БАКАЛАВРСЬКА РОБОТА

КРБ.ІСТ – 25.00.00.000 ПЗ

Група ІСТ-22-1

Юрій ЧУМАКЕВИЧ

2026

Міністерство освіти і науки України
Івано-Франківський національний технічний університет нафти і газу
Факультет інформаційних технологій
Кафедра інформаційно-телекомунікаційних технологій та систем

Чумакевич Юрій Юрійович
(прізвище, ім'я, по батькові)

УДК 004.415.2:004.774
(індекс)

БАКАЛАВРСЬКА РОБОТА

Розроблення інформаційної системи онлайн-сервісу візуальної організації інформації для колективної роботи
(назва роботи)

Інформаційні системи та технології
(назва освітньої програми)

126 − Інформаційні системи та технології
(шифр і назва спеціальності)

Робота містить результати власних досліджень, використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело:

Здобувач освітнього ступеня Чумакечив Ю. Ю.
(підпис, ініціали та прізвище здобувача)

Науковий керівник доцент Волинський О. І.
(підпис, прізвище, ім'я, по батькові, науковий ступінь, вчене звання керівника)

Допущено до захисту
В. о. завідувача кафедри

Заміховська О. Л.
(посада) (підпис) (дата) (ініціали та прізвище)

Івано-Франківськ – 2026

Івано-Франківський національний технічний університет нафти і газу
(повне найменування закладу вищої освіти)

Факультет інформаційних технологій
Кафедра інформаційно-телекомунікаційних технологій та систем
Освітній рівень бакалавр
Спеціальність 126 − Інформаційні системи та технології
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТТС
Заміховська О. Л.
«____» _____2026 року

З А В Д А Н Н Я
НА БАКАЛАВРСЬКУ РОБОТУ СТУДЕНТОВІ

Чумакевичу Юрію Юрійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розроблення інформаційної системи онлайн-сервісу візуальної організації інформації для колективної роботи
керівник роботи, доц. каф. ІТТС Волинський О. І.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом закладу вищої освіти від " 07 " *квітня* 2026 року № 163/7
2. Строк подання студентом роботи_____
3. Вихідні дані до роботи Веб ресурси з аналогами: excalidraw.com, explaineverything.com, Diagrams.net; для реалізації інформаційної системи використано базу даних postgresql.org, фреймворк для серверної частини Ruby on Rails, бібліотеку для клієнтської частини React, CSS фреймворк Tailwind, бібліотеку для роботи з графікою Konva.
4. Зміст пояснювальної записки (перелік питань, що їх належить розробити)
дослідження середовища та логіки візуальної взаємодії; проєктування архітектури та компонентів інформаційної системи; Реалізація програмного забезпечення
5. Перелік презентаційного матеріалу (з точним зазначенням обов'язкових презентацій)
варіанти використання онлайн-сервісу візуальної організації інформації. Usecase діаграма; діаграма робочих потоків веб-дошки для організації візуальної інформації; сценарії ініціалізації нового робочого середовища та фонового збереження графічних маніпуляцій; сценарій налаштування спільного доступу; база даних веб-дошки для організації візуальної інформації; діаграма контролерів; інформаційна система онлайн-сервісу візуальної організації інформації для колективної роботи. робочі вікна

6. Консультанти розділів проекту (роботи)

Розділ	*Консультант*	*Підпис, дата*	
		Завдання видав	*Завдання прийняв*
1-3	Волинський О. І.		

7. Дата видачі завдання: 13.01.2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Термін виконання етапів	Примітка
1.	Аналіз предметної області та огляд існуючих систем для спільної візуальної роботи (Excalidraw, Draw.io)	13.01.2026 - 05.02.2026 р.	Виконано
2.	Моделювання бізнес-процесів (IDEF, DFD, BPMN 2.0), розробка UML-діаграм та карт користувацького досвіду (UJM, CJM)	05.02.2026 - 01.03.2026 р.	Виконано
3.	Проєктування реляційної бази даних (PostgreSQL), побудова концептуальної та логічної моделей із проведенням нормалізації	01.03.2026 - 07.03.2026 р.	Виконано
4.	Програмна реалізація серверної частини (Ruby on Rails API), налаштування автентифікації (Devise) та рольової моделі доступу (Pundit)	07.03.2026 - 18.03.2026 р.	Виконано
5.	Розробка клієнтського SPA-додатку на базі React і TypeScript (каталог дошок, форми авторизації, меню адміністрування команди)	18.03.2026 - 01.04.2026 р.	Виконано
6.	Інтеграція графічного рушія react-konva для обробки об'єктів на полотні та реалізація системи історії локальних змін (Undo/Redo)	01.04.2026 - 1.05.2026 р.	Виконано
7.	Оформлення пояснювальної записки та графічного матеріалу	1.05.2026 - 20.05.2026 р.	Виконано
8.	Подання роботи на рецензування та підготовка до захисту	20.05.2026 - 04.06.2026 р.	

Студент _____ *Чумакевич Ю. Ю.*
(підпис) (прізвище та ініціали)

Керівник роботи _____ *Волинський О. І.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Розрахунково-пояснювальна записка: 70 сторінок, 26 малюнків, 9 таблиці, 31 посилань, 3 додатки.

Об'єкт дослідження – процес проєктування та імплементування інформаційних систем для спільної візуальної роботи та організації даних у групах.

Мета роботи – системний аналіз, проєктування та практична програмна реалізація інформаційної системи (вебдодатка) для колективної візуальної організації даних у режимі реального часу.

У першій частині роботи досліджено теоретичні засади створення цифрових просторів для колективної роботи та проведено системний аналіз існуючих рішень (Excalidraw, Draw.io). На основі виявлених недоліків аналогів обґрунтовано концепцію власного сервісу, що поєднує простий графічний інструментарій із суворою ієрархічною рольовою моделлю. Паралельно було розроблено нормалізовану реляційну базу даних PostgreSQL та, спираючись на аналіз шляху користувача (карти UJM і CJM), спроєктовано ергономічні прототипи інтерфейсу.

Практична реалізація системи ObjectBoard виконана у форматі гібридного Single Page Application. Серверний бекенд створено на базі Ruby on Rails (у режимі API-сервера), де за управління обліковими записами та дозволами відповідають бібліотеки Devise і Pundit. Клієнтська оболонка побудована на екосистемі React із суворою типізацією TypeScript, при цьому рендеринг інтерактивного полотна здійснюється через бібліотеку react-konva, а синхронізація дій команди в реальному часі – через протокол WebSockets (ActionCable). У підсумку здійснено повний цикл створення програмного продукту, успішно протестовано його модулі та підтверджено ефективність розробленої платформи для завдань візуального планування у розподілених командах.

КЛЮЧОВІ СЛОВА: ВІЗУАЛЬНА ВЗАЄМОДІЯ, КОЛЕКТИВНА РОБОТА, ОНЛАЙН-ДОШКА, REACT, RUBY ON RAILS, SINGLE PAGE APPLICATION, WEBSOCKETS, РОЛЬОВА МОДЕЛЬ ДОСТУПУ, POSTGRESQL, ІНТЕРАКТИВНЕ ПОЛОТНО.

ABSTRACT

Calculation and explanatory note: 70 pages, 26 figures, 9 tables, 31 references, 3 appendices.

Object of research – the process of designing and implementing information systems for collaborative visual work and data organization within groups.

Purpose of the work – system analysis, design, and practical software implementation of an information system (web application) for collaborative real-time visual data organization.

The first part of this work examines the theoretical foundations for creating digital spaces for collaborative work and conducts a systematic analysis of existing solutions (Excalidraw, Draw.io). Based on the identified shortcomings of these alternatives, the concept of a proprietary service is proposed, combining simple graphical tools with a strict hierarchical role-based model. In parallel, a normalized PostgreSQL relational database was developed, and, based on user journey analysis (UJM and CJM maps), ergonomic interface prototypes were designed.

The practical implementation of the ObjectBoard system is a hybrid Single Page Application. The server-side backend is built on Ruby on Rails (in API server mode), where the Devise and Pundit libraries handle user accounts and permissions. The client-side shell is built on the React ecosystem with strict TypeScript typing, while the interactive canvas is rendered via the react-konva library, and real-time synchronization of team actions is handled via the WebSockets protocol (ActionCable). As a result, the full software development cycle was completed, its modules were successfully tested, and the effectiveness of the developed platform for visual planning tasks in distributed teams was confirmed.

KEYWORDS: VISUAL INTERACTION, COLLABORATIVE WORK, ONLINE WHITEBOARD, REACT, RUBY ON RAILS, SINGLE PAGE APPLICATION, WEBSOCKETS, ROLE-BASED ACCESS MODEL, POSTGRESQL, INTERACTIVE CANVAS.

ЗМІСТ

ВСТУП

Стрімкий перехід сучасного бізнесу до віддаленого та гібридного форматів роботи кардинально змінив підходи до командної взаємодії. Традиційні текстові канали комунікації часто не здатні забезпечити ефективну передачу складних ідей, архітектурних концепцій чи схем бізнес-процесів. Саме тому критичної ваги набувають інструменти для візуальної колаборації – інтерактивні онлайн-дошки. Вони формують «єдине джерело істини» для розподілених команд, дозволяючи синхронно генерувати ідеї, проєктувати системи та управляти даними.

Незважаючи на наявність на ринку популярних рішень, багато з них або пропонують надто примітивний функціонал без належного контролю доступу, або, навпаки, є перевантаженими та складними в освоєнні. Це зумовлює актуальність створення власного оптимізованого вебдодатка, який поєднає високу швидкість інтерактивної взаємодії, гнучку рольову модель та надійну архітектуру збереження інформації.

Метою курсової роботи є системний аналіз, проєктування та практична програмна реалізація інформаційної системи (вебдодатка) для колективної візуальної організації даних у режимі реального часу.

Відповідно до поставленої мети, було сформульовано такі завдання:

– дослідити предметну область, проаналізувати існуючі програмні аналоги та виявити їхні ключові недоліки;

– розробити комплекс моделей для формалізації вимог, бізнес-процесів та алгоритмів роботи системи (з використанням нотацій IDEF0, DFD, BPMN та UML);

– спроєктувати користувацький досвід (UX/UI) на основі карт шляху клієнта (UJM, CJM) та створити прототипи інтерфейсів;

– розробити логічну та фізичну структуру реляційної бази даних для збереження графічних примітивів і метаданих;

– обґрунтувати вибір технологічного стека (ADR) та здійснити програмну реалізацію клієнт-серверної архітектури вебдодатка. Забезпечити його тестування.

Об’єктом дослідження є процес проєктування та імплементування інформаційних систем для спільної візуальної роботи та організації даних у групах.

Предметом дослідження є архітектурні патерни, алгоритми рендерингу графіки, моделі даних та програмні засоби (фреймворки), що використовуються для створення інтерактивних онлайн-платформ візуалізації.

1 ДОСЛІДЖЕННЯ СЕРЕДОВИЩА ТА ЛОГІКИ ВІЗУАЛЬНОЇ ВЗАЄМОДІЇ

1.1 Аналіз сфери засобів колективного впорядкування інформації

Проєкт розробляється у сфері інструментів для цифрової візуальної співпраці та управління розподіленими командами. У сучасних бізнес-реаліях, де домінує віддалений та гібридний формати роботи, існує критична потреба у платформах, що забезпечують єдине бачення концепцій усіма учасниками процесу. Сфера застосування системи охоплює проведення мозкових штурмів, Agile-планування, побудову дорожніх карт та проєктування системних архітектур. Даний онлайн-сервіс слугує спільним простором, де абстрактні задуми матеріалізуються у графічні артефакти, доступні для колективного редагування.

До цільової аудиторії продукту входять керівники проєктів, яким необхідно організовувати робочі процеси та розподіляти завдання. Також активними користувачами є бізнес-аналітики та IT-спеціалісти, що застосовують платформу для моделювання процесів і технічних схем. Замовники, зі свого боку, виступають стейкхолдерами, яким потрібна зрозуміла та наочна презентація результатів.

Ключова проблема сучасного ринку полягає у роздробленості комунікацій: обговорення в чатах не синхронізовані з візуальними документами, які часто зберігаються на локальних носіях. Відсутність єдиного інформаційного поля («єдиного джерела істини») провокує непорозуміння в команді, суттєво уповільнює процес ухвалення рішень та збільшує час виходу готового продукту на ринок (time-to-market).

Для формування чітких вимог до майбутньої системи необхідно проаналізувати поточний ринок програмного забезпечення у сфері візуальної взаємодії. Одним із відомих рішень є платформа Excalidraw [1]. Це вебдодаток, побудований на базі TypeScript та підтримуваний спільнотою розробників відкритого коду. Головною відмінністю сервісу є специфічний візуальний

стиль, який імітує малювання від руки (рис. 1.1). Платформа забезпечує повноцінну синхронну роботу команди, автоматично зберігаючи всі зміни у власному хмарному просторі. Значною перевагою є наявність вбудованого голосового зв'язку, що дозволяє обговорювати ідеї без використання сторонніх програм.

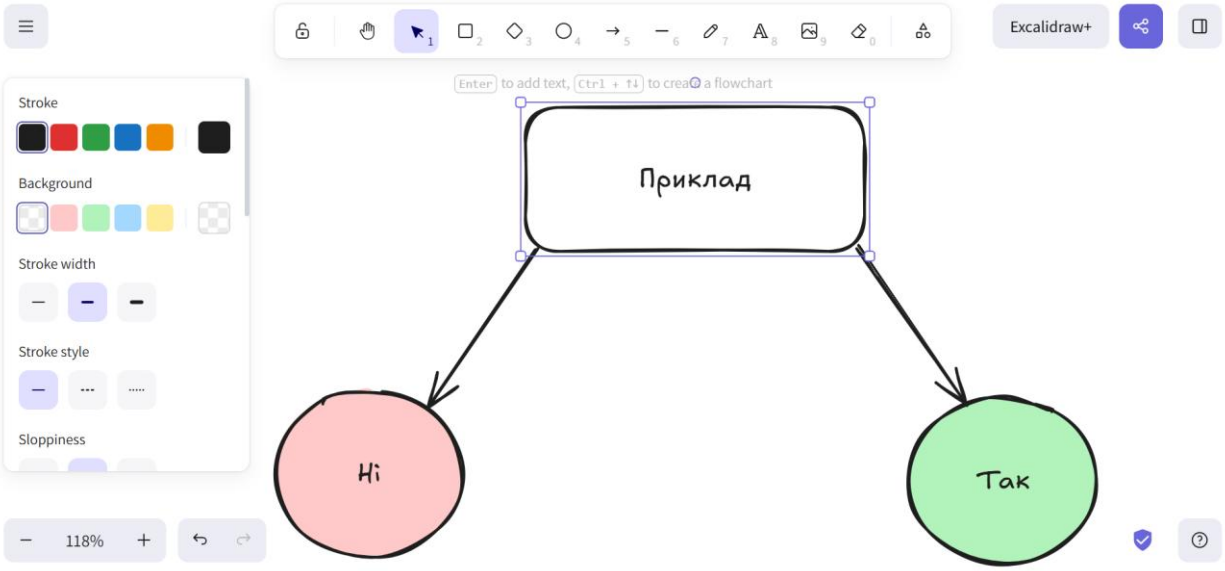

Рисунок 1.1 – Приклад діаграми в Excalidraw [1]

Користувачі мають доступ до великої кількості публічних бібліотек із графічними елементами, які створюються самою спільното. Готові схеми можна експортувати у форматах PNG, SVG, JSON, PDF та PPTX. Попри інтуїтивний інтерфейс та гнучкість, продукт має суттєвий недолік: швидкодія системи помітно знижується під час масштабування великих об'єктів або роботи з перевантаженими дошками.

Інший підхід до візуалізації демонструє платформа Explain Everything від компанії Promethean [2]. Цей вебдодаток орієнтований на освітній сегмент і створення інтерактивних презентацій. Окрім базового інструментарію для побудови схем (рис. 1.2), система пропонує розширені можливості для запису екрана та створення відеоінструкцій. Платформа також підтримує аудіоспілкування та віддалене збереження файлів, а результати можна вивантажити у форматах PDF або PNG. Ергономічний дизайн є безперечною

сильною стороною сервісу. Проте, з точки зору варіативності, система є досить обмеженою: користувачі змушені працювати виключно з офіційними шаблонами від розробників, що робить її менш придатною для складного технічного чи бізнес-моделювання.

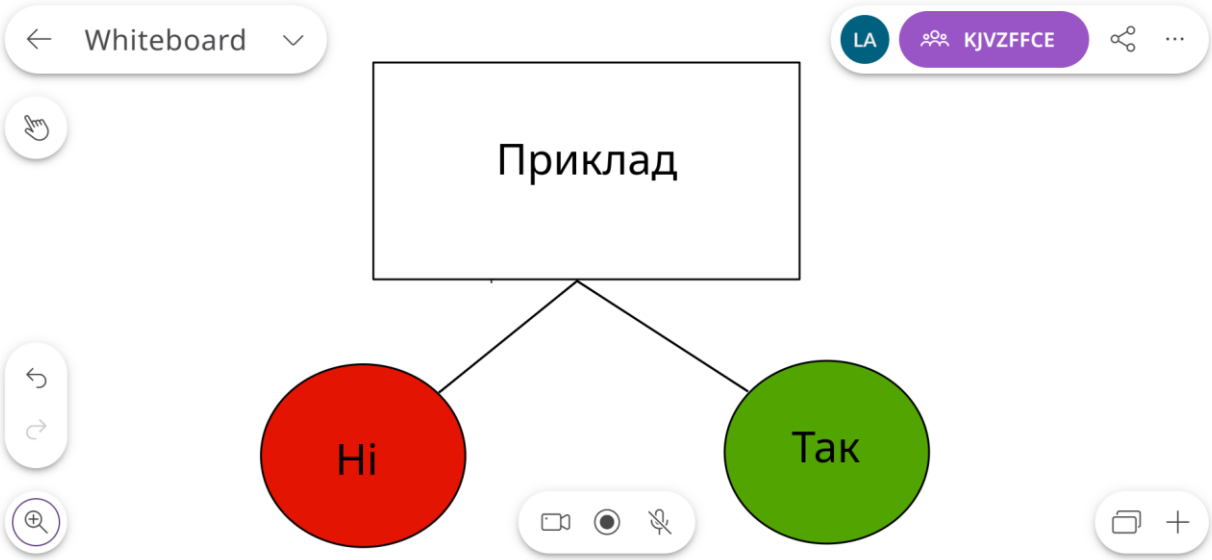

Рисунок 1.2 – Приклад діаграми в Explain Everything [2]

Для вирішення більш формалізованих завдань часто використовується хмарний сервіс Draw.io [3]. Написаний на JavaScript, цей інструмент спеціалізується на створенні строгих технічних, мережевих та архітектурних діаграм (рис. 1.3). На відміну від попередніх аналогів, Draw.io не нав'язує власне сховище, а використовує для збереження даних зовнішні хмарні сервіси. Сервіс пропонує виняткову кількість форматів для експорту (PNG, JPEG, WebP, SVG, PDF, HTML, XML) та володіє масивною вбудованою базою графічних примітивів, яка містить навіть вузькоспеціалізовані стандарти (наприклад, обладнання Cisco). Програма має відкритий вихідний код і проста в освоєнні. Однак недоліком є жорсткість бібліотеки: користувач практично позбавлений можливості зручно розширювати її власноруч намальованими нестандартними фігурами.

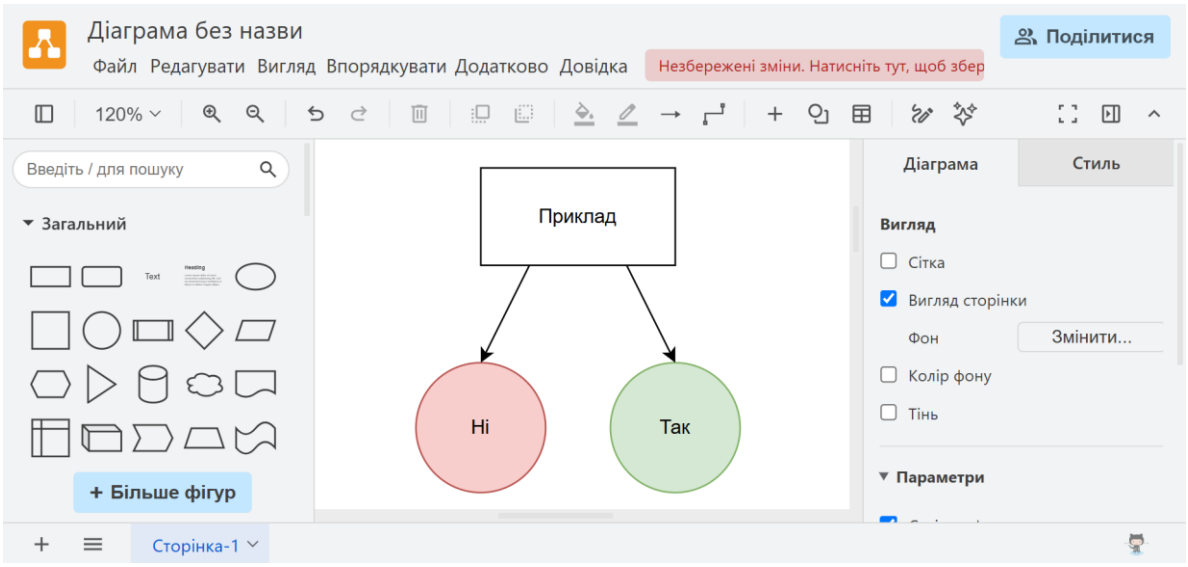

Рисунок 1.3 – Приклад діаграми в draw.io [3]

З метою систематизації зібраних даних та визначення ніші для власної розробки було проведено перехресне порівняння (таблиця 1.1) обраних аналогів за ключовими параметрами.

Таблиця 1.1 – Порівняльний аналіз існуючих аналогів для візуальної співпраці

Критерій порівняння	Excalidraw	Explain Everything	Draw.io
Основна спеціалізація	Неформальні схеми, мозкові штурми (стиль скетчу)	Освітній контент, інтерактивні лекції	Технічні, логічні та архітектурні діаграми
Синхронна робота	Підтримується (включно з вбудованим аудіо)	Підтримується (включно з аудіо та записом екрана)	Підтримується (залежить від підключеного сховища)
Бібліотека об'єктів	Відкрита, постійно поповнюється користувачами	Закрита (лише офіційні базові шаблони)	Велика вбудована (стандартизовані фігури), без власних
Формати експорту	PNG, SVG, JSON, PDF, PPTX	PDF, PNG, формати відео	PNG, JPEG, WebP, SVG, PDF, HTML, XML

Продовження таблиці 1.1

Збереження даних	Власне хмарне середовище	Власне хмарне середовище	Інтеграція зі сторонніми сервісами (Google Drive тощо)
Ключовий недолік	Падіння швидкодії на перевантажених проєктах	Обмеженість візуальних елементів	Неможливість гнучкого розширення власними фігурами
Модель розмежування доступу	Спрощена (загальний доступ за спільним посиланням)	Базова (у межах акаунтів освітнього сервісу)	Делегована (управління правами покладено на Google Drive тощо)
Організація робочого простору	Окремі сесії/дошки без внутрішньої ієрархії учасників	Проєктні папки з орієнтацією на викладача/учня	Файлова система обраного зовнішнього сховища
Навантаженість інтерфейсу	Мінімалістична, сфокусована на швидких начерках	Висока	Висока

Для більш глибокого розуміння місця розроблюваної системи на ринку, проведено кількісний аналіз функціональних параметрів обраних платформ, підсумований у таблиці 1.2. Замість суб'єктивного оцінювання, для порівняння використано абсолютні числові показники, які відображають складність систем та рівень контролю над робочим процесом.

Таблиця 1.2 –Аналіз параметрів систем візуальної співпраці.

Характеристика	**Excalidraw**	**Explain Everything**	**Draw.io**	**Розроблювана система**
Кількість вбудованих рівнів доступу (ролей)	2	2	0	4
Залежність від зовнішніх хмарних сховищ	Ні	Ні	Так	Ні
Можливість точкового управління складом команди	Ні	Так	Ні	Так
Спільна робота в реальному часі	Так	Так	Так	Так

Детальний аналіз наведених у таблиці метрик демонструє суттєві прогалини у популярних глобальних продуктах з точки зору управління робочим процесом. Більшість систем фокусується виключно на розширенні графічного інструментарію, залишаючи питання безпеки та адміністрування на базовому рівні. Наприклад, Draw.io взагалі не має власних механізмів розмежування прав і повністю покладається на зовнішні хмарні сервіси, що унеможливлює використання системи у закритих корпоративних контурах без прив'язки до сторонніх екосистем. Excalidraw використовує максимально спрощену модель з двома рівнями доступу, де управління здійснюється простою передачею відкритого посилання, що створює ризики несанкціонованого втручання.

Розроблювана система компенсує ці недоліки шляхом зміщення акценту на безпечну архітектуру взаємодії. Наявність чотирьох ізольованих рівнів доступу дозволяє гнучко налаштовувати повноваження для різних категорій співробітників безпосередньо в межах платформи. Інтегрована база даних гарантує повну незалежність від сторонніх сховищ, а вбудовані інструменти адміністрування дають змогу власникам дошок точково контролювати список учасників. Завдяки такому підходу спрощений графічний інтерфейс стає свідомим рішенням: він не перевантажує користувача, гарантує високу швидкість освоєння та залишає головним пріоритетом захищену і структуровану роботу з інформацією.

На основі проведеного аналізу предметної області та виявлених недоліків існуючих рішень було сформовано перелік базових вимог до розроблюваного програмного продукту. Головний акцент під час проєктування робиться на забезпеченні стабільної спільної роботи та чіткому розмежуванні прав доступу при збереженні лаконічного інструментарію. Відповідно до цього, спроєктована інформаційна система повинна забезпечувати наступний набір функцій:

1. Адміністрування проєктних середовищ. Користувачі повинні мати інструменти для генерації нових віртуальних полотен, зміни їхніх метаданих

(назв та детальних описів), а також остаточного видалення неактуальних робочих просторів;

2. Взаємодія з елементами візуалізації. Передбачається розміщення на канвасі стандартних геометричних примітивів (прямокутники, кола, лінії) та текстових полів. Доступна зміна їхньої позиції, розмірів, а також кастомізація зовнішнього вигляду, зокрема налаштування відтінків, рівня прозорості та товщини контурів;

3. Ієрархічна система контролю доступу. Це ключовий модуль, що дозволяє гнучко розподіляти права всередині команди. Учасникам призначаються відповідні статуси (Глядач, Редактор, Адміністратор або Власник), кожен з яких жорстко регламентує можливості взаємодії з графічним контентом та управління списком колег;

4. Фонова фіксація змін у реальному часі та керування версіями. Стан поточного проєкту автоматично синхронізується з базою даних, що мінімізує ризик втрати інформації. Крім того, реалізовано механізм покрокового скасування або повернення виконаних маніпуляцій на полотні;

5. Конвертація та завантаження результатів. Розроблена схема може бути швидко експортована на локальний пристрій у вигляді растрового зображення (файли PNG). Це дозволяє зручно інтегрувати готові напрацювання у звітну документацію чи застосовувати їх під час презентацій.

У межах розроблюваної системи ідентифікація зовнішніх акторів (сутностей, які взаємодіють із продуктом) тісно пов'язана із вбудованою ієрархією доступів. Взаємодію з платформою забезпечують чотири типи користувачів:

– Власник (Owner). Особа, яка безпосередньо генерує нове віртуальне полотно. Цей профіль наділений максимальним рівнем привілеїв: він контролює не лише всі графічні примітиви, але й керує статусами запрошених колег. Також лише Власник володіє ексклюзивним правом на повне видалення проєкту.

– Адміністратор (Administrator). Довірений учасник, чиї основні функції зосереджені на модерації команди. Актор із цією роллю відповідає за організаційні моменти – запрошення нових людей до робочого простору, блокування неактуальних користувачів та коригування їхніх повноважень.

– Редактор (Editor). Ключова дійова особа під час роботи з контентом. Права цього актора сфокусовані виключно на маніпуляціях із графікою: він може безперешкодно додавати, налаштовувати або стирати будь-які фігури чи текстові блоки на дошці.

– Глядач (Viewer). Пасивний учасник процесу. Його профіль обмежується базовим дозволом на перегляд поточного стану проєкту. Це дозволяє стейкхолдерам чи колегам безпечно стежити за змінами на полотні без технічної можливості випадково внести власні правки.

1.2 Принципи функціонування полотна для групової роботи

Для функціонального модулювання обрано нотацію IDEF0. Головним завданням розробки моделі в нотації IDEF0 є комплексне відтворення алгоритмів роботи розроблюваної онлайн-платформи для візуальної взаємодії. Відповідно до стандартів цієї методології, будь-який бізнес-процес ілюструється як функціональний блок, що здійснює трансформацію початкової інформації у кінцевий результат. Ця трансформація відбувається за визначеними правилами (управління) та з використанням конкретних механізмів. Важливою особливістю підходу є те, що згенеровані вихідні потоки одного етапу здатні виступати вхідними параметрами, керуючими впливами чи ресурсами для наступних завдань [4].

Під час проєктування було обрано перспективу системного аналітика. Такий ракурс дає змогу абстрагуватися від дрібних технічних особливостей і поглянути на програмний продукт глобально. Основна увага приділяється логіці маршрутизації даних, способам взаємозв'язку між окремими підсистемами та підтримці надійності інформації під час асинхронних

маніпуляцій команди. Завдяки цьому формується максимально вичерпна база вимог, необхідна для коректної автоматизації ключових процесів майбутнього вебсервісу.

Графічне відображення найвищого рівня абстракції розроблюваної системи наведено на рисунку 1.4 у форматі контекстної діаграми A-0. Центральним блоком моделі виступає комплексна функція «Організувати візуальну інформацію та колективний доступ». Застосування стандарту IDEF0 дозволило чітко окреслити межі програмного продукту ObjectBoard та класифікувати всі потоки даних під час взаємодії із зовнішнім середовищем.

Рисунок 1.4 – Концептуальна модель взаємодії сервісу з зовнішнім середовищем

Представлена модель ілюструє глобальний процес перетворення початкових параметрів (інформації про реєстрацію, учасників та їхніх запитів на маніпуляції з контентом) у структурований результат. Регламентують цей процес керуючі впливи: вбудована рольова модель, технічні обмеження системи та алгоритми рендерингу фігур. Безпосереднє виконання операцій покладено на технічні механізми платформи – клієнтський інтерфейс (фронтенд), серверну частину (бекенд) та сховище даних (БД). На виході роботи блоку формується актуалізований стан робочого полотна, оновлена

матриця прав доступу для команди та згенеровані растрові артефакти для експорту. Ця узагальнена схема формує базис для подальшої покрокової декомпозиції процесів.

Для розкриття внутрішньої алгоритмічної структури системи ObjectBoard було побудовано діаграму декомпозиції першого рівня (рис. 1.5). Глобальний процес поділено на чотири взаємопов'язані етапи, які послідовно відображають життєвий цикл сесії користувача.

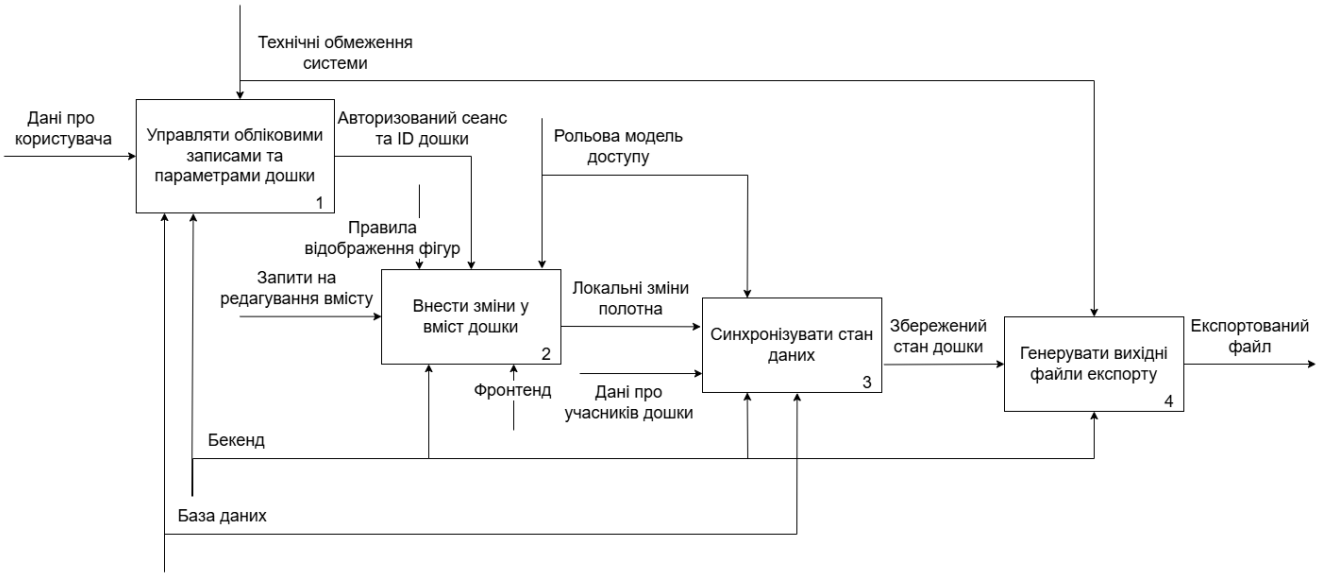

Рисунок 1.5 – Основні процеси організації візуального простору

Перший функціональний вузол – «Управляти обліковими записами та параметрами дошки» – відповідає за стартову ініціалізацію: обробку даних для входу та налаштування нового робочого середовища. Після успішного підключення система переходить до блоку «Внести зміни у вміст дошки», у межах якого реалізується безпосередня взаємодія користувача з графічними об'єктами на полотні.

Третій етап, «Синхронізувати стан даних», є критично важливим для організації колективного доступу: він забезпечує регулярну передачу локальних правок до спільної бази даних із обов'язковим урахуванням рольової моделі. Завершальним кроком є блок «Генерувати вихідні файли

експорту», який дозволяє трансформувати збережену схему у фінальний растровий артефакт для подальшого завантаження.

Усі наведені підпроцеси об'єднані внутрішніми інформаційними потоками (наприклад, передачею ID дошки чи локальних змін) і спільно використовують єдиний набір виконавчих механізмів: клієнтський інтерфейс, серверну логіку та базу даних.

Розшифрування інформаційних потоків (вхідних, вихідних, керуючих), а також механізмів виконання, які фігурують на побудованих IDEF0-діаграмах системи ObjectBoard, зведено у таблицю 1.3.

Таблиця 1.3 – Глосарій стрілок функціональної моделі

Назва стрілки	**Тип**	**Короткий опис**
Дані про користувача	I	Облікові відомості (пароль, email, ім'я), необхідні для ідентифікації та успішного входу в систему.
Запити на редагування вмісту	I	Параметри графічних елементів, що додаються або змінюються на полотні (координати точок, розміри, колір, текстове наповнення).
Дані про учасників дошки	I	Ідентифікатори або поштові адреси колег, яким необхідно надати доступ до спільного робочого простору.
Рольова модель доступу	C	Набір системних політик і статусів (Власник, Адміністратор, Редактор, Глядач), які дозволяють або обмежують певні дії на дошці.
Технічні обмеження системи	C	Внутрішні обмеження платформи: правила валідації введених даних, допустимі формати файлів для експорту, ліміти довжини рядків у БД.
Правила відображення фігур	C	Логіка відмальовування об'єктів на клієнті, правила обробки стилів та накладання шарів (z-index) один на один.
Збережений стан онлайн-дошки	O	Фіналізований вигляд дошки після успішної синхронізації та запису до БД PostgreSQL.

Продовження таблиці 1.3

Експортований файл	O	Графічний документ (растровий артефакт у форматі PNG), створений системою для збереження на пристрій клієнта.
Бекенд	M	Програмний комплекс, який відповідає за бізнес-логіку, обробку API-запитів та безпечний зв'язок із базою даних.
Фронтенд	M	Браузерна частина сервісу, що забезпечує користувацький інтерфейс та динамічне відображення графіки.
База даних	M	Сховище, що гарантує цілісність, надійне збереження та оперативну видачу всіх даних проєкту.
Авторизований сеанс та ID дошки	I/C	Зв'язуючий потік: інформація про поточного активного юзера та унікальний номер відкритого робочого простору.
Локальні зміни полотна	I/O	Зв'язуючий потік: проміжний вигляд об'єктів, який утримується в оперативній пам'яті браузера до моменту фіксації на сервері.

1.3 Моделювання потоків даних в інтерактивному середовищі

Формування вимог до програмного продукту найзручніше здійснювати за допомогою діаграм прецедентів (Use Case UML). Цей інструмент описує виключно зовнішню поведінку системи (які саме функції виконуються), абстрагуючись від внутрішньої технічної реалізації (як саме вони працюють). Головна перевага такого підходу полягає у зміщенні фокусу на потреби та варіанти взаємодії кінцевого споживача із сервісом [5].

Графічне подання прецедентів для системи ObjectBoard проілюстровано на 1 сторінці графічного матеріалу. Архітектура взаємодії спирається на чітку ієрархію повноважень. Виділено чотири типи акторів, які успадковують базові можливості один одного: «Глядач» (має мінімальний рівень – лише перегляд), «Редактор», «Адміністратор» та «Власник» (наділений абсолютними правами управління робочим простором та власне проєктом).

Нижче наведено покрокові сценарії основного потоку подій (Main Flow) для двох найважливіших прецедентів: «Внести зміни у вміст дошки» (табл. 1.4) та «Змінити список учасників» (табл. 1.5).

Таблиця 1.4 – Сценарій прецеденту «Внести зміни у вміст дошки»

Крок	Дія актора (Редактор / Власник)	Відповідь системи
1	Актор виділяє потрібний графічний примітив (наприклад, лінію чи еліпс) на панелі інструментів.	Програмний комплекс активує режим редагування та візуально підсвічує обрану іконку.
2	Актор робить клік у бажаному місці робочого простору.	Відбувається генерація та відображення базової фігури зі стандартними властивостями.
3	Актор коригує візуальні параметри (відтінок, товщину контуру) через меню налаштувань.	Клієнтська частина (фронтенд) миттєво відображає застосовані стилістичні зміни.
4	Актор знімає виділення з елемента, завершуючи маніпуляцію.	Формується пакет даних, який автоматично відправляється на бекенд для фіксації в БД.

Таблиця 1.5 – Сценарій прецеденту «Змінити список учасників»

Крок	Дія актора (Редактор / Адміністратор)	Відповідь системи
1	Актор переходить до меню адміністрування списку команди.	На екран виводиться поточний перелік усіх активних учасників та їхніх статусів.
2	Актор заповнює ідентифікатор нового учасника (ім'я). Актор ініціює відправку запрошення відповідною кнопкою.	Сервер здійснює перевірку наявності вказаного облікового запису в базі даних. У таблиці members формується новий запис.

Для візуалізації маршрутів переміщення інформації всередині програмних комплексів застосовується методологія DFD (Data Flow Diagram). Залежно від потреб проєктування, такі моделі масштабуються від концептуальних загальних схем до багаторівневих ієрархічних структур, що детально описують логіку перетворення даних [6].

Контекстна діаграма (рівень 0) розроблюваного сервісу ObjectBoard, наведена на рисунку 1.6, ілюструє зовнішні інформаційні зв'язки платформи. Головний вузол системи тут виступає єдиним центральним процесом, навколо якого розташовані чотири зовнішні сутності – визначені раніше категорії

користувачів. Такий підхід дає можливість наочно розділити напрямки передачі даних, спираючись на ієрархію повноважень кожного з акторів.

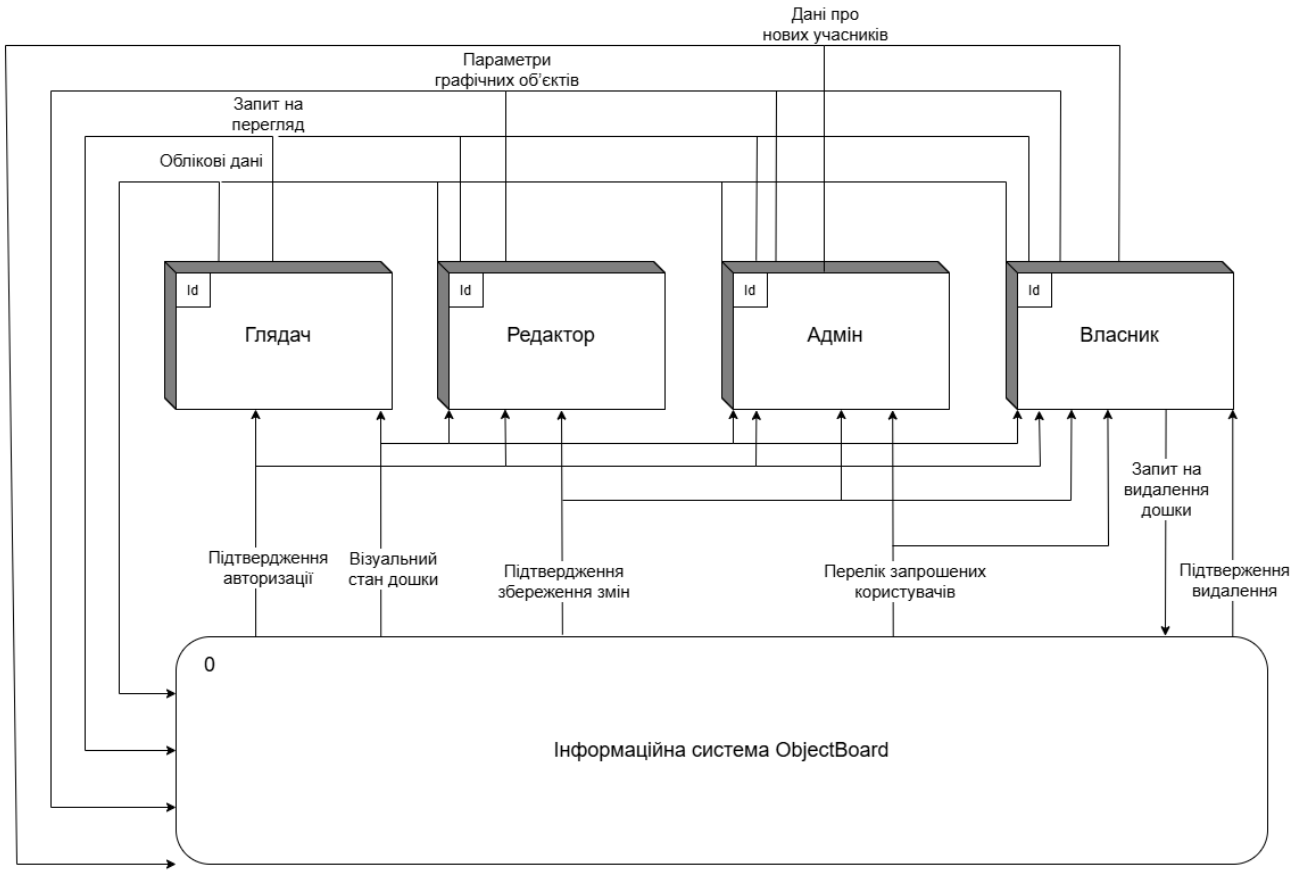

Рисунок 1.6 – Модель інформаційного обміну між учасниками колективної роботи

Базовими вхідними потоками, спільними для всіх типів профілів, виступають запити на відображення робочого простору та авторизаційні відомості. Актори, наділені правом модифікації контенту (Редактор, Адміністратор, Власник), додатково генерують потоки з параметрами візуальних примітивів, які формують вигляд полотна. Своєю чергою, сутності з управлінськими функціями ініціюють передачу інформації щодо складу команди та статусів колег.

Діаграма робочих потоків дозволяє чітко зафіксувати сценарії взаємодії користувача із платформою. Основна відмінність даного підходу від розроблених раніше діаграм полягає в акцентуванні уваги на строгій

хронології подій та причинно-наслідкових залежностях між етапами роботи [7].

З метою глибшого опису алгоритмів функціонування вебсервісу ObjectBoard було розроблено модель IDEF3, яку представлено на 2 сторінці графічного матеріалу. Стартовим кроком змодельованого алгоритму є процедура автентифікації та підтвердження повноважень учасника. Для реалізації вибору між генерацією нового робочого простору та завантаженням уже наявного застосовано ексклюзивний шлюз «виключного або» (XOR). У свою чергу, шлюз типу AND ілюструє синхронність двох процесів: миттєве оновлення візуального інтерфейсу на клієнті та паралельну фіксацію цих же даних на сервері. Щоб прив'язати абстрактні етапи алгоритму до фізичних компонентів архітектури, на схемі задіяно спеціальний вузол Referent, який посилається на СУБД PostgreSQL як на головне джерело збереження поточного стану проєкту.

З метою верифікації цілісності та логічної узгодженості розроблених моделей було побудовано матрицю трасування (таблиця 1.6). Використання такої матриці є надійним способом довести відсутність «ізольованих» елементів: вона підтверджує, що кожна спроєктована сутність БД дійсно бере участь у бізнес-процесах, а будь-яка алгоритмічна дія повністю забезпечена необхідною інформацією для свого виконання.

Таблиця 1.6 – Матриця відповідності кроків IDEF3 та сутностей ERD

Крок процесу (IDEF3)	Користувач	Дошка	Об'єкт	Учасник	Роль
1.0 Авторизувати користувача	X				
1.1 Налаштувати параметри дошки		X		X	
1.2 Завантажити об'єкти з БД		X	X	X	X
1.3 Модифікувати елементи			X		
1.4 Оновити інтерфейс					
1.5 Записати стан у БД			X		
1.6 Експорт результату		X	X		

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА КОМПОНЕНТІВ ІНФОРМАЦІЙНОЇ СИСТЕМИ

2.1 Формалізація алгоритмів колективного створення контенту

Застосування стандарту BPMN 2.0 слугує містком між зібраними вимогами клієнтів та конкретною програмною реалізацією. Ця методологія дає змогу чітко класифікувати події, визначити характер виконання завдань (автоматизований чи ручний) та прописати розгалужену логіку системних рішень [8]. Побудова таких моделей не тільки формує базис для написання програмного коду, але й візуалізує архітектуру обміну даними між користувачем та серверними модулями платформи ObjectBoard.

З метою підтвердження логічної цілісності проєктування була сформована аналітична таблиця 2.1. Вона наочно демонструє, як виявлені на етапі побудови UJM «больові точки» користувачів трансформуються у відповідні технологічні завдання нотації BPMN. Такий підхід доводить доцільність інтеграції кожної функції та кожного алгоритмічного шлюзу в майбутню систему.

Таблиця 2.1 – Відповідність етапів взаємодії користувача операційній логіці сервісу

Етап шляху	**Дія користувача (UJM)**	**Точка контакту**	**Проблема (Біль)**	**Проектована функція ІС (BPMN Task)**
Авторизація	Проходження ідентифікації	Форма логіну	Загроза несанкціонованого доступу до проєкту	Service Task: Верифікація облікових даних
Взаємодія з полотном	Створення та маніпуляція об'єктами	Робоча область (Canvas)	Ризик втрати напрацювань при закритті вкладки	Service Task: Фонове автоматичне збереження

Продовження таблиці 2.1

Командна робота	Делегування прав доступу колегам	Меню адміністрування учасників	Плутанина з версіями схем та повноваженнями	User Task: Призначення ролей за матрицею доступу
Експортування	Вивантаження результату на диск	Кнопка експорту	Складність інтеграції схем у звітну документацію	Service Task: Рендеринг об'єктів полотна у зображення

Для системи ObjectBoard розроблено три діаграми (3 та 4 сторінки графічного матеріалу), які ілюструють процеси різного рівня складності.

Перша діаграма описує базовий лінійний сценарій ініціалізації нового робочого середовища. Тригером процесу виступає дія авторизованого користувача на головному екрані. Схема відображає серверну логіку валідації введеної інформації, автоматичного створення унікального ID та присвоєння автору статусу «Власник». Для захисту бази даних від запису некоректних даних (наприклад, порожньої назви дошки) передбачено ексклюзивний шлюз (XOR), який обробляє виняткові ситуації.

Другою діаграмою є алгоритм фонового збереження графічних маніпуляцій. Цю модель розбито на три горизонтальні доріжки (Lanes): Користувач, Фронтенд та Бекенд. Діаграма чудово передає асинхронний характер роботи сучасних вебдодатків: за рахунок використання шлюзу паралельного розгалуження система миттєво відмальовує зміни на екрані юзера і водночас відправляє запит на сервер через API. Проміжні події-повідомлення використані для моделювання стану очікування відповіді від бази даних. Лише після успішного об'єднання обох паралельних потоків клієнтський інтерфейс виводить статус «Збережено».

Найбільш комплексною є діаграма налаштування спільного доступу. Її архітектура побудована на базі двох окремих пулів: клієнтського («Власник») та серверного («Бекенд»). Пул ініціатора додатково розділений на доріжки

людини та інтерфейсу, що забезпечує чітке розуміння того, де закінчується дія юзера і починається реакція системи. Обмін даними між незалежними пулами реалізується через потоки повідомлень, що імітують мережеві запити.

Центральна логіка цього процесу спирається на перевірку існування запрошеного користувача, яку виконує бекенд за допомогою XOR-шлюзу. За результатами пошуку у БД сервер генерує відповідь: фіксація успішного запису в таблицю members або повідомлення про відсутність акаунта. Інтерфейс перехоплює ці статуси через події-обробники. Важливою інженерною деталлю цієї моделі є інтегрований цикл зворотного зв'язку: у разі помилки алгоритм повертає систему на крок введення email-адреси. Це мінімізує збої та гарантує надійність управління доступами.

2.2 Побудова сценаріїв поведінки та переходів у візуальному полі

Розробка User Journey Map (UJM) є системним підходом до візуалізації та аналізу клієнтського досвіду. Цей інструмент дозволяє відстежити весь цикл взаємодії людини з продуктом, фіксуючи її емоційний стан, послідовність дій та перешкоди на кожному етапі у хронологічному порядку [9].

У таблиці 2.2 розроблено UJM для профілю «Власник». Головна увага тут приділяється етапам первинного налаштування робочого простору та генерації базового контенту.

Таблиця 2.2 – Опис шляху актора «Власник»

Етап	**Дії та точки контакту**	**Емоції та Думки**	**Проблеми та Можливості**
1. Усвідомлення потреби	Дії: Пошук інструменту для побудови схем. Точки: Пошукова система, лендінг.	Емоція: Інтерес (0). Думка: «Потрібен простий сервіс для швидких схем».	Біль: Редактори занадто складні. Рішення: Лаконічний інтерфейс із фокусом на швидкість.

Продовження таблиці 2.2

2. Реєстрація та вхід	Дії: Заповнення форми, верифікація email. Точки: Сторінка Sign-up.	Емоція: Очікування (-1). Думка: «Хоч би це не зайняло багато часу».	Біль: Введення великої кількості даних. Рішення: Мінімалістична форма реєстрації.
3. Створення простору	Дії: Створення дошки, генерація перших фігур. Точки: Дашборд, робоче полотно.	Емоція: Натхнення (+1). Думка: «Чи вистачить тут фігур для мого проєкту?».	Біль: Незручності при малюванні об'єктів. Рішення: Додавання гарячих клавіш.
4. Колективна робота	Дії: Копіювання лінка, відправка колегам. Точки: Вікно «Share», робоча область.	Емоція: Задоволення (+2). Думка: «Чи легко іншим буде зайти сюди?».	Біль: Загроза зміни схеми колегами без потреби. Рішення: Делегування прав доступу (ролей).
5. Завершення та експорт	Дії: Перевірка схеми, завантаження файлу. Точки: Меню файлу, експорт у PNG.	Емоція: Полегшення (+1). Думка: «Чи достатня чіткість для презентації?».	Біль: Складна конвертація векторів у растр. Рішення: Автоматична оптимізація при вивантаженні.

Як видно з графіка емоцій Власника (рис. 2.1), на старті спостерігається невелике зниження задоволеності, спровоковане рутинністю процесу реєстрації. Однак, як тільки користувач переходить до створення схем, показники демонструють впевнене зростання.

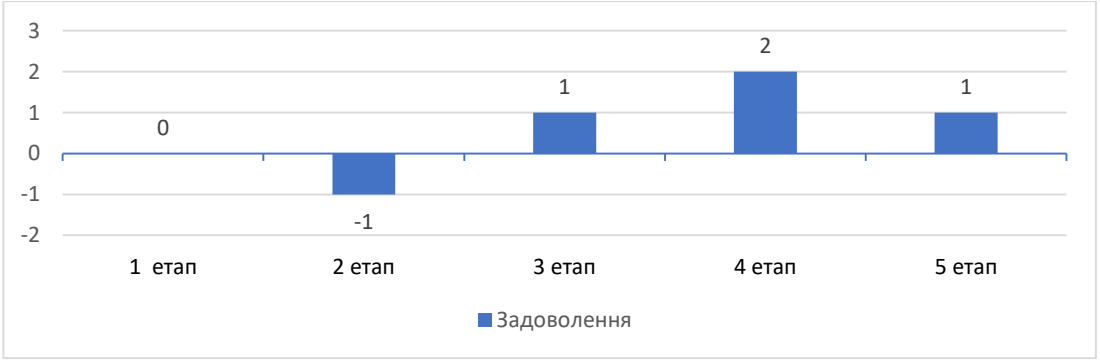

Рисунок 2.1 – Крива емоцій «Власника»

Досвід іншого актора – «Редактора» (табл. 2.3) – суттєво відрізняється, адже він доєднується до вже існуючого середовища. Його головна потреба полягає у миттєвому зануренні в контекст.

Таблиця 2.3 – Опис шляху актора «Редактор»

Етап	**Дії та точки контакту**	**Емоції та Думки**	**Проблеми та Можливості**
1. Отримання доступу	Дії: Перехід за наданим лінком. Точки: Месенджер, сторінка входу.	Емоція: Скептицизм (-1). Думка: «Що це за чергова програма?».	Біль: Небажання довго шукати потрібний проєкт після авторизації. Рішення: Фільтри і можливість пошуку на дашборді.
2. Входження в контекст	Дії: Огляд полотна, масштабування. Точки: Робоча область дошки.	Емоція: Зосередженість (0). Думка: «Логіка ясна, але тут помилка».	Біль: Втрата орієнтації на безкінечному полотні. Рішення: Інтуїтивна навігація та плавне масштабування робочої зони.
3. Активне редагування	Дії: Додавання блоків, налаштування кольору. Точки: Панель інструментів, фігури.	Емоція: Ентузіазм (+1). Думка: «Зміню колір, щоб підкреслити блок».	Біль: Складність у розумінні того, який об'єкт зараз редагується. Рішення: Візуальне підсвічування обраного елемента та зручна бічна панель властивостей.
4. Комунікація	Дії: Створення текстових приміток. Точки: Інструмент «Текст».	Емоція: Впевненість (+1). Думка: «Опишу, чому я це змінив».	Біль: Складність пояснити колегам суть своїх візуальних правок. Рішення: Можливість вільного розміщення текстових блоків-коментарів прямо поруч із фігурами.

Продовження таблиці 2.3

5. Збереження результату	Дії: Перевірка фіксації змін. Точки: Індикатор «Saved».	Емоція: Спокій (+2). Думка: «Чи не зникнуть правки після закриття?».	Біль: Страх втратити напрацювання. Рішення: Динамічний статус збереження.

Крива емоцій актора «Редактор» (рис. 2.2) демонструє стабільний висхідний тренд. Початкова недовіра до нового інструментарію швидко зникає завдяки безшовному входу у проєкт. Найвища точка емоційного комфорту фіксується у фіналі: користувач бачить результат своєї праці та відчуває спокій за збереження даних, що знижує загальний стрес від виконання робочих завдань.

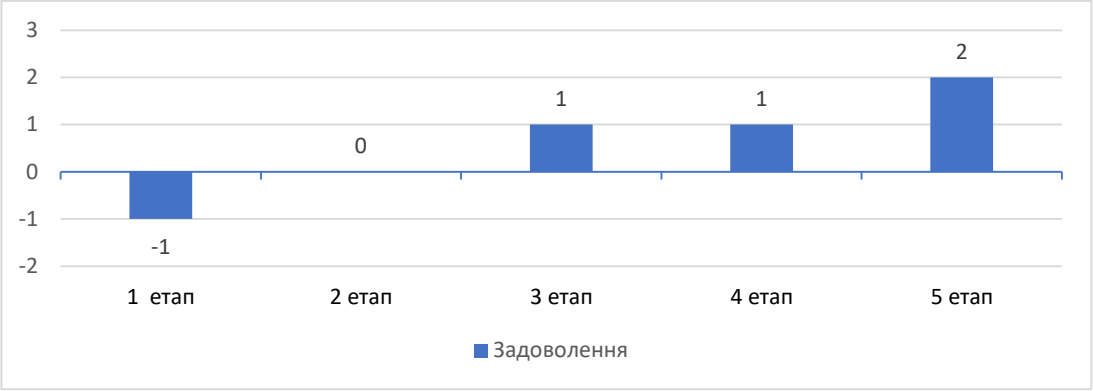

Рисунок 2.2 – Крива емоцій «Редактора»

Зведені дані обох карт засвідчують, що найслабшими місцями (Pain Points) є поріг входу у систему та прозорість збереження інформації. Ці фактори стали ключовими орієнтирами при проєктуванні UI/UX, що вилилося у розробку помітних статусів синхронізації та оптимізацію роботи з графікою.

З метою довгострокового та стратегічного планування була складена CJM (таблиця 2.4). На відміну від UJM, яка фіксує точкові технічні кроки, ця карта відображає еволюцію взаємовідносин клієнта з продуктом. Фокусним актором обрано Проєктного менеджера, адже саме він ініціює використання нових технологій та відповідає за налагодження комунікації в колективі.

Таблиця 2.4 – Аналіз досвіду актора «Проектний менеджер»

Етап	**Дії та точки контакту**	**Біль (Pain Points)**	**Зв'язок з бізнес-метою проєкту**
1. Усвідомлення	Шукає способи візуалізації Agile; читає статті.	«Чати переповнені скріншотами, ніхто не знає, де остання версія схеми».	Формування чіткої потреби у єдиному візуальному хабі.
2. Розгляд	Переглядає огляди ObjectBoard, порівнює експорт з аналогами.	«Чи не виявиться програма надто складною? Чи підтримується експорт PNG?».	Презентація переваг лаконічного інструментарію.
3. Вибір	Реєструється, створює тестове полотно для команди.	Страх за безпеку інформації та труднощі первинного налаштування.	Старт процесу автоматизації створення звітної документації.
4. Використання	Моделює архітектуру в реальному часі разом із колегами.	«Нарешті з'явилося "єдине джерело істини" для наших схем».	Ключовий етап: підтвердження 30% економії часу на узгодженнях.
5. Лояльність	Архівує старі проєкти, робить сервіс щоденним інструментом.	«Це наш новий стандарт. Усі дискусії ведуться навколо дошки».	Стабілізація процесів, усунення хаосу при обміні концепціями.
6. Адвокація	Рекомендує продукт іншим відділам, залишає відгук.	«Потрібно впровадити це у суміжних командах, щоб уникнути плутанини».	Підтвердження рентабельності та успішної інтеграції ПЗ у бізнес-процеси.

Розроблена карта має вирішальне значення для валідації проєкту, оскільки наочно доводить, як суто технічний функціонал конвертується у реальну цінність для бізнесу.

Підсумовуючи, CJM підтверджує, що впровадження ObjectBoard ефективно нейтралізує проблему фрагментованої комунікації. Успішне

проходження клієнтом усіх етапів карти гарантує не лише закриття технічних вимог, але й реальне підвищення продуктивності розподілених команд.

З метою наочного представлення майбутнього інтерфейсу платформи ObjectBoard було створено базові структурні макети (вайрфрейми). Цей підхід застосовується для тестування зручності навігації та логіки взаємодії з системою, повністю ігноруючи графічний дизайн (кольори, шрифти тощо). Відділення етапу проєктування структури від візуального оформлення допомагає розробникам сфокусуватися виключно на розташуванні елементів та їхній корисності. Завдяки цьому ще до початку написання коду можна оцінити доцільність кожної функції та за необхідності прибрати зайві деталі, які не відповідають головним завданням продукту [10].

Специфікацію п'яти базових екранів, які покривають ключові сценарії використання (Use Cases) системи, зведено у таблицю 2.5.

Таблиця 2.5 – Специфікація екранів інтерфейсу

Назва екрана	**Реалізований Use Case**	**Перелік UI-елементів**	**Навігаційні переходи**
1. Головна сторінка	Зареєструватися	Інформаційні блоки, кнопки «Логін» / «Реєстрація», кнопка заклику до дії (CTA).	Клік на вхід/реєстрацію відкриває відповідні форми.
2. Форма входу	Автентифікуватися	Поля введення (email, пароль), опція «Запам'ятати мене», кнопка підтвердження.	Успішний вхід перенаправляє у Каталог дошок.
3. Каталог дошок	Відкрити дошку	Навігаційні вкладки, сітка проєктів, кнопка «Нова дошка».	Клік на картку відкриває Робочий простір.
4. Форма створення	Створити нову дошку	Текстові поля для назви та опису проєкту, кнопка генерації простору.	Підтвердження переводить юзера безпосередньо в редактор.

Продовження таблиці 2.3

5. Робочий простір	Внести зміни у вміст; Модифікувати елементи	Інтерактивний канвас, ліва панель інструментів, права панель властивостей.	Кнопка «Назад» повертає до Каталогу.

Головна сторінка (рис. 2.3) виконує роль презентаційного майданчика для нових відвідувачів. Структурно вона поділена на верхній навігаційний бар (із логотипом та посиланнями на форми доступу) і центральний блок. У центрі розміщено ключову інформацію про переваги продукту, яка підводить користувача до кнопки швидкого старту (Call to Action). Звідси починається шлях клієнта, який веде до створення облікового запису або авторизації.

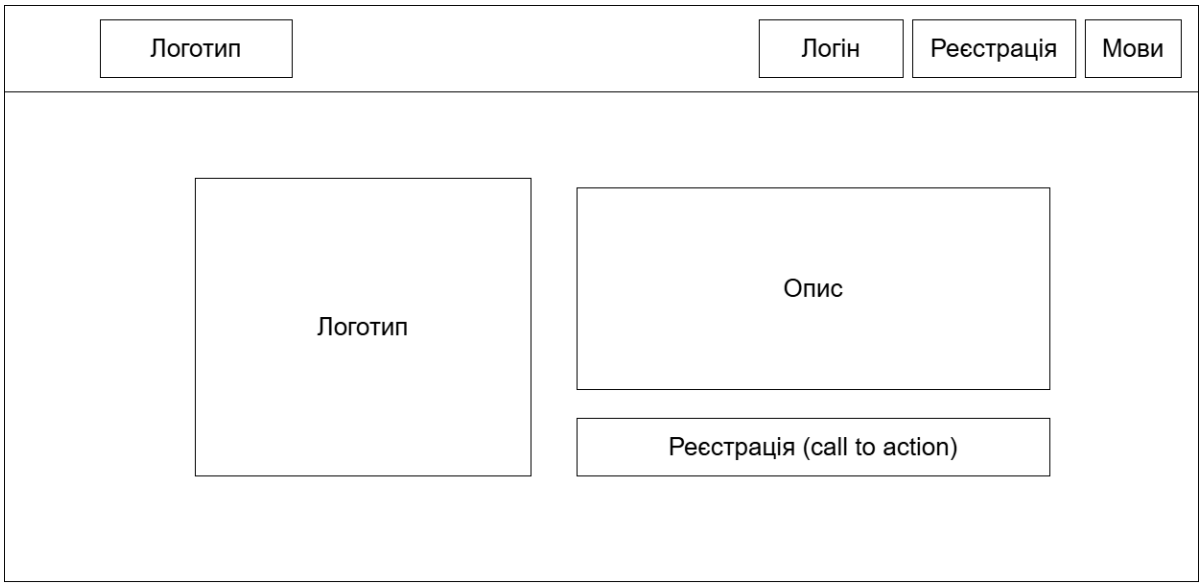

Рисунок 2.3 – Прототип головної сторінки

Екран авторизації (рис. 2.4) відповідає за виконання прецеденту «Автентифікуватися». Його дизайн максимально мінімалістичний: по центру розташовано блок із полями для введення пошти та пароля. Відсутність зайвих графічних елементів дозволяє юзеру швидко сфокусуватися на процесі входу. У разі успішної перевірки даних сервер автоматично направляє людину до її персонального кабінету.

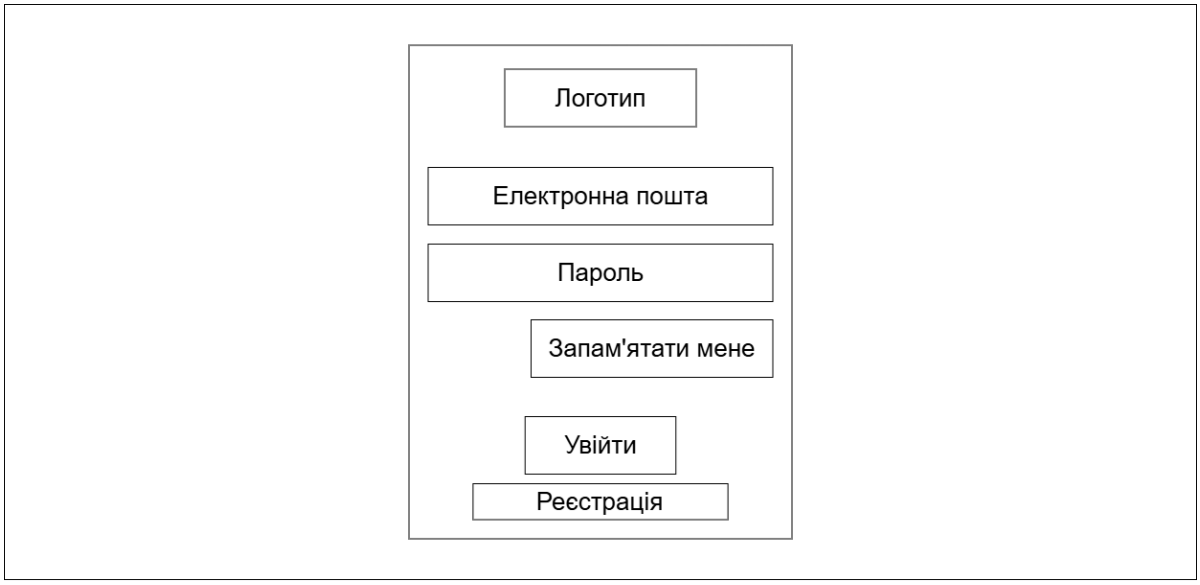

Рисунок 2.4 – Прототип сторінки авторизації

Персональний кабінет, або Каталог дошок (рис. 2.5), слугує основним місцем для управління проєктами та пошуку необхідних схем. Екран складається з шапки (де розташований профіль) та робочої зони. Проєкти відображаються у вигляді зручної сітки карток, що спрощує орієнтацію. Для сортування інформації передбачено вкладки («Мої дошки» та «Запрошення»). Саме з цього екрана юзер може в один клік відкрити існуючу дошку або викликати меню для створення нової.

Рисунок 2.5 – Прототип каталогу дошок

Модуль ініціалізації проєкту (рис. 2.6) реалізує прецедент «Створити нову дошку». Він вимагає від користувача введення найнеобхідніших даних – назви та короткого опису схеми. Натискання кнопки підтвердження завершує процес і відразу завантажує інтерфейс редактора.

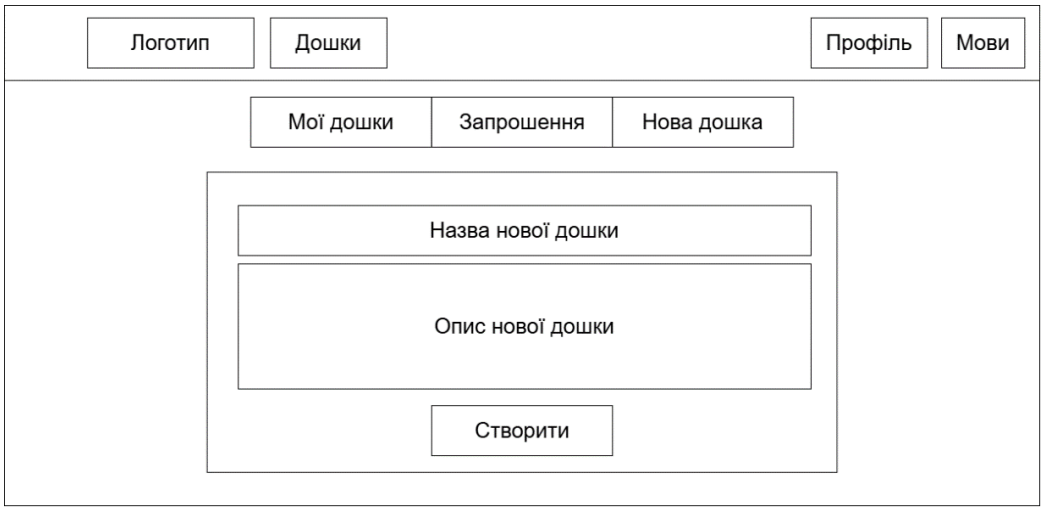

Рисунок 2.6 – Прототип форма створення нової дошки

Безпосередня робота відбувається у робочому просторі (рис. 2.7)., який покриває прецеденти «Внести зміни у вміст» та «Створити об’єкт». Композиція екрана наслідує стандарти графічних редакторів: центральну частину займає безмежне полотно, ліворуч закріплено набір примітивів, а праворуч панель параметрів для налаштування виділеного об'єкта.

Рисунок 2.7 – Прототип робочого простору

Запроєктований ланцюжок екранів формує цілісний, логічно завершений шлях: від першого знайомства із системою на головній сторінці до повноцінного занурення у процес візуального моделювання в робочому просторі.

З метою трансформації концептуальної моделі даних у площину практичної реалізації програмного коду було спроєктовано UML-діаграму класів (рис. 2.8).

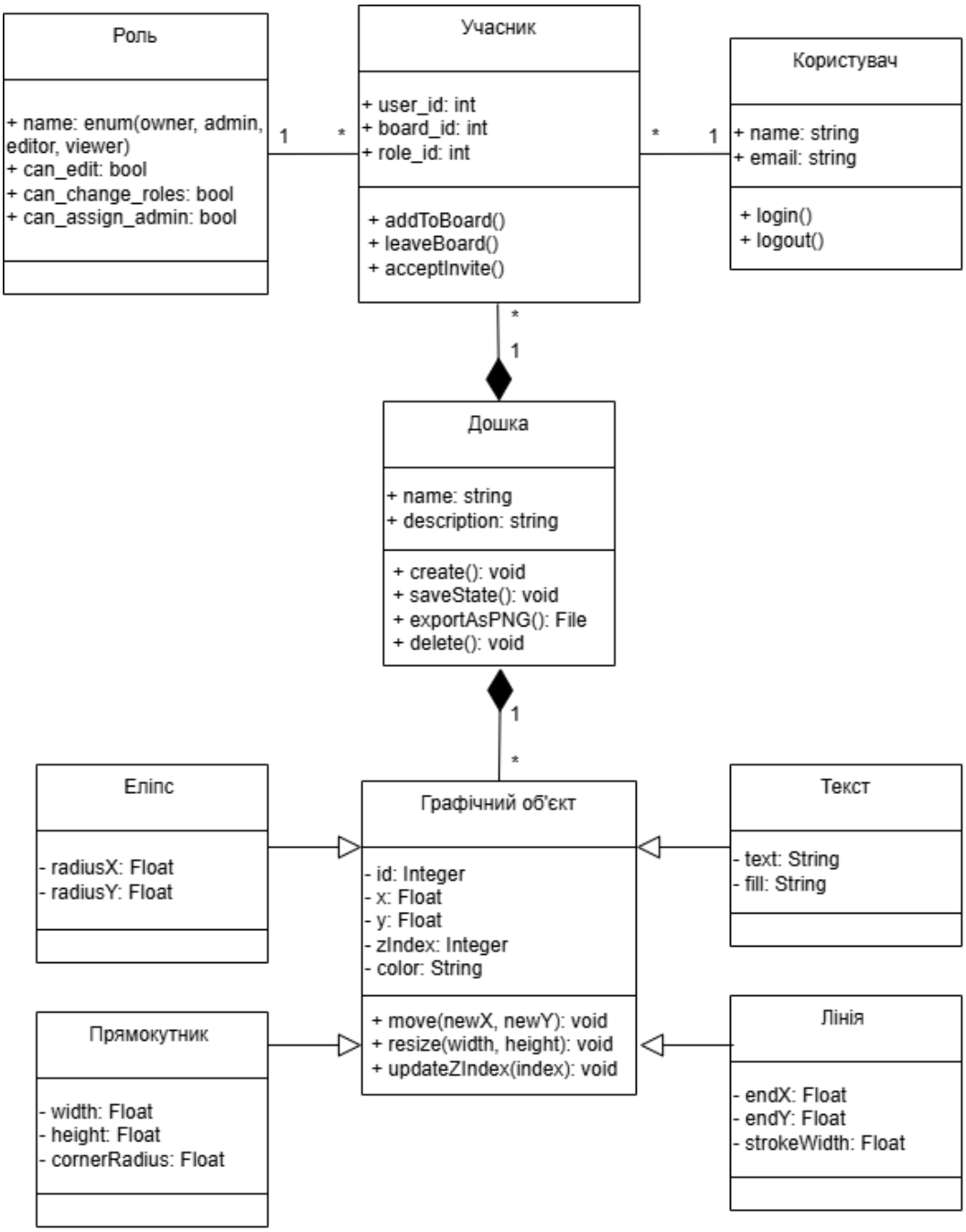

Рисунок 2.8 – Діаграма класів

Ця схема ілюструє архітектуру системи ObjectBoard з позиції об'єктно-орієнтованого програмування. Її головна відмінність від структури БД полягає в тому, що окрім інкапсульованих властивостей (атрибутів), тут зафіксовані поведінкові аспекти – методи (функції) кожної сутності та їхня ієрархічна підпорядкованість.

Ядром представленої структури є клас Board (Дошка), що виконує функцію базового контейнера. За допомогою відношення композиції він жорстко пов'язаний із сутностями Member (Учасник) та CanvasObject (Графічний об'єкт). Такий тип зв'язку гарантує, що життєвий цикл елементів на полотні та прив'язка користувачів до проєкту залежать виключно від існування самої дошки. При цьому клас Member діє як проміжний вузол, що об'єднує фізичну особу з конкретним робочим простором та визначає її повноваження за допомогою зв'язку з класом Role (Роль).

Важливою архітектурною рисою системи є застосування механізму поліморфізму та успадкування для візуальних примітивів. Було створено абстрактний батьківський клас CanvasObject, у якому містяться універсальні параметри: позиціонування на осях координат, z-індекс та колір контуру. Від нього успадковуються вузькоспрямовані дочірні класи (Text, Ellipse, Rectangle, Line), які розширюють загальну модель специфічними атрибутами (наприклад, розміром шрифту або радіусом заокруглення). Завдяки такому підходу програма здатна стандартизовано взаємодіяти з будь-якими фігурами, що суттєво оптимізує процеси їх рендерингу, переміщення та видалення.

Основне призначення UML-діаграм послідовності полягає у візуалізації динамічного обміну повідомленнями між компонентами системи у строгій хронологічній конкатенації. Хоча цей інструмент традиційно вважається суто технічним артефактом для розробників, він має високу цінність і для бізнес-аналітиків. Подібні схеми дозволяють прозоро продемонструвати логіку функціонування сервісу через призму взаємодії об'єктів. Більше того, вони слугують потужним засобом для фіксації поточних процесів та формулювання чітких вимог до майбутнього функціоналу програмного забезпечення [11].

Щоб розкрити внутрішню динаміку обміну даними у вебдодатку ObjectBoard, було спроєктовано діаграми послідовності для двох найважливіших Use Cases. Враховуючи, що застосунок базується на стандартній архітектурі «клієнт-сервер» (без застосування постійного з'єднання реального часу), усі інформаційні транзакції генеруються виключно як прямі HTTP-запити від клієнтської частини до серверної. На рисунку 2.9 змодельовано процес ініціалізації проєкту – «Створення нової дошки».

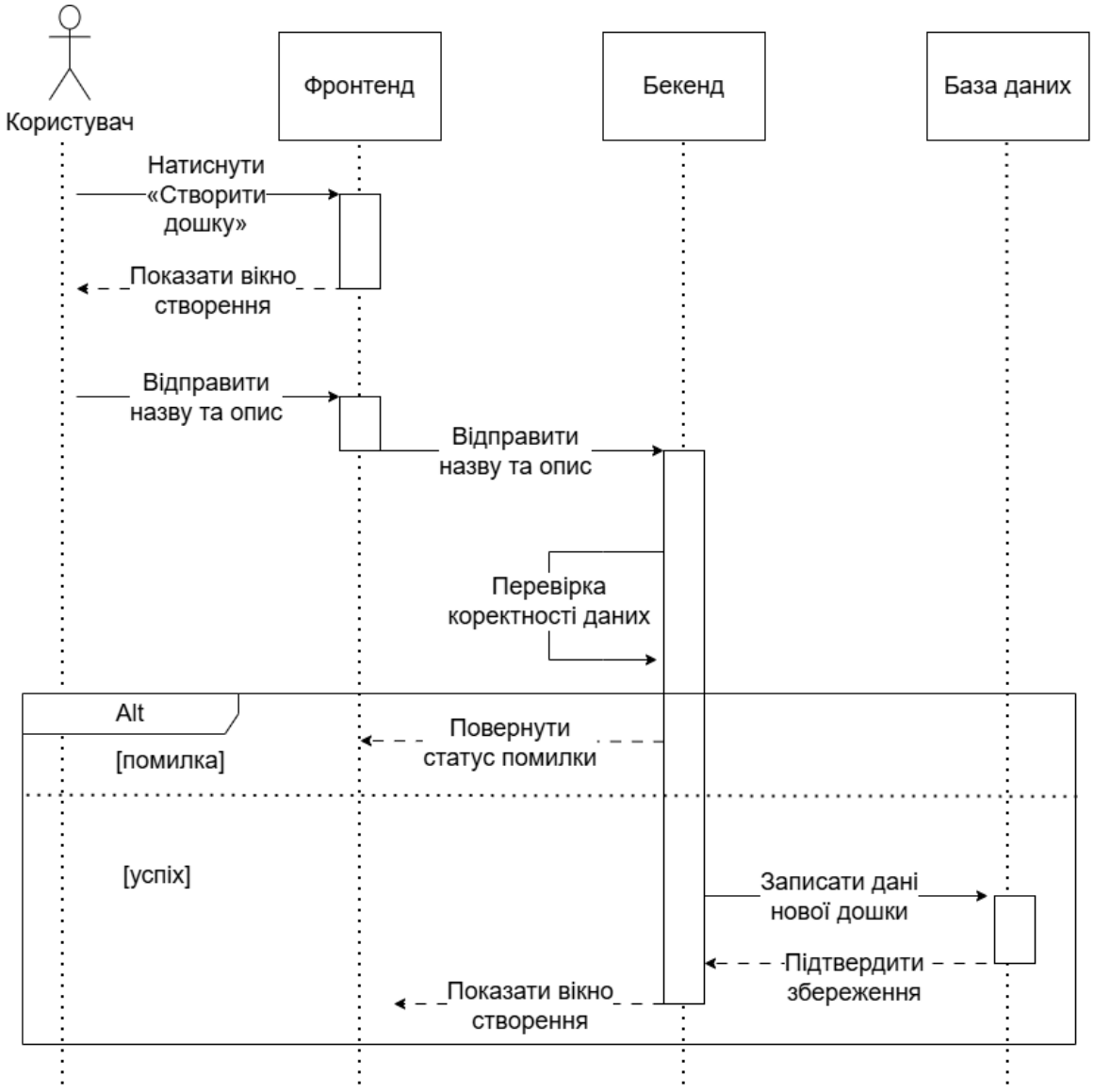

Рисунок 2.9 – Процес створення нової дошки

Тригером виступає дія користувача у браузері: після підтвердження форми фронтенд надсилає пакет інформації на API-сервер. Бекенд бере на себе

функцію перевірки отриманих параметрів. Цей етап валідації відображено за допомогою комбінованого фрагмента альтернативного вибору (alt). Якщо дані не задовольняють умови (наприклад, відсутнє ім'я дошки), сервер відхиляє запит і миттєво генерує помилку. Якщо ж валідація пройдена, ініціюється транзакція до бази даних для збереження сутності. Процес фіналізується передачею ідентифікатора створеної дошки на фронтенд, що супроводжується автоматичним редиректом юзера в робоче середовище.

Логіка прецеденту «Запрошення нового учасника», який відіграє критичну роль у налагодженні командної співпраці, розкривається на рисунку 2.10.

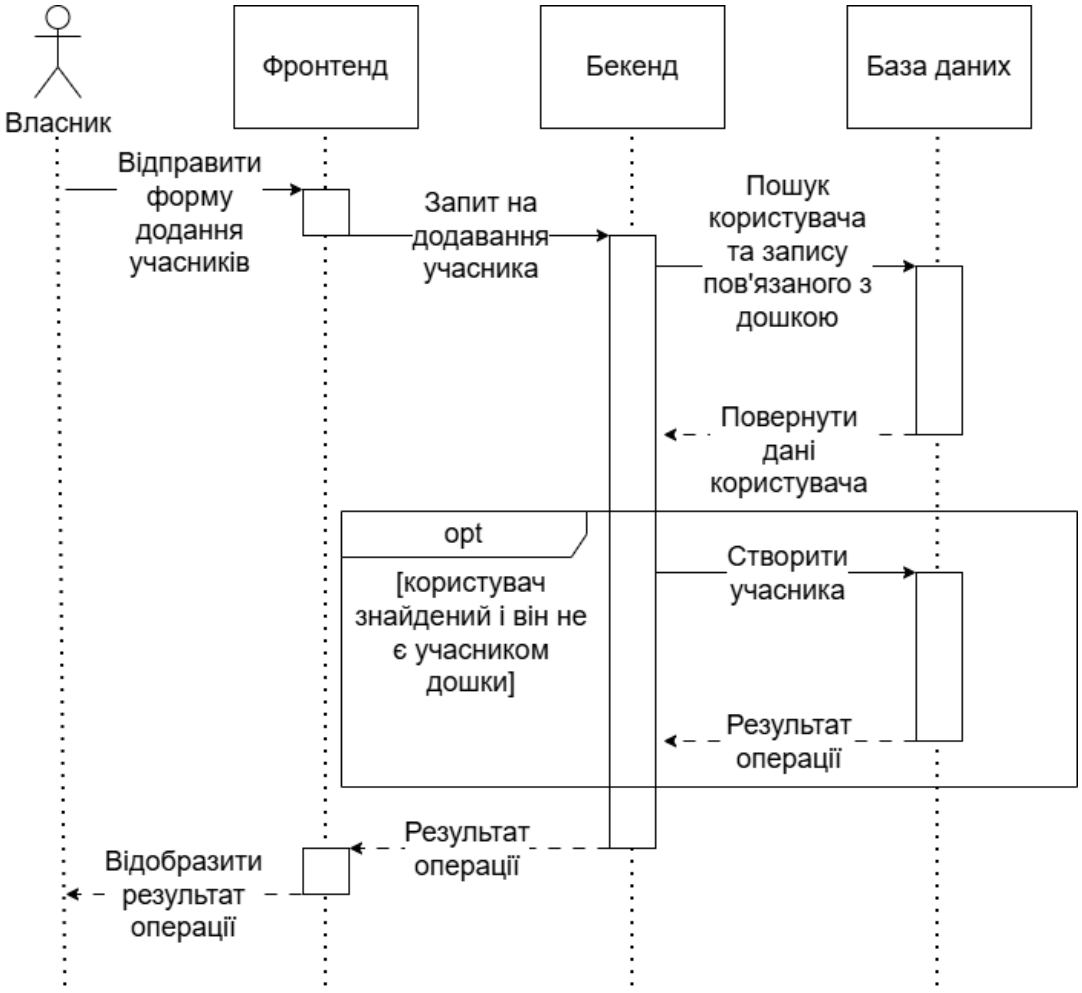

Рисунок 2.10 – Процес запрошення нового учасника

Власник проєкту відправляє запит через інтерфейс, передаючи email колеги та бажаний статус доступу. Серверна частина першочергово опитує

базу даних щодо наявності вказаного облікового запису. За допомогою оператора опціонального виконання проілюстровано подальший розвиток подій: якщо профіль знайдено в системі, у БД генерується новий зв'язковий запис, що наділяє користувача відповідними правами. Після успішного виконання транзакції фронтенд отримує сигнал від сервера, оновлює таблицю на екрані та генерує сповіщення про успішно відправлене запрошення.

2.3 Проєкт бази даних

Проєктування підсистеми зберігання інформації здійснювалося за класичною трирівневою парадигмою. На старті було розроблено концептуальну архітектуру, яка згодом трансформувалася в логічну модель із проведенням процедур нормалізації.

Для наочної візуалізації зв'язків між ключовими інформаційними об'єктами платформи застосовується методологія ER-діаграм (Entity-Relationship). Згідно з канонами інженерії програмного забезпечення, побудова такої концептуальної схеми передує етапу фізичної реалізації та дозволяє зафіксувати фундаментальну архітектуру даних без прив'язки до технічних особливостей конкретної СУБД [12].

На початковій стадії моделювання було виокремлено чотири базові сутності предметної галузі: «Користувач», «Дошка», «Графічний об’єкт» та «Роль». Як демонструє концептуальна схема (рис. 2.11), структурна ієрархія базується на тому, що Дошка виступає своєрідним контейнером для безлічі Графічних об'єктів. Оскільки ключовою фічею сервісу є командна робота, відношення між Користувачами та Дошками реалізовано за принципом «багато-до-багатьох» (кожна особа може мати доступ до різних просторів, а простір – містити різних учасників). Сутність «Роль» впроваджено як зв'язуючу ланку для чіткого розмежування прав кожного конкретного юзера в межах конкретного проєкту.

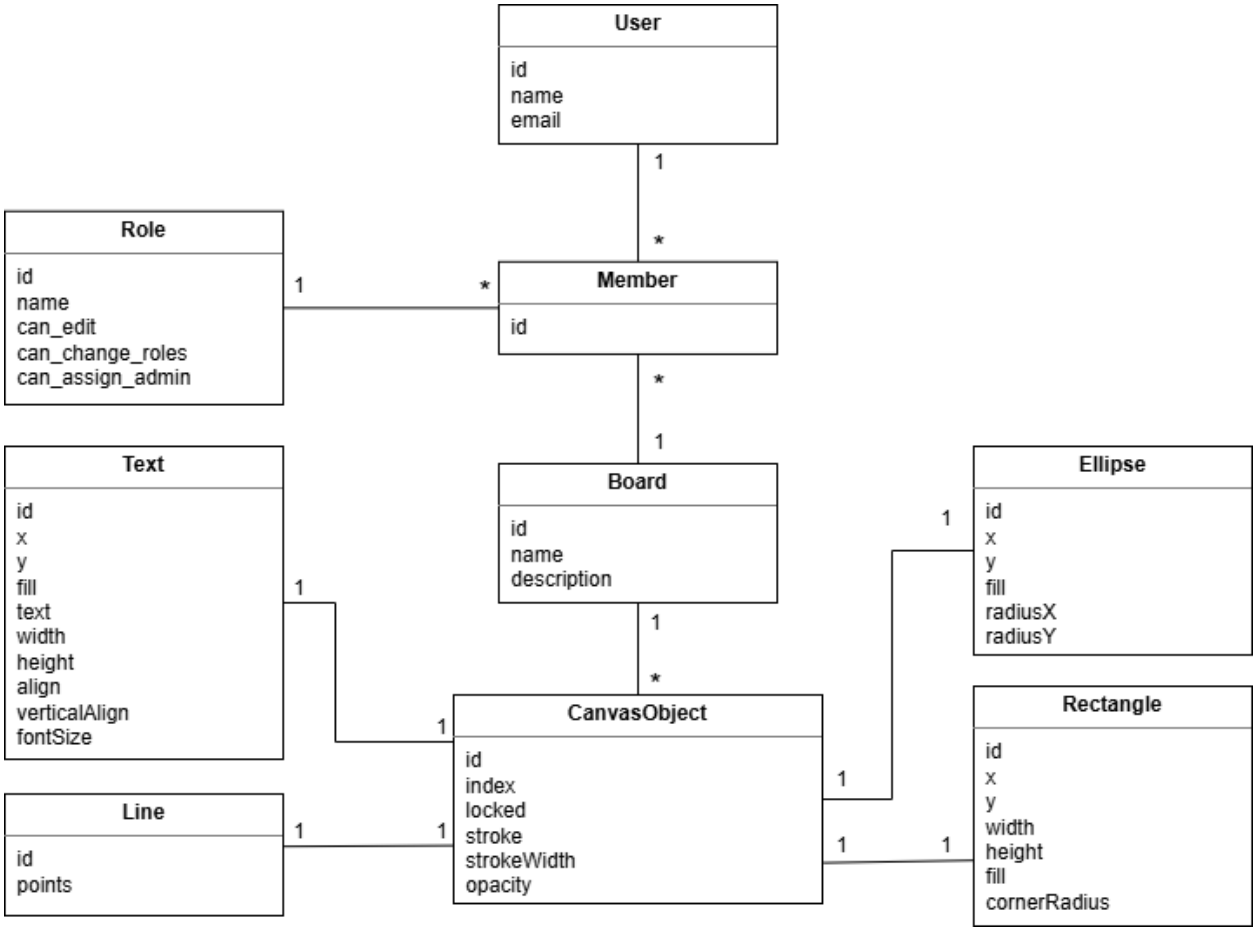

Рисунок 2.11 – Концептуальна модель даних

Наступним кроком еволюції проєкту БД є перехід від ER-діаграми до логічної моделі. Цей етап характеризується поглибленою деталізацією: сутності перетворюються на таблиці, для кожної з них прописуються списки атрибутів, а також жорстко фіксуються первинні та зовнішні ключі для підтримки реляційної цілісності [12].

Отримана логічна модель (5 сторінка графічного матеріалу) детально описує структуру зберігання інформації. Окремо варто відзначити інженерний підхід до збереження графічних примітивів. Розробником було обрано патерн «Class Table Inheritance» (успадкування на рівні таблиць). Загальні характеристики, притаманні всім фігурам (z-індекс, статус блокування, рівень прозорість), записуються до батьківської таблиці canvas_objects. Натомість унікальні властивості конкретних об'єктів (вміст тексту, радіус еліпса чи координати точок) зберігаються у відповідних підлеглих таблицях.

Оптимізація структури реляційної бази здійснювалася через процес нормалізації. Вимога першої нормальної форми (1НФ) щодо атомарності атрибутів успішно дотримана для більшості таблиць (адреси електронної пошти, імена чи статуси є неподільними даними). Проте під час проєктування таблиці lines (лінії) було прийнято свідоме архітектурне рішення на користь використання типу даних «масив» (array) для зберігання координат. Попри те, що з погляду суворої теорії баз даних це є порушенням 1НФ, у контексті специфіки PostgreSQL та роботи з графікою цей крок є критично необхідним. Збереження кожної крапки лінії як окремого рядка у БД призвело б до геометричного зростання навантаження та падіння швидкодії під час рендерингу. Тому масив точок розглядається як єдиний, неподільний об'єкт (суцільний геометричний шлях).

Приведення схеми до другої нормальної форми (2НФ) вимагає ліквідації часткової залежності неключових полів від компонентів складеного ключа. Це правило успішно реалізовано у таблиці зв'язків members. Замість використання складеного ключа, кожен запис тут отримав власний сурогатний id, від якого напряму залежать усі додаткові атрибути (такі як призначена роль).

На завершальному етапі архітектуру було валідовано на відповідність третій нормальній формі (3НФ), що категорично забороняє наявність транзитивних залежностей. Найкращою ілюстрацією виконання цієї вимоги є виокремлення повноважень у самостійну таблицю roles. Перелік дозволів (на редагування малюнків чи запрошення колег) залежить виключно від первинного ключа самої Ролі, а не від даних Учасника. Завдяки цій оптимізації, зміна системних політик для статусу «Редактор» автоматично і миттєво поширюється на всіх відповідних користувачів без потреби редагувати кожен їхній запис індивідуально.

Щоб краще розкрити внутрішню організацію сховища даних вебсервісу ObjectBoard, далі представлено детальну специфікацію стовпців для кожної розробленої таблиці:

1. Таблиця users (Користувачі) – відповідає за збереження реєстраційної інформації про клієнтів платформи:

– id (int): Первинний ключ (PK), який є унікальним ідентифікатором профілю;

– name (varchar): Обов'язкове поле з іменем особи, на яке накладено унікальний індекс;

– email (varchar): нікальна електронна адреса, що слугує логіном при вході;

– encrypted_password (varchar): Хешований рядок пароля, що гарантує захист облікового запису від злому.

2. Таблиця boards (Дошки) – головний елемент архітектури, навколо якого створюються проєкти:

– id (int): Первинний ключ (PK);

– name (varchar): Заголовок робочого простору, який відображається в UI-інтерфейсі;

– description (varchar): Необов'язкове (опціональне) поле для розширеного текстового опису суті проєкту.

3. Таблиця roles (Ролі) – містить конфігурації повноважень для розмежування прав у команді:

– id (int): Первинний ключ (PK);

– name (int): Числове представлення статусу для програмного зв'язку з переліком (enum: owner, admin, editor, viewer);

– can_edit, can_change_roles, can_assign_admin (boolean): Набір булевих прапорців, які вмикають або вимикають конкретні привілеї в системі.

4. Таблиця members (Учасники дошки) – проміжна (асоціативна) сутність, що фіксує зв'язок між людьми та конкретними проєктами:

– id (int): Первинний ключ (PK);

– user_id (int): Зовнішній ключ (FK), що вказує на ідентифікатор учасника;

– board_id (int): Зовнішній ключ (FK), що прив'язує запис до певного полотна;

– role_id (int): Зовнішній ключ (FK), що визначає рівень доступу особи саме у цьому просторі.

Обмеження: Для уникнення дублів у команді на рівні БД створено складений унікальний індекс для комбінації полів user_id та board_id.

5. Таблиця canvas_objects (Об'єкти полотна) – батьківська таблиця, що зберігає універсальні параметри для будь-якої фігури:

– id (int): Первинний ключ (PK);

– board_id (int): Зовнішній ключ (FK), який показує належність графічного елемента до конкретної дошки;

– index (int): Значення z-індексу для розрахунку перекриття шарів (є унікальним показником на одному полотні);

– locked (boolean): Прапорець, що забороняє редагування об'єкта (захист від випадкового зміщення);

– stroke (varchar), strokeWidth (int): Налаштування стилю контуру (відтінок та товщина лінії відповідно);

– opacity (decimal/float): Значення прозорості візуального примітива.

6. Таблиця texts (Текстові блоки) – містить специфічні атрибути інструменту написів:

– canvas_object_id (int): Зовнішній ключ (FK), посилання на базовий об'єкт;

– x, y (int/float): Точні координати позиціювання тексту на осях екрану;

– text (varchar): Безпосередньо набраний користувачем текстовий рядок;

– fontSize, align, verticalAlign (int): Дані для стилізації шрифту та параметри його вирівнювання.

7. Таблиця rectangles (Прямокутники) – відповідає за прямокутні графічні форми:

– canvas_object_id (int): Зовнішній ключ (FK) на загальну таблицю фігур;

– width, height (int): Геометричні розміри фігури;

– fill (varchar): Колір фонової заливки (підтримує HEX-коди або текстові назви);

– cornerRadius (int): Значення для згладжування (заокруглення) кутів форми.

8. Таблиця ellipses (Еліпси) – зберігає параметри для об'єктів без кутів (овали, кола):

– canvas_object_id (int): Посилання на базовий об'єкт (FK);

– radiusX, radiusY (int): Значення радіусів по горизонтальній та вертикальній осях;

– fill (varchar): Колір Відтінок внутрішнього заповнення фігури.

9. Таблиця lines (Лінії) – фіксує результати інструменту вільного малювання або з'єднань:

– canvas_object_id (int): Посилання на базовий об'єкт (FK);

– points (float[]): Масив числових координат, що формують геометричний шлях відмальованої кривої.

Спроєктована таким чином конфігурація таблиць та їхніх полів дає можливість платформі ObjectBoard без затримок обробляти насичені графічні сцени. Окрім швидкодії, грамотне використання зовнішніх ключів (FK) та оптимізованих індексів створює надійний фундамент для ієрархічної системи доступів та гарантує цілісність збережених даних.

2.4 Вибір технологій розробки програмного забезпечення

Фундаментом бекенду розроблюваної платформи виступає Ruby on Rails. Даний фреймворк, створений на базі мови Ruby, містить набір базових конвенцій, що суттєво прискорюють старт розробки вебпроєктів [13].

Ключовою причиною такого вибору є принцип «конвенція замість конфігурації», який позбавляє від необхідності писати шаблонний код під час налаштування маршрутів та контролерів [14]. Додатковою перевагою є наявність масивного репозиторію готових пакетів (гемів). Завдяки їм можна делегувати типові завдання надійним стороннім бібліотекам, а вивільнений час витратити на розробку унікальної бізнес-логіки візуальної дошки.

У питаннях безпеки та ідентифікації було вирішено спиратися на перевірені часом рішення. Процес автентифікації реалізовано за допомогою гема Devise. Це потужний інструмент на базі Warden, який бере на себе повний цикл управління обліковими записами [15]. Для розмежування прав доступу впроваджено бібліотеку Pundit. Її архітектура дозволяє за допомогою звичайних Ruby-класів створити прозору та масштабовану систему авторизації [16]. У контексті нашого проєкту це дає змогу гнучко прописувати специфічні політики доступу (наприклад, право на видалення об'єктів або зміну складу команди), відштовхуючись від поточної ролі юзера.

Підсистема зберігання даних реалізована на базі PostgreSQL. Це об'єктно-реляційна СУБД з відкритим кодом, яка пропонує безкомпромісну надійність та високу продуктивність при обробці запитів [17].

Інтеграція саме цієї системи зумовлена кількома технічними факторами. Насамперед PostgreSQL славиться строгими стандартами збереження цілісності інформації. Крім того, вона ідеально сумісна з ORM-інструментарієм Ruby on Rails. Масштабованість, стабільність роботи під навантаженням та величезна база знань від спільноти роблять цю СУБД оптимальним варіантом для зберігання архітектури проєкту.

Побудова користувацького інтерфейсу здійснювалася за допомогою бібліотеки React. Ця технологія ідеально підходить для розробки складних, динамічних SPA-додатків.

Базовою перевагою є компонентна архітектура: весь інтерфейс (від панелі інструментів до окремого намальованого прямокутника) розбивається на ізольовані модулі. Це гарантує зручність рефакторингу та можливість

багаторазового використання коду. Висока швидкість рендерингу досягається завдяки механізму Virtual DOM, який зберігає структуру сторінки в пам'яті та точково оновлює лише ті елементи, що дійсно змінилися [18]. Ефективне управління внутрішнім станом компонентів та життєвим циклом забезпечується системою хуків (зокрема, useState та useEffect) [19]. Завдяки їм можна легко писати власні абстракції для обробки даних без створення громіздких класів.

Для написання логіки фронтенду було обрано TypeScript. Ця надбудова над класичним JavaScript впроваджує строгу типізацію, що дозволяє відловлювати левову частку помилок ще на етапі компіляції [18]. Розуміння структур даних стає прозорішим, що критично важливо при роботі зі складними об'єктами на полотні.

Візуальне оформлення проєкту базується на фреймворку Tailwind CSS. Його філософія «utility-first» докорінно відрізняється від класичних UI-бібліотек. Замість підключення громіздких файлів зі стилями, розробник оперує базовими утилітарними класами безпосередньо в розмітці, після чого система компілює лише використані стилі у фінальний мініфікований файл [20]. Такий метод ліквідує проблему конфлікту CSS-селекторів та дозволяє стилізувати компоненти максимально швидко. Для ще більшого прискорення верстки було інтегровано рішення shadcn/ui – набір готових доступних компонентів, які не встановлюються як залежності, а копіюються прямо в код проєкту з можливістю повної кастомізації.

Головний функціонал застосунку – малювання на полотні, реалізовано за допомогою бібліотеки react-konva, який є React-обгорткою над відомим графічним рушієм Konva. Цей фреймворк значно розширює стандартні можливості HTML5 Canvas, додаючи підтримку інтерактивності [21]. Завдяки react-konva розробник отримує змогу керувати графічними примітивами так само, як і звичайними DOM-елементами, передаючи їм координати та кольори через React-пропси. Це повністю приховує складність низькорівневого програмування Canvas і дозволяє сфокусуватися на бізнес-логіці дошки.

2.5 Архітектура системи

Базуючись на стандартах методології C4, архітектурне проєктування розпочинається з побудови діаграми контексту. Ця абстракція найвищого рівня діє як відправна точка для розуміння програмного продукту. Її призначення полягає у візуалізації «картини в цілому»: розроблювана система розміщується в центрі, а навколо неї фіксуються всі зовнішні актори та сторонні IT-сервіси, з якими відбувається обмін інформацією [22].

Високорівнева структура взаємодії вебсервісу ObjectBoard із навколишнім середовищем проілюстрована на рисунку 2.12. Цей етап моделювання чітко окреслює межі продукту. На відміну від більш глибоких технічних схем, контекстна діаграма ігнорує конкретні фреймворки чи протоколи [22], зосереджуючи увагу аналітиків виключно на напрямках потоків даних у масштабах усієї системи.

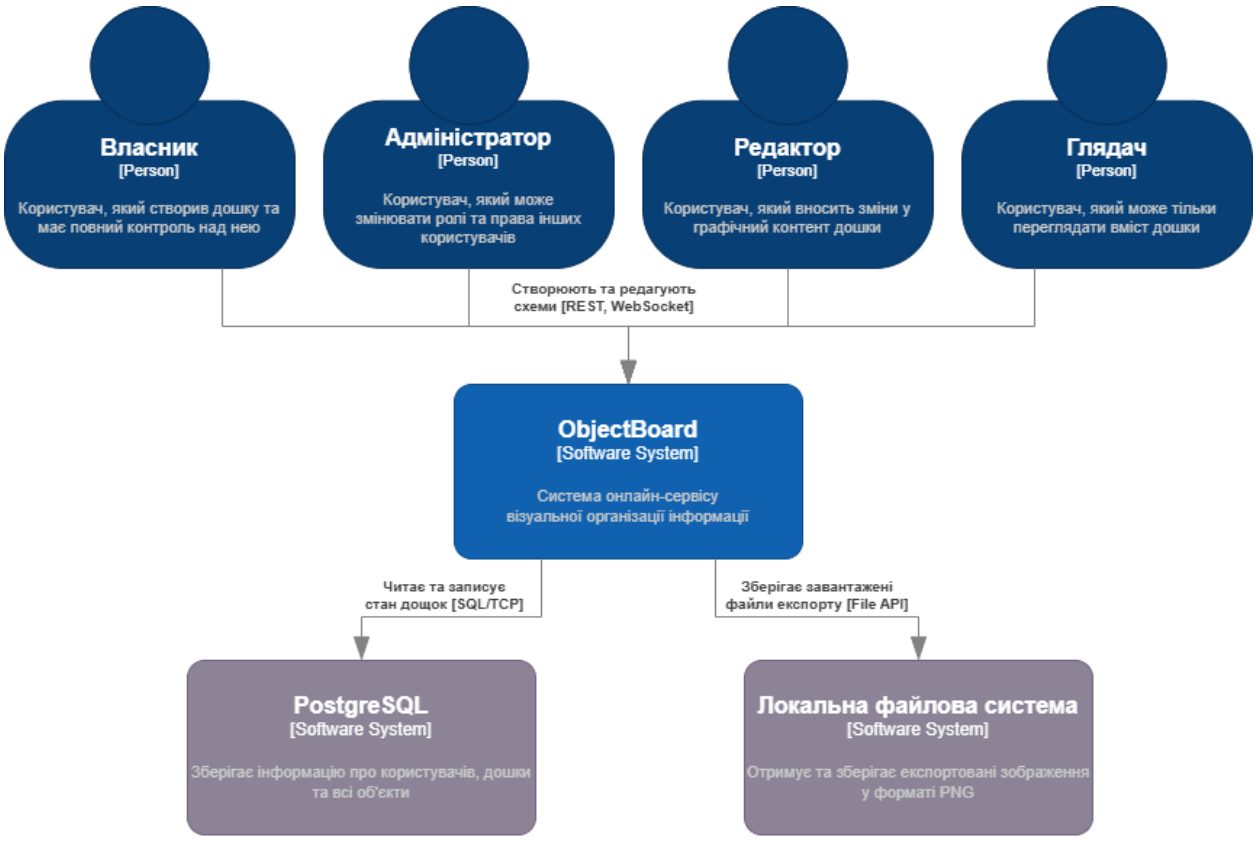

Рисунок 2.12 – Модель зовнішніх зв’язків сервісу візуальної організації інформації

Ядром наведеної схеми виступає сам вебдодаток ObjectBoard. Навколо нього згруповано чотири профілі користувачів (Person), чиї ролі було формалізовано на попередніх етапах аналізу. Ієрархія взаємодії починається з Власника (ініціатора простору) та Адміністратора (керуючого командою), спускаючись до Редактора (генерує контент) і Глядача (здійснює пасивний моніторинг). Обмін даними між цими суб'єктами та платформою відбувається через стандартні браузери із застосуванням вебпротоколів.

Оскільки філософія продукту передбачає максимальну автономність без звернень до сторонніх API, у ролі зовнішніх систем (System_Ext) на діаграмі фігурують лише базові інфраструктурні вузли. Першим таким вузлом є реляційна СУБД PostgreSQL, що гарантує збереження графічної конфігурації дошок та облікових даних. Другим компонентом виступає локальна файлова система клієнта, яка приймає згенеровані растрові зображення (формат PNG) під час процедури експорту. Змодельований контекст повністю релевантний попереднім схемам (Use Case, DFD) і підтверджує, що додаток є інкапсульованим та незалежним рішенням.

Наступним кроком архітектурної декомпозиції за фреймворком C4 є перехід до рівня контейнерів. Цей ракурс розкриває внутрішню топологію програмного забезпечення: єдина система візуально розбивається на окремі розгортані модулі (контейнери), фіксуючи їхні технологічні стеки та контракти взаємодії [22].

Специфіка розробленої архітектури полягає у застосуванні гібридної моделі моноліту. Роль фронтенду виконує клієнтський Single Page Application (SPA), написаний з використанням екосистеми React. Проте його первинна ініціалізація та віддача клієнту делегована серверному фреймворку Ruby on Rails. Такий паттерн поєднує надійність традиційного серверного рендерингу для початкового завантаження із високою динамічністю SPA-додатків.

Для обґрунтування вибору технологій та опису структурних одиниць ObjectBoard було спроєктовано контейнерну діаграму (рис. 2.13). Перший контейнер – React SPA – функціонує безпосередньо у середовищі браузера.

Його зона відповідальності охоплює обробку подій маніпулятора (миші), відмальовування геометричних примітивів на елементі Canvas та формування користувацького інтерфейсу без перезавантаження сторінок.

Рисунок 2.13 – Розподіл функціональних модулів середовища колективної взаємодії

Другим складовим блоком є Backend API на базі Ruby on Rails. Окрім віддачі статики, він функціонує як основний обробник бізнес-логіки. Сервер проводить авторизацію, валідує координати об'єктів та перевіряє рівні доступу. Для забезпечення синхронізації стану дошки між активними

користувачами, окрім класичних REST API запитів, реалізовано базову трансляцію подій через протокол WebSocket (за допомогою модуля ActionCable). Це дозволяє миттєво відправляти зміни графічного контенту на клієнтські додатки інших учасників сесії, суттєво зменшуючи затримки відображення актуальної інформації.

Третій контейнер – Database (PostgreSQL). Він забезпечує персистентне зберігання матриці прав доступу, профілів та властивостей кожного намальованого об'єкта. Серверна частина комунікує із БД через TCP/IP за допомогою ORM-адаптера Active Record. Запроєктована трирівнева ізоляція (клієнт – серверна логіка – дані) робить систему масштабованою та стійкою до збоїв.

Спираючись на передовий інженерний досвід, зокрема на корпоративні стандарти Amazon Web Services (AWS), процес ухвалення фундаментальних технічних рішень супроводжувався веденням документації у форматі Architectural Decision Records (ADR). Ця методологія вимагає фіксувати не просто обрану технологію, але й передумови такого кроку, наявні ліміти, відхилені альтернативи та очікувані результати [23].

Застосовуючи цю методологію, важливим етапом проєктування системи ObjectBoard стало прийняття та документування ключових архітектурних рішень у форматі ADR. Такий підхід дозволив команді чітко обґрунтувати вибір оптимального технологічного стека та зберегти історію проєктування.

Було розроблено три основні документи ADR: вибір системи керування базами даних (ADR-001), вибір загального архітектурного стилю додатка (ADR-002) та вибір технологічного стека для реалізації клієнт-серверної взаємодії (ADR-003). Основний акцент робився на досягненні цільової метрики проєкту – скороченні часу на узгодження візуальних моделей на 30% за рахунок стабільності та швидкодії обраних рішень. Повний текст документів ADR з детальним аналізом альтернатив та наслідків наведено у додатку А.

3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Розробка серверної частини додатку

У рамках архітектурного шаблону MVC, який лежить в основі фреймворку Ruby on Rails, моделі відіграють роль фундаментального шару для управління бізнес-логікою та станом системи. Їхнім головним завданням є взаємодія з базою даних та забезпечення цілісності інформації [14]. Для розроблюваної платформи візуальної співпраці було спроєктовано набір моделей, що повністю відображають структуру БД (див. додаток Б). До цього переліку входять: Board, CanvasObject, Ellipse, Line, Rectangle, Text, User, Member та Role.

Кожна модель інкапсулює правила валідації та налаштування зв'язків (асоціацій) між таблицями, що гарантує надійність даних. Розглянемо конфігурацію на прикладі класу Board:

```
class Board < ApplicationRecord
  has_many :members, dependent: :destroy
  has_many :users, through: :members
  has_many :canvas_objects, dependent: :destroy

  validates :name, presence: true, length: { maximum: 25 }
  validates :description, length: { maximum: 255 }
end
```

У наведеному фрагменті реалізовано відношення «багато-до-багатьох» із користувачами через проміжну таблицю учасників. Опція dependent: :destroy автоматизує каскадне видалення: у разі знищення дошки система самостійно видалить усі пов'язані з нею графічні примітиви та записи про учасників, запобігаючи появі «осиротілих» даних. Валідатори, своєю чергою, блокують збереження дощок без імені або з перевищенням лімітів символів.

Окрім базових налаштувань, моделі містять допоміжні класові методи, які розвантажують контролери («товсті моделі, тонкі контролери»). Яскравим прикладом є метод фабрики у моделі CanvasObject:

```
def self.create_canvas_object(attrs, board_id)
    canvas_object = CanvasObject.new(
      attrs.slice(*CanvasObject.attrs).merge(board_id: board_id)
    )

    type_class = CanvasObject.get_type_class(attrs["type"])
```

```
      type_class.new(
        attrs.slice(*type_class.attrs).merge(canvas_object: canvas_object)
      )

      canvas_object
    end
```

Цей алгоритм приймає JSON-параметри з API і самостійно конструює як базовий запис об'єкта, так і його специфічний дочірній елемент (наприклад, лінію чи текст), що робить код контролера значно чистішим.

Для дотримання принципу DRY (Don't Repeat Yourself) спільну логіку дочірніх графічних об'єктів винесено у спеціальний модуль SpecificCanvasObjectsValidation. Він містить перевірки, зокрема метод, що забороняє зміну батьківського ідентифікатора після створення:

```
def canvas_object_id_immutable
  return unless canvas_object_id_changed?

  errors.add(:canvas_object_id, :canvas_object_id_immutable,
             metadata: {
               object: self,
               new_canvas_object_id: canvas_object_id,
               old_canvas_object_id: canvas_object_id_was
             })
end
```

Цей шар архітектури відповідає за маршрутизацію зовнішніх HTTP-запитів, ініціацію методів моделей та формування кінцевої відповіді для клієнта [14]. Логіка бекенду поділена на вузькоспеціалізовані контролери, що на 6 сторінці графічного матеріалу. Базовим є контролер Pages, який містить єдиний метод app. Його унікальна роль полягає у віддачі стартового HTML-документа для завантаження клієнтського інтерфейсу, тому він відкритий для неавторизованих сесій. Група контролерів Users, Registrations та Sessions є кастомізованою надбудовою над бібліотекою Devise. Оскільки стандартний Devise орієнтований на класичний Rails-моноліт із HTML-переадресаціями, ці класи було перевизначено для коректної обробки JSON-запитів та роботи у форматі REST API.

Задля оптимізації коду створено модуль (concern) BoardRelated, який підключається до контролерів Boards, Members та BoardsContent. Він використовує callback-метод before_action для попереднього пошуку дошки та валідації прав доступу:

```
def set_board
```

```
        @board = Board.find_by(id: params.expect(:id))
        raise BoardErrors::NotFound.new(metadata: { board_id:
params.expect(:id) }) unless @board

        @member = Member.find_by(board: @board, user: current_user)
        raise MemberErrors::NotFound.new(metadata: { board_id: @board.id,
user_id: current_user.id }) unless @member

        authorize @board, :access?
      end
```

Сам контролер Boards забезпечує базові CRUD-операції для просторів, розділяючи отримання даних на спискове (all) та детальне (one). Контролер Members бере на себе всю логіку управління командою: отримання списку колег (others), зміну їхніх статусів (update_role), обробку запрошень (add_to_board, accept_invite) та видалення учасників (kick, leave_board).

Найбільш навантаженим є контролер BoardContent, що обробляє маніпуляції з графікою. Метод save побудовано за принципом пакетної обробки (batch processing):

```
      def save
        authorize @member, policy_class: BoardContentPolicy

        new_params = save_params

        new_ids = []
        ActiveRecord::Base.transaction do
          new_ids = handle_create(new_params[:create])
          handle_delete new_params[:delete]
          handle_update new_params[:update]
        end

        render json: { assigned_IDs: new_ids }
      end
```

Замість надсилання сотень дрібних запитів на кожен рух миші, клієнт агрегує зміни та відправляє їх одним масивом. Бекенд відкриває єдину транзакцію бази даних (ActiveRecord::Base.transaction), що гарантує цілісність інформації у разі збою. Окрім цього, для прискорення синхронізації колективної роботи впроваджено підтримку WebSockets через ActionCable. Після успішного закриття транзакції сервер генерує подію (broadcast) у відповідний канал дошки. Це дозволяє миттєво транслювати актуальний стан об'єктів іншим онлайн-користувачам без потреби ручного оновлення сторінки.

В умовах архітектури SPA (Single Page Application) стандартний механізм представлень Rails (Views) практично не використовується для

генерації контенту. Бекенд функціонує як API-сервер, а єдиний існуючий HTML-файл слугує лише точкою монтування React-додатка:

```
<%= content_tag(:body, "", id:"root", data:{
  controller:"react"
})%>
```

Цей рядок ініціює роботу контролера бібліотеки Stimulus:

```
export default class extends Controller {
  connect() {
    const root = document.getElementById("root");
    createRoot(root).render(<App />);
  }
}
```

Після першого завантаження цієї порожньої сторінки всю подальшу маршрутизацію та відмальовування інтерфейсу перебирає на себе клієнтський JavaScript.

Замість використання стандартних винятків Ruby, у проєкті імплементовано власну ієрархію класів помилок на базі StandardError. Це архітектурне рішення значно спрощує логування та стандартизує структуру відповідей сервера при виникненні непередбачуваних ситуацій.

Було створено абстрактний клас BaseError, який інкапсулює логіку запису в консоль (log_error) та задає інтерфейс для перевизначення методів user_message (текст для клієнта) і status (HTTP-код). На його основі реалізовано п'ять вузькоспеціалізованих модулів (наприклад, user_errors.rb, board_errors.rb). Приклад конкретної реалізації помилки:

```
module UserErrors
  class NotFound < BaseError
    def user_message
      I18n.t("errors.users.not_found")
    end

    def status
      :not_found
    end
  end
end
```

У головному контролері ApplicationController використано макрос rescue_from BaseError, який перехоплює будь-яку з кастомних помилок, автоматично записує трейсбек у лог та повертає клієнту відформатований JSON зі зрозумілим повідомленням.

Конфігурація файлу routes.rb (див. додаток Б) адаптована під потреби гібридного API.

По-перше, налаштовано підтримку локалізації через scope "(:locale)", що дозволяє динамічно перемикати мову (наприклад, /uk або /en). По-друге, шляхи бібліотеки Devise перевизначено (skip: :all) і замінено на власні RESTful ендпоінти для безшовної роботи з React.

Маршрути для дощок (resources :boards) позбавлені методів генерації HTML-сторінок (except: [:show, :new, :index]), але містять вкладені блоки scope :member та scope :content для логічного групування API-запитів.

Критично важливим елементом є наявність fallback-маршруту наприкінці файлу. Оскільки React Router керує історією браузера на клієнті, будь-який прямий перехід за посиланням міг би викликати помилку сервера. Тому бекенд перехоплює всі невідомі URL і примусово віддає єдиний HTML-файл з ініціалізацією SPA, а спеціальне правило match '*unmatched' обробляє некоректні API-запити.

Гарантія стабільності серверної бізнес-логіки забезпечується інструментарієм Minitest. Для об'єктивного вимірювання якості покриття коду застосовано аналізатор SimpleCov. Відповідно до згенерованого звіту (рис. 3.1), загальний показник покриття становить 72.17%. При цьому критично важливі вузли – моделі та контролери – протестовані на 89.33% та 70.25% відповідно.

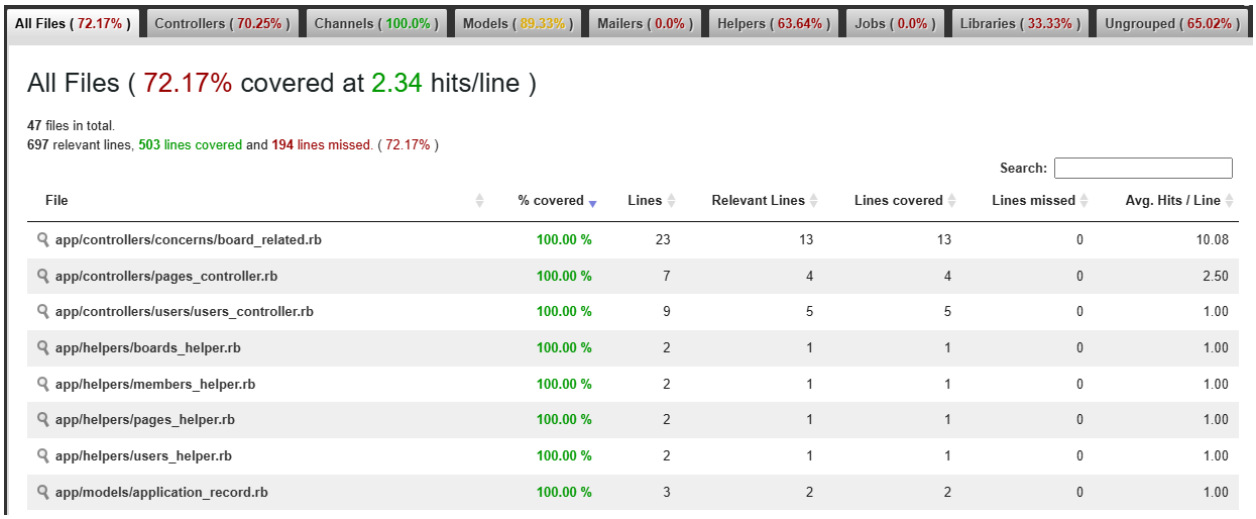

Рисунок 3.1 – Звіт про покриття коду тестами

Тестова база поділяється на два рівні. Інтеграційні тести (ActionDispatch::IntegrationTest) валідують коректність обробки HTTP-транзакцій. Зокрема, у MembersControllerTest симулюються живі сценарії: отримання списку команди, модифікація ролей та обробка інвайтів. У BoardsControllerTest імітується перевірка прав доступу (Pundit), де доведено, що операції видалення проєкту доступні виключно Власнику.

Одиничні тести (ActiveSupport::TestCase) фокусуються на ізольованій перевірці моделей. Наприклад, тести для моделі Member доводять, що система блокує створення дублікатів учасників та забороняє зміну повноважень головного адміністратора (власника) дошки. Детальний приклад реалізації тестів наведено у додатку Б.

3.2 Створення графічної оболонки для маніпулювання об’єктами

На відміну від стандартних елементів інтерфейсу (кнопок чи текстових полів), маршрутизатори в екосистемі React позбавлені візуального представлення. Однак вони є повноцінними вузлами віртуального DOM-дерева, функція яких полягає у відстеженні поточного стану URL-адреси браузера та динамічному монтуванні або демонтажі відповідних компонентів. Саме цей механізм перетворює односторінковий додаток (SPA) на повноцінний програмний продукт із класичною багатосторінковою навігацією [24].

У межах клієнтської частини розроблюваної платформи за маршрутизацію відповідає пакет react-router. Цей інструмент гарантує миттєвий перехід між екранами без відправки додаткових HTML-запитів на сервер. Конфігурація маршрутів ініціалізується у кореневому файлі додатка:

```
const router = createBrowserRouter([
  { path: ROUTES.home(), element: <Home /> },
  { path: ROUTES.board(), element: <BoardWrapper /> },
  { path: ROUTES.boards(), element: <Boards /> },
  { path: ROUTES.signIn(), element: <LogIn /> },
  { path: ROUTES.signUp(), element: <Registration /> },
  { path: ROUTES.profile(), element: <UserProfile /> },
  { path: "*", element: <NotFoundPage /> },
]);
```

Метод createBrowserRouter приймає масив об'єктів, де кожен елемент містить властивість path (відстежуваний URL) та element (компонент, що рендериться за цією адресою).

Запроєктована структура охоплює всі ключові екрани: головну сторінку, панель керування дошками, безпосередньо робоче середовище проєкту, модулі авторизації/реєстрації та особистий кабінет. Для уникнення помилок при зміні адрес використано патерн централізованого зберігання шляхів у словнику ROUTES. Додатково налаштовано fallback-маршрут (path: "*"), який автоматично перехоплює некоректні посилання та виводить компонент помилки <NotFoundPage />. Такий підхід є стандартом для SPA-архітектури, забезпечуючи високу швидкість відгуку інтерфейсу.

Для організації стабільного та ефективного обміну даними між React-клієнтом та бекендом застосовано бібліотеку TanStack Query. Вона вирішує комплекс завдань, пов'язаних з асинхронними запитами, беручи на себе логіку кешування відповідей, фонової синхронізації та управління станом серверних даних безпосередньо у клієнтському коді.

Щоб уникнути дублювання коду, всю базову логіку HTTP-взаємодії було інкапсульовано у двох кастомних хуках: UseCustomMutation (для відправки даних) та UseCustomQuery (для отримання даних). Ці обгортки автоматично обробляють статуси запитів та генерують спливаючі повідомлення для юзера (toast-сповіщення) за допомогою інтегрованого пакета sonner.

Використання такого підходу зводить отримання інформації з API до кількох рядків коду. Наприклад, виклик списку всіх доступних проєктів виглядає наступним чином:

```
const {
  data: boardsData = [],
  isLoading,
  refetch: refetchBoards,
} = UseCustomQuery({
  queryKey: ["boards"],
  path: ROUTES.allBoardsApi(),
});
```

У даному випадку необхідно передати лише унікальний ключ для внутрішнього кешу (queryKey) та URL-адресу ендпоінту (path). Детальна реалізація хука UseCustomQuery наведена у додатку В.

Управління станом у React передбачає не лише збереження змінних, а й фіксацію складних залежностей між ними. Це дозволяє ядру React відстежувати зміни та ефективно перемальовувати лише ті компоненти, які цього дійсно потребують [24].

Для забезпечення плавного та безпомилкового малювання на інтерактивному полотні було спроєктовано спеціальний об'єкт стану та набір допоміжних функцій. Цей тип є дискримінантним об'єднанням (union), що базується на активному режимі (CanvasMode). Система може перебувати у чотирьох фазах: None (режим спокою), Selected (виділення одного чи групи елементів), SelectionNet (малювання рамки виділення) та Inserting (розміщення нової фігури). Залежно від режиму, об'єкт стану містить різний набір даних. Наприклад, статус Selected зберігає масив виділених фігур та параметри їхнього масштабування (resizing), тоді як SelectionNet фіксує лише точки старту та кінця прямокутника виділення.

Для безпечної мутації цього стану імплементовано фабрику createCanvasStateUtils. Вона приймає диспетчер стану setCanvasState і генерує набір методів для управління режимами (об'єкт CanvasStateUtils). Наприклад, метод CanvasStateUtils.None.set() переводить інтерфейс у стан спокою, а CanvasStateUtils.Inserting.updateProperty() динамічно змінює параметри майбутньої фігури. Така інкапсуляція виносить складну логіку переходів за межі UI-компонентів, забезпечуючи високу модульність коду.

Ключовою вимогою до сучасних графічних редакторів є підтримка функцій скасування (Undo) та повторення (Redo). Для вирішення цього завдання, а також для оптимізації відправки даних на бекенд, було розроблено кастомний хук UseHistory.

Усередині хука використовується useRef для створення мутабельного стека історії змін та збереження поточного індексу (позиції юзера в цій історії).

Кожна транзакція (HistoryRecord) формується з масиву індивідуальних правок (ChangeRecord), де фіксується унікальний ID фігури, її тип, а також диференціал між старими та новими параметрами.

Щоб не перевантажувати стек сотнями записів під час одного перетягування мишею, функція historyHandleChanges застосовує метод агрегації (debounce). Зміни тимчасово накопичуються у буфері delayedRecord. Якщо юзер не робить нових дій протягом 200 мс (керується параметром additionalDelay), буфер вважається фіналізованим і метод saveInHistory переносить його у постійну історію. При цьому внутрішня функція noChanges відфільтровує «порожні» рухи, де параметри фігури залишилися незмінними.

Метод saveInHistory виконує подвійну роботу: він не лише зсуває historyIndex та очищає гілку Redo (якщо юзер зробив нову дію після скасування), але й автоматично тригерить відправку інформації на API через хук UseBoardContentMutation.

Спеціалізований мутатор UseBoardContentMutation оптимізовано спеціально для канвасу. Його головна фіча – система дебаунсингу мережевих запитів. Замість миттєвої відправки, хук чекає 0.1 секунди після останньої дії юзера, акумулюючи пакет змін. Однак, щоб уникнути накопичення маси незбережених правок, встановлено жорсткий таймаут у 1 секунд, після якого дані відправляються примусово. Також хук відповідає за підміну тимчасових ідентифікаторів об'єктів (які створюються на клієнті з від'ємними значеннями) на постійні ID, згенеровані сервером після успішного збереження у БД.

Функціонал undo / redo працює шляхом читання попередніх або наступних транзакцій із масиву history (після перевірки лімітів через canUndo / canRedo). Ці функції реверсивно застосовують параметри до об'єктів на полотні та синхронно ініціюють відправку нових координат на сервер. У результаті хук UseHistory забезпечує безперервну роботу клієнта (реактивний UI) і надійну, пакетну (batch) синхронізацію з базою даних, мінімізуючи мережеве навантаження.

Незважаючи на те, що архітектура Single Page Application (SPA) фізично завантажує лише один HTML-документ, для зручності опису ізольованих маршрутами екранів надалі застосовуватиметься термін «сторінки».

Точкою входу до платформи є головна сторінка (рис. 3.2), за рендеринг якої відповідає компонент Home.

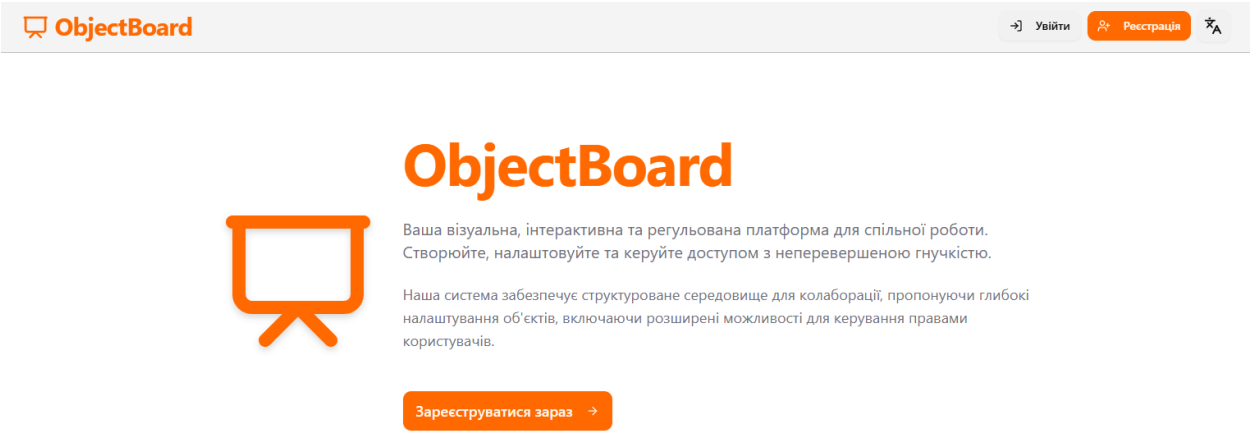

Рисунок 3.2 – Головна сторінка

Вона виконує презентаційну функцію, знайомлячи відвідувача з можливостями сервісу. Інтерфейс поділено на логічні блоки, де за допомогою карток та векторної графіки проілюстровано ключові сценарії використання продукту.

Для керування сесіями та ідентифікації розроблено групу компонентів: сторінки авторизації, реєстрації та налаштувань профілю. На рівні маршрутизатора впроваджено суворі правила доступу: неавторизовані користувачі відкидаються від приватних маршрутів (наприклад, профілю), тоді як залогінені юзери не можуть повторно відкрити форми входу.

Сторінка створення акаунта (рис. 3.3) контролюється компонентом Registration. Усі дані з текстових полів (ім'я, email, пароль та його підтвердження) акумулюються у локальному стані React до моменту відправки на сервер. Після успішного створення профілю система автоматично редиректить юзера на дашборд.

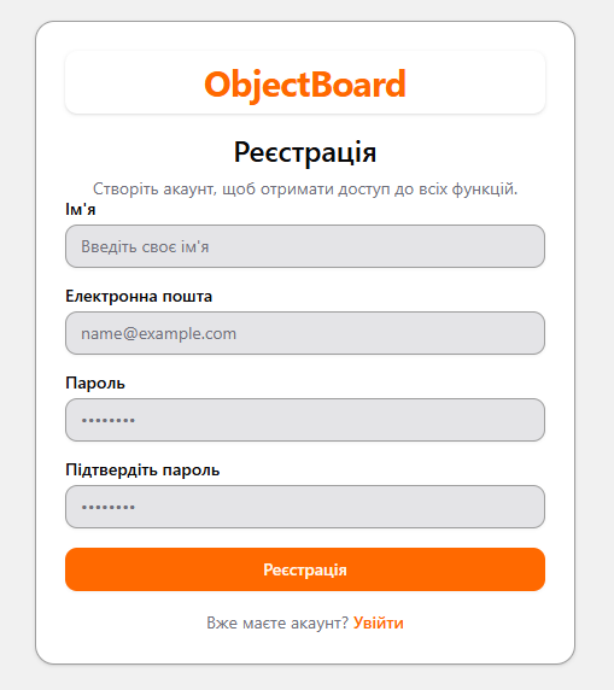

Рисунок 3.3 – Форма реєстрації

Приватні дані керуються через компонент UserProfile (рис. 3.4). Тут реалізовано функціонал зміни персональної інформації та опцію повного видалення облікового запису. Задля безпеки, видалення акаунта вимагає підтвердження через додаткове діалогове вікно.

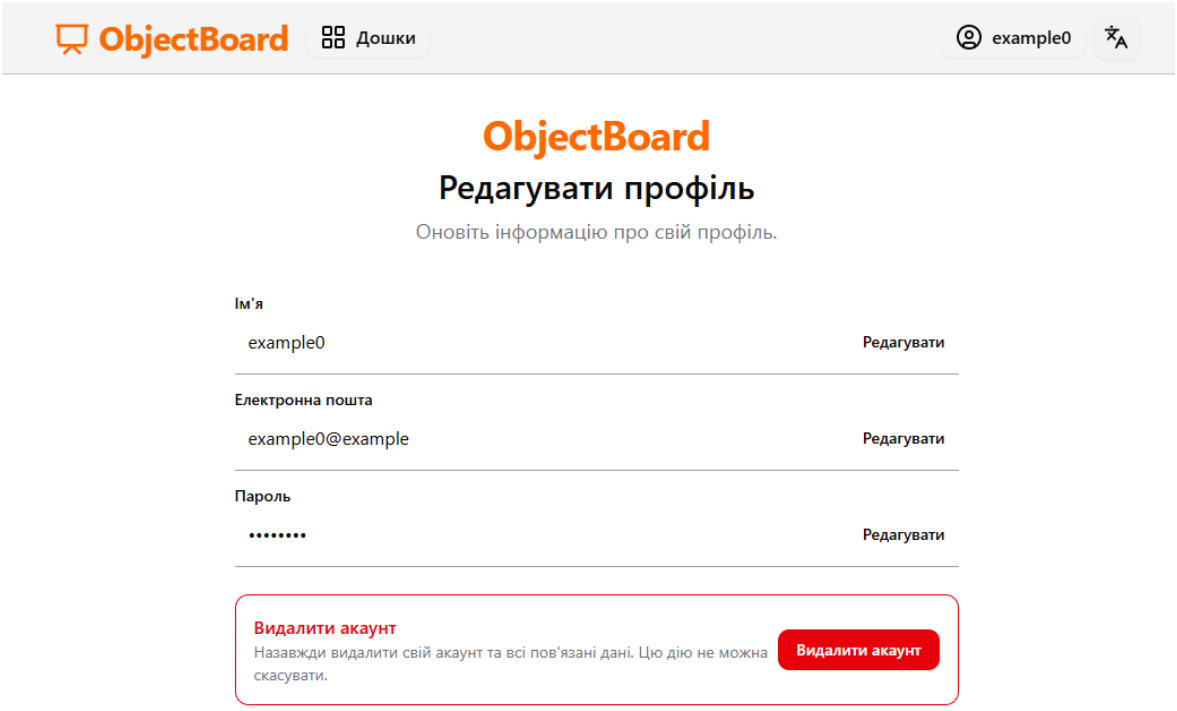

Рисунок 3.4 – Сторінка профілю користувача

Центром управління проєктами виступає сторінка Boards. Вона має зручну систему вкладок, яка розділяє робочі простори на три категорії, що значно полегшує навігацію та управління станом відображення.

Вкладка «Мої дошки» (рис. 3.5), побудована на компоненті BoardsTab, виводить перелік активних проєктів користувача у вигляді сітки. Тут імплементовано рядок пошуку та фільтри за статусом (роллю). У разі відсутності дошок інтерфейс виводить відповідну заглушку-підказку. На кожній картці передбачено кнопки для самостійного виходу з команди або повного видалення проєкту (із захисним вікном підтвердження).

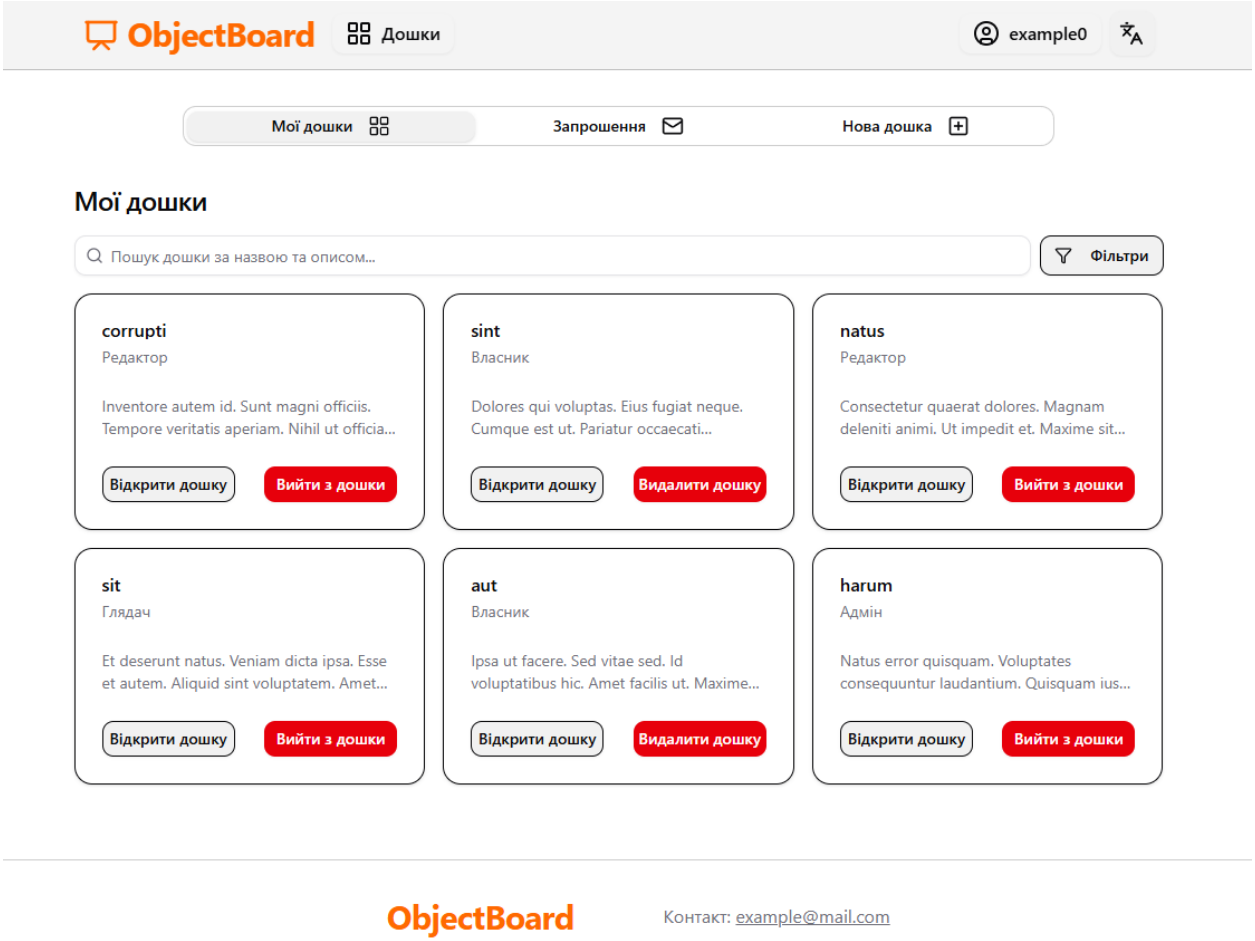

Рисунок 3.5 – Вкладка «Мої дошки»

Розділ «Запрошення» (7 сторінка графічного матеріалу) керується компонентом InvitesTab. Програма автоматично відфільтровує масив дошок, залишаючи лише ті, де користувач має статус «Invited». Юзер може прийняти

або відхилити запит (через підтверджуюче модальне вікно). Внесені зміни миттєво оновлюють масив даних на екрані.

Генерація нових просторів відбувається у вкладці «Нова дошка» (7 сторінка графічного матеріалу) через компонент NewBoardTab. Доки сервер обробляє запит на створення, кнопка підтвердження блокується і змінює текст на «Створення...». Це ефективний патерн UX, що запобігає випадковому дублюванню проєктів. Після відповіді API форма очищується, а клієнта перекидає до загального списку.

Найбільш технічно складним елементом інтерфейсу є сторінка безпосередньо відкритої дошки (рис. 3.6), за яку відповідає компонент Board. Це інтерактивний робочий простір із полотном та набором допоміжних панелей. Візуалізація UI безпосередньо залежить від статусу юзера: наприклад, актор із роллю «Глядач» побачить лише канвас без панелі інструментів.

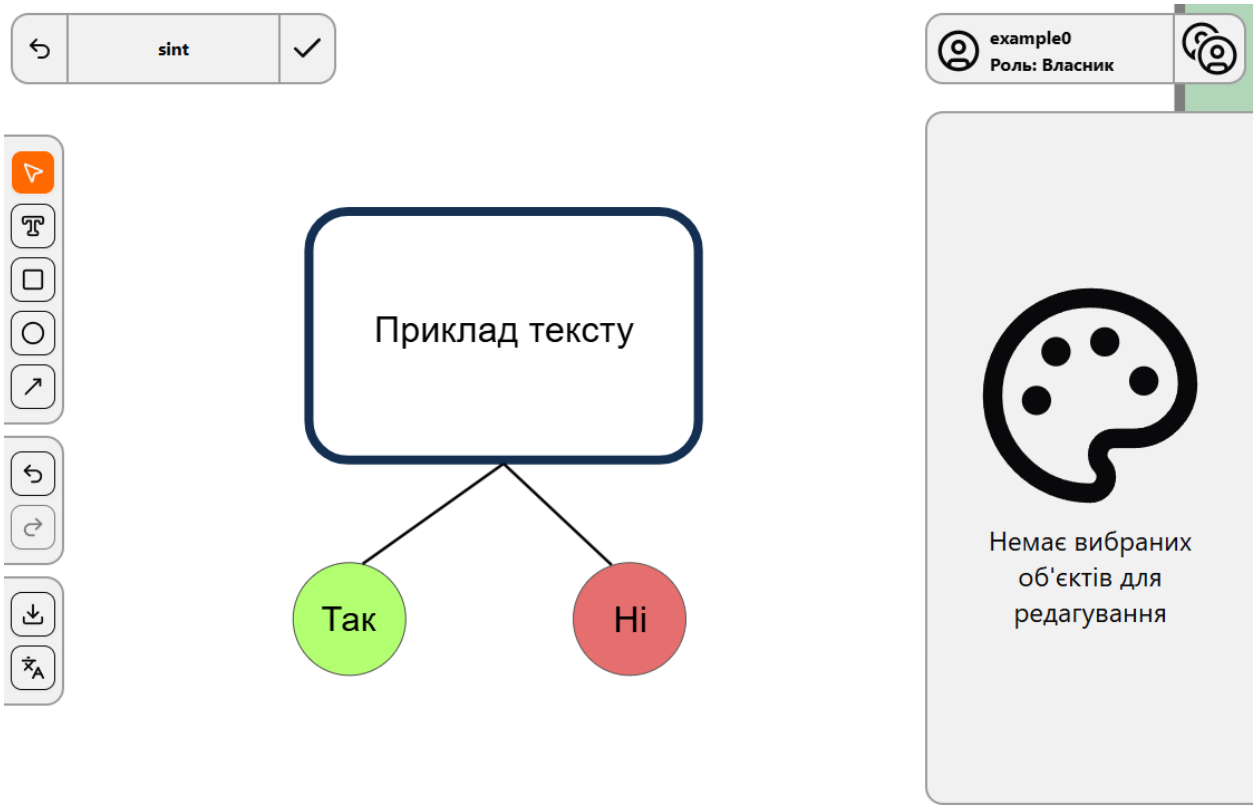

Рисунок 3.6 – Сторінка дошки

Особливістю цього середовища є інтеграція з вебсокетами. Клієнтська частина автоматично підписується на канал оновлень дошки: якщо інший користувач змінює властивості об'єкта або створює новий, ці зміни миттєво надходять через сокет-з'єднання і відображаються на полотні без потреби оновлювати сторінку.

Верхня навігаційна панель BoardMenu інформує про назву проєкту та містить кнопку повернення до дашборда. Якщо система виконує фонове збереження масивів даних у БД, біля назви динамічно з'являється індикатор SavingIcon. Клік по назві дошки викликає модульне вікно ShowBoard (8 сторінка графічного матеріалу), де юзери з правами редагування можуть оновити метадані проєкту.

Зліва у робочому просторі закріплено компонент ToolBar (8 сторінка графічного матеріалу). Перший його блок містить кнопки перемикання режимів (CanvasMode.Inserting) для малювання фігур (лінія, еліпс, прямокутник, текст), де початкові параметри генеруються хуком UseDefaultObjects. Другий блок відповідає за навігацію в історії (Undo/Redo). Третій – за експорт у PNG та перемикач локалізації (LanguageSwitcher).

Для кастомізації графіки використовується права бічна панель ObjectsProperties. Її вміст контекстно-залежний: елементи управління кольором, прозорістю чи товщиною ліній з'являються виключно тоді, коли на полотні виділено конкретну фігуру або активовано режим малювання. Інакше панель виводить стандартну заглушку.

Адміністрування команди здійснюється через випадаюче меню MemberMenu у верхньому правому куті. Воно відображає аватар поточного юзера та перелік інших колег на дошці.

Якщо власник або адміністратор натискає на ім'я колеги, відкривається модальне вікно налаштувань учасника (рис. 3.7). Тут реалізовано механізм перепризначення ролей через селект-меню та відображено набір чекбоксів, що деталізують конкретні дозволи для обраного статусу. У цьому ж вікні знаходиться кнопка для примусового виключення особи з проєкту.

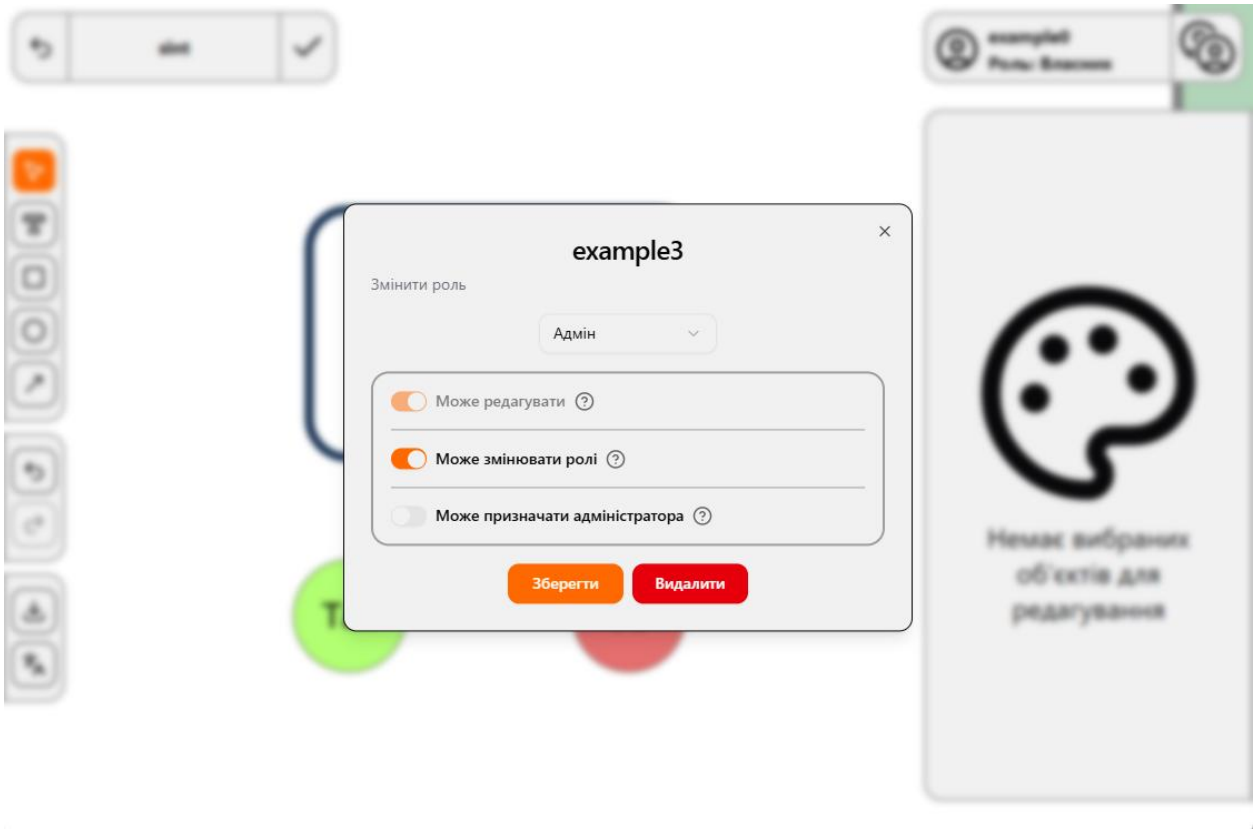

Рисунок 3.7 – Меню налаштувань учасника

Підсумовуючи результати етапу розробки: у рамках проєкту було створено стабільну гібридну архітектуру. Серверна компонента на базі Ruby on Rails взяла на себе функціонал збереження даних (PostgreSQL), перевірки допусків та обслуговування API-маршрутів. Клієнтська сторона, реалізована як React SPA (TypeScript) із застосуванням react-konva для малювання та TanStack Query для синхронізації станів, забезпечила високу швидкість відгуку інтерфейсу. Стилізація за допомогою Tailwind CSS та shadcn/ui дозволила створити сучасний UX/UI дизайн. Комплексне впровадження рольової моделі, системи історії змін та базової інтеграції вебсокетів дозволило досягти головної мети – створення ефективної онлайн-дошки для колективної візуалізації.

ВИСНОВКИ

У результаті виконання курсової роботи було здійснено повний цикл створення програмного продукту – від системного аналізу предметної області до практичної розробки функціонуючого вебдодатка ObjectBoard. На початковому етапі було досліджено потреби сучасних розподілених команд та проведено порівняння провідних аналогів, таких як Excalidraw, draw.io та Explain Everything. Аналіз засвідчив, що ринок потребує рішення, яке б гармонійно поєднувало простоту створення візуальних схем із надійним корпоративним контролем доступу. Це дозволило сформувати чіткі функціональні вимоги до власного сервісу, базовим елементом якого стала ієрархічна рольова модель взаємодії учасників.

Проєктування бізнес-логіки та архітектури системи спиралося на комплексне використання сучасних методологій моделювання. За допомогою нотацій IDEF0 та DFD було визначено глобальні інформаційні потоки, а моделі BPMN 2.0 та IDEF3 дозволили деталізувати критичні сценарії паралельної роботи команди, синхронізації змін на полотні та управління доступами. Спроєктовані UML-діаграми та багаторівнева архітектура за моделлю C4 забезпечили чітке розмежування відповідальності між серверними та клієнтськими компонентами, сформувавши надійний фундамент для написання коду.

Окрема увага в роботі була приділена проєктуванню користувацького досвіду та структури збереження інформації. На основі аналізу потреб клієнтів через побудову карт шляху (UJM та CJM) розроблено ергономічні прототипи інтерфейсів, які гарантують інтуїтивно зрозумілу навігацію. Паралельно з цим було спроєктовано підсистему бази даних: від концептуальної ER-діаграми до нормалізованої фізичної моделі. Прийняті інженерні рішення дозволили оптимізувати збереження великих масивів графічних координат та параметрів фігур у СУБД PostgreSQL, гарантуючи при цьому реляційну цілісність системи.

Успішно імплементовано теоретичні напрацювання у вигляді реального

працюючого програмного продукту. Спираючись на задокументовані архітектурні рішення (ADR), було розроблено гібридну клієнт-серверну платформу. Серверна частина на базі фреймворку Ruby on Rails взяла на себе безпечне управління бізнес-логікою та авторизацією. Клієнтська сторона, реалізована як Single Page Application за допомогою екосистеми React і TypeScript, забезпечила інтерактивність та високу швидкість відгуку графічного канвасу. Завдяки інтеграції протоколу WebSockets вдалося налагодити безперебійну синхронну роботу кількох користувачів на одній дошці в реальному часі.

Таким чином, поставлену мету роботи досягнуто в повному обсязі. Створена платформа ObjectBoard є не просто абстрактним концептом, а готовим та протестованим інструментом, здатним вирішити проблему розрізненості комунікацій і суттєво підвищити продуктивність планування та проєктування у сучасних бізнес-командах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Excalidraw. URL: http://plus.excalidraw.com (дата звернення: 20.01.2026).
2. Explain everything. URL: https://explaineverything.com/ (дата звернення: 22.01.2026).
3. Draw.io. URL: http://drawio.com (дата звернення: 22.01.2026).
4. Figay N. Equivalent IDEF0 and SysML models. URL: https://medium.com/@nfigay/equivalent-idef0-and-sysml-models-8f176ff02b49 (дата звернення: 05.02.2026).
5. What is Use Case Diagram? Visual Paradigm – AI-Powered Visual Modeling Platform. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення: 07.02.2026).
6. What is a Data Flow Diagram. Lucidchart. URL: https://www.lucidchart.com/pages/data-flow-diagram (дата звернення: 012.02.2026).
7. IDEF – Integrated DEFinition Methods (IDEF). IDEF – Integrated DEFinition Methods (IDEF). URL: https://www.idef.com/ (дата звернення: 16.02.2026).
8. Tracking Interoperability and Data Quality: A Methodology with BPMN 2.0 Extensions and Performance Evaluation / X. Heguy та ін. Modelling. 2024. Т. 5, № 3. С. 797–818. URL: https://doi.org/10.3390/modelling5030042 (дата звернення: 16.02.2026).
9. User Journey Mapping: a practical guide | Staff Gateway. URL: https://staff.admin.ox.ac.uk/ux/userjourneymapping (дата звернення: 23.02.2026).
10. What are Website Wireframes. Lucidchart. URL: https://www.lucidchart.com/pages/wireframe (дата звернення: 24.02.2026).
11. Explore the UML sequence diagram. IBM Developer. URL: https://developer.ibm.com/articles/the-sequence-diagram/ (дата звернення: 27.02.2026).

12. Kirvan P., Hanna K. T., Biscobing J. What Is an Entity Relationship Diagram (ERD)? How Does It Work | TechTarget. Search Data Management. URL: https://www.techtarget.com/searchdatamanagement/definition/entity-relationship-diagram-ERD (дата звернення: 01.03.2026).

13. Ruby on Rails. URL: http://rubyonrails.org (дата звернення: 07.03.2025).

14. Ruby S. Agile Web Development with Rails 7. The Pragmatic Programmers, 2023. 606 с.

15. GitHub – heartcombo/devise: Flexible authentication solution for Rails with Warden. GitHub. URL: https://github.com/heartcombo/devise (дата звернення: 12.03.2025).

16. GitHub – varvet/pundit: Minimal authorization through OO design and pure Ruby classes. GitHub. URL: https://github.com/varvet/pundit (дата звернення: 14.03.2025).

17. PostgreSQL: The World's Most Advanced Open Source Relational Database. URL: http://postgresql.org (дата звернення: 16.03.2025).

18. Roldán C. S. React 18 Design Patterns and Best Practices, 4e: Design, Build, and Deploy Production-Ready Web Applications with React by Leveraging Industry-best Practices. de Gruyter GmbH, Walter, 2023. 464 с.

19. React. URL: https://react.dev/ (дата звернення: 18.03.2025).

20. Tailwind CSS – Rapidly build modern websites without ever leaving your HTML. URL: http://tailwindcss.com (дата звернення: 21.03.2025).

21. Konva – JavaScript Canvas 2d Library. Konva – JavaScript Canvas 2d Library. URL: https://konvajs.org/ (дата звернення: 02.04.2025).

22. C4 model. URL: https://c4model.com/ (дата звернення: 04.04.2026).

23. ADR process – AWS Prescriptive Guidance. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/architectural-decision-records/adr-process.html (дата звернення: 08.04.2026).

24. Griffiths D., David G. React Cookbook: Recipes for Mastering the React Framework. O'Reilly Media, Incorporated, 2021. 500 с.

ДОДАТОК А

ADR ДОКУМЕНТИ

ADR-001: Вибір системи керування базами даних (СУБД)

Статус: Ухвалено

Контекст:

Створюваний сервіс ObjectBoard орієнтований на спільну роботу з візуальними даними. Головне завдання продукту – пришвидшити процес затвердження архітектурних та бізнес-схем на 30%. Специфіка графічного онлайн-редактора вимагає безперервного обміну великими масивами даних. Робоча область проєкту може вміщувати тисячі векторних елементів, які редагуються десятками учасників одночасно. Відповідно, інфраструктура зберігання має не лише забезпечувати високу пропускну здатність, але й жорстко контролювати реляційні зв’язки (авторизацію, ієрархію ролей), гарантуючи швидку перевірку прав доступу без затримок.

Рішення:

Як фундаментальне сховище даних обрано об'єктно-реляційну СУБД PostgreSQL.

Обґрунтування:

- Гнучкість роботи з масивами: Для забезпечення плавного рендерингу довільних ліній (малювання від руки) їхні координати зберігаються не як окремі сутності, а як цілісні масиви. Завдяки нативній підтримці типів ARRAY та JSONB, PostgreSQL дозволяє ефективно записувати та індексувати такі структури без втрати продуктивності;
- Транзакційність та безпека (ACID): Проєкт спирається на сувору рольову модель. Застосування реляційної БД із зовнішніми ключами (Foreign Keys) гарантує захист від несанкціонованих змін, а механізм транзакцій запобігає конфліктам (race conditions), коли кілька юзерів намагаються зберегти дані одночасно;
- Швидкодія через індексування: Створення складених унікальних індексів (наприклад, для пари user_id + board_id) дозволяє системі миттєво валідувати доступ юзера до проєкту навіть за умови експоненціального зростання бази даних;
- Екосистемна сумісність: PostgreSQL є оптимальним та найстабільнішим рішенням для роботи в парі з фреймворком Ruby on Rails. Інструментарій Active Record ідеально покриває потреби взаємодії з цією СУБД, мінімізуючи потребу в ручному написанні SQL-коду.

Відхилені альтернативи:

- MongoDB (NoSQL): Документо-орієнтований підхід зручний для збереження параметрів фігур, але критично програє у підтримці строгих зв'язків (ієрархія команд та

просторів). Реалізація перевірки доступів на рівні додатка спровокувала б проблему N+1 запитів і знизила б надійність системи.

- MySQL: Незважаючи на загальну ефективність, інструменти MySQL для роботи з масивами та кастомними типами даних значно поступаються PostgreSQL. Це створило б "вузьке місце" під час запису тисяч координат (з плаваючою комою) для складних фігур.

Наслідки:

- Позитивні: Досягнуто максимальної надійності збереження даних, блискавичної валідації прав та ефективної обробки координат.
- Ризики: Висока структурованість БД вимагає створення міграцій для кожного нового атрибута фігур. У разі зростання кількості записів до мільйонів, виникне потреба у партиціюванні таблиць.

ADR-002: Вибір загального архітектурного стилю додатка

Статус: Ухвалено

Контекст:

Ключова цінність платформи ObjectBoard – швидкість взаємодії та економія часу користувачів. Для цього інтерфейс повинен реагувати на дії (малювання, переміщення об'єктів) за лічені мілісекунди, підтримуючи частоту кадрів на рівні 60 FPS, при цьому фоново синхронізуючи дані з сервером. Враховуючи обмежені ресурси на розробку (один розробник) та початкові очікування трафіку (до 1000 активних юзерів), архітектура має поєднувати високу продуктивність інтерфейсу з простотою розгортання та підтримки.

Рішення:

Затверджено архітектурний патерн «Гібридний модульний моноліт із Single Page Application (SPA)». Backend (Ruby on Rails) та Frontend (React) функціонують як єдина система, де серверний контролер генерує базову сторінку, в яку монтується клієнтський JavaScript-додаток.

Обґрунтування:

- Локальне управління станом (SPA): Делегування відмальовування графіки (Canvas) на сторону браузера забезпечує юзеру миттєвий зворотний зв'язок. React повністю перебирає на себе обробку подій миші, усуваючи необхідність чекати відповіді сервера для кожного пікселя;
- Оптимізація інфраструктури: Монолітний підхід дозволяє запускати проєкт як єдиний процес. Це скасовує потребу у складних DevOps-рішеннях (Kubernetes, Service Mesh), економлячи до 40% часу на етапі налаштування серверного середовища;

- Відсутність CORS-бар'єрів: Оскільки API та клієнтський застосунок знаходяться на одному домені, зникає потреба в обробці кросдоменних запитів та "preflight" перевірках, що економить мережевий час при кожній транзакції;
- Консолідація бізнес-логіки: Усі процеси валідації та збереження зосереджені в одному місці. Це забезпечує передбачуваність системи та гарантує цілісність даних без імплементації складних розподілених транзакцій.

Відхилені альтернативи:

- Мікросервіси: Розподіл функціоналу на незалежні сервіси (Auth, Board, Canvas) створив би колосальний оверхед. Витрати часу на оркестрацію та налаштування міжсервісної мережевої комунікації є абсолютно невиправданими для поточних масштабів проєкту;
- Server-Side Rendering (Багатосторінковий додаток): Використання класичних ERB-шаблонів Rails відхилено. Очікування генерації HTML на сервері після кожної зміни кольору чи позиції фігури зробило б процес малювання неможливим через величезні затримки.

Наслідки:

- Позитивні: Швидкий цикл розробки, єдина кодова база, плавний та реактивний UX завдяки клієнтському рендерингу.
- Ризики: З часом розмір JavaScript-бандла може збільшити час першого завантаження сторінки. У майбутньому, при значному зростанні навантаження, вертикальне масштабування моноліту може виявитися економічно невигідним.

ADR-003: Вибір технологічного стека для реалізації клієнт-серверної взаємодії

Статус: Ухвалено

Контекст:

У системі ObjectBoard ключовою функцією є безперервне збереження графічних маніпуляцій та відображення їх у колег по команді. Оскільки малювання генерує сотні подій на секунду, відправка кожного руху миші на сервер миттєво перевантажила б БД та мережу. Водночас користувачі повинні бачити правки один одного без ручного оновлення сторінки. Необхідно було обрати комбінований підхід, який збалансує стабільність персистентного збереження даних та потребу в синхронізації у реальному часі (real-time).

Рішення:

Затверджено гібридну комунікаційну модель: REST API (JSON / HTTPS) застосовується для пакетного збереження даних (із використанням механізму debounce), а протокол WebSockets (ActionCable) – виключно для трансляції оновлень іншим учасникам сесії.

Обґрунтування:

- Оптимізація запитів (Batching & Debouncing): Використання REST API (бібліотека TanStack Query) дозволяє клієнту накопичувати зміни локально і відправляти їх одним пакетом (наприклад, через 0.1 - 1 сек після зупинки малювання). Це кардинально знижує кількість транзакцій до БД, забезпечуючи високу стабільність бекенду;
- Реактивність через WebSockets: Замість того, щоб перевантажувати сервер постійним HTTP-опитуванням (Short Polling), інтегровано модуль ActionCable. Як тільки REST-запит успішно зберігає пакет даних у БД, сервер генерує подію в сокет-канал. Це дозволяє браузерам інших юзерів миттєво підтягнути актуальний стан дошки;
- Надійність збереження: На відміну від спроб зберігати дані безпосередньо через вебсокети (що може призвести до втрати інформації при розриві з'єднання), класичні HTTP-запити (POST/PATCH) використовують транзакції ActiveRecord і чітко повертають статуси (200 OK або помилки), гарантуючи цілісність "єдиного джерела істини".

Відхилені альтернативи:

- Виключно REST API (без сокетів): Відхилено. Використання лише HTTP-запитів вимагало б від клієнта постійно запитувати сервер про наявність нових змін (Polling). Це створює паразитне навантаження на інфраструктуру та призводить до візуальних затримок у колективній роботі;
- Повний перехід на WebSockets для всіх операцій: Відхилено. Використання сокетів як єдиного каналу (включно із записом у БД) вимагає складної архітектури з In-Memory сховищами (Redis) для утримання поточного стану та вирішення конфліктів. Для поточного етапу використання REST для CRUD-операцій є значно надійнішим і простішим у реалізації.

Наслідки:

- Позитивні: Досягнуто оптимального балансу: сервер захищений від DDoS-ефекту завдяки дебаунсингу HTTP-запитів, а користувачі отримують досвід роботи в реальному часі завдяки broadcast-трансляціям через сокети.
- Ризики: Архітектура фронтенду ускладнюється, оскільки клієнту потрібно одночасно керувати станом HTTP-мутацій (через хуки збереження) та слухати підписку на WebSocket-канал для оновлення локального полотна. Підтримка ActionCable у production-середовищі потребуватиме розгортання та конфігурації Redis.

ДОДАТОК Б

ЛІСТИНГ КОДУ СЕРВЕРНОЇ ЧАСТИНИ

СХЕМА БАЗИ ДАНИХ

```
ActiveRecord::Schema[8.0].define(version: 2025_05_24_124946) do
  enable_extension "pg_catalog.plpgsql"

  create_table "boards", force: :cascade do |t|
    t.string "name"
    t.string "description"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end


  create_table "ellipses", force: :cascade do |t|
    t.bigint "canvas_object_id", null: false
    t.float "x"
    t.float "y"
    t.string "fill"
    t.float "radiusX"
    t.float "radiusY"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["canvas_object_id"], name: "index_ellipses_on_canvas_object_id", unique: true
  end

  create_table "members", force: :cascade do |t|
    t.bigint "user_id", null: false
    t.bigint "board_id", null: false
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.bigint "role_id", null: false
    t.index ["board_id"], name: "index_members_on_board_id"
    t.index ["role_id"], name: "index_members_on_role_id"
    t.index ["user_id", "board_id"], name: "index_members_on_user_id_and_board_id", unique: true
    t.index ["user_id"], name: "index_members_on_user_id"
  end

  create_table "texts", force: :cascade do |t|
    t.bigint "canvas_object_id", null: false
    t.float "x"
    t.float "y"
    t.string "fill"
    t.string "text"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.integer "width"
    t.integer "height"
    t.integer "align"
    t.integer "verticalAlign"
    t.integer "fontSize"
    t.index ["canvas_object_id"], name: "index_texts_on_canvas_object_id", unique: true
  end

  create_table "users", force: :cascade do |t|
    t.string "email", default: "", null: false
    t.string "encrypted_password", default: "", null: false
    t.string "reset_password_token"
    t.datetime "reset_password_sent_at"
    t.datetime "remember_created_at"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.string "name"
    t.index ["email"], name: "index_users_on_email", unique: true
    t.index ["name", "email"], name: "index_users_on_name_and_email", unique: true
    t.index ["reset_password_token"], name: "index_users_on_reset_password_token", unique: true
  end

  create_table "rectangles", force: :cascade do |t|
    t.bigint "canvas_object_id", null: false
    t.float "x"
    t.float "y"
    t.float "width"
    t.float "height"
    t.string "fill"
    t.float "cornerRadius"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["canvas_object_id"], name: "index_rectangles_on_canvas_object_id", unique: true
  end

create_table "canvas_objects", force: :cascade do |t|
    t.bigint "board_id", null: false
    t.integer "index"
    t.boolean "locked"
    t.string "stroke"
    t.integer "strokeWidth"
    t.float "opacity"
```

```
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["board_id"], name: "index_canvas_objects_on_board_id"
    t.index ["index", "board_id"], name: "index_canvas_objects_on_index_and_board_id", unique: true
  end

  create_table "lines", force: :cascade do |t|
    t.bigint "canvas_object_id", null: false
    t.float "points", array: true
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.index ["canvas_object_id"], name: "index_lines_on_canvas_object_id", unique: true
  end

  create_table "roles", force: :cascade do |t|
    t.integer "name"
    t.boolean "can_edit"
    t.boolean "can_change_roles"
    t.boolean "can_assign_admin"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
    t.boolean "can_ignore_rules"
  end

  add_foreign_key "canvas_objects", "boards"
  add_foreign_key "ellipses", "canvas_objects"
  add_foreign_key "lines", "canvas_objects"
  add_foreign_key "members", "boards"
  add_foreign_key "members", "roles"
  add_foreign_key "members", "users"
  add_foreign_key "rectangles", "canvas_objects"
  add_foreign_key "texts", "canvas_objects"
end
```

КОНТРОЛЕР MembersController

```
class MembersController < ApplicationController
  include BoardRelated

  def current
    render json: @member, full: true
  end

  def others
    other_users = []

    @board.members.where.not(user_id: current_user.id).find_each do |member|
      other_users.push(member)
    end

    render json: other_users, each_serializer: MemberSerializer, full: @member.role.can_change_roles
  end

  def update_role
    member_to_update = Member.find_member(params.expect(:member_id))

    new_role = params.expect(newRole: %i[name can_edit can_change_roles can_assign_admin
can_ignore_rules])

    update_role_authorize new_role, member_to_update

    new_role = Role.find_role(new_role)

    member_to_update.handle_update_error unless member_to_update.update(role: new_role)
  end

  def add_to_board
    authorize @member

    new_member_name = params.expect(:name)
    user = User.find_by(name: new_member_name)
    raise UserErrors::NotFound.new(metadata: { name: new_member_name }) unless user

    default_role = Role.find_by(name: :Invited)
    raise RoleErrors::NotFound.new(metadata: { name: :Invited }) unless default_role

    new_member = Member.new(user: user, board: @board, role: default_role)
    return if new_member.save

    new_member.handle_create_error
  end

  # Also used for declining an invite
  def leave_board
    if @member.role.name == :Owner
      raise MemberErrors::OwnerIsImmutable.new(metadata: { user: current_user,
                                                           board: @member.board })
    end

    authorize_with_current_member @member

    @member.destroy
  end
```

```
  def accept_invite
    authorize_with_current_member @member

    minimal_role = Role.find_by(name: :Viewer)

    @member.handle_update_error unless @member.update(role: minimal_role)
  end

  def kick
    member_to_kick = Member.find_member(params.expect(:member_id))

    if member_to_kick.role.name == :Owner
      raise MemberErrors::OwnerIsImmutable.new(metadata: { user: current_user,
                                                           board: member_to_kick.board })
    end

    authorize_with_current_member member_to_kick

    member_to_kick.destroy
  end

  private

  def authorize_with_current_member(member)
    context = {
      current_member: Member.find_by(user_id: current_user.id, board_id: member.board.id)
    }
    record = ApplicationPolicy::PolicyContext.new(
      member,
      context
    )
    authorize record, :leave_board?, policy_class: MemberPolicy
  end

  def update_role_authorize(new_role, member_to_update)
    context = {
      current_member: @member,
      new_role: new_role.as_json
    }
    record = ApplicationPolicy::PolicyContext.new(
      member_to_update,
      context
    )
    authorize record, :update_role?, policy_class: MemberPolicy
  end
end
```

МАРШРУТИЗАЦІЯ

```
Rails.application.routes.draw do
  scope "(:locale)", locale: /en|uk/ do
    root "pages#app"
    devise_for :users, skip: :all

    scope :users do
      get "" => "users/users#current", as: :current_user

      devise_scope :user do
        post 'sign_in', to: 'users/sessions#create', as: :user_session
        delete 'sign_out', to: 'users/sessions#destroy', as: :destroy_user_session


        post '', to: 'users/registrations#create'
        patch '', to: 'users/registrations#update', as: :user_registration
        delete '', to: 'users/registrations#destroy'
      end
    end

    get "boards/all" => "boards#all"
    resources :boards, except: [:show, :new, :index] do
      member do

        get "one" => "boards#one"

        scope :member do
          get "current" => "members#current", as: :current_member
          get "others" => "members#others", as: :others_members
          patch "update_role/:member_id" => "members#update_role", as: :member_update_role
          post "add_to_board" => "members#add_to_board", as: :member_add_to_board
          delete "leave_board" => "members#leave_board", as: :member_leave_board
          post "accept_invite" => "members#accept_invite", as: :member_accept_invite
          delete "kick/:member_id" => "members#kick", as: :member_kick
        end

        scope :content do
          get "get" => "board_content#get", as: :content_get
          post "save" => "board_content#save", as: :content_save
        end

      end
    end
```

```
    get '*path', to: 'pages#app', constraints: lambda { |req|
      !req.xhr? && req.format.html?
    }

    match '*unmatched', to: 'application#route_not_found', via: :all
  end

  get "up" => "rails/health#show", as: :rails_health_check
end
```

ТЕСТУВАННЯ

```
require "test_helper"

class MembersControllerTest < ActionDispatch::IntegrationTest
  setup do
    @board = boards(:one)
    @user = users(:Owner)
    sign_in @user
  end

  test "should get current" do
    get current_member_board_url I18n.locale, @board
    assert_response :success

    current = response.parsed_body
    assert_equal current["name"], @user.name
  end

  test "should get others" do
    get others_members_board_path I18n.locale, @board
    assert_response :success

    current = response.parsed_body
    assert_equal 5, current.length
  end

  test "should update role" do
    role = roles(:Editor)
    patch member_update_role_board_path(I18n.locale, @board, members(:b1Viewer)), params: { newRole: {
      name: role.name,
      can_edit: role.can_edit,
      can_change_roles: role.can_change_roles,
      can_assign_admin: role.can_assign_admin,
      can_ignore_rules: role.can_ignore_rules
    } }

    assert_response :success
  end

  test "should add to board" do
    assert_difference("Member.count") do
      post member_add_to_board_board_path(I18n.locale, @board), params: { name: users(:NonMember).name
}
    end

    assert_response :success
  end

  test "shouldn't add to board twice" do
    assert_no_difference("Member.count") do
      post member_add_to_board_board_path(I18n.locale, @board), params: { name: @user.name }
    end
  end

  test "should leave board" do
    assert_difference("Member.count", -1) do
      delete member_leave_board_board_path(I18n.locale, @board)
    end

    assert_response :success
  end

  test "should accept invite" do
    post member_accept_invite_board_path(I18n.locale, boards(:with_invite))

    assert_response :success
  end
end
```

ДОДАТОК В

ЛІСТИНГ КОДУ КЛІЄНТСЬКОЇ ЧАСТИНИ

UseStageScaleAndPosition

```
import { KonvaEventObject } from "konva/lib/Node";
import { useState } from "react";

const SCALE_BOUNDS = { min: 0.1, max: 20 };
const SCALE_BY = 1.1;

const getLimitedScale = (scale: number, min: number, max: number) => Math.max(min, Math.min(scale, max));

export default function UseStageScaleAndPosition(): {
	onWheel;
	onMouseMove;
	onMouseDown;
	onMouseUp;
	stagePosition;
	stageScale;
	isDragging;
} {
	const [stagePosition, setStagePosition] = useState({ x: 0, y: 0 });
	const [stageScale, setStageScale] = useState(1);

	const onWheel = (e: KonvaEventObject<WheelEvent>) => {
		e.evt.preventDefault();
		const stage = e.target.getStage();
		if (!stage) return;

		const oldScale = stage.scaleX();

		const pointerPosition = stage?.getPointerPosition();
		if (!pointerPosition) return;

		const mousePointTo = {
			x: (pointerPosition.x - stage.x()) / oldScale,
			y: (pointerPosition.y - stage.y()) / oldScale,
		};

		const newScale = e.evt.deltaY < 0 ? oldScale * SCALE_BY : oldScale / SCALE_BY;

		const finalScale = getLimitedScale(newScale, SCALE_BOUNDS.min, SCALE_BOUNDS.max);

		setStageScale(finalScale);
		setStagePosition({
			x: pointerPosition.x - mousePointTo.x * finalScale,
			y: pointerPosition.y - mousePointTo.y * finalScale,
		});
	};

	const [isDragging, setIsDragging] = useState(false);

	const onMouseDown = (e: KonvaEventObject<MouseEvent>) => {
		if (e.evt.button !== 2) {
			return;
		}

		setIsDragging(true);
	};

	const onMouseUp = () => {
		setIsDragging(false);
	};

	const onMouseMove = (e: KonvaEventObject<MouseEvent>) => {
		if (!isDragging) return;

		e.evt.preventDefault();
		const stage = e.target.getStage();
		if (!stage) return;

		setStagePosition({
			x: stage.x() + e.evt.movementX,
			y: stage.y() + e.evt.movementY,
		});
	};

	return { onWheel, onMouseMove, onMouseDown, onMouseUp, stagePosition, stageScale, isDragging
};
}
```

UseObjectsInteraction

```
import { useRef } from "react";
import { Point } from "../../../../Types/CanvasObjects";
import { KonvaEventObject } from "konva/lib/Node";
```

```
import { getCursorOnCanvas, isTooSmallDrag } from "../../../../scripts/canvasUtils";
import getOverlappingObjects from "../../../../scripts/CanvasObjects/getOverlappingObjects";
import { CanvasMode } from "../../../../Types/Canvas";
import UseObjects from "./UseObjects";
import UseTemporaryObject from "./UseTemporaryObject";
import { HistoryRecord } from "../../../../Types/History";
import { UseCanvasState } from "@/components/Board/CanvasStateContext";

type Props = {
	blocked: boolean;
	stageScale: number;
	handleHistory: {
		removeAdditionalHistoryDelay: () => void;
		changeObjects: React.RefObject<() => void>;
		historyHandleChanges: (record: HistoryRecord, waitForFinal?: boolean) => void;
	};
};

export default function UseObjectsInteraction({ blocked, stageScale, handleHistory }: Props) {
	const {
		canvasObjects,
		setCanvasObjects,
		addNewObject,
		moveSelectedObjects,
		moveSelectedLinePoint,
		resizeSelectedObjects,
		resourcesProperties,
	} = UseObjects({ handleHistory });
	const { temporaryObject, createTemporaryObject, updateTemporaryObject, deleteTemporaryObject }
=
		UseTemporaryObject();
	const { removeAdditionalHistoryDelay } = handleHistory;
	const { canvasState, canvasStateUtils } = UseCanvasState();

	const startingPoint = useRef<Point>({ x: 0, y: 0 });
	const mouseDown = useRef(false);

	function clickedOnStage(e: KonvaEventObject<MouseEvent>) {
		return e.target.getType() === "Stage";
	}

	function onMouseDown(e: KonvaEventObject<MouseEvent>) {
		if (e.evt.button !== 0 || blocked) return;
		e.evt.preventDefault();

		const cursorPoint = getCursorOnCanvas(e.target.getStage(), stageScale);
		if (!cursorPoint) return;
		startingPoint.current = cursorPoint;

		switch (canvasState.mode) {
			case CanvasMode.None:
				if (!clickedOnStage(e)) break;

				canvasStateUtils.SelectionNet.set(cursorPoint);
				break;

			case CanvasMode.Inserting:
				createTemporaryObject();
				break;

			case CanvasMode.Selected:
				if (!clickedOnStage(e)) break;

				canvasStateUtils.SelectionNet.set(cursorPoint);
				break;
		}

		mouseDown.current = true;
	}

	function onMouseMove(e: KonvaEventObject<MouseEvent>) {
		if (!mouseDown.current || blocked) return;
		e.evt.preventDefault();

		const currentPoint = getCursorOnCanvas(e.target.getStage(), stageScale);
		if (!currentPoint) return;
		switch (canvasState.mode) {
			case CanvasMode.SelectionNet:
				canvasStateUtils.SelectionNet.update(currentPoint);
				break;

			case CanvasMode.Inserting:
				updateTemporaryObject(startingPoint.current, currentPoint);
				break;

			case CanvasMode.Selected: {
				if (canvasState.resizing) {
					resizeSelectedObjects(currentPoint);
					break;
				}

				const movedBy = {
					x: currentPoint.x - startingPoint.current.x,
					y: currentPoint.y - startingPoint.current.y,
```

```
                                        };

                                        if (canvasState.lineModification) {
                                                moveSelectedLinePoint(movedBy);
                                        } else {
                                                moveSelectedObjects(movedBy);
                                        }

                                        startingPoint.current = currentPoint;
                                        break;
                                }

                                default:
                                        break;
                        }
                }

                function onMouseUp(e: KonvaEventObject<MouseEvent>) {
                        if (e.evt.button !== 0 || blocked) return;
                        e.evt.preventDefault();

                        switch (canvasState.mode) {
                                case CanvasMode.Inserting: {
                                        const currentPoint = getCursorOnCanvas(e.target.getStage(), stageScale);
                                        if (!currentPoint || !startingPoint.current || !temporaryObject) break;

                                        if (isTooSmallDrag(startingPoint.current, currentPoint)) break;

                                        addNewObject(temporaryObject);

                                        canvasStateUtils.Selected.set([temporaryObject]);
                                        break;
                                }

                                case CanvasMode.SelectionNet: {
                                        let overlappingObjects = getOverlappingObjects(canvasObjects,
canvasState.origin, canvasState.current);
                                        overlappingObjects = overlappingObjects.filter((obj) => !obj.locked);

                                        if (overlappingObjects.length === 0) {
                                                canvasStateUtils.None.set();
                                                break;
                                        }

                                        canvasStateUtils.Selected.set(overlappingObjects);
                                        break;
                                }

                                case CanvasMode.Selected:
                                        if (canvasState.lineModification) {
                                                moveSelectedLinePoint({ x: 0, y: 0 }, true);
                                                break;
                                        }

                                        if (canvasState.resizing) {
                                                const currentPoint = getCursorOnCanvas(e.target.getStage(),
stageScale);

                                                if (!currentPoint) break;

                                                resizeSelectedObjects(currentPoint, true);
                                                break;
                                        }
                                        break;
                        }

                        deleteTemporaryObject();
                        mouseDown.current = false;

                        removeAdditionalHistoryDelay();
                }

                return {
                        canvasObjects,
                        setCanvasObjects,
                        canvasUseObjectsInteraction: {
                                temporaryObject,
                                onMouseDown,
                                onMouseMove,
                                onMouseUp,
                        },
                        resourcesProperties,
                };
        }
```

Бібліографічна довідка

Тема роботи: Розроблення інформаційної системи онлайн-сервісу візуальної організації інформації для колективної роботи.

Обсяг бакалаврської роботи 70 сторінок.

Перелік графічного матеріалу:

1. Варіанти використання онлайн-сервісу візуальної організації інформації. Usecase діаграма
2. Діаграма робочих потоків веб-дошки для організації візуальної інформації
3. Сценарії ініціалізації нового робочого середовища та фонового збереження графічних маніпуляцій
4. Сценарій налаштування спільного доступу
5. База даних веб-дошки для організації візуальної інформації
6. Діаграма контролерів
7. Інформаційна система онлайн-сервісу візуальної організації інформації для колективної роботи. Робочі вікна

Дата закінчення БР

Студент Юрій ЧУМАКЕВИЧ