snapshot = queryClient.getQueryData(['board',
boardId]);

      queryClient.setQueryData(['board', boardId], (old) =>
        reorderColumns(old, columnId, targetOrder)
```

```
      );

      return { snapshot };
    },
    onError: (_err, _vars, context) => {
      queryClient.setQueryData(['board', boardId],
context.snapshot);
    },
    onSettled: () => {
      queryClient.invalidateQueries({ queryKey: ['board',
boardId] });
    },
```

У колбеку onError кеш відновлюється зі збереженого знімка, що повертає дошку до стану до початку перетягування. Колбек onSettled викликається незалежно від результату і інвалідує кеш, ініціюючи свіжий запит до сервера для синхронізації фінального стану.

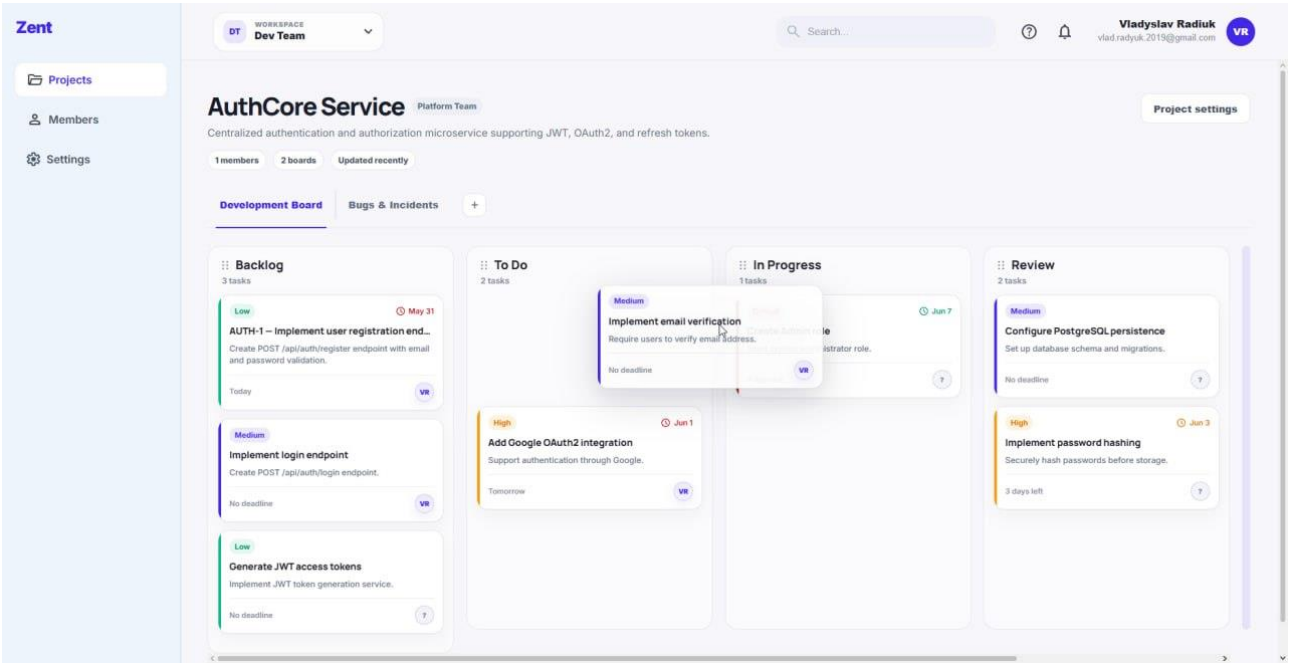

Рисунок 3.3 – Процес перетягування задачі між колонками з відображенням активного стану

Компонентна структура дошки. Компонент Board отримує дані через хук useBoard, який виконує запит GET /api/boards/:id і повертає об'єкт із масивом колонок та вкладеними масивами задач. Локальний стан колонок зберігається

окремо у useState і синхронізується з серверними даними через useEffect. Це дозволяє вносити тимчасові зміни до локального стану під час перетягування, не торкаючись кешу TanStack Query безпосередньо з компонента.

Кожна колонка рендериться компонентом BoardColumn, який містить заголовок, кнопки редагування і видалення, а також список задач через SortableContext. Задачі відображаються компонентом TaskCard, який показує назву, пріоритет, дату та аватари виконавців. SortableTaskCard обгортає TaskCard і додає до нього drag-handle та атрибути сортування від @dnd-kit/sortable.

Додавання нової колонки винесено в окремий компонент AddColumnRail, що рендериться праворуч від усіх колонок. Після введення назви і підтвердження викликається хук useCreateColumn, який через boardsApi.addColumn() відправляє запит і після успіху інвалідує кеш дошки.

Модифікатори та обмеження перетягування. DndContext налаштований із модифікатором restrictToWindowEdges із пакету @dnd-kit/modifiers. Він обмежує переміщення drag-overlay у межах вікна браузера, запобігаючи ситуації, коли елемент виходить за видиму область під час перетягування. Для колонок додатково застосовується горизонтальне обмеження осі, оскільки колонки впорядковуються лише по горизонталі, тоді як задачі всередині колонки переміщуються вертикально.

Реалізована канбан-дошка поєднує декларативну архітектуру компонентів React, реактивне керування серверним станом через TanStack Query та плавну drag-and-drop взаємодію через @dnd-kit. Оптимістичне оновлення разом із механізмом відкату забезпечує відчуття миттєвої реакції інтерфейсу при збереженні консистентності даних.

3.3 Опис екранів та інтерфейсу застосунку

Інтерфейс системи Zent організовано за принципом ієрархічної навігації: від списку команд через проєкти до конкретної дошки з задачами. Кожен екран відповідає одному рівню цієї ієрархії і надає користувачу лише ті дії, що доречні

на відповідному рівні. Навігація між екранами реалізована через React Router з вкладеними маршрутами, тому перехід між розділами не перезавантажує сторінку.

Екрани входу і реєстрації є першими, що бачить новий користувач. Форма входу містить поля для email та пароля, кнопку підтвердження та посилання на реєстрацію. Фоновий відеоролик створює візуальний контекст і відрізняє сторінки автентифікації від решти застосунку. Помилки автентифікації відображаються безпосередньо під формою без перезавантаження сторінки. Після успішного входу токен зберігається у localStorage і користувач перенаправляється до списку команд.

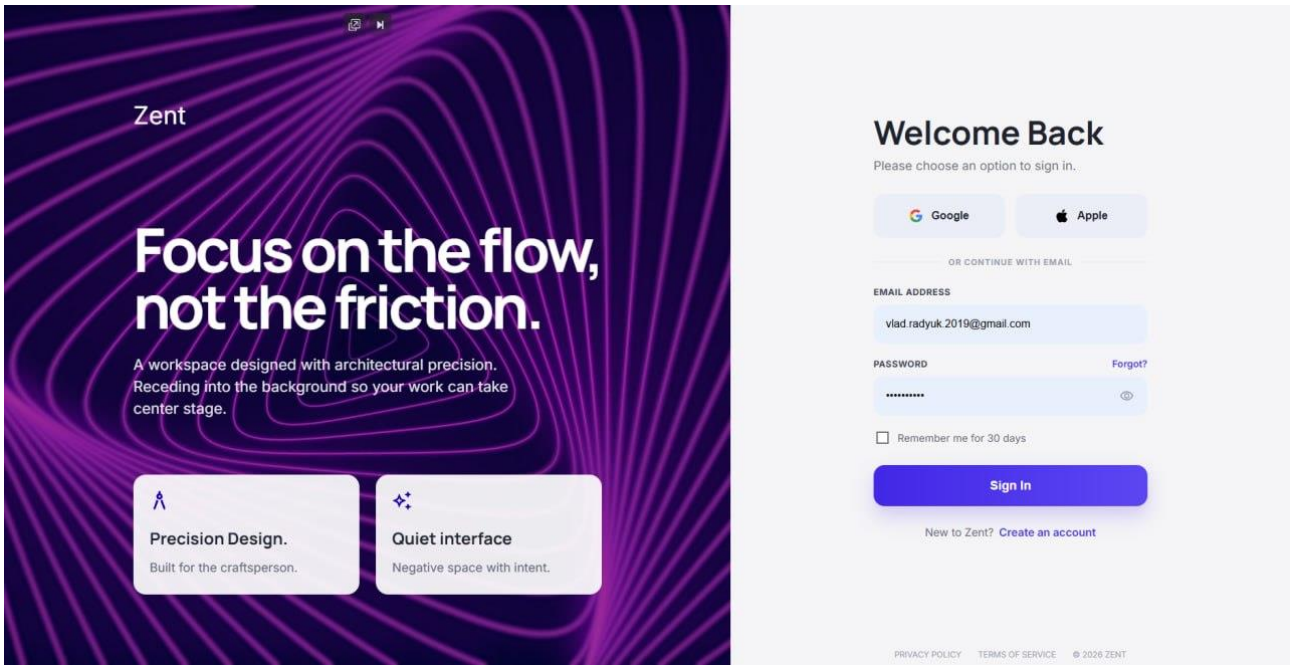

Рисунок 3.4 – Екран входу до системи

Екран реєстрації містить чотири поля: ім'я, прізвище, email та пароль. Валідація відбувається на двох рівнях: на клієнті через validation.ts і на сервері через FluentValidation. Якщо email вже зареєстрований, користувач отримує повідомлення про конфлікт без технічних деталей помилки. Фоновий відеоролик на сторінці реєстрації відрізняється від екрану входу.

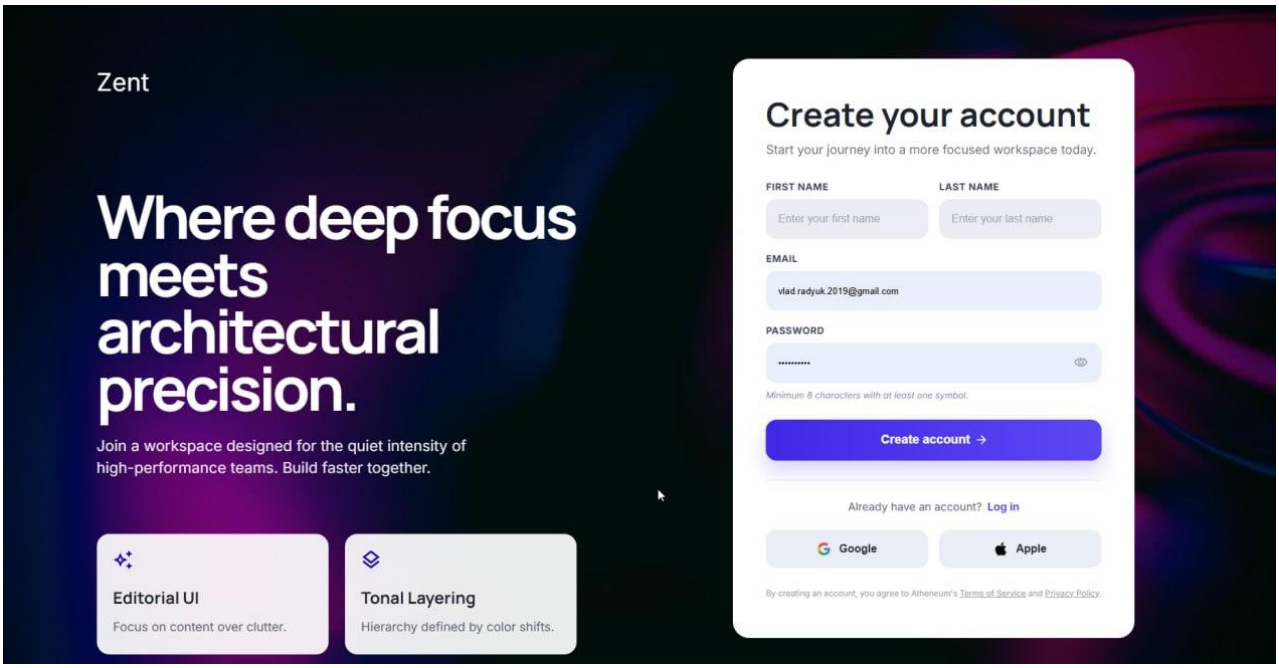

Рисунок 3.5 – Екран реєстрації нового користувача

Після входу користувач потрапляє на головну сторінку із переліком команд, членом яких він є. Кожна команда відображається карткою з назвою та роллю поточного користувача у цій команді. Кнопка створення нової команди розташована у верхній частині списку і відкриває модальне вікно CreateTeamModal. Якщо користувач ще не є членом жодної команди, відображається порожній стан із ілюстрацією та закликом до дії. Клік на картку команди переводить до робочого простору цієї команди.

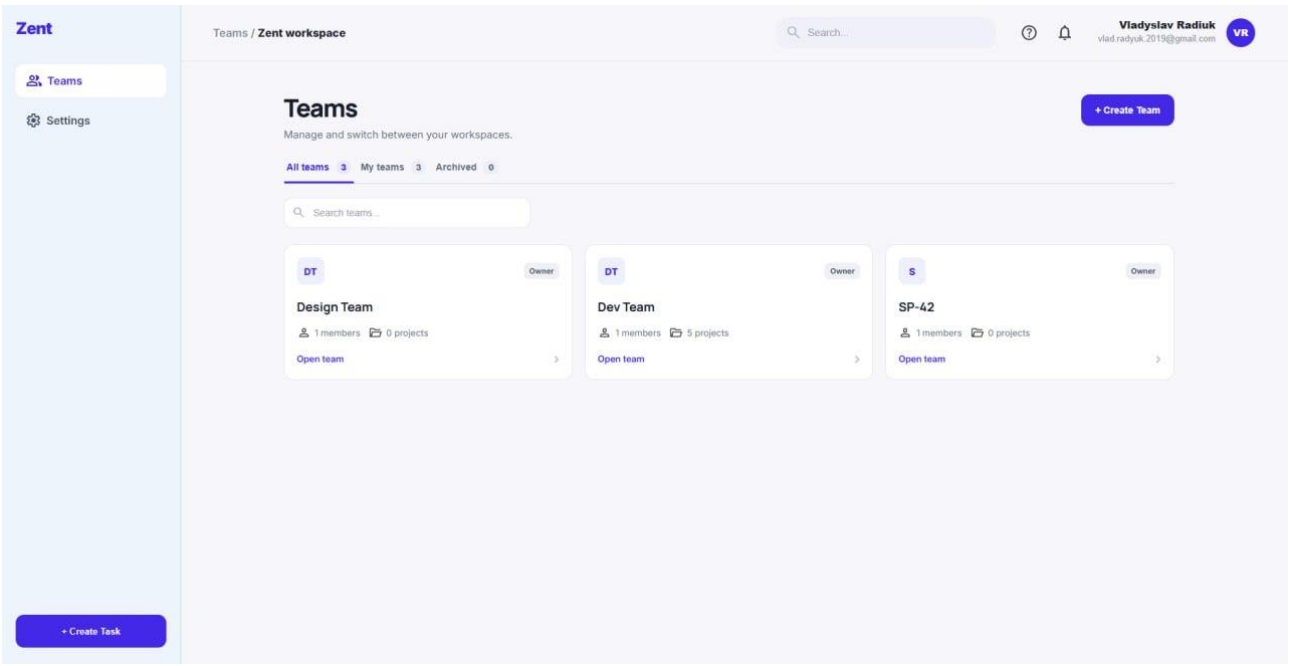

Рисунок 3.6 – Екран списку команд користувача

Після переходу до команди відкривається TeamWorkspaceLayout із бічною панеллю навігації та основним контентом. Бічна панель містить назву команди, посилання на проєкти, учасників та налаштування. Основна область відображає список проєктів команди, впорядкованих за датою створення від найновішого. Кожен проєкт представлений карткою ProjectCard із назвою, описом та кількістю дошок. Кнопка запрошення нового учасника доступна лише користувачам із роллю Admin або Owner і відкриває модальне вікно InviteTeamMemberModal із пошуком за іменем або email.

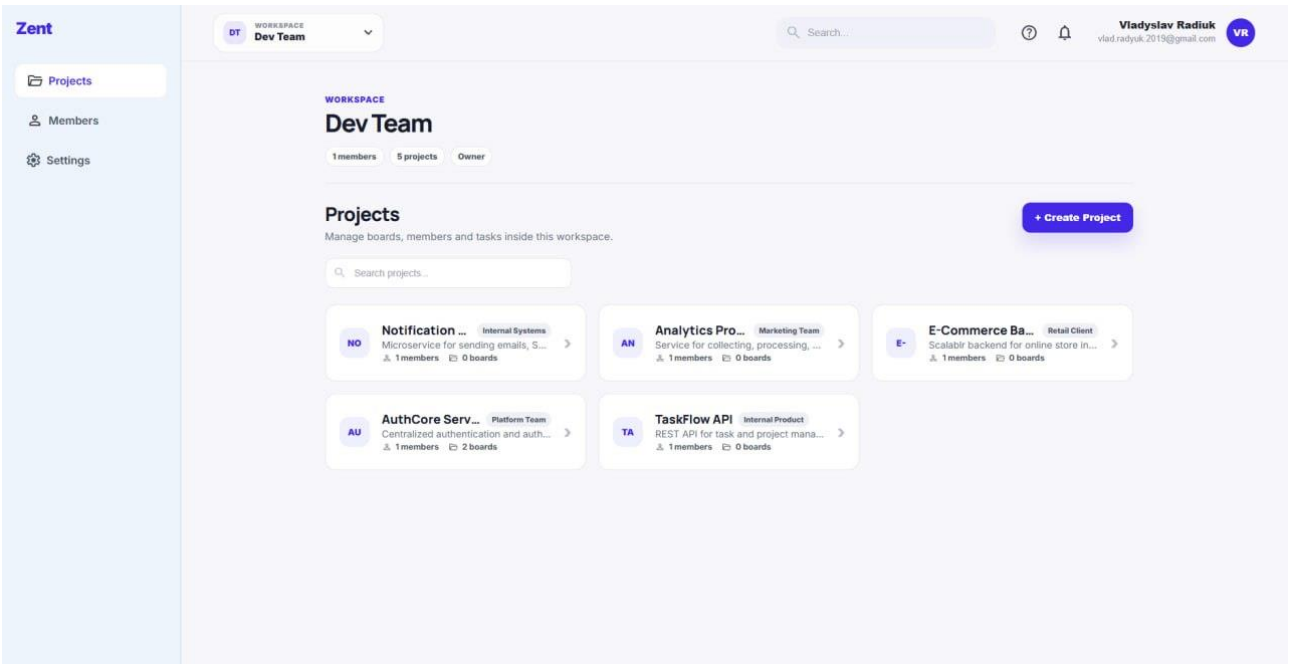

Рисунок 3.7 – Робочий простір команди зі списком проєктів

Сторінка проєкту ProjectPage відображає заголовок із назвою і описом проєкту, список дошок та перелік учасників. Дошки представлені картками, клік на яку відкриває повноцінний канбан-інтерфейс. Кнопка створення нової дошки доступна учасникам із достатніми правами і відкриває модальне вікно CreateBoardModal із полями для назви та опису. Після створення дошка одразу з'являється у списку без перезавантаження сторінки завдяки інвалідації кешу TanStack Query. У правій частині заголовку відображається список учасників проєкту.

Екран дошки є основним робочим простором системи. Колонки відображаються горизонтально і займають всю доступну ширину екрану з горизонтальним прокручуванням при великій кількості колонок. Кожна колонка містить заголовок, кількість задач у ній та список карток. Картка задачі TaskCard відображає назву, пріоритет у вигляді кольорового індикатора, дату дедлайну та аватар виконавця. Колонки і задачі можна переміщувати перетягуванням – під час переміщення елемент набуває напівпрозорого вигляду, а цільова позиція підсвічується.

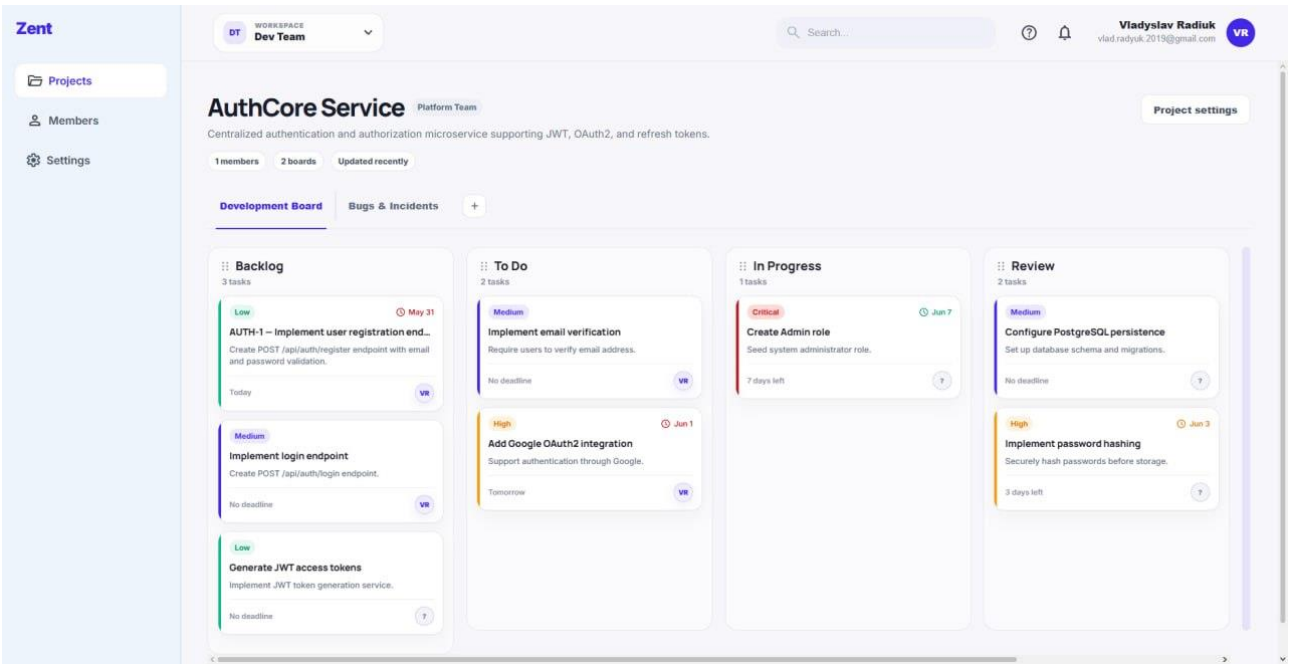

Рисунок 3.8 – Екран канбан-дошки з колонками та задачами

Праворуч від останньої колонки завжди відображається компонент AddColumnRail із кнопкою додавання нової колонки. Клік активує поле введення назви безпосередньо у рейлі без відкриття модального вікна. Кожна колонка має кнопки редагування назви та видалення у заголовку. Видалення колонки запитує підтвердження через DeleteColumnModal, оскільки операція каскадно видаляє всі задачі у ній.

Деталі задачі відкриваються на окремій сторінці TaskDetailsPage за маршрутом /app/:teamId/tasks/:taskId. Сторінка відображає повну інформацію про задачу: назву, опис, пріоритет, дату дедлайну, автора та виконавців. Секція виконавців реалізована компонентом TaskAssigneeControl, який дозволяє додавати і видаляти виконавців через пошук серед учасників проєкту. Кожна зміна виконавця відправляє окремий запит до API і одразу відображається в інтерфейсі без перезавантаження сторінки.

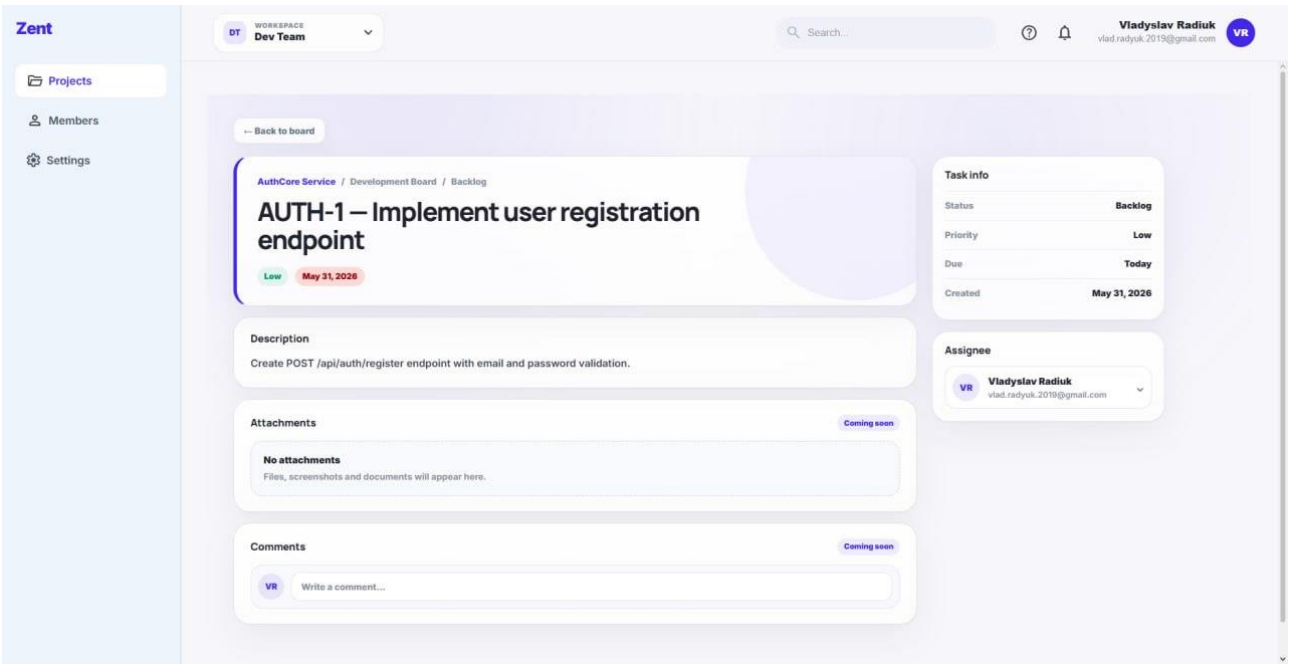

Рисунок 3.9 – Екран деталей задачі

Верхня панель Topbar присутня на всіх захищених сторінках і містить назву поточного розділу та меню користувача UserMenu. Меню відображає ім'я та email поточного користувача, отримані через useCurrentUser, і надає кнопку виходу, яка очищає токен із localStorage і перенаправляє на сторінку входу. Бічна панель Sidebar у робочому просторі команди відображає навігаційні посилання з іконками та підсвічує активний розділ. Загальний макет AppLayout і TeamWorkspaceLayout забезпечують консистентне розташування елементів на всіх сторінках застосунку.

3.4 Тестування програмного забезпечення

Тестування системи Zent організовано у двох окремих проєктах: Zent.Unit.Tests та Zent.Integration.Tests. Кожен із них вирішує свою задачу. Юніт-тести перевіряють бізнес-логіку обробників ізольовано, без залежності від зовнішніх сервісів. Інтеграційні тести перевіряють повний цикл запиту – від HTTP-ендпоінту до бази даних і назад. Обидва проєкти побудовані на xUnit як тестовому фреймворку та FluentAssertions для виразних перевірок результату.

Юніт-тести покривають обробники команд і запитів шару Zent.Application. Кожен тест ізольований: він отримує власний екземпляр бази даних через TestDbContextFactory (див. лістинг 3.7), яка створює ZentDbContext на основі EF Core InMemory провайдера з унікальним іменем для кожного запуску. Завдяки цьому тести не впливають один на одного і можуть виконуватися паралельно.

Лістинг 3.7 – Програмний код

```
public static class TestDbContextFactory
{
    public static ZentDbContext Create()
    {
        var options = new
DbContextOptionsBuilder<ZentDbContext>()
            .UseInMemoryDatabase(Guid.NewGuid().ToString())
            .Options;

        return new ZentDbContext(options);
    }
}
```

Для обробників, що залежать від зовнішніх сервісів – IPasswordHasher та ITokenService – використовується бібліотека Moq. Вона дозволяє підмінити реальну реалізацію тестовою заглушкою і перевірити, що метод був викликаний із очікуваними аргументами.

Для конструювання тестових сутностей у кожному проєкті реалізовано окремий набір Builder-класів. Замість ручного заповнення всіх полів entity в кожному тесті, Builder надає розумні значення за замовчуванням і дозволяє задати лише ті атрибути, що важливі для конкретного сценарію.

Покриття юніт-тестами охоплює такі модулі: автентифікацію (LoginUserHandler, RegisterUserHandler), команди і запити для команд (AddTeamHandler, GetTeamHandler, GetUserTeamsHandler), проєктів (AddProjectHandler, GetTeamProjectsHandler), дошок (AddBoardHandler, GetBoardHandler), задач (AddTaskHandler) та користувачів (GetCurrentUserHandler).

Кожен тестовий клас охоплює декілька сценаріїв: успішне виконання, відмову при відсутності доступу та відмову при некоректних вхідних даних. Наприклад, для AddBoardHandler перевіряється, що дошка зі стандартними колонками створюється при правильній ролі, і що виключення TeamAccessDeniedException кидається при ролі Member.

Рисунок 3.10 – Результати виконання юніт-тестів у середовищі розробки

Окремої уваги заслуговують тести автентифікації, оскільки вони перевіряють не лише результат, а й поведінку: чи дійсно був викликаний IPasswordHasher, чи не викликався ITokenService у разі невірного пароля. Такі перевірки гарантують, що логіка залишається коректною навіть при рефакторингу внутрішньої реалізації.

Інтеграційні тести перевіряють систему як єдине ціле. Запит надходить через реальний HTTP-клієнт, проходить крізь усі шари – валідацію, бізнес-логіку, базу даних – і повертає HTTP-відповідь, яку тест перевіряє за статус-кодом і тілом.

В основі інфраструктури лежить IntegrationTestWebAppFactory, що успадковує WebApplicationFactory<Program>. Вона запускає повноцінний ASP.NET Core хост у пам'яті та підмінює рядок підключення на тестову базу даних PostgreSQL із конфігурації appsettings.Testing.json.

Управління життєвим циклом тестового середовища покладено на TestHostFixture, реалізацію IAsyncLifetime. При ініціалізації fixture виконує міграції, засіває тестового користувача через TestUserSeeder і налаштовує DbCleaner. Усі тестові класи об'єднані в одну xUnit-колекцію [Collection("Integration")] і поділяють один екземпляр fixture – це суттєво скорочує час запуску тестів.

Між кожним тестом база даних скидається до початкового стану через бібліотеку Respawn. На відміну від повного дропу та відтворення бази, Respawn видаляє лише дані з таблиць, зберігаючи схему. Це значно швидше і дозволяє тестам починати з чистого, але передбачуваного стану.

Покриття інтеграційними тестами охоплює автентифікацію, команди, проєкти, дошки, задачі та поточного користувача. Кожен тестовий клас містить Helper-метод для засівання необхідних даних безпосередньо через ZentDbContext, отриманий із DI-контейнера тестового хоста.

Рисунок 3.11 – Результати виконання інтеграційних тестів

Поєднання юніт-тестів на InMemory базі з інтеграційними тестами на реальному PostgreSQL забезпечує двошарове покриття: швидку перевірку бізнес-логіки та надійну верифікацію поведінки системи в умовах, максимально наближених до реальних.

У третьому розділі розкрито практичне створення системи Zent: від серверної архітектури до перевірки готового продукту.

CQRS-диспетчер підтвердив доцільність власного підходу без підключення сторонньої бібліотеки. Чотири базові інтерфейси разом із класом CqrsDispatcher виконують маршрутизацію команд і запитів та залишають код прозорим для аналізу. Команда проєкту згрупувала хендлери за принципом вертикальних зрізів: код окремої бізнес-операції розміщується в одній директорії. Навігація спрощується. Авторизаційна модель спирається на два незалежні рівні перевірки доступу, а розширення TeamMemberQueryExtensions усувають дублювання логіки.

Канбан-дошка поєднує drag-and-drop взаємодію через @dnd-kit, синхронізацію стану із сервером через TanStack Query та оптимістичні оновлення з механізмом відкату. Алгоритм повної перенумерації елементів після переміщення виявився надійнішим за локальний зсув сусідніх позицій.

Інтерфейс системи побудований за принципом ієрархічної навігації. Кожен екран репрезентує окремий логічний рівень і містить лише дії, релевантні поточному контексту користувача. React Router із вкладеною маршрутизацією переводить користувача між рівнями без перезавантаження сторінки, тому взаємодія з застосунком сприймається швидшою.

Тестова стратегія охоплює два рівні. Юніт-тести на основі EF Core InMemory провайдера перевіряють ізольовану поведінку хендлерів, зокрема граничні сценарії та очікувані помилки. Інтеграційні тести проходять шлях від HTTP-запиту до бази даних із використанням PostgreSQL та Respawn для скидання стану між перевірками.

Практична реалізація Zent показує узгодженість між архітектурними підходами й програмним втіленням системи. Обрані технічні рішення підтримують функціональну коректність поточної версії та створюють умови для масштабування без різкого ускладнення супроводу.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ, ОСНОВИ ОХОРОНИ ПРАЦІ

Діяльність розробника програмного забезпечення, попри відсутність явних фізичних небезпек, характеризується специфічним комплексом ризиків: нервово-емоційним напруженням, зоровим навантаженням і гіподинамією. Використання електричного обладнання додатково формує вимоги до організації робочого місця та виробничого середовища. У цьому розділі розглядаються види діяльності та їх ризики в контексті безпеки праці, а також нормативні вимоги до обладнання й організації технологічного процесу розробки системи Zent.

4.1 Діяльність. Її види та розуміння в безпеці праці

Діяльність є фундаментальною формою активності людини, спрямованою на перетворення навколишнього середовища та досягнення свідомо поставлених цілей. На відміну від інстинктивної поведінки тварин, людська діяльність визначається не лише біологічними потребами, а й соціальними мотивами, знаннями та досвідом. У контексті безпеки праці діяльність розглядається як процес взаємодії людини з виробничим середовищем, що супроводжується певним рівнем небезпеки та ризику [32].

Розуміння видів діяльності є необхідною передумовою для правильної організації охорони праці, оскільки кожен вид формує специфічний профіль небезпек і навантажень на організм працівника. Некоректна оцінка характеру діяльності призводить до неправильного визначення заходів захисту і, як наслідок, до виникнення професійних захворювань або нещасних випадків.Класифікація видів діяльності. У науці про безпеку життєдіяльності прийнято кілька підходів до класифікації діяльності. За характером переважного навантаження на організм розрізняють фізичну та розумову працю; за відношенням до виробничого процесу – основну, допоміжну та управлінську; за ступенем автоматизації – ручну, механізовану та автоматизовану [32].Фізична праця характеризується переважним навантаженням на опорно-руховий апарат і

серцево-судинну систему. Основним показником інтенсивності фізичної праці є енерговитрати: легка фізична праця потребує до 150 ккал/год, важка — понад 400 ккал/год. Небезпеки фізичної праці пов'язані передусім із надмірним навантаженням на м'язи та суглоби, несприятливими мікрокліматичними умовами і ризиком травматизму від взаємодії з обладнанням.Розумова праця пов'язана з прийомом та переробкою інформації і потребує напруження уваги, пам'яті та логічного мислення. Незважаючи на відносно невисокі енерговитрати, розумова праця може спричиняти значне функціональне напруження нервової системи. Особливістю розумової праці є те, що суб'єктивне відчуття втоми нерідко запізнюється відносно реального зниження продуктивності – людина продовжує працювати в умовах прихованого перевтомлення, не усвідомлюючи цього.

Розумова праця в контексті розробки програмного забезпечення. Діяльність розробника є яскравим прикладом складної розумової праці з елементами творчої та аналітичної роботи. Проєктування архітектури системи, написання програмного коду, налагодження алгоритмів і тестування – кожна з цих операцій потребує тривалої концентрації уваги та утримання у робочій пам'яті великих обсягів контексту.Характерною особливістю такої діяльності є нерівномірність розподілу навантаження протягом робочого дня: фази інтенсивної концентрації чергуються з відносно менш напруженими періодами. Однак саме під час найбільш складних завдань – наприклад, при пошуку причини помилки або проєктуванні нового модуля – ризик перевтомлення є найвищим, оскільки внутрішня мотивація утримує розробника за роботою всупереч сигналам втоми [33].

Небезпечні та шкідливі чинники розумової праці. Безпека праці оцінюється через призму небезпечних і шкідливих виробничих чинників. Для розумової праці оператора ПК характерні такі групи чинників. Психофізіологічні чинники охоплюють нервово-емоційне напруження, монотонність праці при виконанні одноманітних операцій, а також гіподинамію – зниження рухової активності нижче фізіологічно необхідного рівня. Тривала гіподинамія

уповільнює обмінні процеси, знижує тонус м'язів і підвищує ризик серцево-судинних захворювань.

Фізичні чинники включають вплив електромагнітних полів від монітора та іншого обладнання, підвищений рівень шуму від систем охолодження комп'ютерів, а також несприятливі параметри мікроклімату – підвищену температуру і знижену вологість повітря в приміщеннях із великою кількістю комп'ютерної техніки. Зорове навантаження виділяється в окрему групу чинників через його визначальний вплив на функціональний стан оператора: тривала робота з монітором спричиняє акомодаційну втому, яка при систематичному впливі може призвести до стійкого погіршення зору.

Принцип допустимого ризику в організації діяльності. Сучасна концепція безпеки праці виходить із того, що абсолютно безпечної діяльності не існує – будь-яка праця пов'язана з певним рівнем ризику. Завданням охорони праці є не усунення ризику взагалі, а його зниження до допустимого рівня, при якому ймовірність негативних наслідків для здоров'я працівника є мінімальною.

Для розумової праці принцип допустимого ризику реалізується через нормування тривалості безперервної роботи, встановлення вимог до параметрів мікроклімату та освітлення, ергономічну організацію робочого місця і проведення медичних оглядів. Кожен із цих заходів спрямований на конкретну групу чинників і разом вони формують систему захисту, що знижує кумулятивний ризик до прийнятного рівня.Роль свідомості та мотивації в безпеці праці. Особливістю людської діяльності є її свідомий характер: працівник здатний усвідомлювати небезпеку і цілеспрямовано уникати її. Проте практика показує, що мотивація до дотримання вимог безпеки нерідко поступається мотивації до досягнення виробничого результату. Розробник, що прагне завершити завдання у встановлений термін, схильний ігнорувати сигнали втоми і пропускати регламентовані перерви.

Подолання цієї суперечності є одним із завдань управління охороною праці на підприємстві: формування культури безпеки, за якої дотримання вимог охорони праці сприймається не як обмеження, а як невід'ємна складова

професійної діяльності.Таким чином, розуміння видів діяльності та їх специфічних ризиків є основою для побудови ефективної системи охорони праці. Для розробників системи Zent визначальними є ризики розумової праці, пов'язані з нервово-емоційним напруженням, зоровим навантаженням і гіподинамією, а їх мінімізація потребує комплексного підходу до організації праці та робочого середовища.

4.2 Загальні вимоги безпеки до обладнання та технологічних процесів

Розробка та експлуатація інформаційних систем супроводжується використанням значної кількості електричного обладнання: персональних комп'ютерів, серверів, мережевого обладнання та периферійних пристроїв. Усе це обладнання є потенційним джерелом електричної та пожежної небезпеки, тому до нього висуваються чіткі нормативні вимоги.

Електробезпека обладнання. Відповідно до Правил улаштування електроустановок [32], персональні комп'ютери та периферійні пристрої відносяться до обладнання класу I, що передбачає обов'язкове захисне заземлення металевих корпусів. Опір захисного заземлення не повинен перевищувати 4 Ом. Для підвищення рівня захисту може застосовуватись також занулення або пристрої захисного відключення з порогом спрацювання не більше 30 мА. Прокладання кабелів живлення має виключати їх механічне пошкодження та перегин; використання пошкоджених кабелів і розеток категорично забороняється.

Організація технологічного процесу. Відповідно до НПАОП 0,00-7.15-18, безперервна тривалість роботи з екранними пристроями не повинна перевищувати 2 годин, після чого обов'язкова перерва тривалістю не менше 15 хвилин [36]. Для зниження ризику надзвичайних ситуацій, пов'язаних із відмовою інфраструктури, у системі Zent передбачено розгортання з використанням Docker-контейнерів та резервне копіювання бази даних PostgreSQL із збереженням копій на окремому носієві або у хмарному сховищі.

ВИСНОВКИ

Метою дипломної роботи була розробка вебзастосунку Zent для командного управління проєктами з підтримкою канбан-методології. Робота охоплювала весь шлях від дослідження предметної області до робочої системи з автоматизованим тестуванням: аналіз вимог, архітектурне проєктування та практичне втілення виконувалися послідовно, де кожен етап спирався на результати попереднього.

Предметна область системи описана через п'ять рівнів вкладеності: команда, проєкт, дошка, колонка, задача. Відносини між ними однозначні: кожен рівень підпорядкований рівно одному вищому, зворотних зв'язків немає. Така структура природно задає логіку перевірки прав: коли потрібно з'ясувати, чи може користувач редагувати задачу, система піднімається вгору по ланцюжку аж до рівня команди і перевіряє членство на кожному кроці.

Рольова модель побудована так, що роль зберігається не в обліковому записі, а в записі про членство в конкретній команді. Як наслідок, один і той самий користувач у різних командах матиме різні набори прав, і жодного дублювання для цього не потрібно. Каталог варіантів використання охопив двадцять сценаріїв у семи групах; разом із функціональними та нефункціональними вимогами за ISO/IEC 25010:2023 він склав повну основу для проєктування.

Для серверної частини обрано Clean Architecture у поєднанні з CQRS. Clean Architecture виштовхує всі інфраструктурні деталі на периферію: шар бізнес-логіки нічого не знає про те, яка саме база підключена, як генерується токен і через який транспорт надходить запит – усе це можна замінити, не зачіпаючи ядро. CQRS розводить читання і запис у окремі потоки з власними обробниками, що дає змогу оптимізувати й тестувати кожен незалежно.

Клієнтська частина організована за Feature-Sliced Design – підходом, де залежності між шарами мають лише один дозволений напрямок і де кожна функціональна область ізольована від решти. Поведінкові UML-діаграми

зафіксували три ключових процеси: автентифікацію крізь усі шари системи, переміщення задачі з підтримкою оптимістичного оновлення та повний життєвий цикл дошки від створення до видалення. Структура таблиць PostgreSQL відтворює доменну модель без зайвих відхилень: первинні ключі у форматі UUID, зовнішні ключі на кожному рівні ієрархії та каскадне видалення, що усуває потребу в окремій логіці очищення на рівні застосунку.

На етапі реалізації кожне архітектурне рішення пройшло практичну перевірку. Диспетчер CQRS написано власноруч, без підключення MediatR чи будь-якої іншої бібліотеки. Близько шістдесяти рядків коду закривають задачу маршрутизації команд і запитів повністю, і при цьому вся логіка залишається відкритою для читання без зайвих абстракцій. Хендлери згруповано за вертикальними зрізами, тому весь код конкретної операції знаходиться в одному місці, а не розкиданий між папками різних технічних типів. Алгоритм переміщення елементів на дошці побудований на повній перенумерації після кожної drag-and-drop операції. Інтерфейс реагує на перетягування миттєво завдяки оптимістичному оновленню; якщо сервер повертає помилку, стан дошки відкочується до знімка, зробленого перед початком операції.

Тестування охопило два рівні. Юніт-тести працюють із InMemory провайдером EF Core і перевіряють хендлери ізольовано. Інтеграційні тести піднімають повноцінний ASP.NET Core хост у пам'яті й гоняють запити через реальний PostgreSQL – від HTTP-виклику до збереження в базі і назад, а між тестами Respawn скидає дані, не перебудовуючи схему. Разом два рівні дають швидку перевірку логіки й надійну перевірку поведінки всієї системи в умовах, близьких до реальних.

Zent вирішує задачу, яку жоден із розглянутих аналогів не закрив у безкоштовному варіанті: команда отримує зрозумілий інструмент із невисоким порогом входження, повноцінною ієрархією проєктів і дошок та рольовою моделлю, де права прив'язані до конкретної команди, а не до облікового запису глобально.

ПЕРЕЛІК ДЖЕРЕЛ

1. Методичні вказівки до виконання кваліфікаційної роботи бакалавра для здобувачів спеціальності 121 – Інженерія програмного забезпечення, всіх форм навчання / укладачі: Михалик Д.М., Цуприк Г.Б., Бревус В.М. – Тернопіль: Тернопільський національний технічний університет імені Івана Пулюя, 2024. – 45 с. (https://elartu.tntu.edu.ua/handle/lib/50317)

2. Олянін, Д., Цуприк, Г. (2025) Transformer Neural Networks in Industry 4.0 / Д. Олянін, Г. Цуприк, Т. Говорущенко, О. Багрій-Заяць, І. Андрущак // Computer Information Technologies in Industry 4.0: proceedings of the 3rd International Workshop (CITI-2025), Ternopil, Ukraine, 11–12 June 2025. – Ternopil : Ternopil Ivan Puluj National Technical University, 2025 (Scopus) https://ceur-ws.org/Vol-4057/

3. Tsupryk, H., Olianin, D. (2025). Vydobuvannia danyh z tekstu vykorystovuiuchy transformerni neironni merezhi [Data extraction from text using Transformer Neural Networks]. Information Technology: Computer Science, Software Engineering and Cyber Security, 125–130, DOI: https://doi.org/10.32782/IT/2025-2-13

4. Digital.ai. State of Agile Report. – 2023. – URL: https://digital.ai/resource-center/analyst-reports/state-of-agile-report/

5. Hindarto D. та ін. Agile Project Management Impacts Software Development Team Productivity // Sinkron: Jurnal dan Penelitian Teknik Informatika. – 2024. – URL: https://www.researchgate.net/publication/38296425.

6. Verified Market Reports. Kanban Project Management Software Market. – 2024. – URL: https://www.verifiedmarketreports.com/product/kanban-project-management-software-market/

7. Mpamugo E., Ansa G. Enhancing Network Security in Mobile Applications with Role-Based Access Control // Journal of Information Systems and Informatics. – 2024. – Vol. 6, No. 3. – DOI: 10.51519/journalisi.v6i3.863

8. Rowe D. K. Optimistic Updates with React Query: Enhancing UX in Real-Time // ResearchGate. – 2024. – URL: https://www.researchgate.net/publication/394408260

9. Khanfor A. Tasks Decomposition Approaches in Crowdsourcing Software Development // HCII 2023, Lecture Notes in Computer Science. – Vol. 14016. – Springer, 2023. – P. 488–498. – DOI: 10.1007/978-3-031-35927-9_36

10. Calderon-Tellez J. та ін. Project management and system dynamics modelling: Time to connect with innovation and sustainability // Systems Research and Behavioral Science. – 2024. – DOI: 10.1002/sres.2926

11. Akuthota A. K. Role-Based Access Control (RBAC) in Modern Cloud Security Governance: An In-depth Analysis // International Journal of Scientific Research in Computer Science Engineering and Information Technology. – 2025. – Vol. 11, No. 2. – P. 3297–3311.

12. Oluwatobi A. та ін. The Significance of Use Case Diagrams in Software Development // ResearchGate. – 2025. – URL: https://www.researchgate.net/publication/387903437

13. Molla M. M. I. та ін. A Comparison of Transforming the User Stories and Functional Requirements into UML Use Case Diagram // International Journal of Innovative Computing. – 2024. – Vol. 14, No. 1. – P. 29–36. – URL: https://researchgate.net/publication/384068768

14. Rahman A., Nayem A., Siddik S. Non-Functional Requirements Classification Using Machine Learning Algorithms // International Journal of Intelligent Systems and Applications. – 2023. – Vol. 15, No. 3. – P. 56–69. – DOI: 10.5815/ijisa.2023.03.05

15. Vargas-Enríquez J. та ін. A Review of Non-Functional Requirements Analysis Throughout the SDLC // Computers. – 2024. – Vol. 13, No. 12. – Art. no. 308. – DOI: 10.3390/computers13120308

16. ISO/IEC 25010:2023. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Product Quality Model. – Geneva: ISO, 2023. – URL: https://www.iso.org/standard/78176.html

17. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Addison-Wesley, 2021. 624 p.

18. Martin R. C. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Prentice Hall, 2017. 432 p.

19. Richardson C. Microservices Patterns: With Examples in Java. Manning Publications, 2018. 520 p.

20. Object Management Group. Unified Modeling Language Specification Version 2.5.1. OMG, 2017. URL: https://www.omg.org/spec/UML/2.5.1

21. Wazlawick R. S. Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL, and IFML. 2nd ed. Springer, 2023. P. 89–134

22. Bruegge B., Dutoit A. H. Object-Oriented Software Engineering Using UML, Patterns, and Java. 4th ed. Pearson, 2022. P. 112–147

23. Kiniry J., Cok D. "Verified Optimistic Concurrency in Modern Web Front-ends." IEEE Software, vol. 40, no. 3, pp. 58–66, 2023. DOI: 10.1109/MS.2023.3241180

24. Ferraiolo D., Kuhn R., Chandramouli R. Role-Based Access Control. 2nd ed. Artech House, 2023. 328 p

25. Hu V. C. et al. Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication 800-162. 2019. URL: https://doi.org/10.6028/NIST.SP.800-162

26. Elmasri R., Navathe S. Fundamentals of Database Systems. 7th ed. Pearson, 2015. 1280 p

27. Microsoft. Entity Framework Core documentation. 2024. URL: https://learn.microsoft.com/en-us/ef/core

28. Richards M. Software Architecture Patterns / M. Richards. – Sebastopol : O'Reilly Media, 2015. – 46 p. – URL: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/

29. Young G. CQRS Documents / G. Young. – 2010. – URL: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

30. Microsoft Corporation. Dependency injection in .NET : офіц. документація / Microsoft Corporation. – 2024. – URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection

31. Lerman J. Programming Entity Framework: DbContext / J. Lerman, R. Miller. – Sebastopol : O'Reilly Media, 2012. – 236 p. – URL: https://www.oreilly.com/library/view/programming-entity-framework/9781449331825/

32. Яремко З.М. Безпека життєдіяльності. Навчальний посібник. – Львів: Видавничий центр ЛНУ ім. І. Франка, 2005. 301 с.

33. Жидецький В.Ц. Охорона праці користувачів комп'ютерів : підручник. Львів : Афіша, 2020. 176 с.

34. Желібо Є.П. Безпека життєдіяльності : підручник / В. В. Зацарний. Київ : Каравела, 2023. 344 с.

35. НПАОП 0,00-7.15-18 Вимоги щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями, від 14.02.2018 року №207.

36. ОЛЯНІН D., & ЦУПРИК Н. (2025). Огляд ролі трансформерних нейроних мереж у видобувані інформації із неструктурованих даних. Measuring and computing devices in technological processes, 82(2), 360–364. https://doi.org/10.31891/2219-9365-2025-82-52

37. Tsupryk H. LLM-based Extraction from Resumes / D. Olianin, H. Tsupryk // Advanced Technologies in Scientific Research: collection of scientific papers with proceedings of the 1st International Scientific and Practical Conference, Rotterdam, Netherlands, 20–22 August 2025. – International Scientific Unity, 2025. – 72-76

38. Yaroslav Kotov, Evhenia Yavorska, Halyna Tsupryk, Róża Dzierżak 1 , Oleksandr Reshetnik, Viktoriia Bokovets (2025) Evaluating interoperability and data quality in FHIR-based AI assessment pipelines. Proc. SPIE 14009, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2025, 140091F (30 December 2025) https://doi.org/10.1117/12.310056

ДОДАТКИ

ДОДАТОК А

Тези доповіді на конференції

Міністерство освіти і науки України
Тернопільський національний технічний університет
імені Івана Пулюя
Маріборський університет (Словенія)
Технічний університет в Кошице (Словаччина)
Каунаський технологічний університет (Литва)
Львівський національний університет
імені Івана Франка
Гірничо-металургійна академія ім. Станіслава Сташиця (Польща)
Луцький національний технічний університет
Чернівецький національний університет
імені Юрія Федьковича
Вроцлавський економічний університет (Польща)
Університет технологій та економіки
імені Хелени Ходковської (Польща)
Донбаська державна машинобудівна академія

Студентське наукове товариство

IX МІЖНАРОДНА

студентська науково - технічна конференція

"ПРИРОДНИЧІ ТА ГУМАНІТАРНІ НАУКИ. АКТУАЛЬНІ ПИТАННЯ"

24-25 квітня 2026 р.

(збірник тез конференції)

Тернопіль 2026

УДК 004.41
Радюк В. - ст. гр. СП-42
Тернопільський національний технічний університет імені Івана Пулюя

АРХІТЕКТУРНІ ПІДХОДИ ДО ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ ВЕБ – ЗАСТОСУНКІВ

Науковий керівник: к.т.н., доцент Цуприк Г.Б.

Radiuk V.
Ternopil Ivan Puluj National Technical University

ARCHITECTURAL APPROACHES TO DESIGNING THE BACKEND OF WEB APPLICATIONS

Supervisor: PhD, Associate Professor H. B. Tsupryk

Ключові слова: запити, мікросервіси, моноліт
Keywords: requests, microservices, monolith

Серверна частина веб-додатків відповідає за обробку запитів, реалізацію бізнес-логіки та взаємодію з базою даних. Вона забезпечує автентифікацію та авторизацію користувачів, обробку помилок, логування подій та інтеграцію із зовнішніми сервісами. Крім того, сервер обробляє HTTP-запити, виконує операції створення, читання, оновлення та видалення даних (CRUD) і повертає відповіді клієнту. При проєктуванні таких систем ключовим є вибір архітектурного підходу, який визначає масштабованість, підтримуваність та продуктивність застосунку [1].

Сучасні серверні застосунки також повинні відповідати вимогам високої доступності та відмовостійкості, що досягається шляхом використання балансування навантаження, резервування ресурсів і механізмів автоматичного відновлення. З огляду на зростання кількості користувачів і обсягів даних, особливого значення набуває оптимізація продуктивності, яка включає кешування результатів запитів, ефективну роботу з базами даних та мінімізацію затримок при обробці запитів. У цьому контексті широко застосовуються інструменти моніторингу, які дозволяють відстежувати стан системи в реальному часі та оперативно реагувати на можливі збої.

Найпоширенішими підходами є монолітна та мікросервісна архітектури. Монолітна модель передбачає реалізацію всієї функціональності в межах одного застосунку, що спрощує розробку та розгортання, але ускладнює масштабування окремих компонентів. Мікросервісна архітектура базується на розділенні системи на незалежні сервіси, кожен з яких відповідає за окрему функціональність, що забезпечує гнучкість і горизонтальне масштабування [2].

Крім того, у мікросервісних системах кожен сервіс може бути реалізований із використанням різних технологій і мов програмування, що дозволяє обирати найбільш ефективні інструменти для вирішення конкретних задач. Водночас це створює додаткові виклики, пов’язані з оркестрацією сервісів, забезпеченням їхньої взаємодії та управлінням конфігураціями. Для вирішення цих задач застосовуються контейнери та системи керування контейнерами, такі як Docker і Kubernetes.

При проєктуванні серверної частини використовуються принципи розділення відповідальностей (Separation of Concerns) та багаторівнева архітектура, яка включає рівні представлення, бізнес-логіки та доступу до даних. Застосування шаблонів, таких

як MVC або Clean Architecture, дозволяє підвищити модульність і спростити підтримку коду. Для взаємодії між клієнтом і сервером використовується REST-підхід, який визначає стандартизовану взаємодію через HTTP-протокол і є одним із ключових архітектурних стилів веб-систем [1].

Окрім REST, у сучасних системах також можуть використовуватися альтернативні підходи до обміну даними, зокрема GraphQL або gRPC, які забезпечують більшу гнучкість у формуванні запитів і підвищення ефективності передачі даних. Вибір конкретного підходу залежить від вимог до продуктивності, складності системи та характеру взаємодії між клієнтом і сервером.

Важливим аспектом є забезпечення цілісності даних та коректної роботи в умовах багатокористувацького доступу, що досягається за рахунок використання транзакцій і контролю конкурентного доступу. У мікросервісних системах ці задачі ускладнюються через розподіленість сервісів і потребують додаткових підходів до узгодженості даних [2].

Для забезпечення узгодженості в таких системах застосовуються підходи, як-от eventual consistency, шаблони Saga або використання брокерів повідомлень для асинхронної взаємодії між сервісами. Це дозволяє зменшити зв'язність між компонентами та підвищити стійкість системи до збоїв. Таким чином, ефективне проєктування серверної частини веб-додатків вимагає комплексного підходу, що враховує як архітектурні, так і технологічні аспекти побудови програмних систем.

Література:

1. Fielding R. Architectural Styles and the Design of Network-based Software Architectures. Roy Fielding. URL: https://roy.gbiv.com/pubs/dissertation/fielding_dissertation.pdf.
2. Fowler M. Microservices. *Martin Fowler*. *Articles*. 25.03.2014. URL: https://martinfowler.com/articles/microservices.html.
3. Olianin D., Tsupryk H., Hovorushchenko T., Bahrii-Zaiats O., Andrushchak I. Transformer Neural Networks in Industry 4.0. Proceedings of the 3rd International Workshop (CITI-2025), Ternopil, Ukraine, 11–12 June 2025. Ternopil: Ternopil Ivan Puluj National Technical University, 2025. URL: https://ceur-ws.org/Vol-4057/
4. Tsupryk H., Olianin D. Data extraction from text using Transformer Neural Networks. Information Technology: Computer Science, Software Engineering and Cyber Security. 2025. №2. P. 125–130. URL: https://doi.org/10.32782/IT/2025-2-13
5. Olianin D., Tsupryk H. Огляд ролі трансформерних нейроних мереж у видобувані інформації із неструктурованих даних. Measuring and computing devices in technological processes. 2025. Vol. 82(2). P. 360–364. URL: https://doi.org/10.31891/2219-9365-2025-82-52

ДОДАТОК Б

Код бізнес-логіки

Лістинг Б.1 – Логіка створення задачі

```
using Microsoft.EntityFrameworkCore;
using Zent.Application.Messaging.Abstractions;
using Zent.Common.Exceptions;
using Zent.Data;
using Zent.Data.Entities;

namespace Zent.Application.Features.Tasks.AddTask;

internal sealed class AddTaskHandler(ZentDbContext dbContext) :
ICommandHandler<AddTaskCommand, Guid>
{
    public async Task<Guid> Handle(AddTaskCommand command, CancellationToken ct)
    {
        var board = await dbContext.Boards
            .AsNoTracking()
            .Where(x => x.Id == command.BoardId)
            .Select(x => new { x.Id, x.ProjectId, x.Project.TeamId })
            .FirstOrDefaultAsync(ct);

        if (board is null)
            throw new BoardNotFoundException($"Board with id {command.BoardId}
was not found.");

        var hasAccess = await dbContext.TeamMembers
            .AsNoTracking()
            .AnyAsync(x =>
                x.TeamId == board.TeamId &&
                x.UserId == command.UserId, ct);

        if (!hasAccess)
            throw new TeamAccessDeniedException(
                "You are not a member of this team.");

        var columnExists = await dbContext.Columns
            .AsNoTracking()
            .AnyAsync(x =>
                    x.BoardId == board.Id &&
                    x.Id == command.ColumnId,
                ct);

        if (!columnExists)
            throw new ColumnNotFoundException(
                $"Column with id {command.ColumnId} was not found in this
board.");

        var title = command.Title.Trim();

        var description = string.IsNullOrWhiteSpace(command.Description)
            ? null
            : command.Description.Trim();

        var maxOrder = await dbContext.Tasks
            .AsNoTracking()
            .Where(x => x.ColumnId == command.ColumnId)
            .Select(x => (int?)x.Order)
```

```
            .MaxAsync(ct) ?? 0;

        var task = new TaskEntity
        {
            ColumnId = command.ColumnId,
            CreatorId = command.UserId,
            AssigneeId = command.AssigneeId,
            Title = title,
            Description = description,
            Priority = command.Priority,
            Order = maxOrder + 1,
            CreatedAt = DateTime.UtcNow,
            UntilDate = command.UntilDate
        };

        dbContext.Tasks.Add(task);

        await dbContext.SaveChangesAsync(ct);

        return task.Id;
    }
}
```

Лістинг Б.2 – Логіка переміщення задачі

```
using Microsoft.EntityFrameworkCore;
using Zent.Application.Messaging.Abstractions;
using Zent.Common.Exceptions;
using Zent.Data;

namespace Zent.Application.Features.Tasks.MoveTask;

internal sealed class MoveTaskHandler(ZentDbContext dbContext)
    : ICommandHandler<MoveTaskCommand>
{
    public async Task Handle(MoveTaskCommand command, CancellationToken ct)
    {
        var board = await dbContext.Boards
            .AsNoTracking()
            .Where(x => x.Id == command.BoardId)
            .Select(x => new { x.Id, x.ProjectId })
            .FirstOrDefaultAsync(ct);

        if (board is null)
        {
            throw new BoardNotFoundException(
                $"Board with id {command.BoardId} was not found.");
        }

        var hasAccess = await dbContext.ProjectMembers
            .AsNoTracking()
            .AnyAsync(x =>
                    x.ProjectId == board.ProjectId &&
                    x.UserId == command.UserId,
                ct);

        if (!hasAccess)
            throw new ProjectAccessDeniedException(
                "You are not a member of this project.");

        var task = await dbContext.Tasks
            .FirstOrDefaultAsync(x => x.Id == command.TaskId, ct);
```

```
        if (task is null)
            throw new TaskNotFoundException(
                $"Task with id {command.TaskId} was not found.");

        var taskBelongsToBoard = await dbContext.Columns
            .AsNoTracking()
            .AnyAsync(x =>
                    x.Id == task.ColumnId &&
                    x.BoardId == command.BoardId,
                ct);

        if (!taskBelongsToBoard)
            throw new TaskNotFoundException(
                $"Task with id {command.TaskId} does not belong to this board.");

        var targetColumnExists = await dbContext.Columns
            .AsNoTracking()
            .AnyAsync(x =>
                    x.Id == command.TargetColumnId &&
                    x.BoardId == command.BoardId,
                ct);

        if (!targetColumnExists)
            throw new ColumnNotFoundException(
                $"Column with id {command.TargetColumnId} was not found in this
board.");

        var sourceColumnId = task.ColumnId;
        var targetColumnId = command.TargetColumnId;

        if (sourceColumnId == targetColumnId)
        {
            await MoveInsideSameColumnAsync(
                taskId: task.Id,
                columnId: sourceColumnId,
                targetOrder: command.TargetOrder,
                ct);
        }
        else
        {
            await MoveBetweenColumnsAsync(
                taskId: task.Id,
                sourceColumnId: sourceColumnId,
                targetColumnId: targetColumnId,
                targetOrder: command.TargetOrder,
                ct);
        }

        await dbContext.SaveChangesAsync(ct);
    }

    private async Task MoveInsideSameColumnAsync(
        Guid taskId,
        Guid columnId,
        int targetOrder,
        CancellationToken ct)
    {
        var tasks = await dbContext.Tasks
            .Where(x => x.ColumnId == columnId)
            .OrderBy(x => x.Order)
            .ToListAsync(ct);

        var movingTask = tasks.FirstOrDefault(x => x.Id == taskId);
```

```
        if (movingTask is null)
            throw new TaskNotFoundException(
                $"Task with id {taskId} was not found in this column.");

        tasks.Remove(movingTask);

        var insertIndex = Math.Clamp(targetOrder - 1, 0, tasks.Count);

        tasks.Insert(insertIndex, movingTask);

        for (var i = 0; i < tasks.Count; i++)
            tasks[i].Order = i + 1;
    }

    private async Task MoveBetweenColumnsAsync(
        Guid taskId,
        Guid sourceColumnId,
        Guid targetColumnId,
        int targetOrder,
        CancellationToken ct)
    {
        var sourceTasks = await dbContext.Tasks
            .Where(x => x.ColumnId == sourceColumnId)
            .OrderBy(x => x.Order)
            .ToListAsync(ct);

        var targetTasks = await dbContext.Tasks
            .Where(x => x.ColumnId == targetColumnId)
            .OrderBy(x => x.Order)
            .ToListAsync(ct);

        var movingTask = sourceTasks.FirstOrDefault(x => x.Id == taskId);

        if (movingTask is null)
            throw new TaskNotFoundException(
                $"Task with id {taskId} was not found in source column.");

        sourceTasks.Remove(movingTask);

        var insertIndex = Math.Clamp(targetOrder - 1, 0, targetTasks.Count);

        movingTask.ColumnId = targetColumnId;

        targetTasks.Insert(insertIndex, movingTask);

        for (var i = 0; i < sourceTasks.Count; i++)
            sourceTasks[i].Order = i + 1;

        for (var i = 0; i < targetTasks.Count; i++)
            targetTasks[i].Order = i + 1;
    }
}
```